Symmetry Reduction for STE Model Checking using Structured Models

Ashish Darbari
Balliol College

Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD

To Baba and Daadi

Abstract

Symbolic trajectory evaluation (STE) is not sufficient to handle verification of circuits with large number of state holding elements such as memories. It is also the case that memory based circuits have plenty of symmetry, which can be exploited for computing reductions for STE model checking. This dissertation addresses the problem of symmetry reduction for STE model checking.

There are two main challenges involved in achieving an efficient solution to the problem of symmetry reduction. First is the discovery of symmetry in the circuit, and second, a methodology of computing reductions in the size of the STE model checking run.

To address the problem of finding symmetries in circuits, we propose a method of designing circuit models, so that symmetries in the structure of the circuit, can be recorded at the time of design. We propose a framework that allows us to model circuits using special functions, and a type system is provided to enforce discipline on the usage of these functions. A type soundness theorem then guarantees that circuits constructed using these functions have symmetry.

The other main contribution of our work is the design of a reduction methodology. This is centered around the use of novel STE inference rules, and a symmetry soundness theorem. Inference rules are used to decompose the original property into a set of smaller properties, which is then clustered into several equivalence classes based on the symmetry of circuit model. One representative property is chosen from each equivalence class and verified using an STE simulator, and then using the symmetry soundness theorem correctness of the entire class of equivalent properties is deduced. Inference rules are then used to compose the overall statement of correctness given by the original property.

We demonstrate the strength of our approach on several examples and case studies.

Declaration

This dissertation is the result of my research work carried out at Oxford University Computing Lab (OUCL), University of Oxford. The work described is my own unless otherwise stated in the text. The work presented in this thesis has not been, in full or in part, submitted for any other academic qualification.

Ashish Darbari
Oxford
August, 2006

Acknowledgement

It is a great feeling to get to this point of writing my acknowledgements - having travelled through a wonderful and adventurous journey of my DPhil, rich with various personal and professional experiences from around the world.

Every DPhil requires a reasonably challenging topic to work on, good guidance from a supervisor, and funding to sustain. In my case, I'm fortunate to have got all of the above. Having Tom around, to guide me in my research was the best I could hope for, from a supervisor. He not only advised me on academic, research based issues, but would also guide me in organising my research better. If I would find myself stuck somewhere, Tom would help me think about a solution on my own, and would give me a free rein to steer my research the way I wanted. He also provided valuable tips on theorem proving in HOL.

Throughout my research, I was funded through a combination of grants. I started my DPhil at the University of Glasgow who kindly provided funding for the first two years of my research. Together with the Overseas Research Studentship (ORS), and a donation from Intel Corp., USA, I was able to pay my overseas tuition fee and cover for my maintenance. In the last year of my research, I faced short-term cash flow problem, but I got timely help from Balliol college. Thanks to the college Chaplain Douglas Dupree, for arranging this on a short notice.

I travelled quite a bit during my DPhil and also had opportunities to meet different experts in the field and learn from their experiences. Mike Gordon hosted me in Cambridge in 2002 where I picked the basics of HOL theorem proving from him and Michael Norrish. Mike also encouraged me to implement the STE theory in HOL. This not only gave me an excellent opportunity to learn HOL, but also got an implementation of STE simulator that I used later in my research.

I had the opportunity to spend six months at the Strategic CAD Labs (SCL) at Intel, Oregon, for an internship. During my internship I worked closely with John O'Leary and Robert Jones. While John supervised me regularly, every now and then, Robert would step in with his suggestions and collectively this went a long way in shaping up the ideas that finally led to the development of the structured modelling language, and the symmetry reduction methodology. I would like to thank other members of SCL, including Jim Grundy, Jin Yang, Jeremy Casas, Sava Krstić and Timothy Kam. While Jim would help me understand the reflection features in reFLect, Jin would provide insights on the scope and potential of using different kinds of symmetries in hardware. Jeremy helped me get on with Forte related issues, and Sava helped clarify concepts related to the symmetry theory. Timothy kindly agreed to share insights on his experiences of formally verifying an industrial implementation of a cache. When I visited Intel for a week earlier in 2002, Mark Jones kindly hosted me at Oregon Graduate Institute. Thanks to Mark and her secretary Lorrie Gookin, I had a desk and morning coffee!

Steve McKeever and Bill Roscoe, at Oxford, examined me for my confirmation of status examination. Their feedback proved to be very useful in improving the style and contents of my dissertation.

I have had various fruitful discussions on my research with my friends and colleagues who deserve a special mention - Susanto Kong, Ed Smith, Tom Newcomb and Joe Hurd. I am deeply indebted to Susanto, Ed, Hasan and Tom who took their time to read my thesis and give me useful comments. They provided feedback ranging from minor typos to asking some hard questions, which eventually improved the quality of my presentation.

I would like to take this opportunity to thank the various academic and administrative staff at Glasgow and Oxford. My advisor Frances Kirwan in Balliol college was very welcoming and helped me acclimatize in Balliol when I came to Oxford. She made sure I had no problems throughout my stay. She also helped me apply for a funding to attend the TPHOLs conference in 2003.

Simon Gay was my co-supervisor in Glasgow. He helped me in my beginning stages of DPhil. Whilst I was at Glasgow, in the first two years of my PhD, I always got help from the staff there. Lynn Alexander secretary for the Faculty of Information and Mathematical Sciences, always extended a warm welcome, and helped me getting settled in. Gary Gray in computing support, helped me setting up my machine in the beginning and also helped me get started with VmWare.

When I moved from Glasgow to Oxford, I was embraced with a warm welcome in the ComLab by Julie Sheppard, Christine O' Connor, Wendy Adams, Joe Legett and Mike Field. From providing tips on how to arrange my accommodation in Oxford, to arranging paper work for smooth transfer to Oxford, all these people helped me in ways they could.

What would a lab be without IT administrators, and what would I have done without Terry Brown. On two occasions Terry gave me a life. Once days before the TPHOLs conference in Rome, I accidentally spilled coffee on my laptop, and on another occasion, this time days before I was going to India for my wedding, the boot partition of the laptop was broken. On both the occasions, Terry helped me recover my laptop exactly the way it was prior to the problem, with no data loss! Craig Tranefield often helped me sort out disk quotas, so that I could keep reading my emails.

Being a student at Balliol, has been a cherished experience. I not only met wonderful friends studying all sorts of subjects, I also came across wonderful staff. Senior Tutor's secretary Glynis Baleham and the college secretary Julia Hore deserve special mention.

In the beginning of January 2003, my sister gave birth to a baby girl who unfortunately did not live very long, and passed away in the August 2003. It was a traumatic period for my family in India, and for me in Oxford. I could not visit my sister when she needed me, and I always feel guilty for this, but during those hard days, I met several people in and around college, some of whom went on to become my best friends. These include Melissa Harrison, Emma Cavell, Gergeley Hidas, Patrizia Camelleti, Tobias Galla, Shozeb Haider, Asim Mahmood, Jaideep Das and Rakesh Roshan. Other friends of mine who I met later and who went on to be great pals include Omer Suleman, Anurag Bana, Aman Iqbal and Natasha Kuruppu.

Friends in Glasgow who still continue to be great friends include Abhishek Mahanty, Nikhil Lakhkar and Rajiv Mathur. Other friends and colleagues, who made my stay in Glasgow very enjoyable include Stephan Reiff-Marganiec, Pete Saffrey, Andrew McDonald and Susanto Kong.

During my stay in Oregon, I met Krishna Vaidyanathan, Rajnish Ghugghal and Sanjay Soni. Krishna helped me a great deal in helping me settle in, while Sanjay and I spent many evenings chilling out.

But of all my friends in the last so many years I owe most to Piyush Mishra, my mate from my undergraduate days in India. Piyush has always been a close friend, a guide, and someone who I can ring at 2 am in the morning and still not get admonished. Raghukul Tilak, another friend of mine from undergraduate days, visited me last Winter, and we had a very fruitful discussion on my research. Thanks to both of them for being so patient with me all these years.

In January 2005, I got married to Vidya Chandran. Life has changed for me in so many ways since then. Vidya was also in her final year of her Ph.D. in Cambridge, when we got married. Marrying in the final year of our PhDs was not an easy choice, and we had plenty of stressful times since then, but thanks to her courage and determination, we have both pulled it off. Thanks also to Vidya's brother Vinay and his wife Pratiksha, who work in India but found the time to go through my thesis, and gave me useful feedback. Vidya's parents, Amma and Appa have been a continuous source of inspiration ever since I met them. Even though we have not had a chance to stay with each other for long, on the phone they have always been very encouraging.

I met Tejal Patel through my wife Vidya. She has been a great friend, and I should especially thank her for bringing me the much needed Homeopathic medicines from India when I was suffering from coccydynia in my final stages of thesis writing.

The constant blessings from my well-wishers and elders in the family in India gave me strength during times of adversity, many thanks to them.

Last but not the least, my heartfelt gratitude to Mummy and Papa for being with me always, and telling me that one day it will be over, so I should never loose focus. My brother Abhinav and sister Tripti and her hubby Ashish, always kept cheering me and guess what, all of this worked!

Contents

List of Figures

List of Tables

List of Lemmas and Theorems

Chapter 1

Introduction

In spite of the huge success of STE model checking in large scale hardware verification, there has been little work done on using model specific information such as symmetries to perform property reduction. This dissertation explores the problem of symmetry reduction for STE model checking, and provides a methodology that shows how to make use of symmetries in circuit designs to compute reductions in the size of verification runs using STE model checking.

1.1 Conventional verification

Simulation has been the classic way of validating circuit designs prior to fabrication. It relies on testing a circuit design on a large set of test patterns, and this may help the designer in discovering bugs in the design. However, simulation is a time intensive process which only shows the presence of bugs, but cannot guarantee their absence [49].

An alternative to simulation is the use of formal verification for determining hardware correctness. Formal verification is, in a sense, a mathematical proof [41, 54, 66]. Just as a theorem holds for a particular set of values from a fixed domain, correctness of a formally verified hardware design, holds for all the values from a given domain. Hence one does not need to test the circuit on all or some of the data patterns in the domain, one can rather infer the correctness property from the proof of the theorem.

Formal verification methods, in addition to being theoretically sound, have been demonstrated to be practically effective in industry. In a news article at EETimes.com [79], the world's leading chip manufacturer Intel reported that by using formal verification they found more than 100 “high-quality” logic bugs that would not have been uncovered in any other way. In another recent study, Wolfsthal et al. [111] reviewed the benefits of formal verification by studying its usage in IBM over a period of 12 months. They discovered several hundred bugs using formal methods, and found that corner case bugs were most difficult to find even with carefully planned simulation environments developed with advanced tools.

1.2 Formal hardware verification

A formal hardware verification problem can be understood to consist of establishing, by mathematical proof, that an implementation satisfies a specification. The term *implementation* (*Imp*) refers to the hardware design that is to be verified. This can be a description of the circuit design at any level of abstraction for example a high-level VHDL program, or a gate-level netlist in Blif. A *specification*(*Spec*) refers to the property with respect to which correctness is to be determined. It can be expressed in a variety of ways, for example as a behavioural description, or an abstract structural description.

In formal hardware verification, we do not address the problem of *specification validation* [41]. Specification validation determines whether the specification means what it says, whether it really expresses the property one desires to verify, whether it completely characterizes the correct operation. It can be indirectly formulated in terms of the formal verification framework described above, in the following way.

A specification for a particular verification problem can itself be made the object of scrutiny [54], by serving as an implementation for another verification problem at a conceptually higher level. The purpose of the latter problem is to test if the meaning of the original specification is as intended, its meaning thus serving as a specification at the next higher level. A key thing to observe here is that this principle of scaling the problem to the next higher level does imply that at some level specifications are held to be infallible, a necessary characteristic of any mechanism for formal representation.

Similarly, at the other end of the spectrum, we do not specifically address the problem of *model validation* in formal hardware verification, i.e. whether the model used to represent the implementation is consistent, valid and correct. It is assumed that the quality of verification can only be as good as the quality of the models used.

1.2.1 Implementation

An implementation consists of a description of the hardware design that is to be verified. There are several ways and levels of abstraction hierarchy at which a design can be described. Some of these are:

1. Register transfer level — high level, includes structure and behaviour, generally written in VHDL, Verilog.
2. Gate level — schematic view with gates as building blocks, written in EDIF, Exlif, Blif.
3. Switch level — schematic view using transistors and other circuit components such as resistance, capacitance, inductance.
4. Layout — final design used to create the masks for fabricating the chip.

Some of the issues that a design description needs to address are: whether or not a certain switch level description models analogue behaviour, can a gate level description support model composition, hierarchy, and gate delays, and if a register transfer level description can model non-determinism, concurrency, and composition.

Depending on which choice we make for the description, we shall limit the class of circuits we can verify. For example, an approach that uses the switch level may be too fine grained for circuits involving latches, gates or pipelined components. On the other hand an approach that uses a pure "binary" switch level model may not be able to catch errors that result from analog effects such as leakage capacitances.

Similarly a representation that doesn't support hierarchical designs may not be able to describe large modular designs. We will show later how the choice of an implementation level affects the complexity of a verification task.

1.2.2 Specification

A specification is a description of the intended or required behaviour of a hardware design. Various formalisms have been used to represent specification. Some of the well known formalisms are:

> logic: propositional logic [27], first-order logic [24], higher-order logic [52, 74], temporal logic [35, 90], mu-calculus [10, 28].
>
> automata/language theory: finite state automata [67], trace structures [57].

The goal of these specifications is to be able to express correctness properties. Some of the desired properties are:

> functional correctness properties – e.g. a given adder circuit does indeed perform addition of the input values.
>
> safety and liveness properties – e.g. in a mutual exclusion system with two processors A and B, a safety property can express the fact that no two processors should have simultaneous access to the memory. A liveness property can express the fact that eventually each processor will be able to access the memory.
>
> timing property – access to the memory shall be granted within say 5ms of a processor making a request.

For each type of property to be verified it is the case that some formalisms are more suited for specification than others. For example for specifying liveness properties a logic that reasons explicitly about time, e.g. temporal logic would be more suitable than for example first-order logic. Another issue is the level of abstraction a formalism supports. More will be said on abstraction later, but for the time being, abstraction can be understood as the process of hiding irrelevant information from the description.

We use the language of STE [96], for property specification. We will show examples of doing this in Chapter 3.

1.2.3 Satisfaction

Formal verification involves proving that an implementation satisfies a specification. Some of the commonly encountered forms for establishing the formal relationship are:

theorem proving — relation between a specification and an implementation is regarded as a theorem in logic, to be proved within the context of a proof calculus, where the implementation provides axioms and assumptions that the proof can draw upon.

model checking — in many cases, specification is in the form of a temporal logic formula, the truth of which is determined with respect to a model provided by an implementation. There are however model checking techniques that do not use temporal logic formulas for specifying properties, for example CSP [25] based model checking.

equivalence checking — equivalence of a specification and an implementation is checked, e.g. equivalence of functions or finite state automata.

language containment — the formal language representing an implementation is shown to be contained in the formal language representing a specification.

1.3 STE model checking

Symbolic trajectory evaluation [96] – STE in short, is a model checking technique that combines the ideas of *ternary modelling* with *symbolic simulation*.

Ternary modelling is simulation over a three valued logic. The binary set of values {0, 1} is extended with a third value X. This third value is meant to capture those states in simulation where it is not known for certain whether the value is 0 or 1. By enforcing that the circuit models are monotonic, one can ensure that any binary value that results when simulating patterns containing X's would also result when the X's are replaced by any combination of 0's and 1's. Thus the number of patterns that must be simulated to verify a circuit can be dramatically reduced by representing many different operating conditions by patterns containing X's. By extending the circuit model of the circuit to one over the expanded state set, one can verify circuit behaviour for a set of different operating conditions containing only 0's and 1's, with a single simulation run containing some X's. Though ternary modelling allows us to cover many conditions with a single simulation run, it lacks the power required for complete verification, except for a small class of circuits such as memories [26].

It was shown in [20] that by combining ternary modelling with symbolic simulation, one can model even more complex sets of behaviours with a single simulation run. With ternary symbolic simulation, we go a step further and extend the ternary set of scalar values {0, 1, X} by a set of symbolic values. Each symbolic value indicates the value of a signal for many different operating conditions, parameterised in terms of a set of Boolean variables. Ternary symbolic simulation allows us to combine multiple ternary simulation sequences into a single symbolic sequence.

STE weds the idea of ternary symbolic simulation with the notion of specifying and verifying system behaviour over time.

Specifications take the form of what are known as *symbolic trajectory formulas*. By mixing the vocabulary of Boolean expressions with a *next-time* operator which expresses temporal behaviour, we get a symbolic trajectory formula. The advantage of using a Boolean expression is in specifying conveniently many different operating conditions in a compact form.

Verification takes place by testing the validity of an *assertion* of the form [A $\Rightarrow$ C], where both A and C are trajectory formulas. Intuitively the assertion is valid iff every state sequence that satisfies A must also satisfy C. This is done by generating a symbolic state sequence that satisfies the consequent. In fact there is a unique weakest symbolic sequence satisfying the antecedent.

Some of the important properties of the STE based algorithm is that it requires a small amount of simulation and symbolic manipulation to verify an assertion, when compared to other temporal logic based model checkers. The symbolic manipulation involves only variables explicitly mentioned in the assertion. We do not need to introduce extra variables denoting the initial circuit state or possible primary inputs. It is efficient because the length of simulation sequence depends only on the depth of nesting of temporal next-time operators in the assertion, and it achieves a cone-of-influence reduction [38], by simulating only the cone for those nodes that are mentioned in the consequent of the STE property.

1.4 Symmetry reduction for STE model checking

In this section we will outline the problem of symmetry reduction for STE model checking, and explain our motivation. STE has been successfully applied in several large scale circuit verification tasks [4, 8, 97], due to efficiency reasons explained in the previous section. But for verifying big circuit designs involving a large number of state holding elements like memories [82, 83], it cannot cope well. It is also the case however that circuits like memories intrinsically have structural symmetry and intuition suggests that this could be important for computing reductions of circuit models and properties that have to be verified. The challenge involved in using symmetry for computing the reductions of the circuit models and properties, rests on *finding* symmetry in circuits. One significant piece of research done in this area is Manish Pandey's [82, 83] work on using symmetries for reduction in STE model checking. One of the lessons learnt from his work serves as a prime motivation for our work. The lesson learnt is that the difficulty in a symmetry reduction approach is the *discovery* of symmetry, and proving that the symmetry found is indeed sound.

Depending on how we address this phase of the solution, it could have an impact on the size of the circuits we can verify, and on verification reuse. For example if we represent circuit models as graphs then finding symmetries in those graphs relies on finding appropriate heuristics. For each fresh circuit model that one tries to verify, one may have to come up with a new set of heuristics and, for growing sizes of a given circuit model, the amount of time and space grows accordingly [82].

The next-state function in STE that denotes the circuit model is totally *flat*. This means that it operates on each node in the circuit and gives the next state of the circuit but completely ignores any useful *structural* information that might have been specified in the design phase.

We wanted to investigate connections between structured models having symmetry and the theory of STE. We believe that finding the right link between them will enable us to use STE theory for property verification of symmetric models, and also use symmetry based arguments for computing property reductions.

However we also felt that inventing a whole new theory of STE that relates to a structured model may not be such a good idea. A significant proportion of the industrial community that uses STE for verification knows and understands the existing theory, and there are tools such as Forte [97] and other reduction techniques like symbolic indexing [20, 78, 84, 85] and parametric representation [5, 60], that work with the present theory. So we wanted to design a *minimally invasive* solution that can act as a plug-in to the existing theory, methods and tools thereby providing a seamless solution to the problem of symmetry reduction for STE model checking.

Thus in a nut shell our motivation is *to address the problem of symmetry reduction by finding a link between structured (symmetric) models and the STE theory in a manner that is minimally invasive to existing design and verification flows.*

In the next section we will highlight the important aspects of our proposed solution to the problem of symmetry reduction.

1.4.1 Positioning of this thesis

As stated in the previous section, our intention is to develop a framework whereby we can connect structured models with the logic of STE and do symmetry based problem reductions. To be able to do this we need to have a way to express structured models, discover what attributes of structure we want to capture in our models, and what significance those attributes have in capturing symmetry in the models.

We want to capture the *symmetric* attributes of a structured circuit model. So we set out from the very beginning focussing on that. We believe that symmetry, captured explicitly in the structure of models, can have a significant impact on STE property verification.

The notion of symmetry is formalised. We are interested in structural symmetry – symmetry that arises from the way replicated components in the circuit are used, giving rise to a circuit behaviour that remains independent under uniform permutations of its inputs and outputs. Whenever we say symmetry we mean structural symmetry.

We propose a language for structured models that can capture symmetry by defining a type of structured model and then presenting combinators that are used to construct structured models from other structured models. The choice of these combinators is influenced by our motivation to record symmetric aspects of the circuit. We then define a type system that imposes discipline on the syntax of the models constructed, allowing only those constructions that preserve symmetry. The implementation of the language and the type system was challenging, since we not only wanted to model circuits using the new language but also reason about it.

Reasoning is an important aspect in this approach because we would like to be able to say that if the structured models have been modelled using our language constructs and if they respect the type system of our language then they exhibit structural symmetries. This way we also prove that our type system is sound and by type checking infer that circuits have symmetry. This reduces the problem of symmetry discovery to the problem of type checking.

We deliberately design our type system based on simple type theory, to ensure that type checking is decidable.

We choose to design our language by shallowly embedding it in the language of a formal logic; our logic of choice is the higher-order logic since there has been ample evidence [33, 52, 74] to suggest that higher-order logic is well suited for modelling and reasoning about hardware. We use the interactive theorem prover HOL 4 [3] for our implementation framework. We commonly refer to this as HOL in our presentation.

Once we determine a language for structured models, the next step is to make a connection to the theory of STE logic. We make this connection by giving functions that derive flat STE models from the structured models and proving lemmas that guarantee that if the structured models have symmetry, then the corresponding derived flat STE model will have symmetry as well.

We then make a link that states what we can do with STE property verification once a given flat model has symmetry. This is done by proving a theorem that states that if a given STE model has symmetry then verifying a given property against this model is the same as verifying another property which is a permutation of the given property. This has a deep impact, because this helps cut down the effort of verifying all the STE properties (using an STE simulator) that belong to the same equivalence class modulo permutation to verifying just one – the representative of the equivalence class.

In order to do symmetry based reasoning we need to obtain smaller properties that are symmetric to one another. This means we need to decompose a given STE property with *many* symbolic variables to smaller properties with *fewer* symbolic variables in each one. Decomposing a given STE property into smaller properties relies on employing a suitable set of sound inference rules . We provide such a set of inference rules. Through a tactical (backward) use of these rules, properties can be decomposed into smaller properties and once we have determined that all reduced properties have been verified using STE and the symmetry argument, we can use the rules in the forward direction to construct the overall statement of correctness. This approach combines aspects of theorem proving with STE model checking.

Major steps involved in verifying a property by exploiting symmetries in circuit models are summarised below:

1. Discover symmetry in circuit model by performing type checking on circuit models.
2. Compute a set of reduced properties applying inference rules in the backward manner, to the original property.
3. Cluster the reduced properties into several equivalence classes, using the symmetry specific knowledge derived from circuit models.
4. Run an STE simulator to verify a representative property from each equivalence class.
5. Conclude, by means of a soundness theorem, that all the other properties in the same equivalence class have been verified.
6. Repeat Steps 4 and 5 until properties in all the clusters have been verified.

7. Use inference rules in the forward direction to compose the overall statement of correctness given by the original property.

1.4.2 Contributions

We have designed and developed a framework to address the problem of symmetry reduction in STE model checking. We show in this dissertation how we can cut down the size of an STE verification task significantly by exploiting symmetry that is captured in circuit models. The framework is designed to target symmetry-based reduction of circuit models and STE properties in a way that enables us to understand the connection between the world of structured models and the theory of STE.

We propose capturing symmetries in circuit models by recording it in the structure of the circuit at the time of modelling, using a language that we have designed. Key contributions in the design of our language have been in the choice of a data type of structured models, combinators that can be used to construct these structured models, and a new set of type inference rules that can be used for type checking of these models. We also prove a type soundness lemma that ensures that every circuit validated by our type checking rules has structural symmetry.

Designing the language was just one component in our framework. We need to derive flat models out of these structured models so that we can do STE simulation of these. We provide a method to do this, and ensure that the flat models have symmetry, whenever the structured model has.

The second main contribution of our work is the design of a reduction methodology. This strategy is centered around the use of inference rules, and a theorem that says that if a circuit model has symmetry then verifying a given STE property against that model is equivalent to verifying another one that is symmetric to the given property. Thus by using this theorem we verify one representative from each equivalence class of STE properties and deduce that all the others from the same equivalence class have been verified as well.

We also show several example circuits and case studies to illustrate our methodology and compare it to existing work.

Our work is designed in an open framework. All our results are available as HOL theories or scripts in Moscow ML [89] and Forte [2]. The underlying proof for every lemma and theorem shown in the thesis is a formal proof. It is formal since everything has been derived by way of using smaller properties, definitions, lemmas and theorems using the deductive calculus of higher order logic. Although the proofs have been machine checked with HOL, they were not auto-generated by it, HOL is very much like a programming environment - more so than other popular theorem provers such as PVS [98] or ACL2 [65]. We write proof-scripts in ML that when executed enact the strictly formal proof we have in mind. These scripts use tactics and conversions - technical constructs in HOL that are not very meaningful to readers who are not familiar to HOL. Because of this, we have chosen not to provide the HOL source code for the proofs in the thesis, but to give an informal outline of the formal proof itself. If one wants to explore such a proof, one can re-run it interactively to see what it is doing. On the other hand, because all the proofs are strictly formal, every tiny detail (even if it appears natural and obvious) has to be present explicitly at the time of doing the proofs. In particular, all *assumptions* about variables have to be explicitly present in the statement of the theorems. A lot of theorems in HOL, therefore, are in the form of an implication - *assumptions implies result*. As the reader may note, some of the proofs presented in the thesis are often quite simple. However, the main value in a full presentation of the kind we have given is to show the full development, showing all the definitions and assumptions involved, and how the lemmas fit together.

The work on structured model design has been presented recently at the 13th Workshop on Automated Reasoning (ARW 2006), and in the International Symmetry Conference (ISC 2007). The work on reduction methodology has been presented in the conference on Formal Methods in Computer Aided Design (FMCAD 2006).

1.5 Thesis outline

In the next chapter we shall present an overview of research done in abstraction and its application to formal hardware verification. We feel it is useful to see our research goals from an abstraction point of view.

Chapter 3 presents an introduction to the theory of STE. This chapter provides a useful background to understand the problem with the STE model checking, and appreciate the need to use symmetries.

The essentials of symmetry theory are presented in Chapter 4. We will examine the components of symmetry theory that are applicable to our research setting, and present important related work done in exploiting symmetry for other model checking techniques.

In Chapter 5, we present an overview of the problem of symmetry reduction, and highlight the various components of our solution. We explain the details of each component of our solution, in subsequent chapters.

Chapter 6 presents the structured type of circuit models and the type system for disciplining structured circuit model design. We present the definition of symmetry and prove the type soundness lemma. Several small examples are shown to illustrate the use of combinators in a type safe manner. This chapter also explains how to interpret the run-time behaviour of the structured models.

In Chapter 7, we provide methods to obtain flat STE models from structured models for simulation purposes. We explain the notion of equivalence between different modelling components in our framework and relate the concept of symmetry in structured models to symmetry in flat STE models.

Once the symmetry has been identified through type checking of structured models, the next step is to use symmetry arguments to do justify reduction of STE properties. We shall provide the theoretical details of how we do this in Chapter 8.

Chapter 9 shows examples and case studies to illustrate our solution, and compares our results with the best known in the field.

Chapter 10 summarises and concludes, and also points to future work.

Chapter 2

Role of Abstraction in Verification

Abstraction [35–37, 44, 45, 67, 93, 94] is the key idea for reduction in any model checking based verification. Often the challenges involved in any formal verification effort lie in addressing the problem of abstraction — computing the abstractions and ensuring that they are sound and complete. In this chapter we present different kinds of abstraction techniques widely studied and used in different verification settings.

2.1 Abstraction

In this section we review the standard abstraction techniques and their application to model checking. This will enable us to present our work in perspective of existing work, and motivate the need to discover abstraction and problem reduction techniques for STE model checking.

2.1.1 Classical abstraction

Abstraction is a technique of reducing the size of a model by hiding its irrelevant details. If we denote the original concrete model by $\mathcal{C}$ and an abstract one by $\mathcal{A}$ then when one says $\mathcal{A}$ is an abstraction of $\mathcal{C}$, it means that observable behaviours of $\mathcal{A}$ contain the observable behaviours of $\mathcal{C}$ relative to some *abstraction mapping* that converts behaviours of $\mathcal{C}$ into behaviours of $\mathcal{A}$.

A general framework of abstraction usually consists of the following:

1. choice of a logic for describing the system and expressing properties.
2. a method for obtaining abstract models.
3. a method of relating abstract and concrete models.

In most research the choice of the logic for specifying the properties has been a temporal logic such as CTL* [35] or LTL [90]. To describe the system model, one can use a variety of techniques. One can use an automata [9, 108], or a Kripke structure [35] or a labelled transition system (LTS), to describe the model in terms of its state space. Alternatively one can use a high level language [32, 40] as means of expressing the model, or even use very low level formulations like gate level or transistor level netlists [82, 83]. Netlists are however only used for describing hardware models, whereas the state space approaches and the high-level language descriptions are both capable of describing hardware and software. People have also used *higher order logic* [74] and the language of modal μ-calculus [46] for specifying a system and its properties.

As we shall show in Chapter 3, we use a restricted form of temporal logic in STE for specifying properties. System models are written in transistor level netlists, as a flat FSM [97] or in higher-order logic [47].

The approaches for relating abstract and concrete models generally fall into the following categories depending on whether they use a function or a relation. *Homomorphism* uses a function mapping to relate the concrete and abstract models, while *simulation* and *bisimulation* uses relations to relate the concrete and abstract models. Both homomorphism and simulation are defined as a one way mapping from a concrete to an abstract model, the assumption being that in order to get from the abstract to the concrete world, one can use the inverse of the mapping, as it is the case with bisimulation. While this is reasonable, it is not necessary, valid abstractions can be defined via two mappings, one from the concrete model to the abstract model and another from the abstract to the concrete - this exactly is the case with *Galois connection* - a classical abstraction technique [45].

We will review these abstraction techniques, and their application to model checking in more detail, in the subsequent sections. We will examine the abstraction inherent in STE model checking when we introduce STE in Chapter 3.

2.1.2 Homomorphism

Informally a homomorphism is a mapping from one mathematical object to another such that all relevant structures of the objects are preserved. Homomorphisms or generally morphisms have been extensively studied in several areas of mathematics notably in universal algebra [29], group theory [13] and category theory [71].

We define a base set also known as the *universe*, and then define a set of n-ary operators that operate on this universe. Together the universe and the set of operators, constitute an algebraic structure. The set of operators belong to a set of operation symbols that characterise the *language* or *type* of the algebra. Homomorphism then is a structure preserving mapping from one algebraic structure to another.

Definition 2.1. *A language or a type of an algebra is a set $\mathcal{L}$ of operation symbols such that a non-negative integer n is assigned to each member $op \in \mathcal{L}$.*

This integer is called the arity of *op*; and *op* is said to be an n-ary operator. An operator is finitary if the arity of the operator is some non-negative integer n.

Definition 2.2. *An algebra denoted by $\mathcal{A}$ is defined on a language $\mathcal{L}$ by an ordered pair $(A,\, O)$ where A is a nonempty set and O is a family of finitary operations on A such that corresponding to each n-ary operator op in $\mathcal{L}$ there is an n-ary operation op_A on A.*

Definition 2.3. *Suppose $\mathcal{A}$ and $\mathcal{B}$ are two algebras of the same type $\mathcal{L}$. A mapping $\mathcal{H} : A \rightarrow B$ is called a homomorphism from $\mathcal{A}$ to $\mathcal{B}$ if*

$$\mathcal{H}\, op_A(a_1, \ldots, a_n) \;=\; op_B(\mathcal{H}\, a_1, \ldots, \mathcal{H}\, a_n) \tag{2.1}$$

for each n-ary op in $\mathcal{L}$ and for all $a_1, \ldots, a_n$ in A.

If in addition, the mapping $\mathcal{H}$ is onto then $\mathcal{B}$ is said to be a *homomorphic image* of $\mathcal{A}$, and $\mathcal{H}$ is called an *epimorphism*.

The definition of homomorphism we presented above is very generic as usually found in the books of algebra [29]. However, in the usual discourse on abstraction in model checking, it is common to specialise the definition to the application setting. For example in the application of homomorphisms to abstraction in model checking [37], it is common to define the structure as a *transition system*, and the set of operators is given by an n-ary transition relation on the universe of the set of states of the system. We will present the details of this work in the next section.

An *isomorphism* can be defined as a homomorphism that is injective and surjective. If the mapping is defined from a set onto itself, then a homomorphism is referred to as an *endomorphism*. An *automorphism* is defined as an endomorphism that is also an isomorphism.

Homomorphism is a more general and very useful concept in formalising the notion of abstraction, since it enforces fewer restrictions on the mapping. This is not to say that specialised concepts of isomorphism and automorphism are not useful for abstraction purposes. In fact, automorphisms are a natural way to formalise the concept of symmetry, which we will present in detail in Chapter 4.

2.1.3 Galois connection

Galois connections are a classical abstraction framework [44, 45]. Before we describe this in detail, we recall some basic essential preliminaries.

A *poset* $(S, \widetilde{\sqsubseteq})$ is a partial order $\widetilde{\sqsubseteq}$ on a set S. A complete lattice $S(\widetilde{\sqsubseteq}, \bot, \top, \sqcup, \sqcap)$ is a poset $(S, \widetilde{\sqsubseteq})$ such that any subset X of S has a least upper bound (*lub*) which is denoted by $\sqcup X$ and a greatest lower bound (*glb*) which is denoted by $\sqcap X$. The smallest element of P is $\bot$, and $\top$ is the greatest. Let $(\mathcal{C}, \widetilde{\sqsubseteq})$ and $(\mathcal{A}, \preceq)$, be two *posets*, where $\mathcal{C}$ and $\mathcal{A}$ are sets of concrete states and abstract states respectively, and $\widetilde{\sqsubseteq}$ and $\preceq$ are orderings on the concrete set and the abstract set respectively.

A Galois connection is a pair of maps (α, γ):

$$\begin{aligned} \alpha &\in \mathcal{C} \rightarrow \mathcal{A} \\ \gamma &\in \mathcal{A} \rightarrow \mathcal{C} \end{aligned}$$

such that they obey the following condition:

$$\forall c \in \mathcal{C} \,.\, \forall a \in \mathcal{A}.\ \alpha(c) \preceq a \equiv c \ \widetilde{\sqsubseteq}\ \gamma(a) \tag{2.2}$$

The idea is that for a concrete set of states c there could be two abstract representations e.g., let c_1 be any concrete set of state, and a_1 and a_2 be any two abstract sets of states. If $a_1 = \alpha(c_1)$ and if $a_1 \preceq a_2$ then a_1 and a_2 are both abstractions of c_1, however since a_1 is lower in ordering than a_2, we say that a_1 is more precise than a_2. This can be expressed generally for any $c \in \mathcal{C}$ and $a \in \mathcal{A}$ as:

$$\alpha(c) \preceq a \tag{2.3}$$

Similarly, if $c_1 = \gamma(a_1)$ and $c_2 \tilde{\sqsubseteq} c_1$, then the set a_1 is an abstraction of c_2, and this concrete state c_2 provides more precise information than c_1. This can be expressed generally for any $c \in \mathcal{C}$ and $a \in \mathcal{A}$ as:

$$c \ \tilde{\sqsubseteq} \ \gamma(a) \tag{2.4}$$

When (2.3) and (2.4) are equivalent, we get the condition shown in (2.2).

In [45], P. Cousot and R. Cousot elucidate several properties of Galois connections. The key properties that capture the concept of abstraction are the following:

$$\forall c \in \mathcal{C}.\ c \ \tilde{\sqsubseteq} \ (\gamma \circ \alpha)(c) \tag{2.5}$$

$$\forall a \in \mathcal{A}.\ (\alpha \circ \gamma)(a) \preceq a \tag{2.6}$$

Each function in the pair (α, γ) uniquely determines the other.

$$\forall x \in \mathcal{C}.\ \alpha(x) = \sqcap\{y \in \mathcal{A} \mid x \ \tilde{\sqsubseteq} \ \gamma(y)\} \tag{2.7}$$

$$\forall y \in \mathcal{A}.\ \gamma(y) = \sqcup\{x \in \mathcal{C} \mid \alpha(x) \preceq y\} \tag{2.8}$$

The mappings α and γ preserve the least and the greatest elements of the concrete and abstract lattices. Formally this means:

$$\alpha(\bot^C) = \bot^A \tag{2.9}$$

$$\gamma(\top^A) = \top^C \tag{2.10}$$

Both α and γ are monotonic.

In the presentation above we have always referred to Galois connections on sets of states. One should note here, that during application of the Galois connection to model checking, it has been found useful to consider these sets as sets built over sets, i.e. a power set on the domain on state configurations.

2.1.4 Bisimulation and simulation

Bisimulation and simulation are both very useful abstraction techniques. Bisimulations were introduced by Park [86], and have been used in areas of model checking [59] and games [106].

Bisimulation

Let $\mathcal{M} = (\mathcal{P}, \mathcal{S}, \mathcal{R}, \mathcal{S}_0, \mathcal{L})$, and $\mathcal{M}' = (\mathcal{P}, \mathcal{S}', \mathcal{R}', {\mathcal{S}_0}', \mathcal{L}')$ be two mathematical structures with the same set of atomic propositions $\mathcal{P}$. The set of states are denoted by $\mathcal{S}$ and $\mathcal{S}'$ respectively for the two structures, and $\mathcal{S}_0$ and ${\mathcal{S}_0}'$ are the initial states of $\mathcal{M}$, and $\mathcal{M}'$. The transition relation for the two structures is given by $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, and $\mathcal{R}' \subseteq \mathcal{S}' \times \mathcal{S}'$, and $\mathcal{L}$ and $\mathcal{L}'$ are labelling functions for the two structures respectively. A bisimulation relation $\mathcal{B} \subseteq \mathcal{S} \times \mathcal{S}'$, is defined between $\mathcal{M}$ and $\mathcal{M}'$ if and only if for all $s \in \mathcal{S}$ and $s' \in \mathcal{S}'$, if $\mathcal{B}(s, s')$, then the following conditions hold:

1. $\mathcal{L}(s) = \mathcal{L}'(s')$.

2. For every state s_1, such that $\mathcal{R}(s, s_1)$, there exists another state s_1' such that $\mathcal{R}'(s', s_1')$ and $\mathcal{B}(s_1, s_1')$.

3. For every state s_1', such that $\mathcal{R}'(s', s_1')$, there exists another state s_1 such that $\mathcal{R}(s, s_1)$ and $\mathcal{B}(s_1, s_1')$.

The structures $\mathcal{M}$, and $\mathcal{M}'$ are bisimulation equivalent if there exists a bisimulation relation $\mathcal{B}$ such that for every initial state $s_0 \in \mathcal{S}_0$, there is an initial state $s_0' \in \mathcal{S}_0{}'$ in $\mathcal{M}'$, such that $\mathcal{B}(s_0, s_0')$. In addition, for every initial state $s_0' \in \mathcal{S}_0{}'$ in $\mathcal{M}'$ there is an initial state $s_0 \in \mathcal{S}_0$ in $\mathcal{M}$ such that $\mathcal{B}(s_0, s_0')$.

The concept of bisimulation is closely related to the concept of symmetry. In fact, if models are represented as state transition system, then the relation induced by the symmetry group is a bisimulation on the state transition system. This was observed by Somesh Jha in his Ph.D. thesis [59], and we will explain this a bit more in Section 4.4.

Simulation

Simulation is closely related to bisimulation. Bisimulation guarantees that two structures have the same behaviours, whereas simulation relates a structure to an abstraction of the structure. Since abstraction typically hides some of the details of the original structure, the abstracted structure might have a smaller set of atomic propositions. The simulation guarantees that every behaviour of a structure is also a behaviour of its abstraction. However, the abstraction may have behaviours, that are not possible in the original structure. Given two structures $\mathcal{M}$, and $\mathcal{M}'$ shown above, with the difference that the set of atomic propositions in $\mathcal{M}'$ is $\mathcal{P}'$, and $\mathcal{P}' \subseteq \mathcal{P}$, a relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}'$ between $\mathcal{M}$ and $\mathcal{M}'$ is a simulation relation if and only if for all $s \in \mathcal{S}$, and $s' \in \mathcal{S}'$, if $\mathcal{T}(s, s')$, then the following conditions hold:

1. $\mathcal{L}(s) \cap \mathcal{P}' = \mathcal{L}'(s')$.

2. For every state s_1 such that $\mathcal{R}(s, s_1)$, there is a state s_1', with the property that $\mathcal{R}'(s', s'_1)$ and $\mathcal{T}(s_1, s_1')$.

We say that $\mathcal{M}$ simulates $\mathcal{M}'$ if there exists a simulation relation $\mathcal{T}$ such that for every initial state s_0 in $\mathcal{M}$, there is an initial state s_0' in $\mathcal{M}'$ for which $\mathcal{T}(s_0, s_0')$.

Schmidt and Steffen in [94] use simulation relations to reason about preservation of program models from abstract to concrete domains. They use simulation relation as the basis of safe abstraction of a program model. They also note that the notion of a safe approximation via a Galois connection corresponds to the notion of simulation.

2.1.5 Predicate abstraction

Graf and Saidi [93] proposed a technique for constructing abstractions automatically for an infinite state system. Given a system and a partition of the state space induced by n predicates $\phi_1, \ldots, \phi_n$ on the concrete program variables which defines an abstract state graph starting in the abstract initial state. The possible successors of a state are computed using the PVS theorem prover by verifying for each index i if ϕ_i or $\neg\phi_i$ is a postcondition of it. The reason it is called *predicate abstraction* is because the abstract states have only Boolean variables, and an abstract transition system is formalised by a predicate over these Boolean variables, to capture non-determinism. The abstract state graph is defined in terms of a Galois connection. This approach was used to verify a bounded retransmission protocol developed by Philips. This protocol is an extension of the alternating bit protocol where instead of single messages, message packets are transmitted and the number of possible retransmissions per message is bounded by some upper bound.

2.1.6 Other forms of abstraction

Melham [74], identified the following basic abstraction techniques for hardware specification:

- Structural abstractions: these involve suppression of internal structural details, providing an externally observable view of the implementation. These kind of abstractions are very useful in modelling circuit components. For example, a simple inverter can be modelled by a *not* function at the gate-level, or can be modelled by using CMOS transistors, an example of which can be found in [74]. The impact of structural abstractions on our work is significant, and will become evident when we discuss our work in Chapter 5 and Chapter 6 of our thesis.

- Data abstractions: these change the representation of data between models. We will show later in Chapter 3, examples of data abstraction in STE, via X's and symbolic indexing.

- Behavioural abstractions: these suppress behaviours between models, e.g. by hiding certain modes of operation. This kind of abstraction becomes useful in many complex designs, where one would typically hide a certain non-interesting mode of operation of a device to check properties for other more relevant parts.

- Temporal abstractions: these change the time scale between the models, for example the rise times, setup and hold times in a rising-edge latch can be abstracted and the latch can be modelled as a unit-delay device. In our work, we have used this abstraction model for latches, and flip-flops. Also we will see later that the combinational elements are modelled by zero delays, again a temporal abstraction of the real behaviour of the combinational element.

2.2 Abstraction in model checking

2.2.1 Homomorphism based abstraction

Clarke, Grumberg and Long [37] present a collection of abstraction techniques that can be used for model checking. They use homomorphisms as a central tool to compute abstractions of program models directly from the text of the program, without ever having to examine the unabstracted model.

Programs are modelled as transition systems; the logic for specifying properties and model checking is the propositional temporal logic CTL*. Instead of finding out whether or not a given CTL* formula is satisfied by a given program model, the solution relies on finding out if the same CTL* formula is satisfied by an abstraction of the given program model. The abstraction method is based on the idea of homomorphisms.

We will commonly refer to the original program model as the concrete model and the abstraction of the program model as the abstract model. A state of a program is defined by an assignment of values from a given domain to the program variables occurring in the program.

If we denote the set of all program states in the concrete model by $\mathcal{C}$ then a concrete model denoted by $\mathcal{M}_\mathcal{C}$ is defined by a triple

$$\mathcal{M}_\mathcal{C} = (\mathcal{C},\, I_\mathcal{C},\, R_\mathcal{C})$$

where $\mathcal{C}$ is the set of all concrete states, $I_\mathcal{C}$ is the set of initial states, and $R_\mathcal{C} \subseteq \mathcal{C} \times \mathcal{C}$ is the transition relation.

To perform an abstraction we need to define the notion of an abstract state. An abstract state is an assignment of abstract values to program variables. This of course assumes that we have a domain of abstract values from which we can make such assignments. If we denote the set of all abstract states by $\mathcal{A}$, then an abstract program model is given by a triple

$$\mathcal{M}_\mathcal{A} = (\mathcal{A},\, I_\mathcal{A},\, R_\mathcal{A})$$

where $\mathcal{A}$ is the set of all abstract states, $I_\mathcal{A}$ is the set of initial abstract states, and $M_\mathcal{A} \subseteq \mathcal{A} \times \mathcal{A}$ is the abstract transition relation. Homomorphisms are used to relate the concrete and abstract models. The concrete model $\mathcal{M}_\mathcal{C}$ is abstracted by the abstract model $\mathcal{M}_\mathcal{A}$ when the following conditions are met:

$$I_\mathcal{A}(a) \equiv \exists c.(\mathcal{H}(c) = a \wedge I_\mathcal{C}(c))$$

$$R_\mathcal{A}(a_1, a_2) \equiv \exists c_1 \exists c_2.\ (\mathcal{H}(c_1) = a1) \wedge (\mathcal{H}(c_2) = a_2) \wedge R_\mathcal{C}(c_1, c_2)$$

Homomorphisms as shown above are referred to as *exact* homomorphisms and the abstraction they lead to ($\mathcal{M}_\mathcal{A}$) is known as the *canonical* or exact abstraction of $\mathcal{M}_\mathcal{C}$. Computing this canonical abstraction is expensive since it requires computing a relation over abstract states that involves existential quantifications over concrete states. The authors propose to work around this problem by pushing the existential quantifications inside of the conjuncts thereby yielding an approximation of the canonical abstraction. Approximation however leads one to only check for a weak preservation of the properties; one can only check a subset of CTL* namely $\forall$CTL*. A canonical abstraction allows one to check for a stronger preservation of the property namely the full CTL*.

Homomorphisms induce equivalence relations on the set of concrete program states. If two states map to the same abstract state then they are in the same equivalence class. The approximate abstraction can be exact if the homomorphisms that relate the concrete and abstract model induce an equivalence relation that is congruent with respect to the operations used in the program.

The method of Clarke et al. has several contributions. First they advocate attacking the syntax of the program directly to compute the abstractions. Second they show that not every kind of property can be verified against the abstract model and then they present results that show which fragment of CTL is preserved by which kind of abstraction. Moreover they also provide in the second half of their paper [37] other kinds of useful abstractions that can be used like single bit and product abstractions, congruence modulo n abstraction and symbolic abstractions.

Clarke et al. were not the first to use homomorphisms in performing abstractions. Kurshan [67] also used homomorphisms to relate the concrete and abstract models. He phrased the correctness criterion as an ω-language containment problem rather than model checking using a temporal logic. In his framework both the transition system and the properties to be verified are modelled as automata. Verification is done by checking containment of the language of a system in the language of the property.

In Kurshan's approach the user specifies the homomorphisms and they are used to ensure that the relation of the concrete and the abstract model is sound. The user supplied homomorphisms are checked automatically by exploring the state space of the concrete model. This idea of automation also points out an important deficiency – what if the original model is too big to explore tractably. In this respect Clarke et al's approach is superior since they can handle relatively large models because they do not rely on constructing the original concrete state space of the model.

2.2.2 Counterexample based abstraction refinement

In more recent work, Clarke et al. [36], propose an extension of [37]. In [37], once the abstract transition relation has been constructed, a traditional model checker is invoked to determine whether a given property expressed in $\forall$CTL* is satisfied by the abstract model. If the answer is yes, then the concrete model also satisfies the property, and if the answer is no, then the model checker generates a counterexample. Since the abstract model is approximate, it can have possibly more behaviours than the concrete one, and therefore the counterexample generated may not be a correct one. In [36], the authors propose an automatic technique of refining abstractions, using the information from the spurious counterexamples. The key idea in doing the abstraction refinement here is to obtain the coarsest refinement which eliminates the counterexample. The coarsest abstraction gives the smallest abstract model. The resulting abstract model is used to verify the property, and if the spurious counterexamples are not ruled out, the refinement process continues. The authors note that the problem of finding the coarsest abstraction is NP-hard so they devise heuristics to tackle the computation of refinements in polynomial time. The result is a methodology that is complete for the fragment of $\forall$CTL*. The computation of initial abstraction and refinement steps is entirely automatic and algorithms used are all symbolic. Another advantage of this approach, is that the abstraction refinement is of much finer granularity, and is guaranteed to eliminate spurious counterexamples, while keeping the model small.

2.2.3 Automatic predicate abstraction of C programs

As a part of the SLAM project at Microsoft [18, 19], there has been recent advances [15] in the application of abstraction techniques to model check Windows device drivers, written in the C programming language. The key idea is in automatic abstraction [16] of C programs, which is based on predicate abstraction [93] and includes a purpose built model checking tool [17] designed for checking the abstract programs, and a tool that does abstraction refinement [18].

2.3 Summary

In this chapter we reviewed some basic forms of abstraction and their application in model checking. In the next chapter we present the theory of STE, and we shall see how abstraction is done in STE model checking.

Chapter 3

Essentials of STE Theory

In this chapter we present an introduction to Symbolic Trajectory Evaluation. We present the syntax, semantics and the main artifacts of STE model checking, highlighting its strengths and limitations.

3.1 Symbolic trajectory evaluation

Symbolic Trajectory Evaluation [96], or STE for short, combines the ideas of *ternary modelling* [26] and *symbolic simulation*. In ternary modelling the binary set of values {0, 1} is extended with a third value X that indicates an unknown or indeterminate logic value. Assuming a monotonicity property of the simulation algorithm one can ensure that any binary value resulting when simulating patterns containing X's would also result when each X is replaced by either a 0 or a 1. Thus the number of patterns that must be simulated to verify a circuit can often be reduced dramatically by representing many different operating conditions by patterns containing X's. With ternary simulation, a state with some nodes set to X covers those circuit states obtained by replacing the X values with 0's and 1's. The state with all nodes set to X thus covers all possible actual circuit states. By extending the next-state function of the circuit to one over the expanded state set, we can verify circuit behaviour for a set of different operating conditions with a single simulation run.

Although ternary modelling allows us to cover many conditions with a single simulation run, it lacks the power required for complete verification, except for a small class of circuits such as memories [26]. Symbolic trajectory evaluation goes one step further from the idea of ternary symbolic simulation [20]. It extends the notion of ternary symbolic simulation to include the means of specifying and verifying system behaviour over time.

3.2 The four valued lattice

Symbolic trajectory evaluation employs a ternary circuit state model, in which the usual binary values 0 and 1 are augmented with a third value X that stands for an unknown. To represent this mathematically, we introduce a partial order relation $\sqsubseteq$, with X $\sqsubseteq$ 0 and X $\sqsubseteq$ 1. The relation orders values by information content: X stands for a value about which we know nothing, and so is ordered below the specific values 0 and 1.

We can turn the set of ternary values into a complete lattice where every element has a *join* or a least-upper bound, and a *meet* or a greatest-lower bound, by adding an artificial element - $\top$ (called 'top') to get the set of circuit node values $\mathcal{D} = \{0, 1, X, \top\}$. We use the dual-rail encoding [95], to define the four lattice values in the STE logic. The idea of dual-rail encoding is to use the Boolean set $\mathcal{B}$ of values to denote the four lattice values, by having the concept of a high-rail and a low rail. Whenever a lattice value is 1 the high rail will take on a Boolean T and the low rail a Boolean F. If the lattice value is 0 the high and low rail are just the inverse of the case for 1. Similarly the other lattice values defined as shown below.

Definition 3.1. *Lattice values*

$$\begin{array}{lcl} \top & \triangleq & (F, F) \\ 1 & \triangleq & (T, F) \\ 0 & \triangleq & (F, T) \\ X & \triangleq & (T, T) \end{array}$$

Every pair of lattice values has a least upper bound that is defined as follows.

Definition 3.2. *Least upper bound*

$$(a, b) \;\sqcup\; (c, d) \quad \triangleq \quad (a \wedge c, b \wedge d)$$

The information ordering ($\sqsubseteq$) is formally defined as

Definition 3.3. *Information ordering*

$$a \sqsubseteq b \quad \triangleq \quad (b = a \sqcup b)$$

Lemma 3.1. *Information ordering is reflexive*
$\vdash \; \forall a.\, a \;\sqsubseteq\; a$

Proof: The proof follows trivially from the definition of information ordering ($\sqsubseteq$), definition of least upper bound ($\sqcup$) and the fact that $\wedge$ is idempotent. □

Lemma 3.2. *Information ordering is transitive*
$\vdash \; \forall a\, c.\, (\exists b.\, a \;\sqsubseteq\; b \;\wedge\; b \;\sqsubseteq\; c) \;\supset\; (a \;\sqsubseteq\; c)$

Proof: Case analysis on a and c followed by a case analysis on b, and using the definition of $\sqsubseteq$ and $\sqcup$ to simplify. □

Important properties about least upper bound

In this section we show some of the important properties of least upper bound. All these properties are used in the proofs (Section 3.7) involving trajectory formulas that we will introduce in Section 3.4.

Lemma 3.3. $\sqcup$ *is idempotent*
$\vdash \ \forall a.\ a = (a \sqcup a)$

Proof: Follows by case analysis on a and by unfolding the definition of $\sqcup$. □

Lemma 3.4. $\sqcup$ *is commutative*
$\vdash \ \forall a\, b.\ (a \sqcup b) \ = \ (b \sqcup a)$

Proof: By using the definition of $\sqcup$ and using the commutativity property of the logical and $\wedge$. □

Least upper bound is monotonic

Lemma 3.5. $\sqcup$ *is monotonic*
$\vdash \ \forall a\, b.\ a \ \sqsubseteq \ (a \sqcup b)$

Proof: By a case analysis on a and b and then applying the definition of $\sqsubseteq$ and $\sqcup$ for simplification. □

The following three lemmas are special cases exhibiting the monotonicity of the least upper bound that we need in the proofs in Section 3.7.

Lemma 3.6. $\sqcup$ *is monotonic*
$\vdash \ \forall a\, b\, c\, d.\ ((a \ \sqsubseteq \ c) \ \wedge \ (b \ \sqsubseteq \ d)) \ \supset \ ((a \sqcup b) \ \sqsubseteq \ (c \sqcup d))$

Proof: By case analysis on all the quantified variables followed by simplification using the definitions of $\sqsubseteq$ and $\sqcup$. □

Lemma 3.7. $\sqcup$ *is monotonic*
$\vdash \ \forall a\, b\, c.\ (a \ \sqsubseteq \ b) \ \supset \ (a \ \sqsubseteq \ (b \ \sqcup \ c))$

Proof: By case analysis on all the quantified variables followed by simplification using the definitions of $\sqsubseteq$ and $\sqcup$. □

Lemma 3.8. $\sqcup$ *is monotonic*
$\vdash \ \forall a\, b\, c\, d.\ (a \ \sqsubseteq \ (c \ \sqcup \ d)) \ \wedge \ (b \ \sqsubseteq \ d) \ \supset \ ((a \sqcup b) \ \sqsubseteq \ (c \sqcup d))$

Proof: By case analysis on all the quantified variables followed by simplification using the definitions of $\sqsubseteq$ and $\sqcup$. □

We suppose there is a set of circuit nodes naming observable points in circuits. Since nodes are just names, nodes can be attributed the type of *string*. A state is then an instantaneous snapshot of circuit behaviour given by an assignment of lattice values to nodes. If *bool* denotes the type of Booleans, then formally, a state is represented by a function

$$s : \mathit{string} \rightarrow \mathit{bool} \times \mathit{bool}$$

that maps each node name to a lattice value. Note the use of dual-rail encoding that we explained in the previous section. To model dynamic behaviour, we represent time by the natural numbers *num*. A sequence of states that the circuit passes through as it evolves over time is then represented by a function from time to states:

$$\sigma : \ num \rightarrow string \rightarrow bool \times bool$$

One convenient operation, which will be used below in stating the semantics of STE, is taking the i^{th} suffix of a sequence and is defined as follows.

Definition 3.4. *Suffix of a sequence*

$$\sigma_i \ \triangleq \ \ \lambda\, t\, n.\, \sigma\, (t+i)\, n$$

The suffix function shifts the sequence σ forward i points in time, ignoring the states at the first i time units.

We extend the ordering $\sqsubseteq$ point-wise to get lattices of states and sequences:

Definition 3.5. *Information ordering on states*

$$s_1 \sqsubseteq_s s_2 \ \ \triangleq \ \ \forall n : string.\ s_1\ n \sqsubseteq s_2\ n$$

Definition 3.6. *Information ordering on sequences*

$$\sigma_1 \sqsubseteq_\sigma \sigma_2 \ \ \triangleq \ \ \forall t : num. \forall n : string.\ \sigma_1\ t\ n \sqsubseteq \sigma_2\ t\ n$$

These inherit the information-content ordering from the basic four-valued lattice. If $\sigma_1 \sqsubseteq_\sigma \sigma_2$, then σ_1 has 'less information' about node values than σ_2, i.e. it has X's in place of some 0's and $1's$. We say that σ_1 is 'weaker than' σ_2.

3.3 Circuit models

In STE circuit models are represented by a next-state function from states to states. This is also known as an excitation function and is formally given by a function:

$$\mathcal{M} : (string \rightarrow bool \times bool) \rightarrow (string \rightarrow bool \times bool)$$

A requirement for trajectory evaluation is that the next-state function be monotonic. The notion of monotonicity is closely related to the information ordering on lattice values $\sqsubseteq$. The next-state function cannot gain any information by reducing the information content of the present state. Thus if we changed the present state from a lattice 1 to X, then the next state will either remain unchanged or result in an X as well. Monotonicity for circuit models can be defined as:

Definition 3.7. *Monotonicity of circuit models*

$$Monotonic\ \mathcal{M} \ \ \triangleq \ \forall s\, s'.\, (s\ \sqsubseteq_s\ s') \supset ((\mathcal{M}\ s)\ \sqsubseteq_s\ (\mathcal{M}\ s'))$$

A sequence σ is said to be a *trajectory* of circuit if the set of behaviours that the sequence represents is a subset of the behaviours that the circuit can exhibit. Formally, the set $\mathcal{T}(\mathcal{M})$, is defined as:

$$\mathcal{T}(\mathcal{M}) \triangleq \{\sigma \mid \forall t.\, \mathcal{M}(\sigma\ t) \sqsubseteq_s \ \sigma(t+1)\}$$

This means that the result of applying $\mathcal{M}$ to a state at any time in the sequence σ, must be no more specified with respect to the information ordering ($\sqsubseteq$), than the state at the next moment of time in σ.

Specifications in STE take the form of what are known as *symbolic trajectory formulas*. Trajectory formulas are written by using atomic propositions about nodes in the circuit, a temporal next-time operator (used to write properties which reason about the circuit behaviour over time), and propositional logic formulas, which help us in defining constraints (on circuit node values) under which a given trajectory formula will be true.

3.4 Syntax of STE

Formally, we define the syntax of formulas [76] as follows.

Definition 3.8. *Syntax of STE formulas*

$$\begin{array}{rlll} f & \triangleq & n \ \mathsf{is}\ 0 & \text{- node } n \text{ has value } 0 \\ & \mid & n \ \mathsf{is}\ 1 & \text{- node } n \text{ has value } 1 \\ & \mid & f \ \mathsf{and}\ g & \text{- conjunction of formulas} \\ & \mid & f \ \mathsf{when}\ P & \text{- } f \text{ is asserted only when } P \text{ is true} \\ & \mid & \mathsf{N}f & \text{- } f \text{ holds in the next time step} \end{array}$$

where f and g range over formulas, $n \in$ *string* ranges over the nodes of the circuit, and P is a propositional formula over Boolean variables (i.e. a Boolean 'function') called a *guard*. The advantage of using a Boolean expression is in specifying conveniently many different operating conditions in a compact form. The various guards that occur in a trajectory formula can have variables in common, so this mechanism gives STE the expressive power needed to represent interdependencies among node values. For example consider the following formula:

$$((\text{“}a\text{”} \ \mathsf{is}\ 1) \ \ \mathsf{when}\ v_a) \ \mathsf{and}\ ((\text{“}a\text{”} \ \mathsf{is}\ 0) \ \mathsf{when}\ {\sim} v_a)$$

which expresses the property that the node "a" has the Boolean value v_a at time point 0. This can be succinctly expressed as:

$$\text{“}a\text{”} \ \mathsf{is}\ v_a$$

STE formulas are often implemented as a list of '5-tuples', where the five elements of each tuple are the guard, a node name, a Boolean value (which may be a variable), a natural number for start time and another natural number for end-time. The outcome of this implementation is a set of infix functions (is, and, when, N), that enable us to express the trajectory formulas. The definitions are shown below:

$$\begin{array}{lcl} n\ \mathsf{is}\ v & \triangleq & [(T, n, v, 0, 1)] \\ f\ \mathsf{and}\ g & \triangleq & f\ \mathit{append}\ g \\ f\ \mathsf{when}\ p & \triangleq & \mathit{map}(\lambda(q, n, v, t_0, t_1).(p \wedge q, n, v, t_0, t_1))\ f \\ \mathsf{N}f & \triangleq & \mathit{map}(\lambda(g, n, v, t_0, t_1).(g, n, v, t_0 + 1, t_1 + 1))\ f \end{array}$$

where *append* is an append on lists.

If we wish to express the fact that the node "a" has a Boolean value v_a at present time point, and upto (and including) time point 2, using the above infix functions, it can be stated as

$$(``a"\ \mathsf{is}\ v_a)\ \mathsf{and}\ \mathsf{N}(``a"\ \mathsf{is}\ v_a)\ \mathsf{and}\ \mathsf{N}(\mathsf{N}(``a"\ \mathsf{is}\ v_a))$$

This is clearly cumbersome to express using iterations of the N operator. Thus two convenient functions *from* and *to* are defined. Using the tuple-representation we can define them as:

$$f\ \mathit{from}\ t_0 = \mathit{map}(\lambda(g, n, v, z, t_1).(g, n, v, t_0, t_1))\ f$$

$$f\ \mathit{to}\ t_1 = \mathit{map}(\lambda(g, n, v, t_0, z).(g, n, v, t_0, t_1))\ f$$

Thus the formula stated above can be written conveniently as

$$(``a"\ \mathsf{is}\ v_a)\ \mathit{from}\ 0\ \mathit{to}\ 3$$

and it can be parsed for use with STE implementations, using the above 5-tuple definitions.

3.5 Semantics of STE

We now define when a sequence σ satisfies a trajectory formula f. Satisfaction is defined with respect to an assignment ϕ of Boolean truth-values to the variables that appear in the guards of the formula:

$$\begin{array}{lcl} (\phi, \sigma) \models n\ \mathsf{is}\ 0 & \triangleq & 0 \sqsubseteq \sigma\ 0\ n \\ (\phi, \sigma) \models n\ \mathsf{is}\ 1 & \triangleq & 1 \sqsubseteq \sigma\ 0\ n \\ (\phi, \sigma) \models f\ \mathsf{and}\ g & \triangleq & (\phi, \sigma) \models f \wedge (\phi, \sigma) \models g \\ (\phi, \sigma) \models f\ \mathsf{when}\ P & \triangleq & (\phi \models P)\ \supset\ (\phi, \sigma) \models f \\ (\phi, \sigma) \models \mathsf{N}f & \triangleq & (\phi, \sigma_1) \models f \end{array}$$

where $\phi \models P$ means the assignment of truth-values given by ϕ satisfies the formula P. Logical "and" is denoted by $\wedge$ and "implication" by $\supset$. We write $\phi \models P$ to mean that the propositional formula P is true under the valuation ϕ. Note that $\models$ here is overloaded.

Verification takes place by testing the validity of an *assertion* of the form $A \Rightarrow C$, where both A and C are trajectory formulas. The intuition is that the antecedent A provides stimuli to the circuit nodes, and the consequent C specifies the expected values at circuit nodes. Informally, the assertion is valid iff every state sequence that satisfies A must also satisfy C.

A trajectory assertion is true for a given assignment ϕ of Boolean values to the variables in its guards exactly when every trajectory of the circuit that satisfies the antecedent also satisfies the consequent. For a given circuit model $\mathcal{M}$, we define $\phi \models A \Rightarrow C$ to mean that for all $\sigma \in \mathcal{T}(\mathcal{M})$, if $(\phi, \sigma) \models A$ then $(\phi, \sigma) \models C$. The notation $\models A \Rightarrow C$ means that $\phi \models A \Rightarrow C$ holds for all ϕ.

3.6 STE verification engine

The key feature of this logic is that for any trajectory formula f and assignment ϕ, there exists a unique weakest sequence that satisfies f. This sequence is called the *defining sequence* for f and is written $[f]^\phi$. It is defined recursively as follows:

Definition 3.9. *Defining Sequence*

$$\begin{array}{lcl}
[m \ \mathsf{is}\ 0]^\phi\ t\ n & \triangleq & 0 \ \textit{if}\ m{=}n \ \textit{and}\ t{=}0, \ \textit{otherwise}\ X \\
[m \ \mathsf{is}\ 1]^\phi\ t\ n & \triangleq & 1 \ \textit{if}\ m{=}n \ \textit{and}\ t{=}0, \ \textit{otherwise}\ X \\
[f \ \mathsf{and}\ g]^\phi\ t\ n & \triangleq & ([f]^\phi\ t\ n) \sqcup ([g]^\phi\ t\ n) \\
[f \ \mathsf{when}\ P]^\phi\ t\ n & \triangleq & [f]^\phi\ t\ n \ \textit{if}\ \phi \models P, \ \textit{otherwise}\ X \\
[\mathsf{N} f]^\phi\ t\ n & \triangleq & [f]^\phi\ (t{-}1)\ n \ \textit{if}\ t{\neq}0, \ \textit{otherwise}\ X
\end{array}$$

For any ϕ and σ, we have that $(\phi, \sigma) \models f$ if and only if $[f]^\phi \sqsubseteq \sigma$. The proof can be done by inducting on f and unfolding the definition of the STE semantics ($\models$) and that of the defining sequence. Note that defining sequence for a trajectory formula can return the lattice value $\top$, if the trajectory formula expresses an inconsistent property — for example consider a property that expresses that a node is wired to the Boolean high V_{cc} and the Boolean low *Gnd*.

Generally, inconsistencies can also be a result of a faulty implementation given by a circuit's model or an inconsistent specification (antecedent or a consequent) given by an STE property. The inconsistencies in the consequent of the specification for a given property is captured at the time of computing the defining sequence, while inconsistency in the antecedent or in the circuit's model is discovered at the time of computing what is called a *defining trajectory* of the antecedent of the STE property. These checks for inconsistencies are sufficient to detect bugs in models or specifications because of the way STE model checking is implemented, see Theorem 3.1 below.

Defining trajectory of a formula is its defining sequence with the added constraints on state transitions imposed by the circuit model $\mathcal{M}$.

Definition 3.10. *Defining Trajectory*

$$\begin{array}{lcl}
[\![f]\!]^\phi\ \mathcal{M}\ 0\ n & \triangleq & [f]^\phi\ 0\ n \\
[\![f]\!]^\phi\ \mathcal{M}\ t\ n & \triangleq & [f]^\phi\ t\ n \ \sqcup\ \mathcal{M}\,([\![f]\!]^\phi\ \mathcal{M}\ (t{-}1))\ n
\end{array}$$

It can be shown that for any ϕ and σ, we have that $(\phi, \sigma) \models f$, and $\sigma \in \mathcal{T}(\mathcal{M})$, if and only if $[\![f]\!]^{\phi}\mathcal{M} \sqsubseteq \sigma$. The fundamental theorem of trajectory evaluation [96] follows immediately from the previously-stated properties of $[f]^{\phi}$ and $[\![f]\!]^{\phi}$.

Theorem 3.1. *STE Implementation*

$$\mathcal{M} \models A \Rightarrow C \quad \triangleq \quad \forall t\, n.\ [C]^{\phi}\ t\ n \ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$$

For a detailed proof please refer to [48, 96]. The result of comparing the defining sequence and defining trajectory values is a Boolean expression, encoded as a BDD for efficiency reasons. The intuition is that the sequence characterising the consequent must be 'included in' the weakest sequence satisfying the antecedent, that is also consistent with the circuit.

This theorem gives a model-checking algorithm for trajectory assertions: to see if $\phi \models A \Rightarrow C$ holds for a given ϕ, just compute $[C]^{\phi}$ and $[\![A]\!]^{\phi}$ and compare them pointwise for every circuit node and point in time. This works because both A and C will have only a finite number of nested next-time operators N, and so only finite initial segments of the defining trajectory and defining sequence need to be calculated and compared.

In practice, the defining trajectory of A and the defining sequence of C are computed iteratively, and each state is checked against the ordering requirement as it is generated. Each state of the defining trajectory is computed from the previous state by simulation of a netlist description of the circuit, in which circuit nodes take on values in $\{0, 1, X\}$. The STE implementation algorithm is very efficient because it does not need to represent the whole state transition system explicitly, or encode it symbolically.

Several implementations of STE exist, most notably Intel's Forte [61, 97], which evolved from Seger's Voss system [95]. This industrial version of Forte integrates model checking, BDDs, circuit model manipulating functions, and a theorem prover all of which is built around a custom general purpose functional language FL [6, 61]. FL is a strongly typed language with the distinguishing feature that Booleans are represented as BDDs that are built into the language as first-class objects. FL provides an easy interface for invoking STE model checking runs, and serves as an extensible platform for expressing specifications. Note however, that the public release of Forte does not include a theorem prover.

We implemented an STE simulator [47], by deeply embedding [91] the STE theory, within the logic of the theorem prover HOL 4 [3]. HOL 4 has evolved from the theorem prover HOL88 [53], and is based on the classical higher-order logic. HOL is implemented in the functional programming language ML. Terms in HOL, have the abstract ML type term, and theorems (of ML type thm) are constructed by the application of core axioms and inference rules only, i.e. by proof.

The advantage of having a deep embedding based, STE implementation in HOL is that we could reason about the STE logic, extend the STE model checking by adding deductive inference rules, and use theorem proving support in HOL. However, since each execution step in the STE model checking run only happens via a proof-step in HOL, the speed of the simulation is very slow.

3.7 Properties about trajectory formulas

The syntactic sugar of STE is very limited, as we have seen above, but there are many useful properties that STE formulas have, and in this section, we shall explain them. These properties enable the STE model checking results to be lifted as theorems and used as inference rules in a theorem proving environment [4, 6, 48, 55, 56, 96]. Inference rules provide an effective property decomposition strategy that is used to break the original STE model checking run into smaller runs: use the STE model checker to check the validity of these smaller runs; and then combine the smaller results into a theorem in the theorem proving environment, that represents the result of the overall model checking run. This spirit of combining model checking and theorem proving has drawn considerable attention in other kinds of model checking as well [10, 11, 64, 73, 99, 107].

We present these properties and their proofs here, and, in Chapter 8, when we present an extension of the STE inference rules, we will use many of these properties in the proofs of the new STE inference rules.

Lemma 3.9. *Defining sequence less than or equal to defining trajectory*

$$\vdash\ \ \forall \mathcal{M}\ \phi\ A.\ ([A]^{\phi}\ \sqsubseteq_{\sigma}\ \llbracket A \rrbracket^{\phi}\ \mathcal{M})$$

Proof: Unfolding the definition of $\sqsubseteq_{\sigma}$ we need to prove the following:

$$\forall \mathcal{M}\ \phi\ A.\ \forall t\ n.\ ([A]^{\phi}\ t\ n\ \sqsubseteq\ \llbracket A \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$$

We will prove this by induction on t. For any arbitrary $\mathcal{M}$, antecedent A, and valuation ϕ, we show the following two cases:

Base: We show for any n that

$$\begin{aligned} & [A]^{\phi}\ 0\ n\ \sqsubseteq\ \llbracket A \rrbracket^{\phi}\ \mathcal{M}\ 0\ n \\ \equiv\ & [A]^{\phi}\ 0\ n\ \sqsubseteq\ [A]^{\phi}\ 0\ n \qquad \text{- using Definition 3.10 and Lemma 3.13} \end{aligned}$$

Hence the base is proved.
Step: We prove for any n that

$$\begin{aligned} & [A]^{\phi}\ (t+1)\ n\ \ \sqsubseteq\ \llbracket A \rrbracket^{\phi}\ \mathcal{M}\ (t+1)\ n \\ \equiv\ & [A]^{\phi}\ (t+1)\ n\ \ \sqsubseteq\ ([A]^{\phi}\ (t+1)\ n)\ \sqcup\ (\mathcal{M}\ (\llbracket A \rrbracket^{\phi}\ \mathcal{M}\ t)\ n) \quad \text{- Definition 3.10} \\ \equiv\ & [A]^{\phi}\ (t+1)\ n\ \ \sqsubseteq\ ([A]^{\phi}\ (t+1)\ n)\ \sqcup\ (\mathcal{M}\ (\llbracket A \rrbracket^{\phi}\ \mathcal{M}\ t)\ n) \quad \text{- true from Lemma 3.5} \end{aligned}$$

Hence the step is proved. □

Lemma 3.10. *Defining trajectory preserves information ordering*

$$\begin{aligned} \vdash\ \ & \forall \mathcal{M}.\ Monotonic\ \mathcal{M}\ \supset \\ & \qquad \forall A\ B\ \phi.\ ([B]^{\phi}\ \sqsubseteq_{\sigma}\ \llbracket A \rrbracket^{\phi}\ \mathcal{M})\supset (\llbracket B \rrbracket^{\phi}\ \mathcal{M}\ \sqsubseteq_{\sigma}\ \llbracket A \rrbracket^{\phi}\ \mathcal{M}) \end{aligned}$$

Proof: Unfolding the definition of $\sqsubseteq_\sigma$ we need to prove the following:

$$\begin{array}{l} \forall \mathcal{M}.\ \mathit{Monotonic}\ \mathcal{M}\ \supset \\ \qquad\qquad\qquad \forall A\ B\ \phi.\ (\forall t\ n.\ [B]^\phi\ t\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n) \supset \\ \qquad\qquad\qquad\qquad\qquad\qquad (\forall t\ n. [\![B]\!]^\phi\ \mathcal{M}\ t\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n) \end{array}$$

The proof takes place by induction on t, for any arbitrary $\mathcal{M}$, A, B and ϕ, by assuming the following:

1. $\mathit{Monotonic}\ \mathcal{M}$
2. $(\forall t\ n.\ [B]^\phi\ t\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n)$

we prove:

Base: $(\forall n. [\![B]\!]^\phi\ \mathcal{M}\ 0\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ 0\ n)$

Step: $(\forall n. [\![B]\!]^\phi\ \mathcal{M}\ t\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n)$
$$\supset$$
$$(\forall n. [\![B]\!]^\phi\ \mathcal{M}\ (t+1)\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ (t+1)\ n)$$

Base: For any ϕ and n, we show that

$$\begin{array}{lll} \quad [\![B]\!]^\phi\ \mathcal{M}\ 0\ n & \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ 0\ n & \\ \equiv [B]^\phi\ 0\ n & \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ 0\ n & \text{- follows from assumptions} \end{array}$$

But from assumption (2), we know that for all t and n,

$$[B]^\phi\ t\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n$$

so it will be true for $t = 0$. This proves the base case.
Step: We prove for any ϕ and n that

$$[\![B]\!]^\phi\ \mathcal{M}\ (t+1)\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ (t+1)\ n$$

$$\begin{array}{lll} 3. & \forall n. [\![B]\!]^\phi\ \mathcal{M}\ t\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n & \text{- ind. hypothesis} \\ 4. & \forall s\ s'.\ (s \sqsubseteq_s s')\ \supset\ ((\mathcal{M}\ s)\ \sqsubseteq_s\ (\mathcal{M}\ s')) & \text{- 1 and Definition 3.7} \\ 5. & ([\![B]\!]^\phi\ \mathcal{M}\ t\ \sqsubseteq_s [\![A]\!]^\phi\ \mathcal{M}\ t) & \\ & \qquad \supset\ (\mathcal{M}\ ([\![B]\!]^\phi\ \mathcal{M}\ t)\ \sqsubseteq_s\ \mathcal{M}\ ([\![A]\!]^\phi\ \mathcal{M}\ t)) & \text{- holds for } [\![B]\!]^\phi\ \mathcal{M}\ t \\ & & \quad \text{and } [\![A]\!]^\phi\ \mathcal{M}\ t \\ 6. & (\forall n. [\![B]\!]^\phi\ \mathcal{M}\ t\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n) & \\ & \qquad \supset\ (\forall n.\ \mathcal{M}\ ([\![B]\!]^\phi\ \mathcal{M}\ t)\ n\ \sqsubseteq\ \mathcal{M}\ ([\![A]\!]^\phi\ \mathcal{M}\ t)\ n) & \text{- Definition 3.5} \\ 7. & \forall n.\ \mathcal{M}\ ([\![B]\!]^\phi\ \mathcal{M}\ t)\ n\ \sqsubseteq\ \mathcal{M}\ ([\![A]\!]^\phi\ \mathcal{M}\ t)\ n & \text{- modus ponens on 3, 6} \\ 8. & \mathcal{M}\ ([\![B]\!]^\phi\ \mathcal{M}\ t)\ n\ \sqsubseteq\ \mathcal{M}\ ([\![A]\!]^\phi\ \mathcal{M}\ t)\ n & \text{- holds for any } n \\ 9. & [B]^\phi\ (t+1)\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ (t+1)\ n & \text{- from 2} \\ 10. & [B]^\phi\ (t+1)\ n\ \sqsubseteq\ [A]^\phi\ (t+1)\ n\ \sqcup\ (\mathcal{M}\ ([\![A]\!]^\phi\ \mathcal{M}\ t)\ n) & \text{- Definition 3.10} \\ 11. & [\![B]\!]^\phi\ \mathcal{M}\ (t+1)\ n\ \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ (t+1)\ n & \text{- Lemma 3.8, 8 and 10} \end{array}$$

Hence the step is proved. □

Lemma 3.11. *Defining sequence is monotonic*

$$\vdash \ \ \forall A\ B\ \phi.\ ([A]^{\phi}\ \sqsubseteq_{\sigma}\ [A \ \mathsf{and}\ B]^{\phi})$$

Proof: Unfolding the definition of $\sqsubseteq_{\sigma}$ we need to prove the following:

$$\forall A\ B\ \phi.\ \forall t\ n.\ ([A]^{\phi}\ t\ n\ \sqsubseteq\ [A \ \mathsf{and}\ B]^{\phi}\ t\ n)$$

We prove for any arbitrary A, B, ϕ, t and n that

$$[A]^{\phi}\ t\ n\ \sqsubseteq\ [A \ \mathsf{and}\ B]^{\phi}\ t\ n$$

$$\begin{array}{lll} & [A]^{\phi}\ t\ n\ \sqsubseteq\ ([A]^{\phi}\ t\ n\ \sqcup\ [B]^{\phi}\ t\ n) & \text{- Definition 3.9} \\ \equiv & [A]^{\phi}\ t\ n\ \sqsubseteq\ ([A]^{\phi}\ t\ n\ \sqcup\ [B]^{\phi}\ t\ n) & \text{- Lemma 3.5} \end{array}$$

□

Lemma 3.12. *Defining sequence of a formula is contained in the defining trajectory of the conjunction*

$$\vdash \ \ \forall \mathcal{M}.\ \forall A\ B\ \phi.\ ([A]^{\phi}\ \sqsubseteq_{\sigma}\ [\![A \ \mathsf{and}\ B]\!]^{\phi}\ \mathcal{M})$$

Proof: Unfolding the definition of $\sqsubseteq_{\sigma}$ we need to prove the following:

$$\forall \mathcal{M}.\ \forall AB\phi.\ \forall t\ n.\ ([A]^{\phi}\ \ t\ n\ \sqsubseteq [\![A \ \mathsf{and}\ B]\!]^{\phi}\ \mathcal{M}\ t\ n)$$

For any arbitrary $\mathcal{M}$, A, B, and ϕ we will prove by induction on t

$$\forall t\ n.\ ([A]^{\phi}\ \ t\ n\ \sqsubseteq [\![A \ \mathsf{and}\ B]\!]^{\phi}\ \mathcal{M}\ t\ n)$$

Base: $t = 0$

$$\begin{array}{lll} & [A]^{\phi}\ \ 0\ n\ \sqsubseteq [\![A \ \mathsf{and}\ B]\!]^{\phi}\ \mathcal{M}\ 0\ n & \\ \equiv & [A]^{\phi}\ \ 0\ n\ \sqsubseteq\ [A \ \mathsf{and}\ B]^{\phi}\ 0\ n & \text{- Definition 3.10} \\ \equiv & [A]^{\phi}\ \ 0\ n\ \sqsubseteq\ (([A]^{\phi}\ 0\ n)\ \sqcup\ ([B]^{\phi}\ 0\ n)) & \text{- Definition 3.9} \\ \equiv & [A]^{\phi}\ \ 0\ n\ \sqsubseteq\ (([A]^{\phi}\ 0\ n)\ \sqcup\ ([B]^{\phi}\ 0\ n)) & \text{- Lemma 3.5} \end{array}$$

Hence the base case is proved.
Step: We will prove that

$$[A]^{\phi}\ \ (t+1)\ n\ \sqsubseteq [\![A \ \mathsf{and}\ B]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n$$

$$\begin{array}{lll} \equiv & [A]^{\phi}\ \ (t+1)\ n\ \sqsubseteq\ ([A \ \mathsf{and}\ B]^{\phi}\ (t+1)\ n) \sqcup & \\ & \qquad\qquad\qquad (\mathcal{M}\ ([\![A \ \mathsf{and}\ B]\!]^{\phi}\ \mathcal{M}\ t)\ n) & \text{- Definition 3.10} \\ 1. & [A]^{\phi}\ \ (t+1)\ n\ \sqsubseteq\ [A \ \mathsf{and}\ B]^{\phi}\ (t+1)\ n & \text{- Lemma 3.11} \\ 2. & [A]^{\phi}\ \ (t+1)\ n\ \sqsubseteq\ [\![A \ \mathsf{and}\ B]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n & \text{- Lemma 3.7} \end{array}$$

Hence the step is proved. □

Lemma 3.13. *Defining trajectory is monotonic*

$$\vdash\ \forall \mathcal{M}.\ \mathit{Monotonic}\,\mathcal{M}\ \supset\ \forall A\ B\ \phi.\ (\llbracket A \rrbracket^{\phi}\ \mathcal{M}\ \sqsubseteq_{\sigma}\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M})$$

Proof: Unfolding the definition of $\sqsubseteq_{\sigma}$ we need to prove the following:
$\forall \mathcal{M}.\ \mathit{Monotonic}\,\mathcal{M}\ \supset\ \forall AB\phi.\ \forall t\ n.\ (\llbracket A \rrbracket^{\phi}\ \mathcal{M}\ \ t\ n\ \sqsubseteq\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$

For any arbitrary $\mathcal{M}$, A, B and ϕ we assume that $\mathcal{M}$ is monotonic and prove that

$$\llbracket A \rrbracket^{\phi}\ \mathcal{M}\ \ t\ n\ \sqsubseteq\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M}\ t\ n$$

1. $\mathit{Monotonic}\ \ \mathcal{M}$ - assumption
2. $\forall \mathcal{M}\ A\ B.\ \mathit{Monotonic}\,\mathcal{M}\ \supset\ (\forall t\ n.\ [B]^{\phi}\ t\ n\ \sqsubseteq\ \llbracket A \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$
   $\supset\ (\forall t\ n. \llbracket B \rrbracket^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ \llbracket A \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$ - Lemma 3.10
3. $\forall A\,B.\ (\forall t\ n.\ [B]^{\phi}\ t\ n\ \sqsubseteq\ \llbracket A \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$
   $\supset\ (\forall t\ n. \llbracket B \rrbracket^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ \llbracket A \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$ - modus ponens on 1 and 2
4. $(\forall t\ n.\ [A]^{\phi}\ t\ n\ \sqsubseteq\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$
   $\supset\ (\forall t\ n. \llbracket A \rrbracket^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$ - from 3
5. $\forall t\ n.\ ([A]^{\phi}\ t\ n\ \sqsubseteq\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$ - modus ponens on Lemma 3.12 and 1
6. $\llbracket A \rrbracket^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M}\ t\ n$ - modus ponens on 4 and 5

□

Lemma 3.14. *Defining trajectory is commutative*

$$\vdash\ \forall \mathcal{M}\ A\ B\ \phi.\ (\llbracket B\ \mathsf{and}\ A \rrbracket^{\phi}\ \mathcal{M}\ =\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M})$$

Proof: Two functions are equal if they return the same value for all elements in the domain. Thus we need to prove the following:
$\forall \mathcal{M}\ A\ B\ \phi.\ \forall t\ n.\ (\llbracket B\ \mathsf{and}\ A \rrbracket^{\phi}\ \mathcal{M}\ \ t\ n\ =\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M}\ t\ n)$

For arbitrary $\mathcal{M}$, A, B and ϕ, we prove by induction on t.
Base: $t = 0$

$$\begin{aligned} &([B]^{\phi}\ 0\ n\ \sqcup\ [A]^{\phi}\ 0\ n)\ =\ ([A]^{\phi}\ 0\ n\ \sqcup\ [B]^{\phi}\ 0\ n) && \text{- Definition 3.9, Definition 3.10}\\ \equiv\ &([B]^{\phi}\ 0\ n\ \sqcup\ [A]^{\phi}\ 0\ n)\ =\ ([A]^{\phi}\ 0\ n\ \sqcup\ [B]^{\phi}\ 0\ n) && \text{- Lemma 3.4} \end{aligned}$$

Hence the base case is proved.
Step: We have to prove

$$\llbracket B\ \mathsf{and}\ A \rrbracket^{\phi}\ \mathcal{M}\ (t+1)\ n\ =\ \llbracket A\ \mathsf{and}\ B \rrbracket^{\phi}\ \mathcal{M}\ (t+1)\ n$$

1. $\forall n. [\![B \text{ and } A]\!]^{\phi}\ \mathcal{M}\ t\ n \ = [\![A \text{ and } B]\!]^{\phi}\ \mathcal{M}\ t\ n$ - ind. hypothesis
2. $[\![B \text{ and } A]\!]^{\phi}\ \mathcal{M}\ t \ = [\![A \text{ and } B]\!]^{\phi}\ \mathcal{M}\ t$ - extensionality
3. $\mathcal{M}\ ([\![B \text{ and } A]\!]^{\phi}\ \mathcal{M}\ t) \ = \mathcal{M}\ ([\![A \text{ and } B]\!]^{\phi}\ \mathcal{M}\ t)$ - congruence
4. $\mathcal{M}\ ([\![B \text{ and } A]\!]^{\phi}\ \mathcal{M}\ t)\ n \ = \mathcal{M}\ ([\![A \text{ and } B]\!]^{\phi}\ \mathcal{M}\ t)\ n$ - congruence
5. $[\![B \text{ and } A]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n \ = ([B \text{ and } A]^{\phi}\ (t+1)\ n)$
   $\sqcup (\mathcal{M}\ ([\![B \text{ and } A]\!]^{\phi}\ \mathcal{M}\ t)\ n)$ - Definition 3.10
6. $[\![A \text{ and } B]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n \ = ([A \text{ and } B]^{\phi}\ (t+1)\ n)$
   $\sqcup (\mathcal{M}\ ([\![A \text{ and } B]\!]^{\phi}\ \mathcal{M}\ t)\ n)$ - Definition 3.10
7. $[A \text{ and } B]^{\phi}\ (t+1)\ n \ = [B \text{ and } A]^{\phi}\ (t+1)\ n$ - Definition 3.9, Lemma 3.4
8. $[\![B \text{ and } A]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n \ = [\![A \text{ and } B]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n$ - from 4 and 7

Hence the step is proved. □

Lemma 3.15. *Defining sequence containment implies defining trajectory containment*

$$\vdash\ \forall \mathcal{M}.\ Monotonic\, \mathcal{M}\ \supset\ \forall A\ B\ \phi.\ ([B]^{\phi}\ \sqsubseteq_{\sigma}\ [A]^{\phi}) \supset ([\![B]\!]^{\phi}\ \mathcal{M}\ \sqsubseteq_{\sigma}\ [\![A]\!]^{\phi}\ \mathcal{M})$$

Proof: Unfolding the definition of $\sqsubseteq_{\sigma}$, we need to prove the following:

$$\forall \mathcal{M}.\ Monotonic\, \mathcal{M}\ \supset\ \forall AB\phi.\ (\forall t\ n.\ [B]^{\phi}\ t\ n\ \sqsubseteq\ [A]^{\phi}\ t\ n) \supset$$
$$(\forall t\ n. [\![B]\!]^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n)$$

For any arbitrary $\mathcal{M}$ which is monotonic and any arbitrary A, B and ϕ, we show

$(\forall t\ n. [\![B]\!]^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n)$

We prove this by induction on t.
Base: $t = 0$

$[B]^{\phi}\ 0\ n\ \sqsubseteq\ [A]^{\phi}\ 0\ n$ - Definition 3.10
$\equiv (\forall t\ n.\ [B]^{\phi}\ t\ n\ \sqsubseteq\ [A]^{\phi}\ t\ n)$ - assumption
$\equiv [\![B]\!]^{\phi}\ \mathcal{M}\ 0\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ 0\ n$ - $t = 0$ and using Definition 3.10

Hence the base is proved.
Step:

1. $[\![B]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n$ - to prove
2. $([B]^{\phi}\ \mathcal{M}\ (t+1)\ n\ \sqcup\ \mathcal{M}\ ([\![B]\!]^{\phi}\ \mathcal{M}\ t)\ n)$
   $\sqsubseteq$
   $([A]^{\phi}\ \mathcal{M}\ (t+1)\ n\ \sqcup\ \mathcal{M}\ ([\![A]\!]^{\phi}\ \mathcal{M}\ t)\ n)$ - applying Definition 3.10 on 1
3. $Monotonic\ \mathcal{M}$ - assumption
4. $(\forall t\ n.\ [B]^{\phi}\ t\ n\ \sqsubseteq\ [A]^{\phi}\ t\ n)$ - assumption
5. $\forall n. [\![B]\!]^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - ind. hypothesis
6. $\forall s\ s'.\ (s \sqsubseteq_{s} s')\ \supset\ ((\mathcal{M}\ s)\ \sqsubseteq_{s}\ (\mathcal{M}\ s'))$ - Definition 3.7
7. $(\forall n. [\![B]\!]^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n)$
   $\supset\ (\forall n.\ \mathcal{M}\ ([\![B]\!]^{\phi}\ \mathcal{M}\ t)\ n\ \sqsubseteq\ \mathcal{M}\ ([\![A]\!]^{\phi}\ \mathcal{M}\ t)\ n)$ - holds for $[\![B]\!]^{\phi}\ \mathcal{M}\ t$ and $[\![A]\!]^{\phi}\ \mathcal{M}\ t$
8. $\forall n.\ \mathcal{M}\ ([\![B]\!]^{\phi}\ \mathcal{M}\ t)\ n\ \sqsubseteq\ \mathcal{M}\ ([\![A]\!]^{\phi}\ \mathcal{M}\ t)\ n$ - modus ponens on 5 and 7
9. $[B]^{\phi}\ (t+1)\ n\ \sqsubseteq\ [A]^{\phi}\ (t+1)\ n$ - from 4
10. $\mathcal{M}\ ([\![B]\!]^{\phi}\ \mathcal{M}\ t)\ n\ \sqsubseteq\ \mathcal{M}\ ([\![A]\!]^{\phi}\ \mathcal{M}\ t)\ n$ - from 8
11. $[\![B]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ (t+1)\ n$ - from 9, 10 and Lemma 3.6

Hence the step is proved. □

3.8 Extending the reach of STE

Even though STE achieves significant reduction by using X's and symbolic simulation, in circuits with a large number of state-holding elements, for example memories, the data abstraction of STE via X's is not adequate. Several reduction techniques have been used in the past to complement STE for large scale verification. These include symbolic indexing [20, 78, 84, 85], parametric representation [5, 60] and symmetry [70, 82, 83].

Symbolic indexing is a data abstraction technique that uses symbolic variables to encode or index a large number of cases into a smaller number. Thus if there are n inputs to an AND gate, instead of using n variables to cover all the n symbolic cases, one can use $log_2\, n$ fresh variables to index these n cases and verify all of them at once. The challenge lies in automatically indexing the cases from direct non-indexed set of assertions, given a user specified indexing relation. Melham and Jones [78] provide a sound technique for obtaining an automatic set of indexed cases from a user specified indexing relation. The strength of this technique is demonstrated by verifying content-addressable memories (CAM), a case study also done by Pandey et al. [84, 85] for PowerPC arrays. In both these cases, logarithmic reduction is obtained in the number of CAM lines, tag width and data width. The verification of CAMs is a difficult problem, because the incoming tags are compared in parallel with the stored tags in all the lines at once, and the hit is obtained if there is a match at some location in the CAM. This means that the resulting BDD that gets built during STE model checking run is so big that most STE simulators cannot complete the model checking run. [84, 85].

We will come back to the example of CAM verification in Chapter 9, where we will explore in detail the functionality and architecture of a CAM.

Parametric representation [5] is a technique based on data driven problem decomposition. It is also known as parameterisation. This was proposed by Robert Jones [60]. The idea is similar to the partitioning of data-space into a number of smaller sets and verifying the circuit separately for each smaller set and then knitting the overall verification run from smaller verification runs.

Parameterisation extends the notion of ordinary data-space decomposition further by actually introducing the notion of a *care set* of values and simulating circuits only for those values that belong to the care set. The challenges involved in Parameterisation lie in justifying the correctness of a decomposition and to ensure that the verification done through the case split does indeed cover all the cases. If we denote the care set by a predicate $P(\vec{x})$ where $\vec{x}$ denotes the variable vector that must satisfy P, and Q denotes the function that does STE simulation, then parameterisation will evaluate Q only for values that satisfy P which in this case happens to be the vector $\vec{x}$. Now if *param* is the function that computes a parameterised functional vector representation of P then the correctness statement can be written as

$$Q(param(P(\vec{x})))$$

Thus the constraints imposed by the care set P are encoded inside the symbolic simulator, and it does not have to build expressions for combinations of values that are not of interest. The actual *param* function can be found in [5, 60].

Symmetry [82] has been used with STE model checking to reduce the number of explicit STE cases one needs to verify. A central problem in using symmetry based reduction approach is the *identification* of symmetry in circuit models. If not addressed properly, it could have a substantial effect on the overall verification time. We shall talk about this in more detail when we explain the problem of symmetry reduction for STE model checking in Chapter 5.

3.9 Related work

We have explained in this chapter that STE is an efficient model checking technique, principally because it encapsulates symbolic simulation over a three-valued logic, with limited temporal features. The ability to reason with X's in the logic of STE gives an automatic and powerful data abstraction. Ching-Tsun Chou [34] showed that the data abstraction in STE using X's corresponds to a Galois connection. Recent work by Roorda and Claessen clarifies the semantics of STE model checking by providing closure semantics [92]. They claim that the original semantics of STE [96], also explained in this chapter, is not faithfully implemented in industrial versions of STE simulators such as Forte [97]. The reason stems from the fact that the excitation function in STE only relates the present state (current time point) to the next-state (one unit of time later), and says nothing about the propagation of information between nodes in the same time point. The closure semantics of STE takes as an input a state of the circuit, and calculates all information about the circuit state at the same point in time that can be derived by propagating the information in the input state in a forwards fashion. Subsequently the definition of defining trajectory and the STE implementation is refined to deal with the closure functions rather than the next-state function.

We showed in this chapter, that STE can deal with a restricted form of temporal behaviour, only one next-time operator in the STE logic. However there have been advances in extended forms of STE based model checking [112] that can express all forms of temporal properties that one is able to express using conventional symbolic model checking [35, 43, 63].

Chapter 4

Symmetry

The focus of our research is to exploit symmetry in circuit models, for property reduction with STE model checking. Symmetry has been used by a number of researchers in the past to achieve significant reduction in the size of model checking problems [30, 39, 40, 64, 82, 83, 103].

In this chapter we introduce the basic mathematical foundations of symmetry and its application to the reduction of properties and models during the verification of systems.

In the next section we will introduce the basic concepts of symmetry, and we will present some of the key work done in symmetry based reduction to model checking in Section 4.2. The concept of symmetry is closely linked to the notion of data independence [68], this is reviewed in Section 4.3. Section 4.4 presents another perspective on symmetry; it views symmetry as an instance of abstraction [59]. In the last section of this chapter we will discuss the current state-of-the-art symmetry based reduction in STE model checking, and study its limitations.

4.1 The mathematical foundation of symmetry

An object is said to have symmetry with respect to a given set of *operations* if whenever one of the operations is applied to the object it does not appear to change a certain *property* of the object. An object has a symmetry if we can define a set of operations and identify a property of that object that these set of operations will leave unchanged when applied to that object. These operations that preserve the given property of an object form a *group*. Symmetries of an object can be characterised by a special kind of group known as the automorphism group. In the subsequent section we shall review the basics of group theory in more detail and its relation to symmetry.

4.1.1 Group

A group is given by a triple $(G, *, I)$ where G is a set of elements, $*$ is a binary operation, and $I \in G$ an identity element such that they satisfy the following four axioms:

1. $\mathit{Closure} : \forall a\, b \in G.\ (a * b) \in G$
2. $\mathit{Associativity} : \forall a\, b\, c \in G.\ (a * b) * c = a * (b * c)$

3. $Identity : \forall a \in G.\, I * a = a * I = a$

4. $Inverse : \forall a \in G.\, \exists a^{-1} \in G.\, a * a^{-1} = I$

For a given group, whenever we fix an identity element I and the binary operator $*$, we will just use G to denote the group, rather than the triple notation. Whenever $*$ is commutative, the group is said to be Abelian.

A group is called finite if the underlying set has finitely many elements. A subset of a group that is itslef a group is called a *subgroup*.

4.1.2 Permutation and symmetric groups

Permutation is defined as a bijective mapping of an ordered set onto itself [13]. If we denote a set of elements by S, then the collection of all permutations of S forms a group Sym_S under the function composition $\circ$ which is known as the symmetric group. For a set consisting of n elements, the order of the symmetric group is $n!$. The term permutation group refers to any subgroup of the symmetric group.

The symmetry group of an object is referred to the group of all symmetry preserving operations under which the object is invariant with respect to the function composition operation. For many geometric objects this often includes operations such as rotation, reflection and inversions. In our treatment of symmetry for STE, we are interested in the symmetric group Sym_S which consists of $n!$ permutation operations. We do not deal with other kinds of symmetry preserving operations such as *bit inversion* in this dissertation.

Let S be a set consisting of the following k elements: $\{a_1,a_2,\ldots,a_n\}$. Then the expression $(a_1, a_2, \ldots, a_n)$ denotes a cyclic permutation that takes a_1 to a_2, a_2 to a_3, $\ldots$,a_{n-1} to a_n and a_n to a_1. The number n is the length of this permutation and cyclic permutation of this length n is called a n-cycle. A cyclic permutation of length 2 is usually referred to as a transposition, or a swap.

4.1.3 Generator of a group

A set of elements P of a group G generates the group G, if every element of G can be written as the product of a finite number of factors, each of which is either an element of P or the inverse of such an element. The definition of a group by means of its generators is an exceedingly useful device. For a set S consisting of n elements, pair-wise distinct transpositions (12),(23),$\ldots$,($n-1$ n) together generate the symmetric group Sym_S [13]. This means that for a set consisting of n elements we only ever have to consider $n-1$ pair-wise distinct transpositions rather than $n!$ permutations to completely generate the symmetric group.

4.1.4 Equivalence classes and orbits

Given a group $G \subseteq Sym_S$, any operation of G on a set S, defines an equivalence relation on S: two elements x and y are called equivalent if there exists a g in G with $gx = y$.

The equivalence class of x under this equivalence relation is known as the orbit of x and is given by:

$$\theta(x) = \{gx \mid g \in G\}$$

The elements x and y are equivalent if and only if their orbits are the same. Consider an n element set S given by $\{a_0, a_1, \ldots, a_{n-1}\}$. As explained earlier the symmetric group Sym_S can be generated by the $n-1$ generators defined by the following $n-1$ transpositions: (12),(23),. . .,($n-1$ n). These group generators act on S, to define an equivalence relation on S. The orbit of any element a_i in S, where $0 <= i < n$ gives the set S itself.

4.1.5 Automorphism

We saw in Section 2.1.2 that an automorphism is a special case of a homomorphism. Traditionally symmetries are defined in terms of an automorphism. Therefore we take a quick review of the automorphism definition and explain its relation to the concept of symmetry.

Definition 4.1. *An automorphism of a mathematical object is a mapping that has the following properties:*

1. *its a mapping on some underlying set onto itself - endomorphic*
2. *structure preserving - is a homomorphism*
3. *is bijective - one to one and onto*
4. *and its inverse map is also a homomorphism*

An automorphism of an object is a way of showing its internal regularity. An automorphism of a graph $\mathcal{G}$ is a permutation of its vertices that maps the graph onto itself. The set of automorphisms define a permutation group.

The automorphism groups of a graph characterise its symmetries and are therefore often used in discovering symmetries in circuit models represented as graphs. The set of automorphisms of an object together with the operation of function composition $\circ$ forms a group called the *automorphism group* ($Aut(G)$).

4.2 Applications of symmetry reduction in model checking

4.2.1 Symmetry reduction for CTL* based model checking

Clarke et al. [39] investigate the idea of using symmetries for reduction of state spaces in model checking. Their solution is applicable for both the case, when the transition relation is given explicitly in terms of states, and the one when it is given symbolically as a BDD.

The solution is based on reducing the original *state space* model to a smaller *quotient model*, and checking the validity of the formula, expressed in the logic of CTL*, in this quotient model. The quotient model is generated in terms of orbits. If the model $\mathcal{M}$, for a given system can be expressed by a Kripke structure given by (S, R, L) then a symmetric group G is the set of all the permutations that are a symmetry for $\mathcal{M}$. A permutation is a symmetry of $\mathcal{M}$ iff it preserves the transition relation R, in the Kripke model $\mathcal{M}$. Intuitively the quotient model denoted by $\mathcal{M}_G$ can be understood to consist of the set of all orbits $\theta(s)$ such that the initial state s is a member of the set of initial states S, in the model $\mathcal{M}$. The transition relation R_G has the property that $(\theta(s_1), \theta(s_2)) \in R_G$ iff there exists two states $s_3 \in \theta(s_1)$ and $s_4 \in \theta(s_2)$, and $(s_3, s_4) \in R$. The labelling function L_G is given by $L_G(\theta(s)) = L(rep(\theta_s)))$, where $rep(\theta_s)$ is a *representative state* from the orbit.

For a temporal logic formula (CTL*), if all of its atomic propositions are invariant under the symmetry group G, then this formula is true in the original model $\mathcal{M}$ if and only if it is true in the quotient model $\mathcal{M}_G$. Clarke et al. present techniques for building $\mathcal{M}_G$ without actually building $\mathcal{M}$. The idea is to use the smaller state space of the quotient model, which has only one representative from each orbit.

The authors present lower bounds on the size of the BDDs encoding the orbit relation. Because these bounds are exponential for some important symmetry groups that occur in practice, the authors propose to use *multiple representatives* instead of one from each orbit, and this in turn allows them not to build the full orbit relation.

Emerson and Sistla [51] describe a method of exploiting symmetry based reduction for CTL* model checking by forming a quotient structure using the automorphisms of the original model and the properties. The method is efficient because the authors observe that not only do the models have symmetry, but the formulas expressing CTL* properties have symmetries as well. Thus by effectively decomposing the properties, one can expose the automorphisms in the original properties, and use this to achieve significant reductions. For a given model $\mathcal{M}$, the symmetry of $\mathcal{M}$ is reflected in the group ($Aut\ \mathcal{M}$) of permutations of process indices that define the graph automorphisms of $\mathcal{M}$. The idea of obtaining a quotient structure for a group G contained in ($Aut\ \mathcal{M}$), is based on the equivalence of states, and the internal automorphisms of the CTL* formula f. If the automorphism group of the formula f is denoted by ($Auto\ f$), then if the group G is contained in ($Aut\ \mathcal{M}$) $\cap$ ($Auto\ f$), then the correctness condition is stated as follows:

$$(\mathcal{M}, s) \models f \equiv (\mathcal{M}_G, \vec{s}) \models f$$

where f is a formula of CTL* and $\vec{s}$ denotes the equivalence class of a state s. In fact f can even be a formula of the Mu-calculus [51]. When $G = (Aut\, \mathcal{M}) \cap (Auto\, f)$ then maximal compression is obtained. Obtaining ($Aut\, \mathcal{M}$) is known to be a graph isomorphism problem [42], but fortunately, since the state graph is obtained from a parallel concurrent program, the automorphisms are easily obtained from the process indices of the program. Automorphisms in the formula f are determined by manual inspection. An interesting result given by the authors is an alternative automata-theoretic approach that provides a uniform method permitting the use of a single quotient $\mathcal{M}_{(Aut\, \mathcal{M})}$ to model check formulas f without obtaining the intersection of automorphisms of $\mathcal{M}$ and f. This is done by annotating the quotient with "guides" indicating how the coordinates are permuted from one state to the next in the quotient. An automaton for f designed to run over paths through $\mathcal{M}$, is transformed into another automaton that is run over the quotient structure $\mathcal{M}_{(Aut\,\mathcal{M})}$ using the guides to keep track of shifting coordinates. The concept of annotating the quotient structures is further used in [50] where utilizing symmetry for reduction for CTL* model checking is done ensuring fairness constraints. Recent work by Sistla and Godefroid [103] extends the approach of using annotated quotient structures by employing guards, and identifying symmetries on states that are variable-value pairs, besides process indices. Guards express conditions under which a program transition is executable. The usage of guards allows us to represent a quotient structure in a compact way even for systems that have very little or even no symmetry. The compact form is obtained by first expanding the given graph using guarded annotations and then reducing the graph modulo automorphisms that become explicit through the expansion and annotation step. Experiments on verifying properties of the IEEE Firewire protocol show that the approach using Guarded Annotated Structures outperforms the approaches explained in [50, 51]. Many of these ideas have been implemented by Sistal et al. [104] in a symmetry based model checker called SMC.

4.2.2 Symmetry reduction with Murϕ

Ip and Dill [40] exploited structural symmetries in the description of the systems to achieve reduction for model checking. They proposed that identifying such a symmetry in a system helps to characterize data independence (see Section 4.3) They coined the term scalarset for a new data type that can be used to denote a class of data values that determine the symmetry of the model. Scalarsets record the symmetry in the model. The very presence of the scalarsets in a system is an indication of the presence of certain kinds of symmetry. The authors present a proof of the soundness of the symmetry based verification algorithm, by using the formal semantics of a simple UNITY-like, description language with scalarsets, called Murϕ.

Ip and Dill report the application of their approach to numerous cache-coherence protocols, including the DASH (a scalable shared multi-processor developed at Stanford) and ICCP (industrial cache coherence protocol).

4.2.3 Symmetry reduction using scalarsets in SMV

In [64] Ken McMillan identifies a methodology for verification of data path circuits using several abstraction strategies and symmetry. Symmetry based reduction is one of the main components of his methodology.

The methodology can be summarized in the following steps:

1. **Refinement relations and auxiliary state.** The proof strategy is based on the concept of refinement relations. Refinement relations are a collection of temporal properties. These relate the behaviour (usually the timing models) of a simple abstract model to a more complex and detailed implementation. Once specified by the user, the refinement relations are used to break down the verification problem into smaller parts, using circular compositional reasoning. The

verification problem is decomposed into smaller parts and then the abstract environment (refinement relations) provides the necessary constraints (assume) to prove (guarantee) the existence of the property that talks about the other parts. Thus constraints about one part can be used to prove properties about the other smaller parts of the problem. In an application of this methodology to verification of Tomasulo's algorithm, the author mentions that the refinement relations are written to specify the correct values for the operands and the results obtained in the implementation. To do this, the correct values are obtained from the abstract model and stored in auxiliary state.

2. **Path Splitting.** For pure control logic, using refinement relations is adequate. But for designs with data flow, like memories, buffers and caches, additional reduction strategies are needed. McMillan suggests that by using path splitting we can effect the next step of reduction. The idea is that for structures like memories, or FIFO buffers, we can split the verification problem into cases based upon the path the data values can take.

3. **Symmetry.** After path splitting we can reduce a problem of verifying large number of data values, to the problem of verifying only few representative samples from these values. The selection of the representatives is based on symmetry reduction, which in turn relies on the presence of symmetric data types — scalar sets. This concept of using scalar sets to record symmetry is exactly the same as in Ip and Dill's approach [40].

4. **Data type reduction.** For types with large or infinite values, data type reduction is used. Large or infinite types such as data words are reduced to small finite types, representing all the irrelevant values by a single abstract value. A special case of this is the uninterpreted function abstraction, which in [64] uses a lookup table to represent an arbitrary function, then splits cases so that we use only one element of the table for each case.

   This methodology was used to test the packet buffering application, the cache coherent multiprocessor, and a version of Tomasulo's algorithm.

4.3 Symmetry and data independence

We have seen earlier that diagnosing structural symmetry in a state transition system amounts to identifying a set of automorphisms (that induce a congruence relation on the state transition system). Extracting such a set of automorphisms amounts to identifying a new data type in the state transition system such that a function π that permutes the elements of a data structure indexed by this new data type consistently throughout a state generates symmetrically equivalent states. This new data type is exactly the scalarset data type in [40]. Ranko Lazic [68] views a system with scalarsets as a data-independent concurrent system with respect to the scalarset data type.

Lazic observes that a system with a scalarset data type is a special case of a family of parameterised labelled transition systems (LTS). A data type with respect to which a system is data independent can be treated as a parameter of the system, i.e. as a type variable that can be instantiated by any concrete type. Based on this intuitive definition of data independence, it is straightforward to formulate syntactic restrictions which ensure that a system is data independent as is the case with a scalarset data type in [40]. However Lazic points out that is hard to see how the restrictions and the results in different formalisms are related and therefore they motivate the need to understand data independence semantically. The important thing to keep in mind is that the semantic definition must not be stronger than the syntactic restrictions, but just strong enough to prove results which enable reductions to finite instantiations.

In order to understand the semantic definition of data independence, it is important to capture the idea of what it means for an LTS to be parametric. Any system or a program that uses type variables and operations on them in a polymorphic manner gives rise to a parametric family of LTSs, where each LTS is the semantics of the program or system with respect to an instantiation of the type variables and operations. In the special case where the only operations are equality testing, the family of LTS is data independent.

The major contribution of Lazic's work is the fact that the theorems and definitions are not in terms of any particular formalism, but are semantic.

4.4 Symmetry as an instance of abstraction

Symmetry reduction techniques when applied in practice appear quite different from the several abstraction techniques that we reviewed in the previous chapter. Although both abstraction and symmetry reduction are employed for state space reduction, their role in practice is a bit different. Abstraction typically hides the irrelevant details of the system while symmetry collapses parts of the state space that are structurally equivalent. This is one reason why we chose to present abstraction and symmetry as separate from one another so far. In this section however we wish to explain that symmetry is theoretically just another kind of abstraction, but we will also explain why we do not take this particular perspective in practice.

Given a state transition system $\mathcal{M}_\mathcal{C}$ and a symmetry group G, the orbit relation induced by G is a bisimulation. This was shown by Somesh Jha in his PhD thesis [59]. We have seen in Section 2.1.4 that bisimulation relations are a way to compute sound abstractions. Even though this is the case, we do not use bisimulation minimisation algorithms to obtain quotient groups. The reason is that bisimulation minimisation algorithms require that a transition graph is constructed first and then minimised using the fixpoint characterisation of bisimulation. But if one has to build the transition graph first then the very idea of using symmetry reduction gets defeated since the state graph might have a huge number of states or may be infinite.

4.5 Symmetry reduction for STE model checking

In the past, Manish Pandey has used symmetries to compute reduction for STE based model checking [82, 83]. He identifies three special class of symmetries, data, structural and mixed (data and structural). Pandey uses sub-graph isomorphism based techniques to identify structural symmetries and uses symbolic simulation to discover data symmetries, and mixed symmetries. Structural symmetries as in Pandey's work, are exactly the symmetries we are interested in this thesis. Structural symmetry is the one that comes into being if, by permuting certain inputs and outputs of the circuit, the behaviour of the circuit remains unchanged. Some examples of circuits that have structural symmetry include comparators, multiplexers, registers, and memory based circuits. A circuit is said to have data symmetry if the behaviour of the circuit remains unchanged under a data complement operation of some of its inputs and outputs; for example a decoder circuit. Mixed symmetries are said to be present in a circuit if structural, and, data symmetries are both present for example an $n - bit$, register.

The symmetries Pandey found most useful, were structural symmetries. Circuits were represented as transistor level netlist graphs. Since the problem of sub-graph isomorphism is known to be NP-complete, he used graph colouring to discover symmetric segments in a circuit model. He used symmetries to compute significant reductions for verifying SRAMs, however a significant proportion of his time and space was consumed in discovering symmetries. In fact he notes in his thesis, that "*It is interesting to note that symmetry checks dominate much of this total time*" (Chapter 3, page 77 of [82]). Also it is not clear from his work, how much impact, data symmetries and mixed symmetries have in property reduction.

4.6 Summary

This chapter presented the theory of symmetry, and its mathematical foundations. We presented applications of symmetry in problem reductions for different kinds of model checking. We presented an overview of an important work done in symmetry reduction for STE model checking, and highlighted its limitations.

In the next chapter we will explain the problem of symmetry reduction for STE model checking in more detail, and will present a top-level view of our solution, that circumvents the limitations of the existing work done on symmetry reduction for STE model checking by Pandey.

Chapter 5

Symmetry Reduction Framework

Many large circuits, specially the ones with a large number of state holding elements, pose a considerable challenge to STE model checking. It is, however, also the case that many of these circuits exhibit symmetry in their structure. We showed in Chapter 4, that many researchers in the past have used symmetries for computing reductions in the size of the model checking problem.

In this chapter, we present the problem of symmetry reduction for STE model checking, and describe our approach by highlighting the key components of our solution. Details of each component will be explained in subsequent chapters.

5.1 The problem of symmetry reduction

In a typical STE verification cycle, we use a simulator, and ask if a given circuit model satisfies a certain STE property. A circuit's model is an internal representation of its behaviour which itself is usually specified by a finite state machine [9, 58, 108], or a state transition system [35]. This is usually derived from the circuit's netlist description [1, 105], which is obtained through a series of synthesis steps, from a high-level description specified in a language such as VHDL or Verilog.

The netlist description of circuits is quite low level. Information about the circuit is modelled at the level of each single node and its connections to other nodes in the circuit. At this level of abstraction a lot of information about the structure of the circuit, perhaps described in the high-level language is lost. For example, two nodes "a_0" and "a_1" may belong to a common structure (an electrical bus) which might be responsible for generating symmetry, but this information is not present in the netlist. We therefore refer to the netlist description of the circuit's model as a flat model.

We use Intel's STE simulator Forte [97], which represents circuit models internally using a method, similar to a finite-state machine and is therefore known as an FSM. Informally, the FSM is a collection of hash tables containing information about circuit nodes and their fanin-fanout dependency. In STE model checking theory, the term model is often referred to the next-state function on lattice states. In practice, the next-state function, is constructed on-the-fly at run time, from the FSM. We showed in Chapter 3, how the next-state function takes a lattice state, and for each node returns the lattice state of the node at the next point in time. This level of circuit operation given by the next-state function is too low-level, since it specifies the behaviour of each single node, and does not have the ability to talk about a group of nodes (bits). For this reason we refer to the FSM in Forte, and the equivalent next-state function derived from it, as a *flat* circuit model.

If we denote any given STE property by STE, and a circuit model by FSM, then the question asked in a STE verification cycle can be stated as

$$\mathsf{FSM} \models \mathsf{STE}$$

where $\models$ denotes the STE satisfiability relation shown in Chapter 3.
The above question can be answered by running an STE simulator and finding out if the given property is satisfied by the circuit model. But, in many cases, this may not be possible directly, especially when the model and property in question is too large. In this case, we would like to answer this question in another way, by exploring possible symmetries in the circuit model. If a circuit model has symmetry, it can be used to justify the construction of a reduced circuit model (FSM'), and compute reduced properties (STE'), in such a way that verifying a reduced property, possibly against the reduced model, entails that the original property in question has been verified against the original model. Together with symmetry, we can use other forms of reduction algorithm as well, such as symbolic indexing, and parametric representation. This reduction philosophy is shown in Figure 5.1. The complexity of STE model checking depends on

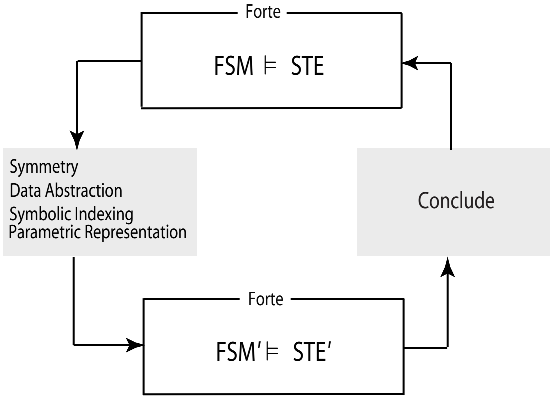


Figure 5.1: Problem reduction philosophy. FSM' and STE' are reduced circuit models and properties respectively. We would like to add symmetry based reduction to the existing set of reduction techniques, such as data abstraction, symbolic indexing, and parametric representation.

the nodes and number of next time operators in the consequent of the assertion. The STE implementation algorithm automatically achieves cone-of-influence reduction [38] , by querying for the defining sequence of only those nodes that are mentioned in the consequent. If we can decompose the STE property in such a way that the resulting properties have fewer nodes in the consequent than the original property, then verifying these smaller properties has an effect of achieving reduction in the circuit model. This is because the cone-of-influence of fanin nodes that will be constructed at the time of STE model checking run for the consequent nodes in a given property, would be smaller, due to fewer nodes in the consequent of the property due to decomposition. Thus, computing smaller properties in a sound way, and then asking if these smaller properties are satisfied by the original model, has the effect of asking if the reduced circuit model satisfies reduced STE properties. This can be formally stated as

$$\mathsf{FSM}' \models \mathsf{STE}'$$

where FSM' denotes a reduced STE model, and STE' denotes the reduced STE property.

There are three main challenges in using symmetry to compute reduced circuit models (FSM'), and reduced STE properties (STE'). They are as follows:

1. Discovering symmetries in a given circuit model.
2. Making the link between symmetry in circuit models and properties expressed in the logic of STE.
3. Developing a reduction methodology for performing sound property reduction for STE model checking.

We pointed out in Chapter 4, that symmetries were used by Pandey et al. [70, 82, 83] to compute reductions in the size of STE model checking run; however one of the key limitations of his work was the way symmetries were discovered in circuits. Circuit models were represented by transistor-level netlist graphs, and discovering symmetry then amounted to an instance of the sub-graph isomorphism problem, a known NP-complete problem [42]. The main difficulty with this approach was that for each new circuit in question, new heuristics had to be discovered.

The time spent on detecting symmetries accounted for most of the time used overall in reduction, and for growing sizes of the same circuit model (for example different sizes of SRAM), the amount of time required to find symmetries grew accordingly with the size of the model.

In fact as we pointed out in Chapter 4, Pandey observed in his dissertation, that "*It is interesting to note that symmetry checks dominate much of this total time*"(Chapter 3, page 77 of [82]). This key limitation in Pandey's work is one of the prime factors to consider for our research.

We do not wish to pursue the problem of symmetry discovery by trying to find it in flat FSMs which are represented as graphs. In the next section, we shall outline our methodology of symmetry reduction, that addresses the problems we have outlined in this section.

5.2 The proposed solution

The solution to the problem of symmetry reduction lies in addressing the issues we have highlighted in the previous section; the solution should handle the problem of symmetry

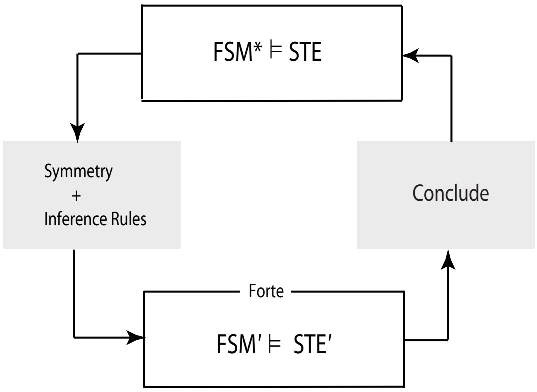


Figure 5.2: FSM′ and STE′ are derived by identifying symmetries in circuit models, and performing sound property decomposition.

discovery in circuit models, articulate the connection between symmetric circuit models and STE properties, and propose a sound methodology for property reduction. In the subsequent subsections, we will outline our solution by providing answers to all these issues.

5.2.1 Discovering symmetry

Structured modelling of circuits

We believe that the problem of symmetry detection in circuit models is best solved by lifting the representation of models to a higher level of abstraction. If the symmetry in models can be captured in the description of an HDL [40], then the problem of symmetry detection can reduce to the problem of type checking. This is the approach that we have taken for our work on symmetry detection.

High-level description languages come in several flavours. On one hand we have VHDL, Verilog and SystemC which have imperative features, and have been used extensively in high level design. On the other hand there are functional hardware description languages such as Lava [22] and reFLect [75], which though relatively new, have been used for several industrial designs. Although both the imperative and the functional languages offer many high-level features, they do not offer any possibility to express symmetry in circuits by using types.

There are special purpose languages such as Murϕ that allow more abstract descriptions of systems, and come enriched with a rich set of types (notably the scalarset type), that allows one to capture many symmetric aspects of the system. Murϕ in particular has been used in the design of several cache coherence protocols, that have structural symmetry. However, we have not seen any application where Murϕ was used to design circuit components, the way we wanted to do for our research.

We therefore wanted to design a framework whereby symmetry in a circuit can be recorded at the time of its design, and verified later by type checking. Our solution is centered around the design of an abstract data type (ADT) for symmetric circuits. The ADT is defined by presenting a type for circuit models, and defining functions that let us construct symmetric circuit blocks in a bottom up fashion. Since, the ADT can capture the structure of circuit models, we refer to it as a *structured model* (FSM*). The '$*$' in FSM* suggests that the circuit model has structural information; something that is missing in an FSM.

The symmetries we are interested are those that arise out of groups of wires that are held together as a unit. The unit of wires is typically known as a bus. Often, there are several such input and output buses in a circuit. Symmetry is exhibited by a circuit across some of its input and output buses, when the behaviour of the circuit remains unchanged under related permutations of its inputs and outputs. This kind of symmetry is also referred to as structural symmetry, since it is an outcome purely of the structure of the circuit.

The major challenge in our work was in discovering a reasonably structured *type* for circuit models that would capture the structural symmetries we were interested in. Also once we fixed a base type, the next challenge was to define a set of functions, that are used to construct symmetric circuit models. The abstract data type is then defined by providing type judgement rules that distinguish symmetric circuits from non-symmetric ones. We justify that the ADT we define does indeed capture the symmetry we want, by proving a type soundness theorem. This theorem states that all well-behaved circuits that respect the typing rules will have structural symmetry. This theorem then becomes the basis for deciding by type checking, whether or not a given circuit has symmetry.

We implement the circuit construction functions, and the type judgement rules, shallowly in the logic of the HOL theorem prover. The proof of the type soundness theorem is also carried out in HOL. We shall explain all of this in detail in Chapter 6.

In Figure 5.3, we show our framework of circuit modelling. The block labelled FSM* can be seen in this figure. This denotes the structured circuit modelling, together with the typing rules, and the type soundness theorem.

Simulating structured models in HOL

One of the well known benefits of modelling hardware using a functional language is the ability to rapidly prototype and simulate the circuit models [22, 69, 100]. Simulating structured circuit models in the HOL logic is achieved by a combination of ML preprocessing functions and writing *conversions* [88] in HOL that can simulate the circuit's behaviour over the Boolean domain. This is shown in Figure 5.3, by an arrow going from structured models to simulatable models in HOL. This is in contrast to the STE models in HOL logic [47], which work on a three-valued domain. We will present the details of this flow in the next Chapter.

From structured models to flat netlists

Once we model circuits using our ADT, and we have the proof of the type soundness lemma, then the next step is to compile the circuits into FSMs that can be used for

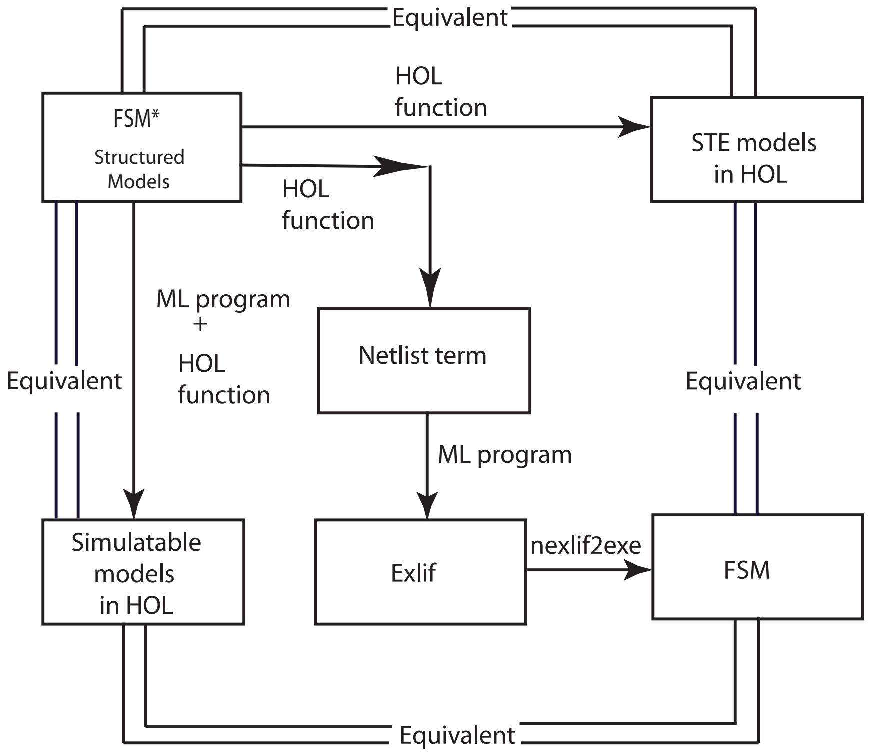


Figure 5.3: The world of circuit models.

simulation with an STE simulator. The flow of the compilation process that we use is shown in Figure 5.3. The first step in this process is to flatten the structured models, and this is achieved by writing a function in HOL. This can be seen in Figure 5.3 as an arrow between FSM* and the block labelled Netlist term. This netlist term is a relation between input and output Boolean states. This relation is then converted by an ML program to an Exlif format. Exlif is an internal netlist format used by the STE simulator Forte, for representing circuits, and its syntax is very similar to the more well known Blif [1] representation. The final step in producing the FSM is to use the nexlif2exe program distributed with the Forte simulator.

We mentioned in Chapter 3 that we have implemented an STE simulator [47] by formalising the theory of STE logic in the theorem prover HOL. The three-valued STE models in the logic of HOL, are derived from structured models, by providing a function in HOL, which is shown in Figure 5.3. These STE models must have the property of monotonicity. We prove a theorem that the derived STE models are monotonic. Also, it can be seen that whenever the structured circuit models have symmetry, the equivalent STE model in HOL will have symmetry as well. More on this will be said later in Chapter 7.

The STE models in HOL are behaviourally equivalent to the FSMs we compile. This equivalence can be observed by running an STE simulator, on the models [47] in HOL, and running the STE simulator on the FSMs in Forte. This equivalence is depicted in Figure 5.3.

5.2.2 Relation between symmetric circuit models and STE

Recording symmetry in circuit models serves as the beginning point of our property reduction methodology. The next important question is what is the connection between symmetry in circuit models and STE properties. The answer is that symmetry in circuit models gets mirrored by the symmetry in STE properties. We formalise this precisely by providing a *soundness theorem* in Chapter 8. This theorem is based on the observation that trajectory formulas that form the basis of STE properties, can have symmetry amongst them as well. We learnt in Section 5.2.1, that the symmetry we are interested in is the one that arises out of uniform permutations of wires in the input and output buses. Since wires have names, and the same names are used in the trajectory formulas expressing the STE property, these names provide the basis for generating the notion of equivalence (and hence symmetry) amongst STE properties. We will explain the details of how this is done precisely, in Chapter 8.

5.2.3 STE property reduction

Reduction begins by decomposing the given STE property in a sound manner into a set of smaller properties, using a set of *inference rules*. Inference rules serve the general purpose of decomposing properties [4, 62, 97, 98]; in our case the rules become all the more useful, since they help expose the symmetric equivalence class of properties. When they are used in a backward manner, like tactics [53] in HOL, they decompose a property into smaller properties. The set of decomposed STE properties obtained by using inference rules is then clustered into several equivalence classes. A representative from each equivalence class is then picked and verified explicitly by using an STE simulator such as Forte. Then using the soundness theorem that we just mentioned, we can conclude that those properties that belonged to the same equivalence class as the representative, are verified as well. The correctness result about all the smaller properties is determined, by repeatedly using the flow:

- *cluster*,
- *pick and verify the representative*, and
- *use the soundness theorem to conclude correctness of entire class of properties*

Inference rules are then used in the forward direction, to reconstruct the overall statement of correctness, expressed by the original property.

5.3 Summary

In this chapter we explained the problem of symmetry reduction, and presented a top-level view of our solution. We showed that our solution is based on addressing three key issues, and in the subsequent three chapters, we will explain each one of them in detail.

Chapter 6

Structured Models

In the previous chapter we motivated the need to record symmetries in a high level structured description. In this chapter we will present a framework for modelling circuits in a way that enables us to capture symmetries. This focusses on the design of an abstract data type for circuits. We will provide a type for circuits, and a set of functions that help us design circuits with this type. Then a set of type judgement rules are provided that become the basis for type checking circuits to infer symmetry in them. In other words, the type judgement rules enforce a discipline on the way circuit constructing functions could be used to guarantee symmetry in them. We give the definition of symmetry that we are interested in and prove that those circuit models that respect our type judgement rules have this symmetry.

6.1 A language for structured models

The symmetry that we are interested in, arises in circuits due to groups of wires that are treated like a unit. Each wire can take on a *bit* value, and we will refer to such groups of wires collectively as a *bus*. Thus, if the group has 8 wires, then we will refer to it as an 8-*bit* bus.

Many circuits have symmetry; for example, a comparator used for equality comparison, a multiplexer, a random access memory, and so on. These circuits have symmetry, because they treat the buses of inputs and outputs in a special way, by allowing only restricted set of operations to take place on the bus values. Thus it makes perfect sense to treat these buses in a special way.

The concept of "treating buses in a special way" can be formalised by an abstract data type (ADT) which we also refer to as *structured models*.

Our methodology of designing the structured models can be summarised in the following steps:

1. Propose a sufficiently generic type for circuits.

2. Define the functions that will be used for constructing symmetric circuit blocks.

3. Provide the type judgement rules.

4. Articulate the mathematical definition of structural symmetry.

5. Prove a type soundness theorem saying that every circuit that is well typed with respect to the typing rules will have structural symmetry.

In the subsequent sections, we shall explain the details of our steps.

6.2 The type of circuits

We want to model a collection of bit values which are treated in a special way. As explained in the previous section, a group of wires is known as a bus in electrical terms. We choose to model the collection of values on a single bus, by the type of Boolean lists, where the Boolean type *bool* denotes the bit value on a wire. The collection of several such buses can be modelled by the type of *list* of Boolean lists. A circuit is said to have symmetry with respect to a specific sets of inputs and outputs, when the circuit's behaviour remains unchanged under permutation of those inputs and outputs. The specific inputs are known as symmetric inputs. Every circuit can have some inputs which will be symmetric and there would be some inputs that would not have any role in the symmetry of the circuit; they are referred to as non-symmetric inputs. Thus a circuit has the following type:

$$circ : (bool\ list)\ list \rightarrow (bool\ list)\ list \rightarrow (bool\ list)\ list$$

where the type (*bool list*) denotes the set of values of any single bus, and the list of Boolean lists ((*bool list*) *list*) denotes the collection of several buses. The first argument of the circuit type denotes the values of non-symmetric inputs, the second argument denotes the values for symmetric inputs and the third argument the outputs of the circuit.

We provide a set of functions that allow us to construct circuits with the above type. The definition of these circuit constructing functions is structured into layers. The reason for this is that we want to first develop a family of functions that take a Boolean list and return a Boolean list, in a way that these functions preserve the symmetry property that we explained intuitively above, and then we can use these functions to construct other higher-order functions for circuit construction. We tend to distinguish between the term *function* and the term *combinator*. We refer to the term functions when we mean functions of the type *bool list* → *bool list*, and we refer to the term combinator, when we mean functions of the type (*bool list list* → *bool list list* → *bool list list*), or of the type *bool list list* → *bool list list*.

The first level, called Level 0 in our framework, is the layer that consists of function definitions over Boolean lists, and provides type judgement rules, to identify safe functions that are used to design symmetric functional blocks. The second layer, referred to as Level 1, is the layer where we provide circuit combinators, and a type system that is used for identifying ways in which these circuit combinators are combined to generate circuits that have symmetry. We show that these circuits indeed have symmetry by proving a type soundness lemma.

We have designed the ADT in the language of higher-order logic. There has been a lot of work done in modelling hardware in HOL [33, 52], so this offered a natural point to start. The choice of logic being higher-order was significant, since the implementation of the ADT operators becomes a lot easier in terms of abstraction [22, 69]. We implemented the ADT shallowly, in the theorem prover HOL 4. The shallow embedding [12, 91] enabled us to reuse many existing theories in HOL over Booleans, lists, arithmetic, combinators and so on, and also laid the foundation for carrying out the proof that the design and the implementation of the type system was correct. The notion of correctness is that circuits that are constructed using operations of the abstract data type, do indeed have the symmetry we want to capture. In principle proofs could have been done using a pencil and paper, but we believe doing proofs using a machine offers a considerably higher degree of confidence in the proof, and also serves as a blue-print for any future work, a sentiment also espoused recently by Aydemir et al. [14].

In subsequent sections, we present the details of our approach, but before (we do that a note) on the presentational style and notations. Although the theory presented in the next few sections is the one implemented in the theorem prover HOL 4 [3], we present the definitions using the more familiar functional style used in standard texts on functional programming [21, 89] rather than using the syntax specific to HOL 4.

6.3 Functional blocks — Level 0 framework

We use several well known functions on lists from functional programming [3, 21, 89], such as *hd*, *tl*, *append*, *map*, *map2*, *fold* and so on, as building blocks for defining other higher-order combinators that are used to define symmetric circuit blocks. Here for the sake of completeness we shall present these functions. These definitions are all very simple and self explanatory.

6.3.1 Some useful functions on lists

Definition 6.1. *Head and tail of a list*

$$\begin{array}{lcl} hd\ (h :: t) & \triangleq & h \\ tl\ (h :: t) & \triangleq & t \end{array}$$

Definition 6.2. *List membership*

$$\begin{array}{lcl} e\ \in\ [\,] & \triangleq & F \\ e\ \in\ (h :: t) & \triangleq & (e = h)\ \lor\ (e\ \in\ t) \end{array}$$

Definition 6.3. *Length of a list*

$$\begin{array}{lcl} length\ [\,] & \triangleq & 0 \\ length\ (h :: t) & \triangleq & 1\ +\ (length\ t) \end{array}$$

Definition 6.4. *Selecting an n^{th} element from a list*

$$\begin{aligned} &el\ \ 0\ \ l &&\triangleq\ \ hd\ \ l\\ &el\ \ (n+1)\ \ l &&\triangleq\ \ el\ \ n\ \ (tl\ \ l) \end{aligned}$$

Definition 6.5. *Append lists*

$$\begin{aligned} &append\ \ [\,]\ \ l &&\triangleq\ \ l\\ &append\ \ (x :: l_1)\ \ l_2 &&\triangleq\ \ (x :: (append\ \ l_1\ l_2)) \end{aligned}$$

Definition 6.6. *Fold on lists*

$$\begin{aligned} &foldr\ \ f\ \ e\ \ [\,] &&\triangleq\ \ e\\ &foldr\ \ f\ \ e\ \ (x :: l) &&\triangleq\ \ f\ \ x\ \ (foldr\ f\ e\ l) \end{aligned}$$

Definition 6.7. *Drop i elements from a list*

$$\begin{aligned} &drop\ \ 0\ \ l &&\triangleq\ \ tl\ \ l\\ &drop\ \ (i+1)\ \ l &&\triangleq\ \ drop\ \ i\ \ (tl\ \ l) \end{aligned}$$

Definition 6.8. *Take i elements from a list*

$$\begin{aligned} &take\ \ 0\ \ l &&\triangleq\ \ [\,]\\ &take\ \ (i+1)\ \ (x :: xs) &&\triangleq\ \ (x :: (take\ \ i\ \ xs)) \end{aligned}$$

Definition 6.9. *Update an element at a given position in a list*

$$update\ \ elem\ \ i\ \ lst\ \ \triangleq\ \ append(take\ \ i\ \ lst)(elem :: (drop\ \ i\ \ lst))$$

Definition 6.10. *Map a given function on a list*

$$\begin{aligned} &map\ \ f\ \ [\,] &&\triangleq\ \ [\,]\\ &map\ \ f\ \ (h :: t) &&\triangleq\ \ f\ \ h :: (map\ \ f\ \ t) \end{aligned}$$

Definition 6.11. *Apply a function pointwise on a pair of lists*

$$\begin{aligned} &map2\ \ f\ \ [\,]\ [\,] &&\triangleq\ \ [\,]\\ &map2\ \ f\ \ (h_1 :: t_1)\ \ (h_2 :: t_2) &&\triangleq\ \ f\ h_1\ h_2 :: map2\ \ f\ t_1\ t_2 \end{aligned}$$

Note that the definitions of *drop* and *take* are not exactly the same as those given in standard texts on functional programming such as ML or Haskell. The function *map2* is referred to as *zipwith* in some functional programming languages such as Haskell.

6.3.2 Some useful properties about functions over lists

Before we present the Level 0 layer, we will present some useful properties on lists, that will be found useful for proofs of other lemmas and theorems later on. The first of these states the fact that the length of the list remains invariant under the application of the function *map*.

Lemma 6.1. *Length remains invariant under application of map*

$$\vdash \ \forall inp\, f.\ length(map\ f\ inp)\ =\ length\ inp$$

Proof outline: By induction on *inp*, and rewriting with the definitions of *map*, and *length*. □

Similarly, the length of the list remains unchanged under application of the function *map2*, which is stated in the next Lemma.

Lemma 6.2. *Length remains invariant under application of map2*

$$\begin{aligned} \vdash \ \forall inp_1\ inp_2.\ (length\ inp_1\ = length\ inp_2) \supset \\ \forall f.\ (length(map2\ f\ inp_1\ inp_2)\ =\ length\ inp_1) \wedge \\ (length(map2\ f\ inp_1\ inp_2)\ =\ length\ inp_2) \end{aligned}$$

Proof outline: The proof takes place by induction on inp_1 and inp_2, and some simple rewrites using the definitions of *map2* and *length*. □

The following lemmas state the relation of *map* and other functions such as *append*, *take*, *drop* and *el*. All these lemmas become very useful in the proofs of soundness lemmas.

The first of these states that *map* distributes over *append*. So the order in which we apply *map* does not matter, we can do it before doing the append or after, the result is the same. This is stated in Lemma 6.3.

Lemma 6.3. *Map distributes over append*

$$\vdash \ \forall inp_1\, inp_2\, f.\ map\ f\ (append\, inp_1\ inp_2)\ =\ append\ (map\ f\ inp_1)(map\ f\ inp_2)$$

Proof outline: By induction on inp_1, and inp_2, and rewriting with the definitions of *map* and *append*. □

Next, we have the property that *map* distributes over *take* and *drop* and *el*. This is given by Lemma 6.4, Lemma 6.5 and Lemma 6.6 respectively.

Lemma 6.4. *Map distributes over take*

$$\vdash \ \forall inp\, i.\ (i < (length\ inp))\ \supset\ \forall f.\ (map\ f\ (take\ i\ inp)\ =\ take\ i\ (map\ f\ inp))$$

Proof outline: By induction on *inp*, then solving the base case by falsifying the assumption ($i < (length\ [\,])$), and the step is solved by induction on i, and using the definition of *take*. □

Lemma 6.5. *Map distributes over drop*

$$\vdash \;\; \forall inp\, i.\; (i < (length\; inp)) \;\supset\; \forall f.\; (map\; f\; (drop\; i\; inp) \;=\; drop\; i\; (map\; f\; inp))$$

Proof outline: By induction on *inp*, then solving the base case by falsifying the assumption ($i < (length\; [\,])$), and the step is solved by induction on i, and using the definition of *drop*. □

Lemma 6.6. *Map distributes over el*

$$\vdash \;\; \forall inp\, i.\; (i < (length\; inp)) \;\supset\; \forall f.\; (f\; (el\; i\; inp) \;=\; el\; i\; (map\; f\; inp))$$

Proof outline: By induction on *inp*, base is solved by falsifying the assumption, and the step is done by induction on i, and rewriting with the definition of *length*. □

The next two lemmas explains the rather complex dynamics when the functions *map*, *append*, *take*, *drop*, and *el* interact. These two lemmas are used in the proof of Lemma 6.10.

Lemma 6.7. *Map-Take-Append*

$$\begin{aligned}\vdash \;\; & \forall inp\; i\; j.\; (i < (length\; inp)) \;\supset\; (j < (length\; inp)) \;\supset \\ & \quad \forall f.\; (map\; f\; (take\; i\; (append (take\; j\; inp)(el\; i\; inp :: (drop\; j\; inp))))) \\ & \quad = \\ & \quad take\; i\; (append\; (take\; j\; (map\; f\; inp))(el\; i\; (map\; f\; inp) :: (drop\; j\; (map\; f\; inp))))\end{aligned}$$

Proof outline: The proof begins by induction on *inp*. The base case is solved trivially by falsifying the assumption. The step case is solved by a further induction on i and j. The case for $i = 0$ and $j = 0$ is solved by rewriting with the definitions of *map*, *take*, *drop*, *append* and *el*. Rest of the cases are solved by rewriting with the above definitions and Lemma 6.4, Lemma 6.5 and Lemma 6.6. □

Lemma 6.8. *Map-Drop-Append*

$$\begin{aligned}\vdash \;\; & \forall inp\; i\; j.\; (i < (length\; inp)) \;\supset\; (j < (length\; inp)) \;\supset \\ & \quad \forall f.\; (map\; f\; (drop\; i\; (append (take\; j\; inp)(el\; i\; inp :: (drop\; j\; inp))))) \\ & \quad = \\ & \quad drop\; i\; (append\; (take\; j\; (map\; f\; inp))(el\; i\; (map\; f\; inp) :: (drop\; j\; (map\; f\; inp))))\end{aligned}$$

Proof outline: The proof of this lemma is exactly similar to the proof of the above theorem. □

The last of the properties in this section, states that the non-empty lists are closed under the *tl* function. That is for all lists that are non-empty, if an element is contained in the tail of a list, it is contained in the parent list itself.

Lemma 6.9. *Non-empty lists are tail closed*

$$\vdash \;\; \forall inp.\; (inp \;\neq\; [\,]) \;\supset\; \forall m.\; (m \;\in\; (tl\; inp)) \;\supset\; (m \;\in\; inp)$$

Proof outline: By induction on *inp*, and rewriting with the definitions of list membership $\in$, *length* and *tl*. □

Functional blocks that exhibit symmetry should have the property of being commutative and associative.

Definition 6.12. *Commutativity and Associativity*

$$\begin{array}{lcl} comm\ f & \triangleq & \forall xy.\ f\ x\ y\ =\ f\ y\ x \\ assoc\ f & \triangleq & \forall xyz.\ f\ x\ (f\ y\ z)\ =\ f\ (f\ x\ y)\ z \end{array}$$

6.3.3 Level 0 functions

Together with the function *map*, the following functions constitute the Level 0 framework. The role of the functions is illustrated in Figure 6.1. The function *id* simply returns the Boolean list. We use the function *map*, to apply a function of the type (*bool* → *bool*), to a list of Boolean values. The serial composition operator ∘ composes two functions of the type (*bool list* → *bool list*), and the function *fold*, folds a function on Booleans, of the type (*bool* → *bool* → *bool*), onto a list of Booleans. The base value for the fold function is derived from the head of the list.

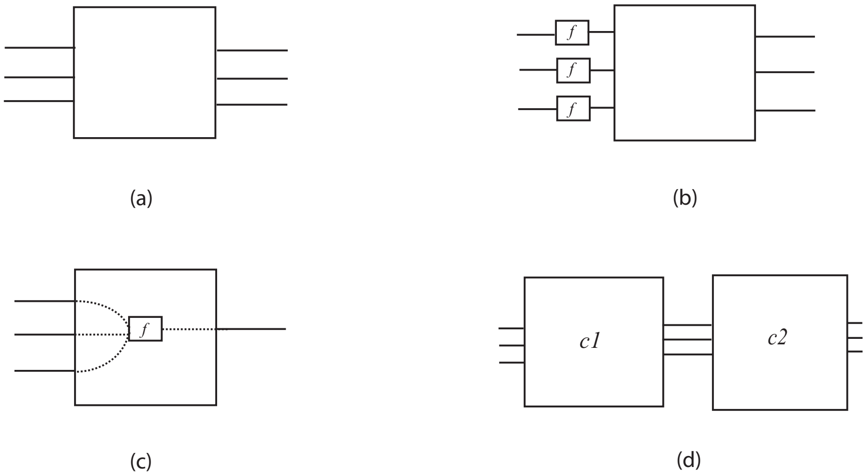


Figure 6.1: Functional blocks — Level 0. Identity is shown in (a), the function *map* in (b), *fold* is shown in (c), and (d) shows serial composition.

Definition 6.13. *Identity, serial composition and fold*

$$\begin{array}{lcl} id & \triangleq & \lambda inp : bool\ list.\ inp \\ f\ \circ\ g & \triangleq & (\lambda inp.\ f\ (g\ inp)) \\ fold\ \ f\ \ (c : bool\ list\ \rightarrow bool\ list) & \triangleq & \lambda inp.\ \left[foldr\ \ f\ \ (hd\ (c\ inp))\ (tl\ (c\ inp)) \right] \end{array}$$

$$\frac{}{\mathit{safe}\ \ \mathit{id}}$$

$$\frac{f : \mathit{bool} \ \rightarrow\ \mathit{bool}}{\mathit{safe}\ \ (\mathit{map}\ \ f)}$$

$$\frac{(\mathit{safe}\ \ c)\quad (f : \mathit{bool}\ \rightarrow\ \mathit{bool}\ \rightarrow\ \mathit{bool}) \qquad (\mathit{assoc}\ f)\ \ (\mathit{comm}\ f)}{\mathit{safe}\ \ (\mathit{fold}\ \ f\ \ c)}$$

$$\frac{\mathit{safe}\ \ c_1 \qquad \mathit{safe}\ \ c_2}{\mathit{safe}\ \ (c_1 \circ\ c_2)}$$

Table 6.1: Rules for safe functional blocks.

6.3.4 Functional blocks are safe

The functional blocks defined at Level 0 have a symmetry property. We define a predicate *safe*, ensuring that functional blocks are always used in a certain designated manner — the result of the construction entails the symmetry property of the functional block. The predicate *safe* is defined inductively, by the rules given in Table 6.1. We have used the inductive definition package [31, 77] to define these rules. The result of such a definition typically is a set of rule definitions and tactics in HOL, that can be used for doing proofs about the predicate defined inductively.

Table 6.1 presents the type safety rules for Level 0. The function *id* is a safe function, since it simply returns the list that is supplied to it as the argument. The function $(\mathit{map}\ \ f)$ is a safe function, if f is a function from *bool* to *bool*. The third rule states that if a given function c is safe, then it is safe to fold an associative and commutative function onto the function c. This is because the ordering of elements in the list will not change the output of the fold function. Serial composition of two different safe functions is safe. This is the fourth rule.

The definition of symmetry we are going to present relies on the concept of a permutation. From Chapter 4 we know that any permutation on an n-element set, can be obtained by composing swap functions on the n-element set. Thus if we define the swap function to range between 0 and the length of the bus (Boolean list), then these swaps can act on the Boolean list to compute arbitrary permutations on the elements of the list. The Boolean lists themselves denote the state of a bus, whereby it is assumed that the Boolean value at each position i in the list, represents the state of the i^{th} bit of a bus, for all i between 0 and $n - 1$, where n is the width of the bus. Below we present the definition of the swap function on a list.

Definition 6.14. *Swap on lists*

$$\begin{array}{lll} swap\ \ (i,\, j)\ \ lst & \triangleq & if\ \ (i\ <\ length\, lst)\ \wedge\ (j\ <\ length\, lst)) \\ & & then\ \ (update\ (el\ j\ lst)\ \ i\ \ (update\ (el\ i\ \ lst)\ \ j\ \ lst)) \\ & & else\ \ lst \end{array}$$

A sequence of applications of swap can be used to compute arbitrary permutations of lists.

Before we present the definition of symmetry of a functional block, we will present some useful properties about the function *swap* and its relation with other functions on lists. The first of these is about the function *map*, which distributes over *swap*. This property becomes useful later when we try and prove the soundness lemma that all safe functions have symmetry.

Lemma 6.10. *Map distributes over swap*

$\vdash\ \ \forall inp\, f\, i\, j.\ map\ f\ (swap(i, j)\ inp)\ =\ swap(i, j)\ (map\ f\ inp)$

Proof outline: The proof begins by induction on *inp*. We get the following two cases:

1. $\forall f\, i\, j.\ map\ f\ (swap(i, j)\ [\,])\ =\ swap(i, j)(map\ f\ [\,])$
   This is proved trivially by rewriting with the definition of *map* and *swap*.
2. Assuming the following: $\forall f\, i\, j.\ map\ f\ (swap(i, j)\ inp)\ =\ swap(i, j)(map\ f\ inp)$ we have to prove the following:
   $$\forall h\, f\, i\, j.\ map\ f\ (swap(i, j)\ (h :: inp))\ =\ swap(i, j)(map\ f\ (h :: inp))$$

The goal is proved by doing an induction on i, and j. Thus we have to prove the following four sub-cases:

1. $map\ f\ (swap(0, 0)(h :: l))\ =\ swap(0, 0)(map\ f\ (h :: l))$
   This case is proved by rewriting the goal with the definition of *swap*.
2. $map\ f\ (swap(0, j + 1)(h :: l))\ =\ swap(0, j + 1)(map\ f\ (h :: l))$
   This goal is proved by using the definition of *swap* to rewrite the goal first, and then we use Lemma 6.3, Lemma 6.4, Lemma 6.5 and Lemma 6.6 to complete the proof for this case.
3. $map\ f\ (swap(i + 1, 0)(h :: l))\ =\ swap(i + 1, 0)(map\ f\ (h :: l))$
   This goal is proved by using the definition of *swap* to rewrite the goal first, and then we use Lemma 6.3, Lemma 6.4, Lemma 6.5, and Lemma 6.6 to complete the proof for this case.
4. $map\ f\ (swap(i + 1, j + 1)(h :: l))\ =\ swap(i + 1, j + 1)(map\ f\ (h :: l))$
   This goal is proved by using the definition of *swap* to rewrite the goal first, and then we use Lemma 6.6, Lemma 6.7, and Lemma 6.8 to complete the proof for this case. □

The following lemma, though not used in the proof of the soundness lemma at Level 0, is used later on in the proofs relating the symmetry of circuits with the fact they have been type checked. This lemma is about *map2* being distributive.

Lemma 6.11. *Map2 distributes over swap*

$$\begin{aligned}\vdash\ \ & \forall inp_1\ inp_2.\ (length\ inp_1\ =\ length\ inp_2)\ \supset \\ & \qquad \forall f\ i\ j.\ (map2\ f\ (swap(i,j)\ inp_1)\ (swap(i,j)\ inp_2)) = \\ & \qquad\qquad\qquad swap(i,j)\ (map2\ f\ inp_1\ inp_2)\end{aligned}$$

Proof outline: The proof takes place by induction on inp_1, inp_2, i and j, and the structure of the proof is similar to the one shown for Lemma 6.10.

A useful property about safe functions is that they are length preserving. This means that for two different input buses, if their lengths are equal, then the length of the outputs obtained by applying the safe function on each one of them is equal. This is given in Lemma 6.12.

Lemma 6.12. *Safe functions are length preserving*

$$\vdash\ \ \forall c_0.\ safe\ c_0\ \supset\ \forall l\ m.\ (length\ l = length\ m)\ =\ (length\ (c_0\ l) = length\ (c_0\ m))$$

Proof outline: The proof begins by rule induction [110] on predicate *safe*. The first case is solved trivially by a rewrite using the definition of *id*. The next case uses Lemma 6.1, and the remaining two cases are discharged by rewriting with the definitions of *fold* and the serial composition ∘ respectively. □

Now we shall provide some useful properties about certain functions on lists, which we shall use at a later stage in defining circuit construction combinators. The first of these properties says that the effect of applying the swap function to a non-empty list of lists, and then taking the tail of that, is same as taking the tail of the non-empty list of lists first, and then applying the swap function. This is stated in Lemma 6.13.

Lemma 6.13. *Mapping swap on non-empty list distributes over tail*

$$\vdash\ \ \forall inp.\ (inp\ \neq\ [\,])\ \supset\ \forall i\,j.\ (map(swap(i,j))\ (tl\ inp))\ =\ tl(map(swap(i,j))\ inp)$$

Proof outline: Induction on *inp*, using the definitions of *map* and *tl*. □

An interesting property about swap is that, we can select an n^{th} element from a list of buses, and the apply swap to it, and it has the same effect as first selecting the n^{th} list, and then applying swap on it. This is the property stated in Lemma 6.14.

Lemma 6.14. *Mapping the swap is distributive over element selection*

$$\begin{aligned}\vdash\ \ & \forall inp.\forall n.\ (n\ <\ (length\ inp))\ \supset \\ & \qquad\qquad \forall i\,j.\ (el\ n\ (map(swap(i,j))\ inp))\ =\ (swap(i,j)(el\ n\ inp))\end{aligned}$$

Proof: By induction on *inp*, we get two cases. The base case is solved trivially, since the assumption is falsified by $n\ <\ length\,[\,]$. The step case is solved by inducting on n, and making use of the definition of *el*. □

The function *swap* does not change the length of a list. This is given by Lemma 6.15.

Lemma 6.15. *Length remains invariant under swap*

$$\vdash \;\; \forall inp. \forall i\, j.\; length\; inp \;=\; length\; (swap(i,j)\; inp)$$

Proof: By induction on *inp*, we get two cases. The base case is solved trivially since the assumption is falsified by $i \;<\; length\,[\,]$. The step case is solved by rewriting with the definition of *swap*, and then doing a case-split on the conditional $if - then - else$. The subsequent proof is completed by making use of properties on lists that deal with *take*, *drop* and *append* operations. □

Applying the *swap* function to each element of a list of buses does not change the length of the list of buses. This is stated in Lemma 6.16.

Lemma 6.16. *Length remains invariant when mapping the swap function*

$$\vdash \;\; \forall inp. \forall i\, j.\; length\; inp \;=\; length\; (map(swap(i,j))\; inp)$$

Proof: We begin by doing the induction on *inp*. The base case is solved trivially using the definition of *map* and *swap*. The step case is solved by doing an induction on i, and j, and using the definition of *swap*. □

The following lemma relates the application of the function *foldr* on lists, with the function *swap*. It says that the function *swap* distributes over *foldr*. An important constraint on the function f is that it should be associative, and commutative.

Lemma 6.17. *Swap distributes over Foldr*

$$\begin{aligned} \vdash \;\; \forall f.\; assoc\; f \;\supset\; comm\; f \supset \\ \forall inp\, i\, j.\; swap(i,j)[foldr\; f\; (hd\; inp)\; (tl\; inp)] \\ =\; [foldr\; f(hd(swap(i,j)\; inp))(tl(swap(i,j)\; inp))] \end{aligned}$$

Proof outline: The proof takes place by induction on *inp*, i, and j. □

Now we are ready to define precisely the concept of symmetry for functional blocks.

Definition 6.15. *Symmetry of functional blocks*

$$sym\; c \;\;\triangleq\;\; \forall inp\, i\, j.\; c\;\; (swap\; (i,j)\; inp) \;\;=\;\; swap\; (i,j)\; (c\; inp)$$

Symmetry of functional blocks is the property that the behaviour of the functional block will not change under the permutation of the input and output states of the functional block, and this is formalised in Definition 6.15.

All functions that are checked by the predicate *safe* have the property of symmetry. This is articulated by the Level 0 Safety Lemma.

Theorem 6.1. *Level 0 Safety Theorem*

$\vdash \ \forall c.\ safe\ \ c\ \supset\ \ sym\ \ c$

Proof outline: The proof takes place by an induction on the rules defining safe. Thus we have four cases to prove:

1. $sym\, id$

   This is proved by rewriting with the definition of *sym* and *id*.
2. $\forall f.\ sym\ (map\ f)$

   By rewriting the goal with the definition of *map*, and *sym*, we need to prove the following sub-goal:

   $map\ f\ (swap(i,j)\ inp) = swap(i,j)(map\ f\ inp)$

   which is an instance of Lemma 6.10.
3. Given the following assumptions
   (a) $safe\, c$
   (b) $sym\, c$
   (c) $assoc\, f$
   (d) $comm\ f$

   we have to prove

   $sym\ (fold\ f\ c)$

   Rewriting this goal using the definition of *fold*, *sym*, we get the following sub-goal:

   $$\begin{aligned}\forall inp\ i\ j.[foldr\ f\ (hd\ (swap(i,j)(c\ inp)))(tl\ (swap(i,j)(c\ inp)))]\\ =\ swap(i,j)[foldr\ f\ (hd\ (c\ inp))(tl\ (c\ inp))]\end{aligned}$$

   which is an instance of Lemma 6.17.
4. Given the following assumptions
   (a) $safe\, c_0$
   (b) $sym\, c_0$
   (c) $safe\, c_1$
   (d) $sym\, c_1$

   we have to prove

   $sym\ (c_0\ \circ\ c_1)$

   This goal is easily discharged by using the definitions of *sym* and serial composition $\circ$.

□

The Level 0 functional blocks presented above serve as atomic components for defining circuit construction combinators later on. The concept of symmetry of functional blocks is also an atomic concept, in the sense that since all functional blocks at Level 0 have symmetry, circuits constructed by employing symmetric functional blocks, will have symmetry. Level 0 safety theorem is used in the proof of the lemma we present next.

The following lemma articulates the connection between safe functions and their application to construct circuits that preserve symmetry. Lemma 6.18 states that the application of swap to a list of inputs can be done before mapping the safe function onto the inputs, or after mapping the safe function; the effect is the same in both the cases.

Lemma 6.18. *Okay to map safe functional blocks*

$$\vdash \ \ \forall c_0.\ \mathit{safe}\ c_0 \ \ \supset \ \ \forall \mathit{inp}. \forall i\, j.\ \mathit{map}(\mathit{swap}(i,j))\ (\mathit{map}\ c_0\ \mathit{inp}) \ = \\ \mathit{map}\ c_0\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$$

Proof outline: Using the rule induction tactic we can decompose the proof into four cases:

1. $\forall \mathit{inp}\, i\, j.\ \mathit{map}(\mathit{swap}(i,j))(\mathit{map}\ \mathit{id}\ \mathit{inp}) \ = \ \mathit{map}\ \mathit{id}\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$. By induction on *inp*, we get two cases:
   (a) $\forall i\, j.\ \mathit{map}(\mathit{swap}(i,j))(\mathit{map}\ \mathit{id}\ [\,]) \ = \ \mathit{map}\ \mathit{id}\ (\mathit{map}(\mathit{swap}(i,j))\ [\,])$
   This case is proven trivially by unfolding the definition of *id*, and *map*.
   (b) Assuming, the following
   $\forall i\, j.\ \mathit{map}(\mathit{swap}(i,j))(\mathit{map}\ \mathit{id}\ \mathit{inp}) \ = \ \mathit{map}\ \mathit{id}\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$
   we have to prove
   $\forall h\, i\, j.\ \mathit{map}(\mathit{swap}(i,j))(\mathit{map}\ \mathit{id}\ (h :: \mathit{inp})) \ = \ \mathit{map}\ \mathit{id}\ (\mathit{map}(\mathit{swap}(i,j))\ (h :: \mathit{inp}))$
   The goal can be proven by unfolding the definition of *map* and *id*.
2. The second case we have to prove is:
   $\forall f\ \mathit{inp}\, i\, j.\ \mathit{map}(\mathit{swap}(i,j))(\mathit{map}(\mathit{map}\ f)\ \mathit{inp}) \ = \ \mathit{map}(\mathit{map} f)\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$
   This is proved by an induction on *inp*. The base case is trivially discharged by using the definition of *map*, and the step case we have to prove is an instance of Lemma 6.10.
3. Given we know that c_0 is safe, and f is associative, and commutative, and we know the following
   $\forall \mathit{inp}\, i\, j.\ \mathit{map}(\mathit{swap}(i,j))(\mathit{map}\ c\ \mathit{inp}) \ = \ \mathit{map}\ c\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$
   we have to prove
   $$\forall \mathit{inp}\, i\, j.\ \mathit{map}(\mathit{swap}(i,j))\ (\mathit{map}(\mathit{fold}\ f\ c)\ \mathit{inp}) \\ = \ \mathit{map}(\mathit{fold}\ f\ c)\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$$
   The proof begins by induction on *inp*, and the base case is discharged by rewriting the goal by the definition of *map*. For the step case we have the following assumptions:
   (a) *safe c*
   (b) *assoc f*
   (c) *comm f*
   (d) $\forall \mathit{inp}\, i\, j.\ (\mathit{map}(\mathit{swap}(i,j))\ (\mathit{map}\ c\ \mathit{inp})) \ = \ (\mathit{map}\ c\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}))$
   (e) $\forall i\, j.\ \mathit{map}(\mathit{swap}(i,j))(\mathit{map}(\mathit{fold}\ f\ c)\ \mathit{inp}) \ = \ \mathit{map}(\mathit{fold}\ f\ c)\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$
   and we have to prove the following
   $$\forall h\, i\, j.\ \mathit{map}(\mathit{swap}(i,j))\ (\mathit{map}(\mathit{fold}\ f\ c)\ (h :: \mathit{inp})) \\ = \ \mathit{map}(\mathit{fold}\ f\ c)\ (\mathit{map}(\mathit{swap}(i,j))\ (h :: \mathit{inp}))$$
   If we rewrite the goal using the definition of *fold*, for arbitrary h, i and j, we get the new goal as:
   $$\mathit{map}(\mathit{swap}(i,j))\ (\mathit{map}(\lambda l.\ [\mathit{foldr}\ f\ (\mathit{hd}(c\ l))\ (\mathit{tl}(c\ l))])\ (h :: \mathit{inp})) \\ = \ \mathit{map}(\lambda l.\ [\mathit{foldr}\ f\ (\mathit{hd}(c\ l))\ (\mathit{tl}(c\ l))])\ (\mathit{map}(\mathit{swap}(i,j))\ (h :: \mathit{inp}))$$

But since c is safe, we can deduce from Theorem 6.1, that *sym c* holds, which if we use to rewrite the goal, we get the following two sub-goals as:

$$\begin{aligned} &map(swap(i,j))\ (map(\lambda l.\ [foldr\ f\ (hd(c\ l))\ (tl(c\ l))])\ inp) \\ &\qquad = \ map(\lambda l.\ [foldr\ f\ (hd(c\ l))(tl(c\ l))])\ (map(swap(i,j))\ inp) \end{aligned}$$

This sub-goal can be discharged by rewriting the sub-goal with the definition of *fold*.

$$\begin{aligned} &swap(i,j)\ [foldr\ f\ (hd(c\ h))\ (tl(c\ h))] \\ &\qquad = \ [foldr\ f\ (hd(swap(i,j)(c\ h)))(tl(swap(i,j)(c\ h)))] \end{aligned}$$

This sub-goal is an instance of Lemma 6.17. Thus can be rewritten easily using that lemma.

4. The last case to be proven is:

$$\forall inp\, i\, j.\ map(swap(i,j))\ (map(c_0\ \circ\ c_1)\ inp)\ =\ map(c_0\ \circ\ c_1)(map(swap(i,j))\ inp)$$

The proof in this case also begins by induction on *inp*. The base case is solved by rewriting the goal with the definition of *map*. The step case is discharged by deducing from the assumptions, that c_0 and c_1 are symmetric, and then rewriting the goal by using the definition of symmetry. □

6.4 Level 1 framework

In this section we shall present the elements of the next layer of our framework of designing structured models. This layer, called Level 1, consists of circuit construction combinators that allow us to construct symmetric circuits. We use the safe functions from Level 0 in the definition of circuit construction, and also define a new set of higher-order combinators that are used for designing bottom-up symmetric circuits. We provide type judgement rules that govern the way symmetric circuits can be constructed, and later on prove that the type judgement rules are sound, meaning that every piece of circuit definition that is well-behaved with respect to the type judgement rules has symmetry.

Before we proceed with the definitions of our combinators, we define a predicate asserting that all buses in a list are of equal length. This is vital for us since circuits that do bitwise operations on a pair of inputs, require the inputs to be of equal length.

Definition 6.16. *Buses are of equal length*

$$CheckLength\ \ inp\ \ \triangleq\ \ \forall l.\ l \in inp\ \supset\ \forall m.\ m \in inp\ \supset\ (length\, l\ =\ length\, m)$$

Immediately following from the definition of *CheckLength* are some Corollaries used in the proofs later on about lengths of buses and different functions operating on the buses.

Corollary 6.1. $\forall inp\ h.\ CheckLength(h :: inp)\ \supset\ CheckLength\ inp$

Corollary 6.2. $\forall inp\ h'\ h.\ CheckLength(h' :: (h :: inp))\ \supset\ CheckLength(h' :: inp)$

Corollary 6.3. $\forall inp\ h.\ CheckLength(h :: inp)\ =$
$(CheckLength\ inp)\ \wedge\ \forall m.\ m\ \in\ inp\ \supset\ (length\, m = length\, h)$

Corollary 6.4. $\forall inp\ h'\ h.\ CheckLength(h' :: (h :: inp))\ \supset\ (length\ h' = length\ h)$

The following lemma states that if a list of buses has the property that all buses are of equal length, then if the swap operation is applied to the tail of the list of buses, then all the resulting buses in the list have the same length as the first element of the original list.

Lemma 6.19. $\forall inp\ h.\ CheckLength(h :: inp)\ \supset$
$\qquad\forall i\ j.\ CheckLength(map(swap(i,j))\ inp)\ \supset$
$\qquad\qquad\forall m.\ m\ \in\ (map(swap(i,j))\ inp)\ \supset\ (length\ m =\ length\ h)$

Proof outline: The proof begins by induction on *inp*. The base case is proven by first rewriting with the definition of *map* and then falsifying the assumption $m\ \in\ [\,]$. The step case that we have to solve is the following:

$$\begin{array}{l}\forall h\ h'.\ CheckLength(h' :: (h :: inp))\ \supset \\ \qquad \forall i\ j.\ CheckLength(map(swap(i,j))(h :: inp))\ \supset \\ \qquad\qquad \forall m.m\ \in\ (map(swap(i,j))\ (h :: inp))\ \supset\ (length\ m = length\ h')\end{array}$$

Stripping the quantifiers, and pushing the antecedent of the implication into the assumption, and rewriting with the definition of *map*, we get the following assumptions

(a) $\forall h.\ CheckLength(h :: inp)\ \supset$
$\qquad\forall i\ j.\ CheckLength(map(swap(i,j))inp)\ \supset$
$\qquad\qquad\forall m.m\ \in\ (map(swap(i,j))\ inp)\ \supset\ (length\ m = length\ h)$

(b) $CheckLength(h' :: h :: inp)$

(c) $CheckLength((swap(i,j)\ h) :: (map(swap(i,j))\ inp))$

and the following sub-goal

$$\forall\ m.\ m\ \in\ ((swap(i,j)\ h) :: (map(swap(i,j))\ inp)\ \supset\ (length\ m = length\ h')$$

Using Corollary 6.1, we can add the following assumptions:

(d) $CheckLength(h :: inp)$

(e) $CheckLength(map(swap(i,j))\ inp)$

Rewriting the goal with the definition of list membership $\in$, we get the following two sub-goals:

(i) $length\ (swap(i,j)\ h) = length\ h'$

(ii) $length\ m = length\ h'$

The first sub-goal (i) can be proven by using Lemma 6.15, and transitivity. The second sub-goal (ii) is proven by using Corollary 6.2. □

The following lemma is very similar to the above lemma, in that it relates the length of the buses before and after mapping swap onto a list of buses. It states that if a swap function is applied onto each bus in a list of buses (*inp*), and all the resulting buses are of equal length, and if all the buses in *inp* are of equal length, then the length of a bus which is in *inp* is the same as the length of the first bus that belongs to the list of swapped buses.

Lemma 6.20. $\forall inp\ h. \forall i\ j.\ \ CheckLength((swap(i,j)\ h) :: (map(swap(i,j))\ inp))\ \supset$
$CheckLength\ inp\ \supset$
$\forall m.\ m\ \in\ inp\ \supset\ (length\ m =\ length\ (swap(i,j)\ h))$

Proof outline: The proof for this lemma proceeds exactly in the same manner as the proof of the previous lemma. □

Both Lemma 6.19 and Lemma 6.20 are used in the proof of Lemma 6.28 at a later stage.

Lemma 6.21, also relates the property of length preservation of buses, under swap. This lemma is about the length of all the buses being equal if a composition of the safe functions is mapped onto a list of buses all of whose buses are of equal length and is equal to the length of the first bus. This lemma is used for the proof of Lemma 6.27.

Lemma 6.21. *Serial composition of safe functions preserves the length*

$$\vdash\ \ \forall c_0\ c_1.\ safe\ c_0\ \supset\ safe\ c_1\ \supset\ \forall inp\ h.\ CheckLength(h :: inp)\ \supset$$
$$\forall m.\ m\ \in\ (map(c_0\ \circ\ c_1)\ inp)\ \supset\ (length\ m\ =\ length\ (c_0(c_1\ h)))$$

Proof outline: The proof is carried out by inducting on *inp*, and solving the resulting sub-goals by using the definition of *CheckLength* and using Lemma 6.12. □

The following lemma states that if the length of all the buses within the list is equal then if this list is appended with itself, the resulting list will have all the buses of equal length.

Lemma 6.22. *Bus lengths stay fixed under bus append*

$$\vdash\ \ \forall inp.\ CheckLength\ inp\ \ \supset\ \ CheckLength(append\ inp\ inp)$$

Proof outline: By induction on *inp*, then using the definition of *CheckLength* and *append* to rewrite the goal. □

Since *id* is the function that gives the same output as the input, mapping this onto a list of buses, will give a list that is equal to the original list of buses. This is stated in Lemma 6.23.

Lemma 6.23. *Mapping identity on a list*

$$\vdash\ \ \forall inp\, m.\ (m\ \in\ (map\ id\ inp))\ \equiv\ (m\ \in\ inp)$$

Proof outline: Proof takes place by induction on *inp*, and rewriting with the definitions of *map*, list membership $\in$, and *id*. □

Thus if an identity function is mapped onto a list of buses all of which are of equal length, then the resulting bus also has the buses all of equal length. Lemma 6.24 states precisely this.

Lemma 6.24. *Length is invariant when mapping the identity function*

$\vdash\ \ \forall inp.\ CheckLength\ inp\ \supset\ CheckLength\ (map\ id\ inp)$

Proof outline: By induction on *inp*, we get two cases. The base case is trivially solved by using the definition of *map*. The step case is rewritten using the definition of *CheckLength*, and the resulting sub-goal is solved using Lemma 6.23. □

This brings us to another related lemma, which states that if we map a function ($map\ f$) the length of the buses remain equal if they were equal in the original list. This is given in Lemma 6.25.

Lemma 6.25. *Length is invariant when mapping a* ($map\ f$)

$\vdash\ \ \forall inp\ f.\ CheckLength\ inp\ \supset\ CheckLength(map(map\ f)\ inp)$

Proof outline: The proof takes place by induction on *inp*. The base case is solved by rewriting with the definition of *map*. The step case is solved by rewriting the sub-goal with the definition of *CheckLength*, and then using Lemma 6.1 and using the following property:

$$\forall inp\ h.\ CheckLength(h :: inp)\ \supset\ \forall m\ f.\ m\ \in\ (map(map\ f)\ inp)\ \supset \\ (length\ m = length\ h)$$

This property is proven by inducting on *inp* and using the definition of *CheckLength*. □

Lemma 6.26 states that mapping a function ($fold\ f\ c_0$) onto a list of buses reduces the length of all buses in the resulting list to 1.

Lemma 6.26. *Mapping a fold reduces the length of the buses to one*

$\vdash\ \ \forall inp.\ \forall c_0\ m\ f.\ \ m\ \in\ (map(fold\ f\ c_0)\ inp)\ \supset\ (length\ m = 1)$

Proof outline: We do an induction on *inp*, and simplify the resulting sub-goals using the definition of *fold* and list membership $\in$. □

This brings us to one of the most important lemmas in this section, which states that mapping a safe function is a length preserving operation on the buses of a list.

Lemma 6.27. *Mapping a safe function preserves the length*

$\vdash\ \ \forall c_0.\ safe\ c_0\ \ \supset\ \forall inp.\ CheckLength\ inp\ \ \supset\ \ CheckLength(map\ c_0\ inp)$

Proof outline: The proof begins by rule induction on the the rules defining safe. We have four cases to prove:

1. $\forall inp.\ CheckLength\ inp \supset\ CheckLength(map\ id\ inp)$
   This case is exactly same as Lemma 6.24, and thus can be used to rewrite the goal in one step.
2. $\forall f\ inp.\ CheckLength\ inp \supset\ CheckLength(map(map\ f)\ inp)$
   This goal is same as Lemma 6.25, and thus a single rewrite step using this lemma can discharge this goal.

3. Assuming the following:
   (a) $safe\ c$
   (b) $\forall inp.\ CheckLength\ inp \supset CheckLength(map\ c\ inp)$
   (c) $assoc\ f$
   (d) $comm\ f$

   we have to prove

   $\forall inp.\ CheckLength\ inp \supset CheckLength(map(fold\ f\ c)\ inp)$

   The proof for this sub-goal begins by induction on inp. Thus we have two cases to prove:
   (i) $CheckLength\ [\,] \supset CheckLength(map(fold\ f\ c)\ [\,])$
   This can be discharged by rewriting with the definition of map.
   (ii) $\forall h.\ CheckLength(h :: inp) \supset CheckLength(map(fold\ f\ c))(h :: inp)$
   The proof takes place by some obvious rewriting steps using the definition of $CheckLength$, and then using the property given by Lemma 6.26.
4. Assuming the following:
   (a) $safe\ c_0$
   (b) $safe\ c_1$
   (c) $\forall inp.\ CheckLength\ inp \supset CheckLength(map\ c_0\ inp)$
   (d) $\forall inp.\ CheckLength\ inp \supset CheckLength(map\ c_1\ inp)$

   we have to prove

   $\forall inp.\ CheckLength\ inp \supset CheckLength(map(c_0 \circ c_1)\ inp)$

   The proof begins by induction on inp. Thus we have two cases to prove:
   (i) $CheckLength\ [\,] \supset CheckLength(map(c_0 \circ c_1)\ [\,])$
   This can be discharged by rewriting with the definition of map.
   (ii) $\forall h.\ CheckLength(h :: inp) \supset CheckLength(map(c_0 \circ c_1))(h :: inp)$
   This sub-goal can be proved by rewriting with the definition of map, $\circ$, and using the property given by Lemma 6.21.

□

The next lemma states that applying a swap onto each bus in a given list of buses, preserves the length of the list. If the buses in the original list were of equal length, then after the swap the resulting list of buses will have all the buses of equal length.

Lemma 6.28. *Length invariant is preserved under swap*

$$\vdash\ \forall inp.\ \forall i\ j.\ (CheckLength\ inp) \equiv (CheckLength(map(swap(i,j))\ inp))$$

Proof outline: First we shall prove the $\supset$ direction, and then the *only-if* direction. Thus we have the following two immediate goals:

1. $\forall inp.\ \forall i\ j.\ (CheckLength\ inp) \supset (CheckLength(map(swap(i,j))\ inp))$
   The proof starts by induction on inp. The base case is trivially solved by rewriting the goal with the definition of map. The step case proceeds by a rewriting step using the definition of map. The resulting sub-goal is stripped of all its quantifiers and by having the following assumptions:

(a) $\forall ij.\ \mathit{CheckLength}\ \mathit{inp}\ \supset\ \mathit{CheckLength}(\mathit{map}(\mathit{swap}(i,j)\ \mathit{inp}))$

(b) $\mathit{CheckLength}(h :: \mathit{inp})$

we are left to prove:

$\mathit{CheckLength}((\mathit{swap}(i,j)\ h) :: (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}))$

By using Corollary 6.3, we can rewrite the goal to

$$\mathit{CheckLength}(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}) \wedge \\ \forall m.\ m\ \in\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})\ \supset\ (\mathit{length}\ m = \mathit{length}\ (\mathit{swap}(i,j)\ h))$$

From Lemma 6.15 and Lemma 6.19, we can easily prove the above goal.

2. $\forall \mathit{inp}.\ \forall i\ j.\ (\mathit{CheckLength}(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}))\ \supset\ (\mathit{CheckLength}\ \mathit{inp})$
   The proof for this direction proceeds similar to the other direction, by induction on *inp*, and discharging the base case trivially by rewriting with the definition of *map*. The step case is also solved similarly, by rewriting with Corollary 6.3. The resulting sub-goal is solved by doing rewriting using Lemma 6.15 and Lemma 6.20.

□

Lemma 6.28 is immediately used in the proof of the next lemma. The next lemma states that if by appending two different list of buses, we know that the resulting list has all the buses of equal length, then we can conclude that if we map the swap function onto each one of the inputs, and then append the resulting lists, all the buses in the resulting list will be of equal length.

Lemma 6.29. *Length of the buses are preserved when swap is applied*

$$\vdash\ \forall \mathit{inp}_1\ \mathit{inp}_2.\ \mathit{CheckLength}(\mathit{append}\ \mathit{inp}_1\ \mathit{inp}_2)\ \equiv \\ \forall i\ j.\ \mathit{CheckLength}(\mathit{append}(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}_1)(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}_2))$$

Proof outline: The equivalence is proved in two directions. Thus we have the following two sub-goals:

1. $\forall \mathit{inp}_1\ \mathit{inp}_2.$
   $$\mathit{CheckLength}(\mathit{append}\ \mathit{inp}_1\ \mathit{inp}_2)\ \supset \\ \forall ij.\ \mathit{CheckLength}(\mathit{append}\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}_1)\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}_2))$$
   The proof for this case proceeds by induction on inp_1. Thus we have the following two cases:

   (i) $\forall \mathit{inp}_2.\ \forall i\ j.$
   $$\mathit{CheckLength}(\mathit{append}\ [\,]\ \mathit{inp}_2)\ \supset \\ \mathit{CheckLength}(\mathit{append}(\mathit{map}(\mathit{swap}(i,j))\ [\,])(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}_2))$$
   This goal is reduced using the definition of *append*, and the subsequent goal is an instance of Lemma 6.28, hence is proved by a rewriting step.

   (ii) With the following assumptions:
   $$\forall \mathit{inp}_2\ i\ j.\ \mathit{CheckLength}(\mathit{append}\ (\mathit{inp}_1\ \mathit{inp}_2))\ \supset \\ \mathit{CheckLength}(\mathit{append}(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}_1)(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}_2))$$
   we have to prove:
   $$\forall h\ \mathit{inp}_2\ i\ j. \\ \mathit{CheckLength}(\mathit{append}\ (h :: \mathit{inp}_1)\ (\mathit{inp}_2))\ \supset \\ \mathit{CheckLength}(\mathit{append}(\mathit{map}(\mathit{swap}(i,j))\ h :: \mathit{inp}_1)(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}_2))$$

Rewriting the assumptions and the goal using the definition of *append*, and then rewriting the sub-goal with Corollary 6.3, we get the following two sub-goals:

(ii)-(a) $CheckLength(append\ (map(swap(i,j))\ inp_1)\ (map(swap(i,j))\ inp_2))$

(ii)-(b) $length\ m = length\ (swap(i,j)\ h)$

The first sub-goal (ii)-(a), can be discharged easily by using Corollary 6.1 and the assumptions. The second sub-goal (ii)-(b), can be solved by using Lemma 6.19. If we specialise the quantified variable *inp* in that lemma with $(append\ inp_1\ inp_2)$, and the variable h with h, and add it to our list of assumptions, we can subsequently rewrite the assumptions using the Lemma 6.3, Corollary 6.3 and Lemma 6.15 to prove the goal.

2. $\forall inp_1\ inp_2\ i\ j.$

$$CheckLength(append\ (map(swap(i,j))\ inp_1)\ (map(swap(i,j))\ inp_2)) \supset$$
$$CheckLength(append\ inp_1\ inp_2)$$

The proof for this direction starts in a very similar fashion to the proof for the other direction, except that when we have to prove the last sub-goal, similar to (ii)-(b) above, we use Lemma 6.20, and then rewrite the assumptions using the Lemma 6.3, Corollary 6.3 and Lemma 6.15 to prove the goal.

□

6.4.1 Combinators for constructing circuits

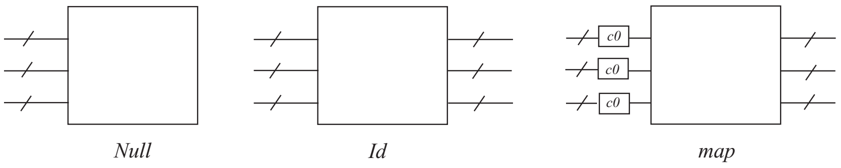


Figure 6.2: Circuit construction combinators showing the circuit that generates no output (*Null*), the Identity circuit (*Id*) and the function that maps a safe function (*map*).

The combinators used for defining circuit blocks are defined below. Note that all these combinators take a list of Boolean lists and produce a list of Boolean lists as an output. The intuition is that these combinators are all functions from symmetric inputs to outputs, where the type of list of Boolean lists is used for representing the symmetric input and outputs. When defining a circuit with non-symmetric inputs, one can pass them as the first argument and the symmetric inputs can be the second argument. Thus the combinators we define encapsulate the non-symmetric inputs (list of Boolean lists), and hence have the following type:

$(bool\ list)\ list \rightarrow (bool\ list)\ list$

This is the type we use to represent the class of symmetric circuit blocks, and the definitions of circuit construction combinators that we are going to present next, are all combinators that allow us to define symmetric circuits. Note however, that not every circuit with the above type will be symmetric, only those that are constructed using the combinators that we will soon show, have symmetry.

The combinator *Null*, shown in Figure 6.2 does not produce any output, and is useful for circuit designs, where for a certain configuration of inputs, the user wants to have an empty list of outputs. Then we have the identity or the buffer circuit, shown in Figure 6.2, and is defined by the combinator *Id*, which simply returns the list of symmetric input buses in the output. The third combinator is the polymorphic combinator *map*, that we presented in Section 6.3. In Level 1, the function *map* maps a safe function c_0, onto a list of symmetric input buses.

$$\begin{aligned} \mathit{Null} &\triangleq \lambda \mathit{sym}.\, [\,] \\ \mathit{Id} &\triangleq \lambda \mathit{sym} : (\mathit{bool\ list})\ \mathit{list}.\ \mathit{sym} \end{aligned}$$

The serial composition operator $\circ$, shown in Figure 6.3, is the polymorphic function $\circ$, defined at Level 0 in Section 6.3.3. It is used here to compose two circuits serially.

Parallel composition of two different symmetric circuit blocks is accomplished by a parallel composition operator ($\|$) shown below in Figure 6.3. One important constraint to keep in mind is that both these circuit blocks should generate output lists where each bus in the list is of equal length. Also the lengths of bus in one of the output lists, generated from circuit block c_1, should be equal to the length of the bus in the other output list, generated from the circuit c_2.

This is important to ensure since we do not want to cluster unequal buses together in one list. The reason is that we would like to be able to do a bitwise operation on this list, and to do this all the inputs have to be of equal length.

$$\begin{aligned} (c1\ \|\ c2) \triangleq\ & \lambda \mathit{sym}.\ \mathit{if}\ \ \mathit{CheckLength}(\mathit{append}\ (c1\ \mathit{sym})(c2\ \mathit{sym})) \\ & \qquad\qquad \mathit{then}\ \ \mathit{append}\ \ (c1\ \mathit{sym})(c2\ \mathit{sym})\ \ \mathit{else}\ \ [\,] \end{aligned}$$

Given a circuit c, if we want to duplicate it, we can use the *Fork* combinator, shown in Figure 6.3. The definition of this is presented below.

$$\mathit{Fork}\ \ c \triangleq \lambda \mathit{sym}.\ \ \mathit{append}\ (c\ \mathit{sym})\ (c\ \mathit{sym})$$

Amongst a list of n buses, we may wish to select any one of the n buses, and this is done by the *Select* combinator shown in Figure 6.4. Again note that it uses the definition of list selection function *el*.

$$\begin{aligned} \mathit{Select}\ n\ c \triangleq\ & \lambda \mathit{sym}.\ \mathit{if}\ (n\ <\ \mathit{length}(c\ \mathit{sym})) \\ & \qquad\qquad \mathit{then}\ \ [\mathit{el}\ n\ (c\ \mathit{sym})]\ \ \mathit{else}\ [\,] \end{aligned}$$

Given a list of buses, we wish to have an operator that can allow us to take all the buses but the first one. This is a feature that is similar to the *tl* function on lists. This is done by the *Tail* combinator shown in Figure 6.4. We define the combinator *Tail* in terms of *tl*.

$$\begin{aligned} \mathit{Tail}\ c \triangleq\ & \lambda \mathit{sym}.\ \mathit{if}\ (1\ <\ \mathit{length}(c\ \mathit{sym})) \\ & \qquad \mathit{then}\ \ \mathit{tl}\ (c\ \mathit{sym})\ \ \mathit{else}\ [\,] \end{aligned}$$

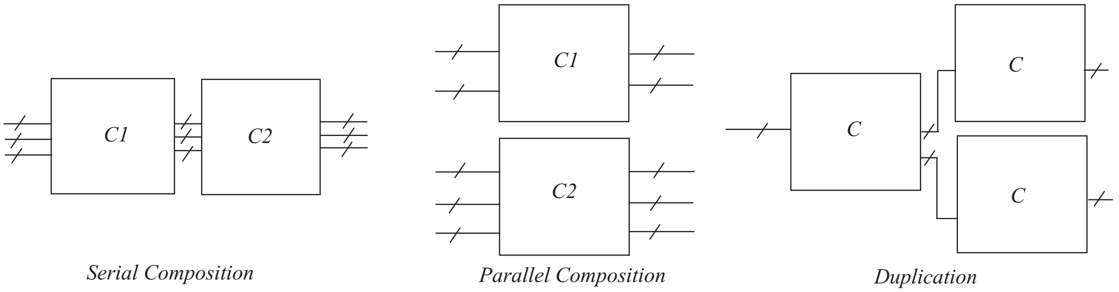


Figure 6.3: Circuit construction combinators showing serial composition ($\circ$), parallel composition ($\|$) and duplication (*Fork*).

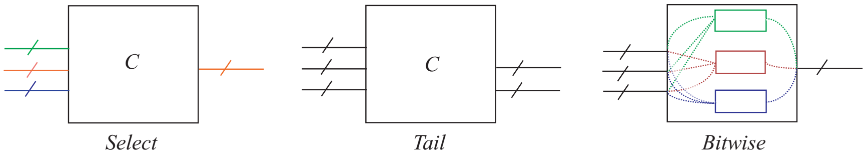


Figure 6.4: Circuit construction combinators showing the combinator that selects one of the several buses (*Select*), the one that obtains the tail (*Tail*), and the combinator that does the bitwise operation (*Bitwise*).

Our last combinator definition is also one of the most useful ones. This is the definition of the *Bitwise* operation (see Figure 6.4) on a list of buses, all of which are of equal length. The function *Bitwise*, applies the function f bitwise to all the buses, and it does this by using the bitwise function on lists, *map2*. *Bitwise* is defined by folding the combinator (*map2* f), onto a list of buses. The starting value for the *foldr* function comes from the head of the list of Boolean lists, and then *Bitwise* traverses the remainder of the list of buses, and using the function f, applies (*map2* f) bitwise to all the buses.

$$\begin{array}{lll} \mathit{Bitwise}\ f\ c & \triangleq & \lambda sym.\ \mathit{if}\ ((c\ sym)\ \neq\ [\,]) \\ & & \qquad\qquad \mathit{then}\ \ [\mathit{foldr}\ (\mathit{map2}\ \ f)(\mathit{hd}\ (c\ sym))(\mathit{tl}\ (c\ sym))] \\ & & \qquad\qquad \mathit{else}\ [\,] \end{array}$$

Once we define the combinators, we have to establish rules for combining them. In the next section we will present these rules. These lay the foundation of distinguishing symmetric circuits from non-symmetric ones. By using these rules to type check a given piece of circuit syntax, we can conclude that the circuit has symmetry. This is possible because we prove a type soundness theorem.

6.4.2 Typing rules for symmetric circuits

In this section we shall present a type system that provides the type judgement rules for building symmetric circuits. The type for symmetric circuits is given by

$$c : \mathit{bool\ list\ list} \rightarrow \mathit{bool\ list\ list}$$

The typing rules are defined by inductively defining the predicate *SS*, which represents the concept "symmetry-safe". Again we use the inductive rule definition package in HOL to define the rules. The outcome of such a definition in HOL is a set of rule definitions and a set of tactics that can be used to do proofs by rule induction. The rules are shown in Table 6.2.

$$\frac{}{\mathit{SS}\ \ \mathit{Null}} \qquad \frac{}{\mathit{SS}\ \ \mathit{Id}} \qquad \frac{\mathit{safe}\ \ c_0}{\mathit{SS}\ \ (\mathit{map}\ \ c_0)}$$

$$\frac{\mathit{SS}\ \ c_1 \qquad \mathit{SS}\ \ c_2}{\mathit{SS}\ \ (c_1 \circ\ c_2)} \qquad \frac{\mathit{SS}\ \ c_1 \qquad \mathit{SS}\ \ c_2}{\mathit{SS}\ \ (c_1 \parallel\ c_2)} \qquad \frac{\mathit{SS}\ \ c}{\mathit{SS}\ \ (\mathit{Fork}\ \ c)}$$

$$\frac{\mathit{SS}\ \ c \qquad n : \mathit{num}}{\mathit{SS}\ \ (\mathit{Select}\ \ n\ \ c)} \qquad \frac{\mathit{SS}\ \ c}{\mathit{SS}\ \ (\mathit{Tail}\ \ c)}$$

$$\frac{\mathit{SS}\ \ c \qquad \mathit{assoc}\ f \qquad \mathit{comm}\ f \qquad f : \mathit{bool}\ \rightarrow\ \mathit{bool}\ \rightarrow\ \mathit{bool}}{\mathit{SS}\ \ (\mathit{Bitwise}\ \ f\ \ c)}$$

Table 6.2: Type judgement rules for symmetric circuits.

Clearly *Null*, and *Id* are symmetry safe functions, so they are type safe. If we map (*map*) a safe functional block c_0, the result is a symmetry safe circuit. The serial ($\circ$) and parallel ($\|$) composition of two symmetry-safe circuits is a symmetry-safe circuit. Forking (*Fork*) is a symmetry-safe combinator. For a given natural number (denoted by type *num*), the circuit (*Select n c*) that selects an n^{th} bus from the output of a symmetry-safe function is symmetry-safe. The combinator *Tail*, is a symmetry-safe combinator, provided it is applied to a symmetry-safe circuit. The last rule is about the symmetry-safe property of the *Bitwise* combinator. We enforce the constraint that the function f of the type $\mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool}$ is associative and commutative.

6.4.3 Symmetry of circuits

In this section we will present the definition of symmetry. A circuit is symmetric if its behaviour remains unchanged under permutation of its symmetric input and output states.

$$\begin{array}{lll} \mathit{Sym}\ \ c & \triangleq & \forall inp.\ \ \mathit{CheckLength}\ inp\ \ \supset \\ & & \qquad\qquad \forall i\, j.\ \ map(swap(i,j))(c\ \ inp)\ =\ c\,(map(swap(i,j))\ \ inp) \end{array}$$

The definition shown above enforces the constraint that all buses in symmetric inputs are of equal length. This is to be consistent with the bitwise operations, which require equal lengths of inputs.

Now that we have shown the definition of symmetry, we would like to prove the type soundness theorem. Before we show the theorem and its proof, we need to take a look at two important properties that are needed in the proof of the type soundness theorem. One that symmetry-safe circuits ($\mathit{SS}\ \ c$) preserve the length invariant i.e., about the buses being of equal length. This is presented in the form of a theorem, and another lemma that states a relationship between folding a ($\mathit{map2}\ \ f$) operator and the swap function. This lemma is needed to prove the type soundness theorem in the case when the circuit is formed by a *Bitwise* combinator.

We first present the theorem about length invariance property of an input, when a symmetry safe function is applied to it.

Theorem 6.2. *Symmetry safe circuit blocks preserve length*

$\vdash \forall c.\ \mathit{SS}\ \ c\ \ \supset\ \ \forall inp.\ \ \mathit{CheckLength}\ inp\ \ \supset\ \ \mathit{CheckLength}(c\ inp)$

Proof outline. The proof starts by using the rule induction tactic on the predicate *SS*. Thus we have nine cases to prove.

1. $\forall inp.\ \mathit{CheckLength}\ inp\ \ \supset\ \ \mathit{CheckLength}(\mathit{Null}\ inp)$
   Follows easily from the definition of *CheckLength* and *Null*.
2. $\forall inp.\ \mathit{CheckLength}\ inp\ \ \supset\ \ \mathit{CheckLength}(\mathit{Id}\ inp)$
   Follows easily from the definition of *Id*.
3. Assuming that c_0 is safe, we have to prove that the following sub-goal:

   $\forall inp.\ \mathit{CheckLength}\ inp\ \ \supset\ \ \mathit{CheckLength}(map\ c_0\ inp)$

   But this is exactly the same as Lemma 6.27.
4. Given we have the following assumptions:

   $\forall inp.\ \mathit{CheckLength}\ inp\ \ \supset\ \ \mathit{CheckLength}(c_0\ inp)$
   $\forall inp.\ \mathit{CheckLength}\ inp\ \ \supset\ \ \mathit{CheckLength}(c_1\ inp)$

   we need to prove the following sub-goal:

   $\forall inp.\ \mathit{CheckLength}\ inp\ \ \supset\ \ \mathit{CheckLength}((c_0\ \circ\ c_1)\ inp)$

   Rewrite the sub-goal using the definition of serial composition $\circ$, and using the definition of *CheckLength* to rewrite the subsequent goal and assumptions.
5. Given we have the following assumptions:

   $\forall inp.\ \mathit{CheckLength}\ inp\ \ \supset\ \ \mathit{CheckLength}(c_0\ inp)$
   $\forall inp.\ \mathit{CheckLength}\ inp\ \ \supset\ \ \mathit{CheckLength}(c_1\ inp)$

   and we rewrite the goal with the definition of $\|$, we need to prove the following sub-goal:

$$\forall inp.\ CheckLength\ inp\ \supset\ CheckLength(if\ (CheckLength(append(c_0\ inp)(c_1\ inp)))\ then\ append\ (c_0\ inp)(c_1\ inp))\ else\ [\,])$$

By doing a case-split on the conditional *if–then–else*, and using our assumptions, we need to discharge the following two sub-goals:

(a) Under the assumption that the *if* part holds, we need to prove

$CheckLength(append(c_0\ inp)(c_1\ inp))$

which is the same as the assumption, so proved.

(b) When the *else* part of the conditional holds, then we need to prove

$CheckLength\ [\,]$

which is easily discharged by rewriting the sub-goal with the definition of *CheckLength*.

6. Rewriting with the definition of *Fork*, under the assumptions that $(CheckLength\ inp)$ holds and $(\forall inp.\ CheckLength\ inp\ \supset\ CheckLength(c\ inp))$ holds, we can easily prove from Lemma 6.22 that the following sub-goal holds:

   $CheckLength(append\ (c\ inp)(c\ inp))$

7. Rewriting with the definition of *Select*, we get the following two sub-goals:

   (a) Under the assumption that $\forall inp.\ CheckLength\ inp\ \supset\ CheckLength(c\ inp)$ holds, and $(n\ <\ (length(c\ inp)))$, it follows from the definition of *CheckLength* that the following sub-goal is true.

   $CheckLength\ [\,(el\,n(c\ inp))\,]$

   (b) The sub-goal $(CheckLength\ [\,])$ holds trivially from the definition of *CheckLength*.

8. Rewriting with the definition of *Tail*, we have the following two cases:

   (a) Under the following assumptions:

      1. $\forall inp.\ CheckLength\ inp\ \supset\ CheckLength(c\ inp)$
      2. $1\ <\ length(c\ inp)$
      3. $CheckLength\ inp$
      4. $CheckLength(c\ inp)$

   We have to show that $CheckLength(tl\ (c\ inp))$ holds. From $(1\ <\ length(c\ inp))$, it follows that $(0\ <\ length(c\ inp))$. From Lemma 6.9, and the assumptions, the goal trivially follows.

   (b) From the definition of *CheckLength*, it follows trivially.

9. Rewriting the goal with the definition of *Bitwise*, we get two sub-goals that we can discharge easily by using the definition of *CheckLength*.

□

Now we come to the lemma that is needed for the proof of the last case for the type soundness theorem. This lemma states that for all non-empty lists of buses, if all the buses are of equal length in the list, then the swap distributes across the folding of the $(map2\ f)$ operator over the list of buses. The proof of this theorem relies on proving another property, which we present in Lemma 6.31.

Lemma 6.30. *Swap distributes over the fold of* $(map2\ f)$

$$\begin{aligned}
\vdash\ & \forall inp.\ (inp \neq [\,])\ \supset \\
&\quad CheckLength\ inp\ \supset \\
&\quad\quad \forall f.\ assoc\ f\ \supset\ comm\ f\ \supset \\
&\quad\quad\quad \forall i\, j.\ (foldr\ (map2\ f)(hd\ (map(swap(i,j))\ inp))(tl(map(swap(i,j))\ inp)) \\
&\quad\quad\quad\quad = \\
&\quad\quad\quad\quad (swap(i,j)(foldr\ (map2\ f)(hd\ inp)(tl\ inp)))
\end{aligned}$$

Proof outline: We do an induction on *inp*. The base case is solved trivially by falsifying the assumption. The step case begins by rewriting the goal using the definitions of *foldr*, *map*, *map2*, *hd* and *tl*. This reduces the goal to following:

$$\begin{aligned}
&foldr\ (map2\ f)\ (swap(i,j)\ h)(map(swap(i,j))\ inp)\ = \\
&\qquad\qquad swap(i,j)\ (foldr\ (map2\ f)\ h\ inp)
\end{aligned}$$

and we have the following assumptions:

(a) $inp \neq [\,] \supset CheckLength\ inp \supset$
$\forall f.\ assoc\ f \supset comm\ f \supset$
$(foldr\ (map2\ f)\ (hd\ (map(swap(i,j))\ inp))$
$(tl\ (map(swap(i,j))\ inp)) =$
$swap(i,j)\ (foldr\ (map2\ f)\ (hd\ inp)\ (tl\ inp)))$

(b) $(h :: inp) \neq [\,]$

(c) $CheckLength(h :: inp)$

(d) $assoc\ f$

(e) $comm\ f$

Using Corollary 6.1, we can conclude *CheckLength inp*. Adding Lemma 6.31, we can add the following on the stack of assumptions:

$$\begin{aligned}
&\forall inp.\ CheckLength\ inp\ \supset \\
&\quad \forall h.\ (\forall m.\ m\ \in\ inp\ \supset\ (length\ m = length\ h))\ \supset \\
&\quad\quad \forall i\ j\ f.\ swap(i,j)\ (foldr\ (map2\ f)\ h\ inp) = \\
&\quad\quad\quad foldr\ (map2\ f)\ (swap(i,j)\ h)\ (map(swap(i,j))\ inp)
\end{aligned}$$

Resolving this assumption with others and from the definition of *CheckLength*, we can easily prove the goal. □

The following lemma states that when folding the $(map2\ f)$ operator onto a list of buses, and supplying an element h as the base value of the fold function, where h is the first element of the list, whose length being equal to the rest of the buses in the list, the order of applying the swap onto each bus in the list, will not matter. The proof of this theorem involves another property, that we present in Lemma 6.32.

Lemma 6.31. *Folding $(map2\ f)$ distributes over swap*

$$\begin{array}{l} \vdash\ \forall inp.\ CheckLength\ inp\ \supset \\ \qquad \forall h.\ (\forall m.\ m\ \in\ inp\ \supset\ (length\, m\ =\ length\, h))\ \supset \\ \qquad\qquad \forall f\, i\, j.\ (swap(i,j)\ (foldr\ (map2\ f)\ h\ inp) \\ \qquad\qquad\qquad = \\ \qquad\qquad\qquad foldr\ (map2\ f)\ (swap(i,j)\ h)(map(swap(i,j))\ inp)) \end{array}$$

Proof outline: We induct on *inp*. The base case is solved easily by rewriting with the definition of *map*, and *foldr*. The proof for the step case proceeds by rewriting with the definition of *foldr*, and *map*. Thus we have the following assumptions:

(a) $CheckLength\ inp\ \supset\ \forall h.\ (\forall m.\ m\ \in\ inp\ \supset\ (length\, m = length\, h))\ \supset$
$\qquad \forall f\ i\ j.\ swap(i,j)(foldr\ (map2\ f)\ h\ inp) =$
$\qquad\qquad (foldr\ (map2\ f)\ (swap(i,j)\ h)\ (map(swap(i,j))\ inp))$
(b) $CheckLength(h :: inp)$
(c) $\forall m.\ m\ \in\ (h :: inp)\ \supset\ (length\, m = length\, h')$

and we have to prove the following goal:

$$\begin{array}{l} swap(i,j)\ (map2\ f\ h\ (foldr\ (map2\ f)\ h'\ inp)) = \\ \qquad map2\ f\ (swap(i,j)\ h)\ (foldr\ (map2\ f)\ (swap(i,j)\ h')\ (map(swap(i,j))\ inp)) \end{array}$$

Unfolding the definition list membership $\in$, we get the following assumptions:

(a) $CheckLength\ inp\ \supset\ \forall h.\ (\forall m.\ m\ \in\ inp\ \supset\ (length\, m = length\, h))\ \supset$
$\qquad \forall f\ i\ j.\ swap(i,j)(foldr\ (map2\ f)\ h\ inp) =$
$\qquad\qquad (foldr\ (map2\ f)\ (swap(i,j)\ h)\ (map(swap(i,j))\ inp))$
(b) $CheckLength(h :: inp)$
(c) $\forall m.\ ((m\ = h)\ \vee\ (m\ \in\ inp))\ \supset\ (length\, m = length\, h')$

From (c) it follows that $(length\ h) = (length\ h')$. Modus Ponens with Lemma 6.32, leads us to have the following assumptions:

(a) $CheckLength\ inp\ \supset\ \forall h.\ (\forall m.\ m\ \in\ inp\ \supset\ (length\, m = length\, h))\ \supset$
$\qquad \forall f\ i\ j.\ swap(i,j)(foldr\ (map2\ f)\ h\ inp) =$
$\qquad\qquad (foldr\ (map2\ f)\ (swap(i,j)\ h)\ (map(swap(i,j))\ inp))$
(b) $CheckLength(h :: inp)$
(c) $(length\ h = length\ h')$
(d) $\forall f.\ (length\ h = length\ (foldr\ (map2\ f)\ h'\ inp))$

Using Lemma 6.11, we get the following assumptions:

(a) $CheckLength\ inp\ \supset\ \forall h.\ (\forall m.\ m\ \in\ inp\ \supset\ (length\, m = length\, h))\ \supset$
$\qquad \forall f\ i\ j.\ swap(i,j)(foldr\ (map2\ f)\ h\ inp) =$
$\qquad\qquad (foldr\ (map2\ f)\ (swap(i,j)\ h)\ (map(swap(i,j))\ inp))$
(b) $CheckLength(h :: inp)$
(c) $(length\ h = length\ h')$

(d) $\forall f.\ (\mathit{length}\ h = \mathit{length}\ (\mathit{foldr}\ (\mathit{map2}\ f)\ h'\ \mathit{inp}))$

(e) $\mathit{map2}\ f\ (\mathit{swap}(i,j)\ h)\ (\mathit{swap}(i,j)\ (\mathit{foldr}\ (\mathit{map2}\ f\ h'\ \mathit{inp})) =$
$\qquad \mathit{swap}(i,j)\ (\mathit{map2}\ f\ h\ (\mathit{foldr}\ (\mathit{map2}\ f)\ h'\ \mathit{inp}))$

The goal we need to prove is:

$$\mathit{swap}(i,j)\ (\mathit{map2}\ f\ h\ (\mathit{foldr}\ (\mathit{map2}\ f)\ h'\ \mathit{inp})) =$$
$$\mathit{map2}\ f\ (\mathit{swap}(i,j)\ h)\ (\mathit{foldr}\ (\mathit{map2}\ f)\ (\mathit{swap}(i,j)\ h')\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}))$$

Right hand side of assumption (e) is the same as the left hand side of the goal. After a rewriting step, and using extensional equality of functions, we get to prove:

$$\mathit{swap}(i,j)\ (\mathit{foldr}\ (\mathit{map2}\ f\ )\ h'\ \mathit{inp}) =$$
$$\mathit{foldr}(\mathit{map2}\ f)\ (\mathit{swap}(i,j)\ h')(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$$

which can be easily proved by using Corollary 6.1, and the assumptions. □

The following lemma states that folding the ($\mathit{map2}\ f$) operator onto a tail of a list of buses, with the base bus h', will result in a bus whose length is equal to the length of the first element of the list (h), of the buses, provided the length of bus h is equal to the length of the bus h'. As we have pointed out, this lemma is needed in the proof of the previous lemma, and therefore we have presented it here.

Lemma 6.32. *Folding* ($\mathit{map2}\ f$) *preserves the length of the buses*

$$\vdash\ \forall h\,\mathit{inp}.\ \mathit{CheckLength}(h :: \mathit{inp})\ \supset$$
$$\forall h'\,f.\ (\mathit{length}\ h\ =\ \mathit{length}\ h')\ \supset$$
$$(\mathit{length}\ h\ =\ \mathit{length}\ (\mathit{foldr}\ (\mathit{map2}\ f)\ h'\ \mathit{inp}))$$

Proof outline: The proof takes place by induction. The base case is solved trivially by rewriting the goal with the definition of *foldr*. The step case is solved by first stripping the quantifiers and resolving the assumptions with Lemma 6.2 and Lemma 6.4, and rewriting with the definition of *foldr*. Thus the goal would look like:

$$\mathit{length}\ h = \mathit{length}\ (\mathit{map2}\ f\ h\ (\mathit{foldr}\ (\mathit{map2}\ f)\ h''\ \mathit{inp}))$$

and the set of assumptions appear as:

(a) $\forall h.\ \mathit{CheckLength}(h :: \mathit{inp})\ \supset$
$\qquad \forall h'.\ (\mathit{length}\ h) = (\mathit{length}\ h')\ \supset$
$\qquad\qquad \forall f.\ \mathit{length}\ h = \mathit{length}\ (\mathit{foldr}\ (\mathit{map2}\ f)\ h'\ \mathit{inp})$

(b) $\mathit{CheckLength}(h' :: h :: \mathit{inp})$

(c) $\mathit{length}\ h' = \mathit{length}\ h''$

(d) $\mathit{length}\ h' = \mathit{length}\ h$

(e) $\mathit{CheckLength}(h' :: \mathit{inp})$

At this stage, the goal is easily proved by using Lemma 6.2. □

6.4.4 Type soundness theorem

Now we are ready to present the type soundness theorem and the proof. We used the inductive rule definition package in HOL 4 to prove this theorem, here we show the complete proof.

Theorem 6.3. *Symmetry safe circuits have symmetry*

$\vdash \forall c.\ \mathit{SS}\ \ c\ \ \supset\ \ \mathit{Sym}\ \ c$

Proof outline. The proof begins by invoking rule induction on the predicate *SS*. Thus we have nine cases to prove.

1. *Sym Null*
   Follows from the definitions of *Sym* and *Null*.
2. *Sym Id*
   Follows from the definitions of *Sym* and *Id*.
3. Assuming ($\mathit{safe}\ c_0$), we shall prove $\mathit{Sym}\ (\mathit{map}\ c_0)$. Rewriting with the definition of *Sym*, we need to prove the following:

   $$\forall \mathit{inp}.\ \mathit{CheckLength}\ \mathit{inp}\ \ \supset\ \ \forall i\, j.\ \mathit{map}(\mathit{swap}(i,j))\ (\mathit{map}\ c_0\ \mathit{inp})\ =\ \mathit{map}\ c_0\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$$

   From Lemma 6.18, we can conclude that the above holds, given we know from our assumption that ($\mathit{safe}\ c_0$) holds.
4. Assuming ($\mathit{SS}\ c_0$), ($\mathit{SS}\ c_1$), ($\mathit{Sym}\ c_0$) and ($\mathit{Sym}\ c_1$), we shall prove ($\mathit{Sym}(c_0\ \circ\ c_1)$). Rewriting the goal with the definition of *Sym*, and the serial composition $\circ$, we need to prove the following:

   $$\mathit{map}\ (\mathit{swap}(i,j))(c_1(c_0\ \mathit{inp}))\ =\ c_1(c_0\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})$$

   From Lemma 6.2 and our assumptions ($\mathit{SS}\ c_0$), and ($\mathit{SS}\ c_1$) we know that $\mathit{CheckLength}(c_0\ \mathit{inp})$ and $\mathit{CheckLength}(c_1\ \mathit{inp})$ hold. By unfolding the definition of $\mathit{Sym}\ c_0$ and $\mathit{Sym}\ c_1$ in the assumptions, and by subsequent rewriting, we can deduce our goal.
5. Assuming ($\mathit{SS}\ c_0$), ($\mathit{SS}\ c_1$), ($\mathit{Sym}\ c_0$) and ($\mathit{Sym}\ c_1$), we shall prove ($\mathit{Sym}(c_0\ \ \|\ \ c_1)$). By rewriting the goal with the definition of $\|$ and *Sym*, we get the following sub-goal, under the assumption that *CheckLength inp* holds for any arbitrary *inp*.

   $$\begin{aligned}&\mathit{map}(\mathit{swap}(i,j))(\mathit{if}\ \ \mathit{CheckLength}(\mathit{append}(c_0\ \mathit{inp})(c_1\ \mathit{inp}))\\ &\qquad\qquad \mathit{then}\ (\mathit{append}(c_0\ \mathit{inp})(c_1\ \mathit{inp}))\ \mathit{else}\ [\,])\\ &=\\ &(\mathit{if}\ \ \mathit{CheckLength}(\mathit{append}(c_0(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}))(c_1(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})))\\ &\qquad \mathit{then}\ (\mathit{append}(c_0\ (\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}))(c_1(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})))\\ &\qquad\qquad \mathit{else}\ [\,])\end{aligned}$$

   The proof now proceeds with a case-split on the conditional, and we get four sub-cases.

   (a) $\mathit{map}(\mathit{swap}(i,j))(\mathit{append}(c_0\ \mathit{inp})(c_1\ \mathit{inp}))\ =$
   $(\mathit{append}(c_0(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp}))(c_1(\mathit{map}(\mathit{swap}(i,j))\ \mathit{inp})))$

Unfolding the definition of $Sym\ c_0$, and $Sym\ c_1$ in the assumptions, and rewriting the sub-goal using Lemma 6.3, we can prove this case.

(b) $map(swap(i,j))(append\ (c_0\ inp)(c_1\ inp))\ =\ [\,]$
This case is discharged using Lemma 6.29, given that we have the following assumptions:

(i) $CheckLength(append\ (c_0\ inp)(c_1\ inp))$

(ii) $\sim CheckLength(append\ (c_0\ (map(swap(i,j))\ inp))(c_1\ (map(swap(i,j))\ inp)))$

(c) $map(swap(i,j))\ [\,]$
$= append(c_0(map(swap(i,j))\ inp))(c_1(map(swap(i,j))\ inp))$
The proof steps are similar to the above case.

(d) $map(swap(i,j))\ [\,]\ =\ [\,]$
By rewriting this goal with the definition of *map*.

6. Assuming ($SS\ c$) and ($Sym\ c$), we need to prove ($Fork\ c$). By rewriting the assumption and the goal, using the definition of *Sym*, and then rewriting the subsequent goal again with the definition of *Fork*, we can prove this case.

7. Assuming ($SS\ c$) and ($Sym\ c$), we need to prove ($Select\ n\ c$). Rewriting the assumption and the goal, using the definition of *Sym*, and then rewriting the subsequent goal again with the definition of *Select*, we get the following sub-goal to prove:

$$\begin{aligned} &map(swap(i,j))\ (if\ (n\ <\ length(c\ inp))\ then\ [\,el\ n\ (c\ inp)\,])\ else\ [\,] \\ &\quad = if\ (n\ <\ (length\ (c\ (map(swap(i,j))\ inp))) \\ &\qquad then\ [el\ n\ (c\ (swap(i,j))\ inp)]) \\ &\qquad else\ [\,] \end{aligned}$$

The proof now proceeds with a case-split on the conditional, $if-then-else$, and we get four sub-cases.

(a) $map(swap(i,j))\ [\,el\ n\ (c\ inp)\,]\ =\ [\,el\ n\ (c\ (map(swap(i,j))\ inp))]$
This case is discharged by using Lemma 6.14.

(b) $map(swap(i,j))\ [\,el\ n\ (c\ inp)\,]\ =\ [\,]$
This case is discharged by using Lemma 6.16, given that we have the following assumptions:

(i) $length\ (c\ inp)\ >\ n$

(ii) $\sim(length\ (c\ (map(swap(i,j))\ inp))\ >\ n)$

(c) $map(swap(i,j))\ [\,]\ =\ [\,el\ n\ (c\ (map(swap(i,j))\ inp))\,]$
The proof steps are similar to the above case.

(d) $map(swap(i,j))\ [\,]\ =\ [\,]$
The last sub-case follows easily by rewriting the goal with the definition of *map*.

8. Assuming ($SS\ c$) and ($Sym\ c$), we need to prove ($Tail\ c$). If we rewrite the goal with the definition of *Tail* and *Sym*, and rewrite the assumption with the definition of *Sym*, we get the following sub-goal:

$$\begin{aligned} &map(swap(i,j))\ (if\ 1\ <\ (length(c\ inp))\ then\ (tl(c\ inp))\ else\ [\,]) \\ &\qquad = \\ &if\ (1\ <\ length(c\ (map(swap(i,j))\ inp)))\ then\ (tl(c\ map(swap(i,j))\ inp))\ else\ [\,] \end{aligned}$$

The proof now proceeds with a case-split on the conditional, $if - then - else$, and we get four sub-cases.

(a) $map(swap(i,j))\ (tl\ (c\ inp)) \ = \ tl(c\ (map(swap(i,j))\ inp))$
The proof for this sub-goal takes place by rewriting the assumptions using the definition of *Sym* and using Lemma 6.13 to rewrite the goal.

(b) $map(swap(i,j))\ (tl(c\ inp)) \ = \ [\,]$
This is proved by using Lemma 6.16, given that we have the following assumptions:

(i) $length\ (c\ inp) \ > \ 1$

(ii) $\sim(length\ (c\ (map(swap(i,j))\ inp)) \ > 1$

(c) $map(swap(i,j))[\,] \ = \ tl(c\ (map(swap(i,j))\ inp))$
The proof steps are similar to the above case.

(d) $map(swap(i,j))\ [\,] \ = \ [\,]$
Rewriting with the definition of *map*.

9. Assuming $(assoc\ f)$, $(comm\ f)$, $(SS\ c)$ and $(Sym\ c)$, we shall prove $(Sym\ (Bitwise\ f\ c))$. The proof begins by rewriting the goal and the assumptions with the definition of *Sym*, and rewriting the goal with the definition of *Bitwise*. Thus we need to prove the following sub-goal:

$$\begin{array}{l}(map(swap(i,j))\ (if\ ((c\ inp) \ \neq \ [\,])\ then \\ \qquad\qquad [foldr\ (map2\ f)\ (hd(c\ inp))(tl(c\ inp))]\ else\ [\,])) \\ \qquad = \\ (if\ ((c\ (map(swap(i,j))\ inp)) \ \neq \ [\,])\ then \\ \qquad\qquad [\,foldr\ (map2\ f)(hd(c\ (map(swap(i,j))\ inp))) \\ \qquad\qquad\quad (tl(c\ (map(swap(i,j))\ inp)))\,]\ else\ [\,])\end{array}$$

The proof now proceeds with a case-split on the conditional, $if - then - else$, and we get four sub-cases.

(a) $map(swap(i,j))\ [\,foldr\ (map2\ f)\ (hd(c\ inp))\ (tl(c\ inp))\,]$
$=$
$[\,foldr\ (map2\ f)\ (hd(c\ (map(swap(i,j))\ inp)))\ (tl(c\ (map(swap(i,j))\ inp)))\,]$
This case is proven by first rewriting the sub-goal with *map*, and then using Lemma 6.30 and Lemma 6.2.

(b) $map(swap(i,j))\ [\,foldr\ (map2\ f)\ (hd(c\ inp))\ (tl(c\ inp))\,] \ = \ [\,]$
This case is proven by using Lemma 6.16, given that we have the following assumptions:

(i) $length\ (c\ inp) \ > \ 0$

(ii) $\sim(length\ (c\ (map(swap(i,j))\ inp)) \ > 0$

(c) $map(swap(i,j))\ [\ ] \ =$
$[\,foldr(map2\ f)\ (hd(c\ (map(swap(i,j))\ inp)))\ (tl(c\ (map(swap(i,j))\ inp)))\,]$
The proof steps are similar to the above case.

(d) $map(swap(i,j))\ [\,] \ = \ [\,]$
By rewriting with the definition of *map*.

□

Circuits take a non-symmetric input, a symmetric input and give an output. In the type system above, the symmetric circuit blocks had encapsulated the type of non-symmetric inputs (*bool list list*). We showed definitions of functions that let us construct larger symmetric blocks from smaller ones by using combinators judiciously. The type judgement rules together with the type soundness theorem guarantees that the resulting circuit blocks are symmetric. Now we will present what it means generally for a circuit to have symmetry. This notion is expressed by the predicate *Validate*.

Definition 6.17. *Validating circuits to be symmetric*

$$Validate\ (c : \mathit{bool\ list\ list} \rightarrow \mathit{bool\ list\ list} \rightarrow \mathit{bool\ list\ list}) \quad \triangleq \quad \forall nsym.\ SS\ (c\ \ nsym)$$

Lemma 6.33. *Type safe circuits have symmetry*

$$\vdash\ \ \forall c.\ Validate\ c\ \supset\ \forall nsym.\ Sym\ (c\ \ nsym)$$

Proof outline: The proof takes place by unfolding the definition of *Validate* and rewriting by using Theorem 6.3. □

In practice, for each new circuit definition c in question, we use tactics in HOL to prove that $SS\ \ c$ holds. The tactics are usually very simple, made up of the rule definitions and certain basic simplifications based on the circuit definition. The strength of this proof oriented approach to type checking is that starting from the most basic gates onwards, we can re-use the proof results bottom-up building a library of circuits that have been certified to be symmetric via type checking.

Another advantage of our type checking method is that for different sizes for of a given circuit (for example memories of different sizes), doing the type checking, requires constant amount of time, which is usually of the order of few seconds, and there are no constraints on space.

6.5 Adding time to the combinational circuits

The symmetry we are interested in capturing are the ones that arise due to special treatment of groups of wires, and the way they are combined with other functional blocks. In the previous sections, we showed different ways in which combinational blocks of circuits can be combined to generate symmetry. A key observation here is that symmetry is a structural property, it arises purely due to the structure of the circuit, and this precisely why we did not consider the delay elements so far in our presentation of the abstract data type. Delay elements such as latches and flip-flops do not contribute explicitly to the symmetry of the circuit. However they can be present in the circuit, and the overall circuit may or may not have symmetry depending on whether or not the circuit is type checked according to the typing rules we provided earlier. In other words, the temporal behaviour of the delay elements does not influence the way the delay elements are connected with other components to give rise to structural symmetry. This is why we chose to abstract the temporal aspect of the behaviour in our representation of delay elements.

We also do a structural abstraction of the delay elements. We consider each delay element to take an input, a clock and an output, and leave out the complementary output and the resets. Since the values that appear at the input, output and the clock are bit values, we use the type of Boolean to denote this. Thus each delay element has the following type:

$$bool \rightarrow bool \rightarrow bool$$

Delay elements that we have used in our experiments are the rising-edge latch (*RE*), and an active-high (*AH*) latch. The rising-edge latch samples the input value at the output at every rising edge of the clock, whereas the active-high latch also known as the level-triggered latch samples the input during the period that the clock is high.

$$RE\ (clk : bool) \ \triangleq\ \lambda inp : bool.\, inp$$

$$AH\ (clk : bool) \ \triangleq\ \lambda inp : bool.\, inp$$

The definition shown above simply returns the input value at the output, in the same way as a buffer does. However, the exact interpretation, about when these values appear is left for a later stage, and will depend on the context. If we wish to simulate the circuits containing delay elements, in HOL, then we can use an interpreter function written using functions in ML and HOL, that models the actual behaviour of the device. We shall explain about this in detail in a later section. If we want to simulate the circuits containing delay elements in Forte, then they have to be compiled to a flat FSM, and the interpretation of the delay elements takes place in that process. We shall explain more about this in Chapter 7.

This idea of using different interpretations, is similar to the notion of alternative base functions [80] and the non-standard interpretations based circuit analysis [102].

We have shown two different delay elements, so that when we compile the circuits in FSM* to FSMs in Forte, we can interpret these delay elements then, and illustrate the behavioural differences. We have modelled simple circuits such as a unit-delay multiplexer, and a comparator using a *RE* latch while we used the *AH* latch, to model SRAM and CAMs later in Chapter 9. If one desires to extend the library of delay elements, it can be done by adding more definitions, and suitably interpreting them in the two separate compilation phases later on.

6.6 Examples

In this section we will show examples of modelling circuits using the definitions of functions shown in Level 0 and Level 1.

Modelling basic circuits

Our first example shows how to model basic gates. The definitions of some simple gates are shown in Definition 6.19. The definition relies on using functional blocks, which are presented in Definition 6.18. The functional blocks use the definition of Boolean and, or and not, already defined in HOL.

Definition 6.18. *Functional blocks*

$$\begin{aligned}
inv &\triangleq map\ (\sim) \\
and &\triangleq fold\ (\wedge)\ id \\
or &\triangleq fold\ (\vee)\ id \\
nand &\triangleq inv \circ and
\end{aligned}$$

Definition 6.19. *Basic gates*

$$\begin{aligned}
Inv &\triangleq map\ inv \\
And &\triangleq map\ and \\
Or &\triangleq map\ or \\
Nand &\triangleq map\ nand
\end{aligned}$$

Our next example is the one showing Boolean bitwise operations.

Definition 6.20. *Bitwise operations*

$$\begin{aligned}
bAnd &\triangleq Bitwise\ (\wedge)\ Id \\
bOr &\triangleq Bitwise\ (\vee)\ Id
\end{aligned}$$

Now we present the definition of an n-bit comparator. There is nothing in the definition which restricts the size of the comparator. Figure 6.5 shows a 2-bit comparator with a unit-delay.

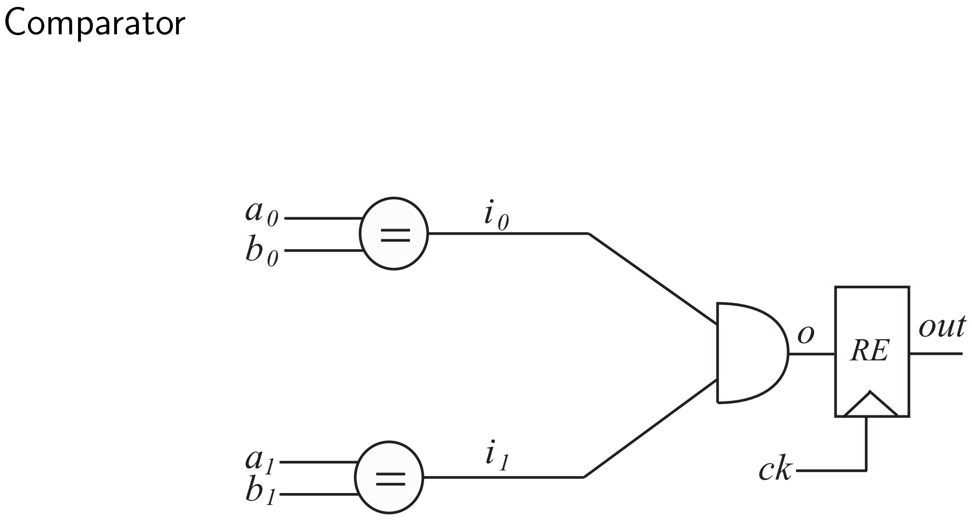


Figure 6.5: A unit-delay, 2-bit comparator.

Definition 6.21. *Unit-delay comparator*

$$\begin{aligned} \mathit{xnor}\ a\ b &= (a \ \wedge\ b) \ \vee\ (\sim a \wedge\ \sim b) \\ \mathit{Comp}\ \mathit{ck} &= \mathit{let}\ \mathit{comp} = \mathit{Bitwise}\ \ \mathit{xnor}\ \ \mathit{Id} \\ &\qquad \mathit{in} \\ &\qquad \mathit{map}(\mathit{map}(\mathit{RE}\ (\mathit{hd}\ (\mathit{hd}\ \mathit{ck})))) \circ \mathit{And} \circ \mathit{comp} \end{aligned}$$

When we get to evaluate the definition of the comparator, we can specify the buses as a list of Boolean lists. When we say evaluate here, we mean evaluating the function comparator to produce a flat term in HOL, with no structure any more, just a term with each node connected to another by Boolean connectors like ∧, ∨, ∼ and so on. Below we show a HOL session, where we specify the size of the comparator by specifying the two input buses (a and b, of size 2) and the clock. Note that clock (`ck`) is a non-symmetric input, and the buses `[a0;a1]` and `[b0;b1]` are symmetric inputs, clubbed together in a list. A HOL conversion `comp_conv` built from the combinator definitions and a list simplifier (`SIMP_CONV list_ss`), is used to evaluate the comparator definition. The output of the conversion is a theorem in HOL, the right-hand side of which is the intended structural flat definition of the comparator. Notice that the semantics of the delay elements have not been interpreted at all, they appear as a primitive in the output. In the next section, we shall provide one way to interpret the delay elements, in a manner that the exact behaviour of the circuit can be simulated at the HOL prompt.

Evaluating the structure

```
val comp_model_list =
[comparator_def, And_def, and_def, fold_def, xnor_def, toTime_def,
  map, Mem, Null_def, Id_def, o_DEF, Foldr, ||_def, id_def, Fork_def, el,
  Select_def, toTime_def, Tail_def, Bitwise_def, append, hd, tl];

val comp_conv = SIMP_CONV list_ss (comp_model_list);

`` comparator [[ck]] [[a0; a1];[b0; b1]] ``;
comp_conv it;

> val it =`` comparator [[ck]] [[a0; a1]; [b0; b1]] `` : term
- > val it =
    |- comparator [[ck]] [[a0; a1]; [b0; b1]] =
       [[RE ck ((b1 ∧ a1 ∨ ∼b1 ∧ ∼a1) ∧(b0 ∧ a0 ∨ ∼b0 ∧ ∼a0))]] : thm
```

Mux

The last example in this section is a unit-delay 2-to-1 multiplexer. Again a particular size is not specified in the definition; it is left until we are ready to evaluate the particular instance later. This example shows how more than one non-symmetric input is clubbed together in a single list of non-symmetric inputs. The definition is self explanatory. Figure 6.6 shows an n-bit multiplexer.

Definition 6.22. *Unit-delay multiplexer*

$$
\begin{array}{lcl}
\mathit{ctrl\_and}\ \mathit{inp} & = & \mathit{map}\ \ (\wedge\ (\mathit{hd}\ \mathit{inp})) \\
\mathit{not\_ctrl\_and}\ \mathit{inp} & = & \mathit{map}\ \ (\wedge\ ({\sim}(\mathit{hd}\ \mathit{inp}))) \\
 & & \\
M1\ \mathit{inp} & = & (\mathit{map}(\mathit{ctrl\_and}\ \mathit{inp}))\ \circ\ \mathit{Select}\ 0\ \mathit{Id} \\
M2\ \mathit{inp} & = & (\mathit{map}(\mathit{not\_ctrl\_and}\ \mathit{inp}))\ \circ\ \mathit{Select}\ 1\ \mathit{Id} \\
 & & \\
\mathit{Auxmux}\ \mathit{inp} & = & ((M1\ \mathit{inp})\ ||\ (M2\ \mathit{inp})) \\
 & & \\
\mathit{Mux}\ [\mathit{ck}; \mathit{ctrl}] & = & \mathit{map}(\mathit{map}(\mathit{RE}(\mathit{hd}\ \mathit{ck})))\ \circ\ \mathit{Bitwise}\ (\vee)\ (\mathit{Auxmux}\ \mathit{ctrl})
\end{array}
$$

Notice, that the effect of mapping the delay element `RE` onto the output of the mux circuitry, is to map an instance of `RE` onto each element of the output bus. This becomes visible in the HOL snippet of the Mux example shown next.

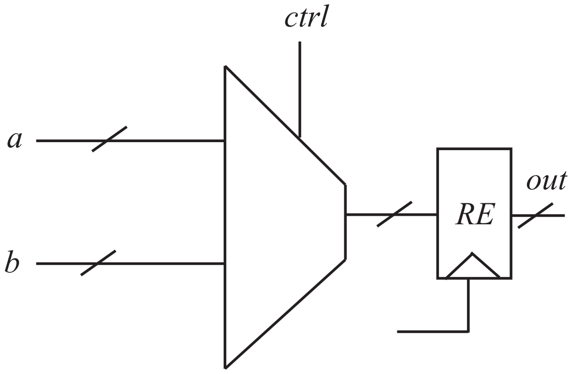


Figure 6.6: A unit-delay, 2-to-1 multiplexer.

Evaluating the structure

```
val mux_model_list = [Mux_def, Auxmux_def, M1_def, M2_def,
  ctrl_and_def, not_ctrl_and_def, toTime_def, map, Mem, Null_def,
  Id_def, o_DEF, Foldr, ||_def, id_def, Fork_def, el, Select_def,
  toTime_def, Tail_def, Bitwise_def, append, hd, tl];

val mux_conv = SIMP_CONV list_ss (mux_model_list);

val MUX_CONV = SIMP_CONV std_ss [DISJ_IMP_THM, FORALL_AND_THM,
  UNWIND_FORALL_THM1, length, hd, tl, Foldr, map2]);


`` Mux [[ck]; [ctrl]] [[a0; a1]; [b0; b1]] ``;
mux_conv it;
RIGHT_CONV_RULE(SIMP_CONV std_ss [ONE]) it;
RIGHT_CONV_RULE(mux_conv) it;
RIGHT_CONV_RULE(REWRITE_CONV [CheckLength_def, map, append]) it;
RIGHT_CONV_RULE(MUX_CONV) it;

> val it =
    ⊢ Mux [[ck]; [ctrl]] [[a0; a1]; [b0; b1]] =
        [[RE ck (~ctrl ∧ b0 ∨ ctrl ∧ a0);
          RE ck (~ctrl ∧ b1 ∨ ctrl ∧ a1)]] : thm
```

We use conversion (`mux_conv`) designed specifically around the definition of functions used in the Mux definition, and the combinators for circuit construction defined at Level 1 and Level 0. In the specific example of Mux we also have another conversion `MUX_CONV` that helps to simplify the terms with quantifiers over implications and connectors, by making use of in-built theorems `DISJ_IMP_THM` and `FORALL_AND_THM`, and `UNWIND_FORALL_THM1`.

In the next section, we show how to interpret the delay elements and evaluate the circuits for their behaviour in HOL.

6.7 Simulating models in HOL – interpreting time

So far we have shown how to model circuits using the circuit constructing combinators and functional blocks in FSM*, in a type safe manner ensuring that the resulting circuit model has symmetry. We also showed how we model delay elements, and combine them to produce circuits that have symmetry and delay as well.

In this section, we will present a method of interpreting these circuits in HOL, for evaluating their behaviour. This is done by using an ML function that does some preprocessing, followed by conversions in HOL to complete the simulation.

The task of the ML function is to take a circuit term written in the HOL logic (using the combinators of FSM*) and return another term in the HOL logic, which is the equivalent definition of circuit, but that can be now simulated over streams. Thus the

ML function looks at the right-hand side of the definition of the circuit, a HOL term with the HOL type *bool list list* → *bool list list* → *bool list list*, and returns another HOL term that is an interpretation of the original circuit definition, with the following type

$$(num \rightarrow (bool\ list)\ list) \rightarrow (num \rightarrow (bool\ list)\ list) \rightarrow (num \rightarrow (bool\ list)\ list)$$

The ML function works by systematically applying a HOL function *toTime*, to the right hand side of the circuit definition, and for a given time point t, delays the sequence of symmetric and non-symmetric inputs, if they are in the cone of a delay primitive. Thus the intended behaviour of a delay element is captured by delaying the sequences of inputs. If there are no delay elements, then the input sequences are not delayed. The result of applying *toTime* and delaying the sequence of inputs is another term in HOL that models the behaviour of the original circuit definition.

For a given sequence of symmetric inputs, non-symmetric inputs, and time, if the behaviour of the circuit needs to be determined, one can simply rewrite the circuit term with the definitions of *toTime*, definitions of delay primitives such as *RE*, and *AH*, and other circuit combinators. We do not show the implementation of the ML function here, since it is simply a term processing step which is done easily by using the built-in ML term construction and destruction functions.

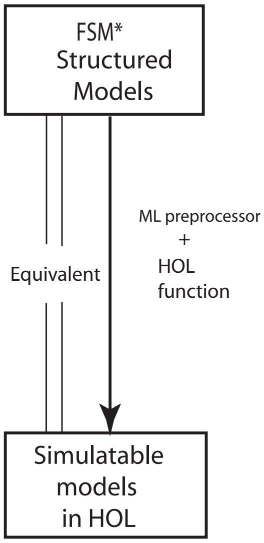


Figure 6.7: Interpreting structured models for simulation in HOL.

Now we show the definition of the function *toTime*.

Definition 6.23. *Interpreting circuits temporally*

$$toTime\ \ (c : bool\ list\ list \rightarrow\ bool\ list\ list)\ \ \sigma_{sym}\ \ (t : num)\ \ = c\ \ (\sigma_{sym}\ \ t)$$

One interesting property we want to establish is that all circuits in FSM* that have the symmetry expressed by *Sym*, their equivalent interpreted version on streams has symmetry as well. To define a mathematical definition of symmetry of circuits over streams, we need the concept of applying the permutation on a sequence of Boolean streams, and the notion of length invariance for the sequence. The latter is needed to establish that the length of the buses in symmetric inputs, stays equal for all time points. The function *apply* applies a permutation (swap) on the sequence of list of buses.

Definition 6.24. *Applying a swap on a sequence*

$$apply\ \ \pi\ \ \sigma\ \ t\ \ =\ \ map\ \ \pi\ \ (\sigma\ \ t)$$

The predicate *length_inv* states that for all time points the list of buses should have each bus of equal length.

Definition 6.25. *Length invariant of sequence*

$$length\_inv\ \ \sigma\ \ = \forall t.\ \ CheckLength(\sigma\ \ t)$$

Now we can present the definition of symmetry of circuits, interpreted over Boolean streams. The definition captures the notion of structural symmetry. The behaviour of the circuit stays constant, under permutations of input and output states, over all time points.

Definition 6.26. *Symmetry for timed versions of circuits*

$$\begin{aligned} &Sym_{\tau}\ \ (c : (num \rightarrow bool\ list\ list) \rightarrow (num \rightarrow bool\ list\ list))\ \ = \\ &\qquad \forall \sigma_{sym}.\ length\_inv\ \ \sigma_{sym}\ \ \supset \\ &\qquad\qquad \forall t\ i\ j.\ map(swap(i,j))(c\ \ \sigma_{sym}\ \ t) = c\ \ (apply(swap(i,j))\ \ \sigma_{sym})\ \ t \end{aligned}$$

The next lemma articulates the connection between symmetries in the two worlds, the world of FSM*, where the time is denoted by delay primitives, and the world where circuits are interpreted over streams. The merit of this lemma lies in showing that by interpreting the circuits over streams, using the function *toTime*, does not destroy the symmetry of the circuit. Thus the two circuit models with the different type are equivalent because they both have symmetry. This is shown in Figure 6.7.

Lemma 6.34. *Relation between Sym and* Sym_{τ}

$$\vdash\ \ \forall c.\ \ (Sym\ c)\ \ \supset\ \ Sym_{\tau}\ \ (toTime\ c)$$

Proof outline: Proof takes place by rewriting with the definitions of *Sym*, Sym_{τ}, *toTime*, *length_inv* and *apply*. □

Thus having established the relation between two different symmetries, we can state the following theorem, that says that all type safe circuits validated by the predicate *SS*, show symmetry when executed over a sequence of time.

Lemma 6.35. *Type safe circuits exhibit* Sym_{τ}

$$\vdash\ \ \forall c.\ \ (SS\ c)\ \ \supset\ \ Sym_{\tau}\ \ (toTime\ c)$$

Proof outline: Follows from Lemma 6.34. □

Examples

In this section we revisit the comparator and the multiplexer example, and show how the function *toTime* turns them to their versions over streams.

Comparator

We presented the example of a unit-delay in the last section. Now we show how to interpret it using the function *toTime*. The circuit definition presented in the earlier section is stripped, and the function *toTime* is applied consistently onto each sub-term on the right hand side of the definition. This is because a delay element in the form of *RE* was encountered in the definition. The first session shows the output of an ML interpreter, for the comparator. Notice, that the output of the circuit is delayed by one time point, asking the value of the output, by executing the circuit at the previous time point. Thus the effect of mapping a delay element `RE` is to delay the input sequence σ_{sym} by one time point (`t-1`).

```
val CompTimed =`` $\lambda\sigma_{nsym}\sigma_{sym}$ $t$.(toTime (map (map (RE (hd (hd (ck))))))
                                    $\circ$(toTime (And $\circ$ comp)) $\sigma_{sym}$ (t-1) `` ;
```

Now consider two streams of inputs, non-symmetric input `nsym`, given by a constant value of a symbolic value `ck`, and the symmetric input given by `sym` whose value changes over time. Thus the values returned by the symmetric input sequence at time points 0, 1, and 2 are all different.

```
val nsym =  `` $\lambda$t. (ck:bool list list) ``;

val sym =`` $\lambda$t. if (t=0) then ([[a0; a1];[b0; b1]])
                   else if (t=(SUC 0)) then [[T;T];[T;F]]
                            else ([[a0; a1];[c0; c1]]) `` ;
```

We can write the conversion `comp_conv` again using the list simplifier, and the definitions of combinators, but this time adding the definition of *toTime*.

```
val comp_model_list =
[comparator_def, And_def, and_def, fold_def, xnor_def, toTime_def,
  map, Mem, Null_def, Id_def, o_DEF, Foldr, ||_def, id_def, Fork_def, el,
  Select_def, toTime_def, Tail_def, Bitwise_def, append, hd, tl];

val comp_conv = SIMP_CONV list_ss (comp_model_list);
```

When executed over time points 0, 1 and 2, the trace of the circuit execution can be seen in the HOL session shown in Table A.1, A.2 and A.3, in the Appendix.

Mux

The multiplexer example is shown in the HOL session below. This example shows that both the non-symmetric and symmetric sequence of inputs is delayed by one time point. The reason is because the value at output is the value that gets computed by the state of the inputs at the previous time point. In the case of Mux, since control pin is another non-symmetric input besides the clock, its state has to be delayed by one time point.

```
val MuxTimed = `` λσ_nsym σ_sym t.(toTime(map(map(RE(hd ck)))) ∘
      toTime(Bitwise(∨)(Auxmux (el (SUC 0) (σ_nsym (t-1))))))σ_sym (t-1) `` ;
```

Conversions are written in the usual manner using list simplifiers, and using the conversions shown earlier for reducing the terms with implications, and quantification.

```
val mux_model_list = [Mux_def, Auxmux_def, M1_def, M2_def,
  ctrl_and_def, not_ctrl_and_def, toTime_def, map, Mem, Null_def,
  Id_def, o_DEF, Foldr, ||_def, id_def, Fork_def, el, Select_def, toTime_def,
  Tail_def, Bitwise_def, append, hd, tl];

val mux_conv = SIMP_CONV list_ss (mux_model_list);

val MUX_CONV = SIMP_CONV std_ss [DISJ_IMP_THM, FORALL_AND_THM,
  UNWIND_FORALL_THM1, length, hd, tl, Foldr, map2]);
```

Now consider the train of input sequences shown in the following HOL session.

```
val nsym = `` λt. if (t=0) then [ck;[T]] else [ck;[F]] `` ;

val sym = `` λt. [[a0; a1];[b0; b1]] `` ;
```

The non-symmetric input sequence changes with time, so the control will select the first bus at time point 1, but will select the second bus at time point 2. The execution trace in HOL for the two time instances is shown in Table A.4 and Table A.5, in the Appendix.

6.8 Related Work

The idea to use a type to capture symmetry was first given by Ip and Dill [40]. They implemented scalarsets in Murϕ, to capture symmetry, and verified cache-coherence protocols. Ken McMillan [64] also used scalarsets to verify several circuits, including verifying Tomasulo's out-of-order execution algorithm [87].

Our work on the design of the abstract data type has been inspired by the work done by Luk et al. [69] on regular circuit design. Luk et al. presented several higher-order combinators to model circuits in a functional language based on μFP [101]. Some of the combinators we have used in our work, like *map*, *map2*, and the serial ($\circ$) and parallel ($\|$) composition operators, are very similar to the combinators in [69]. Higher-order combinators appear regularly in many functional and hardware design languages such as Hydra [80, 81], Lava [22], and reFLect [75]. Our motivation was to design a subset of regular circuits, namely the ones that are symmetric, and therefore our set of circuit construction functions is small compared to the set of combinators proposed in [69].

In principle, an ADT can be implemented in any language, but there are features in some languages that provide a natural setting for their implementation. For example we believe that strongly typed, functional languages such as ML, Haskell, Lava and reFLect are all well suited for implementing the ADT. This is because commonly used data types such as lists are inductively defined in these languages, and common operations on these data types such as taking a head of a list, or tail of a list are trivial to implement. If we implement the ADT in any of the above languages, we do not have to reimplement many of the basic operations on list, since they already exist.

When we define the circuits using the ADT, we need to guarantee that the circuit will have the kind of structural symmetry we intend to discover. This cannot be taken for granted, and we are obligated to prove that this is indeed the case. This is why we chose a theorem prover so that proofs about the type safety theorem could be done with full rigour, and this was done within HOL by doing a shallow embedding of our theory.

In our work we took a functional approach to model circuits, given that more recent advances in HDL designs [22, 75, 80, 100] have advocated use of functional languages. Also, it is easier to simulate functions than relations, for rapid prototyping in the early stages of design cycle. The idea to keep structure and the run-time behaviour of circuit models, is inspired by the work done by John O'Donnell [81], and Satnam Singh [102]. Their idea is to use alternative functions for interpreting circuits as either netlists, or to evaluate them to check their behaviour.

6.9 Summary

In this chapter we presented our structured modelling language and showed how to model circuits starting from simple combinational gates to a unit-delay multiplexer and a comparator.

We presented a type of structured circuit models and presented a set of combinators that can be used for constructing structured symmetric circuits. Since symmetry detection relies on type checking we presented a novel set of type inference rules that can be used for type checking. We then formulated the definition of symmetry that we are interested in and proved a theorem that those circuit models that respect our type judgement rules have this symmetry. The theory presented in this chapter has been implemented in HOL 4.

We also showed how to interpret the run-time behaviour of the circuits expressed in the language of FSM*.

In the next chapter we will explain the process of compiling structured models to a flat netlist, that can be simulated in Forte for simulation, or compiled to a flat STE model that can be simulated in HOL.

Chapter 7

Compiling Structured Models to STE Models

In the previous chapter we have seen the modelling of circuits at a high level of abstraction that allows us to capture structural symmetries in circuits. Once it has been determined that the circuits have symmetry by way of type checking the circuit's syntax, we would like to obtain the netlists (FSM) from the structured descriptions, so that they can be simulated and verified using an STE simulator such as Forte. This is crucial for our research since without the FSMs we will not be able to illustrate the complete symmetry reduction approach. So the work presented in this chapter can be viewed as a bridge from "symmetry identification" (presented in the last chapter) to "using symmetry for reduction in STE model checking" that we will present in the next chapter.

Although not a part of our symmetry reduction methodology, we will show in this chapter a method of obtaining flat STE models in the HOL logic.

7.1 Deriving STE models from structured models

We mentioned in Chapter 3 that we wrote an STE simulator by a deep embedding of the STE theory in the logic of HOL [47]. This STE simulator is a good tool for understanding property verification over a three-valued logic of STE and how it is connected to the Boolean logic. STE circuit models and Boolean relational models have to be written in the logic of HOL, and then an STE simulator is invoked at the ML-HOL prompt. One has the choice of using the tool as a stand-alone simulator or use it to derive theorems in HOL about STE verification runs.

We used that simulator to test several small examples [47]. However, for reasonably big examples the simulator is very slow. This is because the execution of STE implementation algorithm, described in Chapter 3, relies on executing conversions in HOL, a way of execution by forward proofs, which though being transparently sound, is also very slow. Besides, the use of the HOL type `bool` to denote Boolean guards in STE, also slows down the execution speed [10].

For model checking large examples we wish to use Intel's symbolic simulator Forte. We mentioned in Chapter 5 that circuit models in Forte are represented as flat FSMs. An FSM is also known as an exe in Forte. More precisely an exe in Forte is essentially a collection of hash tables containing information about the circuit nodes and their dependencies. Forte builds a three-valued STE model on-the-fly from its FSM.

The FSMs are usually obtained from an exlif representation of a circuit model. Thus our target is to obtain an exlif from the structured descriptions of circuit models. Once we obtain an exlif, we can use Intel's *nexlif2exe* tool to get an exe of the circuit, that can used for STE simulation.

7.1.1 Deriving FSM from structured models

The figure below shows our framework, with two different ways to obtain a three-valued STE model. One leads us to obtain an exe and the other to obtain an STE model in the HOL logic. We can use Forte to simulate the exe, and use our simulator [47] to simulate the STE models in the HOL logic. We shall explain in detail both these approaches in the subsequent sections. Let us begin by explaining the path that enables us to extract exe from the structured models. This is a two step process. The first step consists of obtaining the netlist term in HOL, using the HOL function *ckt2netlist*, and the second step uses an ML function *netlist2exlif* to turn this netlist term to an exlif representation. Then using the *nexlif2exe* tool we obtain an exe. Given a structured circuit model defined in HOL with the type *bool list list* → *bool list list* → *bool list list*, our goal is to obtain a netlist term in HOL, which is a relation on input and output Boolean states. This is achieved by employing two state functions, the first (s_b) that assigns a Boolean value to each *input* node in the circuit, and the other (${s_b}'$) that assigns the Boolean value to *output* nodes. Node names are represented by the type *string*, thus the two state functions have the type: *string* → *bool*.

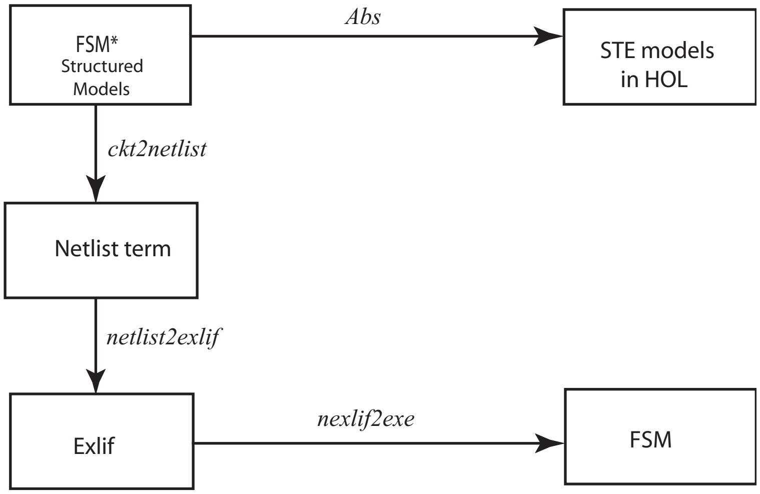


Figure 7.1: Compiling structured models to FSMs in Forte, and deriving flat STE models in HOL.

The definition of *ckt2netlist* is presented in Definition 7.1. This definition uses a function *flat*, which for a given structured circuit *c*, and non-symmetric and symmetric inputs, produces an output list of Boolean lists, of values that would be given by *c*. The function *ckt2netlist* needs to know the names of input and output nodes. Since the inputs are of non-symmetric and symmetric kind, they are specified separately as two different arguments to *ckt2netlist*. Each input can have more than one non-symmetric, symmetric or output buses. Names of nodes that belong to one bus are kept together in a list of strings, and several such buses are grouped together as a list of string lists (denoted by *nsym*, *sym* and *out* in the definition below). This is similar to the way we group Boolean input and output values at the time of designing structured circuits in the last chapter. We use the positional notation to group corresponding names and values at corresponding positions in each individual list of string lists and list of Boolean lists respectively. Thus the function *ckt2netlist* turns a circuit (a function in FSM*), into a relation and changes the positional notation into a named notation.

Definition 7.1. *From structured models to netlist terms in HOL*

$$flat\;\; c\; nsym\; sym\; (s_b : string \rightarrow bool) \;\; \triangleq \;\; c\; (map(map\; s_b)\; nsym)(map(map\; s_b)\; sym)$$

$$\begin{aligned} &ckt2netlist\;\; c\;\; nsym\;\; sym\;\; out\;\; (s_b : string \rightarrow bool)\;\; (s_b' : string \rightarrow bool) \\ &\quad \triangleq \;\; (map(map\; s_b')\; out) \;\; = \;\; flat\;\; c\; nsym\; sym\; s_b \end{aligned}$$

An example of how this definition is used in practice is shown below for a 2-to-1, 2-bit unit-delay Mux.

```
************** Obtaining a netlist from the structured model ***************


-`` ckt2netlist Mux
     [["ck"];["ctrl"]][["a0";"a1"];["b0";"b1"]][["out0";"out1"]]
s_b s_b'``


⊢ ckt2netlist Mux
     [["ck"];["ctrl"]] [["a0";"a1"];["b0";"b1"]] [["out0";"out1"]] s_b s_b'
=
(s_b' "out0"= RE(s_b "ck")(∼s_b "ctrl" ∧ s_b "b0'" ∨ s_b "ctrl" ∧ s_b "a0"))
     ∧
(s_b' "out1"= RE(s_b "ck")(∼s_b "ctrl" ∧ s_b "b1" ∨ s_b  "ctrl" ∧ s_b "a1"))

val mux_thm = it;
```

The next step in obtaining an exlif from the netlist term in HOL, relies on an ML program *netlist2exlif*, that we have written.

The program *netlist2exlif* takes the theorem obtained in the previous step, the name of a file where exlif has to be written, specified by a string, and whether or not the circuit has a unit-delay specified by another string. If there is a delay, it is specified by the string *"clock"*, else the string is empty, and the resulting exlif has no delay.

The following session shows how we run the *netlist2exlif* program for the unit-delay multiplexer circuit at the ML-HOL prompt.

```
************** Obtaining an exlif from a netlist  **********************

-  netlist2exlif Mux [mux_thm] "mux" "clock"
```

The result of the above is a mux.exlif file, that can be processed to an exe by running the *nexlif2exe* program at the Linux command shell, outside the ML-HOL framework.

```
************** Obtaining an exe from an exlif  **********************

[ashish@localhost ashish]  nexlif2exe mux.exlif
```

The session below shows the mux.exlif file.

```
.model mux
.inputs a0 a1 b0 b1
.outputs out0
.expr  n18 = ctrl `
.expr  n16 = a0 `
.expr  n15 = ctrl `
.expr  n13 = b0 `
.expr  n12 = n18 ’
.expr  n10 = n15 & n16
.expr  n9 = n12 & n13
.expr  n5 = n9 + n10
.expr  n4 = ck `
.latch n5 out0 re clock
.inputs a0 a1 b0 b1
.outputs out1
.expr  n42 = ctrl `
.expr  n40 = a1 `
.expr  n39 = ctrl `
.expr  n37 = b1 `
.expr  n36 = n42 `
.expr  n34 = n39 & n40
.expr  n33 = n36 & n37
.expr  n29 = n33 + n34
.expr  n28 = ck `
.latch n29 out1 re clock
.end
```

7.1.2 Deriving STE Models in HOL from structured models

The notion of a circuit model in the STE theory is the one of excitation function, as explained in Chapter 3. The excitation function is the flat STE model that gets constructed on-the-fly by the STE simulators such as Forte. In this section, we shall show how to derive such an excitation function in the HOL logic from the structured circuit models described in HOL.

This is accomplished by writing a function in HOL (*Abs*), that takes a circuit of the type *bool list list* → *bool list list* → *bool list list* and returns an excitation function that takes a lattice state and returns a lattice state at the next point in time. The function *Abs* needs to derive the excitation function that accurately reflects the behaviour of the structured circuit (c), which means that the next-state obtained by applying the derived excitation function on the initial state should be consistent with any Boolean state that the circuit can drive. The function *Abs* needs to know the names of all the input and output node names, the given Boolean state the circuit is in, and it will produce an STE model that will be consistent with the circuit definition and the initial Boolean state.

Before we formalise the definition of *Abs*, we will formalise the notion of dropping a Boolean state and dropping the Booleans.

Definition 7.2. *Drop Booleans*

$$\begin{array}{lcl} F\downarrow & = & 0 \\ T\downarrow & = & 1 \end{array}$$

The drop of the Boolean value T is the lattice value 1, and the drop of the Boolean false is 0. This is the same definition that appeared in [7, 47].

The function that drops a Boolean state to a lattice state is slightly more subtle than simply lifting the definition of drop on Boolean values. We present the definition of this function below.

Definition 7.3. *Dropping Boolean states to lattice states*

$$(drop_b\ [\,]\ [\,]\ (s : string \rightarrow bool \times bool)\ n\ \triangleq\ X)$$
$$(drop_b\ \ [\,]\ \ \_\ \ s\ \ n\ \ \triangleq\ \ X)$$
$$(drop_b\ \ \_\ \ [\,]\ \ s\ \ n\ \ \triangleq\ \ X)$$
$$\begin{array}{l} (drop_b\ \ ((a : string) :: alist)\ \ (b :: blist)\ \ s\ \ n\ \ \triangleq\ \ (if\ \ (n\ =\ a)\ \ then\ \ ((b\downarrow)\ \sqcup\ (s\ n)) \\ \hfill else\ \ drop_b\ \ alist\ \ blist\ \ s\ \ n)) \end{array}$$

The function $drop_b$ needs to know the state of the circuit, before it can drop the Boolean state to a lattice state. This is provided by passing the list of node names, and a list of Boolean values at each node, whereby a positional correspondence between names and the values is assumed. Third argument to the function $drop_b$ is a lattice state function, that describes the initial state of the circuit and is denoted by s. Finally the last argument denotes the name of a given node n.

This function $drop_b$, has to ensure that the drop of the Boolean value is consistent with the supplied lattice state. This means that given any Boolean state for a given c, the next-state obtained by the derived excitation function should be the least-upper-bound of the drop of the Boolean state, and the initial lattice state. If the initial lattice state and the drop of the Boolean state drives the circuit in an inconsistent state, then the resulting state should be a $\top$.

The function $drop_b$ is used to obtain the next-state lattice value for a given bus in the circuit. To obtain the next-state value for several buses in the circuit, we use the function *bool2ste*. The function definition is shown below and it takes a list of bus names (list of string lists) and a list of Boolean bus values (list of Boolean lists) as arguments, besides taking the lattice state function s and a node name. If the given node n is contained in the first bus, the corresponding Boolean value is obtained by using the function $drop_b$, else recursively this process is repeated on for all the buses in the list. If the given node n is not contained in any of the buses, then an X is returned. Intuitively it means that for all nodes that are not in the circuit, the next state is given by X, else it is the drop of the Boolean state corresponding to the node n.

Definition 7.4. *Obtaining the next-state lattice value*

$$\begin{array}{l}
(bool2ste\ \ [\,]\ \ [\,]\ \ s\ \ n\ \ \triangleq\ \ X)\\
(bool2ste\ \ [\,]\ \ \_\ \ s\ \ n\ \ \triangleq\ \ X)\\
(bool2ste\ \ \_\ \ [\,]\ \ s\ \ n\ \ \triangleq\ \ X)\\
(bool2ste\ \ (a :: alist)\ \ (b :: blist)\ \ s\ \ n\ \ \triangleq\ \ if\ \ (n\ \in\ a)\ \ then\ \ (drop_b\ \ a\ \ b\ \ s\ \ n)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad else\ \ (bool2ste\ \ alist\ \ blist\ \ s\ \ n)
\end{array}$$

The function *Mem*, determines whether or not a given node is a member of the list of buses. This is needed in the definition of *Abs*.

Definition 7.5. *Membership of an element in a list of lists*

$$\begin{array}{l}
Mem\ \ elem\ \ [x]\ \ =\ \ (elem\ \ \in\ \ x)\\
Mem\ \ elem\ \ (x :: xs)\ \ =\ \ (Mem\ \ elem\ \ x)\ \ \vee\ \ (Mem\ \ elem\ \ xs)
\end{array}$$

The function *Abs* is defined below. The next-state value is X for all input node names, and for the output nodes, the lattice value is the one given by *bool2ste*. The first two arguments of *Abs* are a pair of lists. The first argument denotes the state of the non-symmetric inputs, and the second argument represents the state of the symmetric inputs. The state in each argument is given by a pair, where the first component in each pair is the names of input buses (list of string lists), and the second component in each pair is the Boolean valued state at each node in the bus (list of Boolean lists). Again we assume the positional notation between names and the Boolean values. Output bus names are represented by out_{name}, and the initial lattice state by s, and n is used to represent any given node.

Definition 7.6. *Deriving STE models from* FSM^*

$$\begin{array}{l}
Abs\ \ c\ \ (nsym_{name}, nsym)\ \ (sym_{name}, sym)\ \ out_{name}\ \ \triangleq\ \ \lambda s : string \rightarrow bool \times bool.\ \lambda n.\\
\qquad\qquad\qquad\qquad\qquad if\ \ ((Mem\ n\ nsym_{name})\ \vee\ (Mem\ n\ sym_{name}))\\
\qquad\qquad\qquad\qquad\qquad then\ \ X\\
\qquad\qquad\qquad\qquad\qquad else\ \ (bool2ste\ \ out_{name}\ \ (c\ \ nsym\ \ sym)\ \ s\ \ n)
\end{array}$$

Below we show an example of a three-valued STE model for the 2-to-1, 2 bit Mux circuit. The function `Abs_MuxConv` is a HOL conversion that we wrote.

```
- `` Abs Mux ([["ck"];["c"]], [[ck];[c]])
([["a0";"a1"];["b0";"b1"]],[[a0;a1];[b0;b1]]) [["out0";"out1"]] `` ;

- rhs(concl (Abs_MuxConv it);
- > val it =
       (λs n.
          (if
             ((n = "clk") ∨ (n = "c")) ∨ ((n = "a0") ∨ (n = "a1")) ∨
             (n = "b0") ∨ (n = "b1")
           then
             X
           else
             (if (n = "out0") ∨ (n = "out1") then
                (if n = "out0" then
                   (if ~c ∧ b0 ∨ c ∧ a0 then One else Zero) lub (s "out0")
                 else
                   (if n = "out1" then
                      (if ~c ∧ b1 ∨ c ∧ a1 then One else Zero) lub
                      (s "out1")
                    else
                      X))
              else
                X))) : thm
```

Consider the following initial lattice state, that simply assigns the lattice value (drop of the Boolean value) to each input node including the clock (`ck`) and control (`c`) and an X to the output node.

```
val mux_lattice_st =`` λn.
(if (n="a0") then drop (a0)
    else if (n="a1") then drop (a1)
       else if (n="b0") then drop (b0)
                 else if (n="b1") then drop (b1)
                        else if (n="ck") then drop (ck)
                           else if (n="c") then drop (c)
                                       else X) `` ;
```

The result of the execution of `mux_lattice` on the initial lattice state `mux_lattice_st,`is an X for all input nodes. For the output nodes, the calculation in Appendix A-3.

7.2 STE models are monotonic

One very useful property the derived STE models have is that they are monotonic. The definition of monotonicity is the one presented in Definition 3.7.

Lemma 7.1. *Derived STE models are monotonic*

$$\vdash\ \forall c\, nsym_{name}\ \ sym_{name}\ \ out_{name}\ \ nsym\ \ sym.$$
$$Monotonic\ (Abs\ \ c\ \ (nsym_{name}, nsym)\ (sym_{name}, sym)\ out_{name})$$

Proof outline: The proof of this theorem relies on simple rewriting with the definition of *Monotonic*, *Abs*, *bool2ste*, and then using Lemma 7.3. (see below). □

Lemma 7.2. *Property about* $drop_b$

$$\vdash\ \forall s\, s'.\, (\forall n.\, (s\ n)\ \sqsubseteq\ (s'\ n)) \supset$$
$$\forall out_{name}\ v\ n.\, (drop_b\ \ out_{name}\ v\ s\ n)\ \sqsubseteq\ (drop_b\ out_{name}\ v\ s'\ n)$$

Proof outline: The proof takes place by induction on out_{name}. For the base case, given we know that $\forall n.\ s\ n\ \sqsubseteq\ s'\ n$, we have to prove:

$(drop_b\ [\,]\ v\ s\ n)\ \sqsubseteq\ (drop_b\ [\,]\ v\ s'\ n)$

This can be proven by inducting on v. The base case is proven by rewriting with the definition of $drop_b$, and noting that $X \sqsubseteq X$.

The step case is proven in a similar manner. Given that we know the following:

1. $\forall n.\ s\ n\ \sqsubseteq\ s'\ n$
2. $\forall v\ n.\ (drop_b\ out_{name}\ v\ s\ n)\ \sqsubseteq\ (drop_b\ out_{name}\ v\ s'\ n)$

we need to prove:

$$\forall h'\ v\ n.\ (drop_b\ (h' :: out_{name})\ v\ s\ n)\ \sqsubseteq\ (drop_b\ (h' :: out_{name})\ v\ s'\ n)$$

Again by induction on v, we get two cases. The base is solved by rewriting with the definition of $drop_b$, and noting that $X \sqsubseteq X$. The step case we have to solve after rewriting it with the definition of $drop_b$ is:

$$(drop_b\ h''\ \sqcup\ s\ h')\ \sqsubseteq\ (drop_b\ h''\ \sqcup\ s'\ h'))$$

We know from assumptions that $(s\ h'\ \ \sqsubseteq\ \ s'\ h')$, and since know from Lemma 3.1 that $\sqsubseteq$ is reflexive; we can use Lemma 3.6 to conclude the above goal. □

Lemma 7.3. *Property of bool2ste*

$$\vdash\ \ \forall s\, s'.\, (\forall n.\, (s\, n)\ \sqsubseteq\ (s\,'n)) \supset$$
$$\forall out_{name}\, v\, n.\, (bool2ste\ out_{name}\ v\ s\ n)\ \sqsubseteq\ (bool2ste\ out_n\ v\ s'\ n)$$

Proof outline: The proof takes place by induction on out_{name}. The base case is solved by inducting on v and solving the resulting sub-goals by rewriting with the definition of *bool2ste*, and noting that $X \sqsubseteq X$. For the step, given we know the following:

1. $\forall n.\ s\ n\ \sqsubseteq\ s'\ n$
2. $\forall v\ n.\ (bool2ste\ out_{name}\ v\ s\ n)\ \sqsubseteq\ (bool2ste\ out_{name}\ v\ s'\ n)$

we have to prove the following:

$$\forall h\ v\ n.\ (bool2ste\ (h :: out_{name})\ v\ s\ n)\ \sqsubseteq\ (bool2ste\ (h :: out_{name})\ v\ s'\ n)$$

This goal can be proved by inducting on v. The base is solved by rewriting with the definition of *bool2ste*, and noting that $X \sqsubseteq X$. The last case is solved by rewriting the goal with the definition of *bool2ste*, and then using the Lemma 7.2 and Lemma 3.6 to complete the proof. □

7.3 Symmetry of STE models

In Chapter 6, we presented a definition of structural symmetry. The concept of symmetry that we are interested in capturing is that the behaviour of the circuit does not change under permutations of its input and output states. The states could be Boolean or lattice valued. We want to relate symmetries of the STE model to the STE property reduction. Thus we need to define the symmetry of the STE model. Since the STE model is over the three-valued domain containing an X, we denote the symmetry of the STE models by Sym_χ. The definition of Sym_χ is shown below. It relies on a function $apply_s$. This function applies a swap on node name specified by π, onto the given state to give a new permuted state.

Just as the swap function on nodes can be composed serially to obtain arbitrary permutations, we can compose the function $(apply_s\ \pi)$, with different values of π to obtain a permutation on state. Thus $apply_s$ lifts the notion of a swap on node names to a swap on states, and it is then composed serially to obtain a single chosen permutation on states.

Definition 7.7. *Permutation on states*

$$apply_s\,\pi\,s \;\triangleq\; \lambda\, n.\; s(\pi\ n)$$

Definition 7.8. *Symmetry preserves excitation function*

$$Sym_\chi\ \mathcal{M}\,\pi \;\triangleq\; \forall s.\ apply_s\,\pi\,(\mathcal{M}\,s) \;=\; \mathcal{M}\,(apply_s\,\pi\,s)$$

We know that the STE model represented as FSM has symmetry since its corresponding high-level structured definition using the combinators has symmetry. In the next chapter, we will present the theory that enables us to connect the property verification to the fact that the STE model has symmetry. For the moment we shall move on and discuss the notion of equivalence between different kind of model representations that we have discussed so far.

7.4 Equivalence

In the previous chapter we explained the equivalence between the block labelled FSM* and the block labelled Simulatable models in HOL. This can also be seen in Figure 7.2.

Circuit models obtained by using the function *Abs* have symmetry whenever the corresponding structured circuit model has one. This is why we have shown in Figure 7.2, an equivalence between the block labelled FSM* and the block labelled STE models in HOL. We can observe this equivalence for each instance of the circuit model. We can conclude that the structured circuit model has symmetry by way of type checking, and using Lemma 6.33. Then we obtain the corresponding three-valued STE model in HOL by using the function *Abs*, and then prove for the specific instance that Sym_χ holds.

The STE model in HOL, obtained by using the function *Abs* is equivalent to the FSM representation in Forte. This means that if we evaluate the run-time behaviour of a given STE model in HOL and the run-time behaviour of the FSM for the same

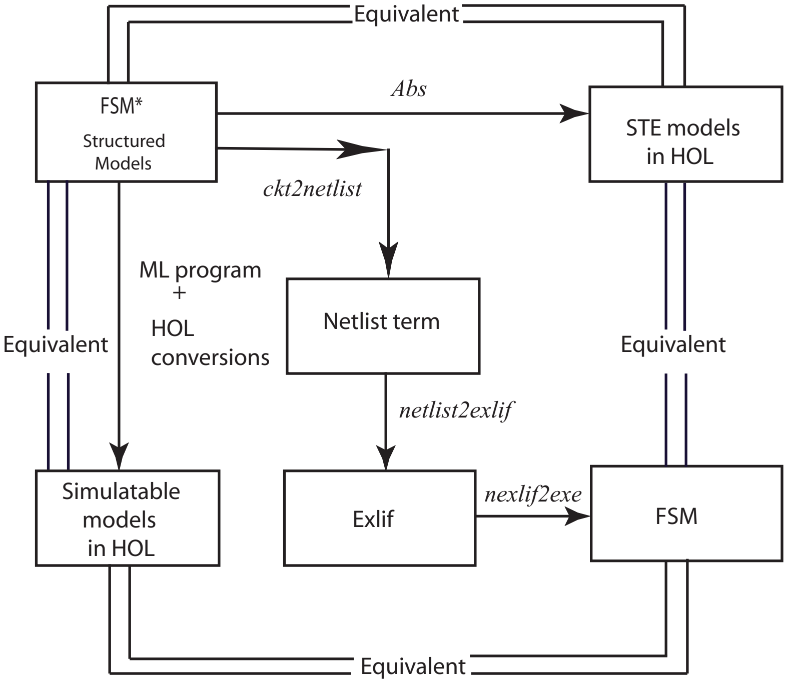


Figure 7.2: Overall framework of circuit modelling.

model, it will be identical on all sets of input and output lattice states. We depict this equivalence graphically in Figure 7.2.

We showed in the previous chapter, how to interpret the run time behaviour of the structured models over Boolean streams, by using an ML pre-processing step and the HOL conversions. The run-time interpretation of those models is equivalent to the run-time behaviour of the corresponding FSMs. If we evaluate the run-time behaviour of those models in HOL, and evaluate the corresponding FSMs in Forte, we will get identical behaviour for all values over the Boolean domain. This is shown in the Figure 7.2 by an equivalence between the block labelled Simulatable models in HOL and the block labelled FSM.

7.5 Summary

In this chapter we presented techniques to derive the STE model from the structured description. We showed two different approaches to doing this. One approach extracts the STE model in the HOL logic, so it can be simulated using the STE simulator in HOL, and the other approach derives the Forte's FSM, where a three-valued model is constructed on-the-fly by the STE simulator in Forte. This approach enables us to extract several reasonably large FSMs, thereby enabling STE simulation in Forte. We show that the extracted STE models are monotonic, and provide informal notions of equivalences of the different modelling formats for circuits.

Thus this chapter has provided both a theoretical and practical link between symmetry identification theory, that we presented in Chapter 6, and the theory of symmetry reduction that we will present in the next chapter.

Chapter 8

Reduction Methodology

This chapter examines the other side of our research, which is a strategy of reduction based on symmetry identification. Once we have diagnosed the presence of symmetries via type checking of circuit models, we need to carry out a process of property reduction and use arguments from the symmetry theory to justify a reduction of the verification problem. In this chapter, we present a novel set of inference rules that can be used for property decomposition in a tactical manner. We use symmetry based arguments to justify clustering of symmetric STE properties. Then we can pick one representative STE property from each equivalence class, verify that property by running an STE simulator, and conclude because of symmetry that all the other equivalent representatives have been verified as well. Once we have done this we can use the inference rules in the forward direction to deduce the overall statement of correctness of the original STE property. We show how this is done on the multiplexer and the comparator circuit, whose modelling we showed in the previous chapter.

8.1 Overview of reduction

In Figure 8.1, we show the big picture of the overall property reduction framework. The framework shows how by modelling circuits using a higher, more abstract and structured description (shown in Chapter 6) enables us to record that the equivalent FSM has symmetries (Chapter 7). This information is then used for performing a reduction based verification of STE properties. The reduction approach is centered around the use of a set of STE inference rules, and observing that symmetry in circuit models is mirrored by symmetry in STE properties. This gives us a sound basis for justifying that verifying reduced properties against the original circuit model (FSM) is sufficient to guarantee that the original STE properties have been verified against the FSM.

The question we ask in a typical property verification, is whether or not the FSM satisfies ($\models$) the STE property. Rather than trying to feed the STE property directly into an STE simulator to verify it, we decompose the property using the STE inference rules into smaller properties.

The reduced STE properties are then partitioned into different equivalence classes, and one representative from each equivalence class is fed into Forte for explicit STE simulations.

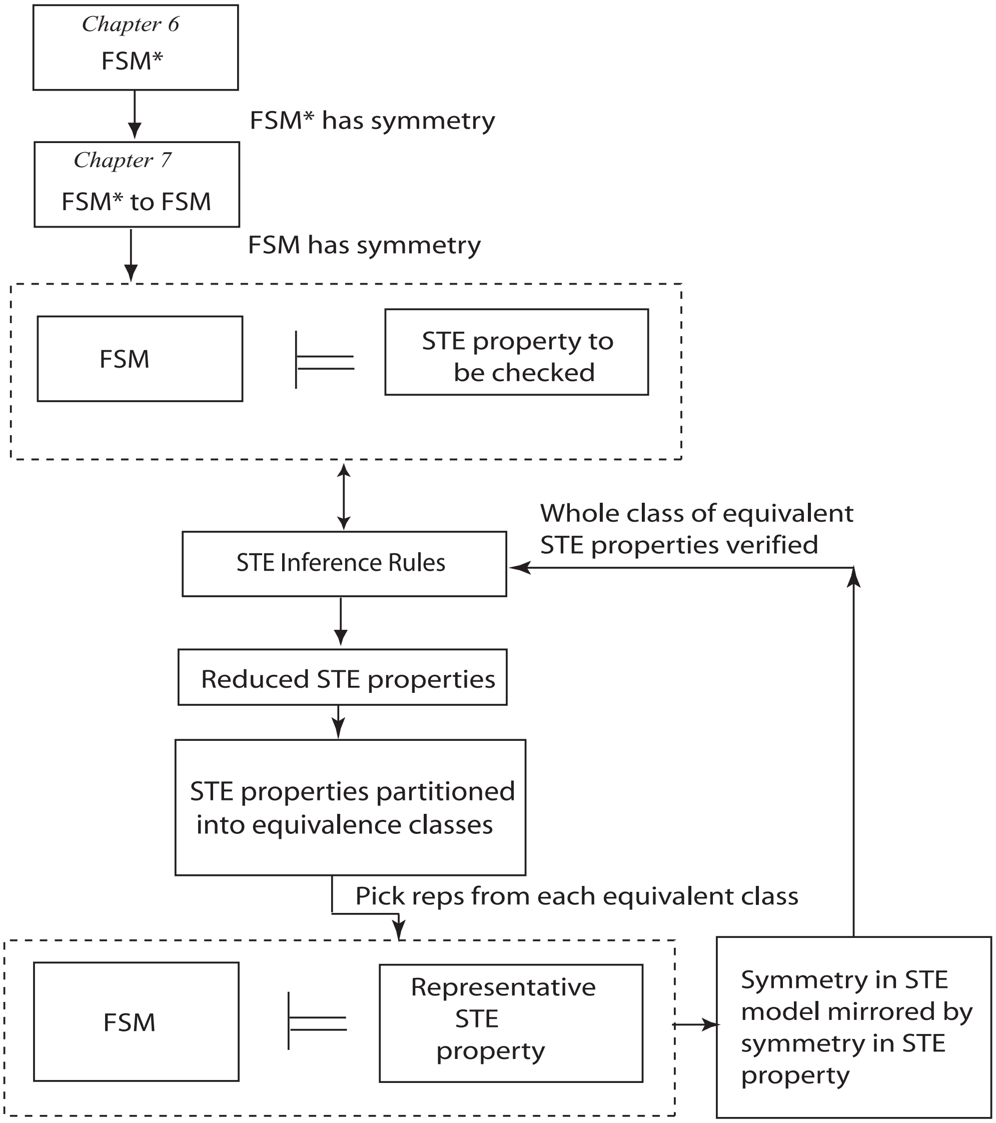

Chapter 6
FSM*
FSM* has symmetry
Chapter 7
FSM* to FSM
FSM has symmetry
FSM
STE property to be checked
Whole class of equivalent STE properties verified
STE Inference Rules
Reduced STE properties
STE properties partitioned into equivalence classes
Pick reps from each equivalent class
FSM
Representative STE property
Symmetry in STE model mirrored by symmetry in STE property

Figure 8.1: Reduction framework.

The partitioning of the properties is based on having identified the names of circuit nodes that belong to a bus in the symmetric inputs. Nodes that belong to the symmetric buses, and whose names appear in the decomposed STE properties, generate the notion of equivalence on the set of STE properties. Thus if we verify a decomposed STE property that talks about a node "a_0" and there are other smaller properties that talk about nodes "a_1", "a_2", and nodes "a_0", "a_1" and "a_2" form a symmetric bus then, having verified the property that talks about "a_0", means we have verified the corresponding properties that talk about "a_1" and "a_2". This is possible because we prove a soundness theorem that justifies the concept that symmetry in circuit models is mirrored by symmetry in STE properties.

Therefore implicitly, by using the soundness theorem, we prove the existence of the correctness of the whole class of equivalent STE properties. All we need to do now is to use the STE inference rules in the forward direction and compose the overall statement of correctness about the original STE property.

The notion of using STE inference rules [55, 56, 97] for property decomposition is not new, however they are not used that much in practice. We have increased the scope of their application significantly by inventing new STE inference rules. The new rules work in tandem with the known existing STE inference rules.

8.2 Key components

As already explained in the previous section, the two vital components for accomplishing property reduction are the existence of a soundness theorem, which enables the symmetry in circuit models to be used to partition the set of STE properties into equivalence classes, and the novel set of STE inference rules that allow very effective property decomposition and recomposition. In this section we will discuss the details of each of these components.

8.2.1 The soundness theorem

Before we present the soundness theorem we need to explain the notion of permuting lattice valued states, sequences and trajectory formulas. We begin by defining the permutation on a sequence of lattice values. We do this by providing a function $apply_\sigma$.

Definition 8.1. *Permutation on lattice sequences*

$$apply_\sigma\, \pi\, \sigma \triangleq \lambda\, t\, n.\ \sigma\ t\ (\pi\ n)$$

The function $apply_\sigma$ applies the swap function on node names specified by π, onto a given sequence σ and returns a new permuted sequence. This definition is similar to the definition of applying the swap on states, that we presented in the previous chapter.

Just as the swap function on nodes can be composed serially to obtain arbitrary permutations, we can compose the function ($apply_\sigma\ \pi$), with different values of π to obtain a permutation on the sequence. Thus $apply_\sigma$ lifts the notion of a swap on node names to a swap on sequences, and it is then used to get permutation on sequences.

One useful property of a swap function is that of self inverse. If a swap function is applied twice on a given element, it gives the same element back. This is formalised below, by the predicate *self_inverse*.

Definition 8.2. *Self inverse property*

$$self\_inverse\ \ \pi\ \ \triangleq\ \ \forall\, n.\ \pi(\pi\ n)\ =\ n$$

As explained in Chapter 3, fundamental to an STE implementation is the computation of the defining sequence and the defining trajectory, and checking if the information ordering on lattice values ($\sqsubseteq$) is preserved for all nodes in the consequent of the trajectory formula.

Thus we need to know what happens to the information ordering of two sequences if a permutation is applied on them. The answer to this is in Lemma 8.1, which says that the information ordering is preserved under permutation of sequences. As we shall see later, this lemma becomes very useful in the proof of the soundness theorem, when the sequences in question become instantiated to *defining trajectory* of a given STE formula.

Lemma 8.1. *Information ordering is preserved under permutation of sequences*

$$\vdash \;\; \forall \pi.\ \mathit{self\_inverse}\ \pi \;\supset\; \forall \sigma_1\ \sigma_2.\ (\sigma_1 \sqsubseteq_\sigma \sigma_2 \;\equiv\; (\mathit{apply}_\sigma\, \pi\, \sigma_1) \sqsubseteq_\sigma (\mathit{apply}_\sigma\, \pi\, \sigma_2))$$

The proof involves proving both the directions of the equivalence. We first show the *if* part.

For any arbitrary π, if π is a swap, then we show for any arbitrary σ_1 and σ_2 the following holds.

$$(\sigma_1 \sqsubseteq_\sigma \sigma_2) \supset (\mathit{apply}_\sigma\, \pi\, \sigma_1) \sqsubseteq_\sigma (\mathit{apply}_\sigma\, \pi\, \sigma_2)$$

From the definition of ordering on sequences we get

$$\forall t\, n.\, (\sigma_1\, t\, n \sqsubseteq \sigma_2\, t\, n)) \supset (\mathit{apply}_\sigma\, \pi\, \sigma_1\, t'\, m) \sqsubseteq (\mathit{apply}_\sigma\, \pi\, \sigma_2\, t'\, m)$$

By applying the definition of apply_σ we have

$$\forall t\, n.\, (\sigma_1\, t\, n \sqsubseteq \sigma_2\, t\, n)) \supset (\sigma_1\, t'\, (\pi\, m)) \sqsubseteq (\sigma_2\, t'\, (\pi\, m))$$

But since the left-hand side of the implication is true for all t and n, it is true for t' and $(\pi\, m)$, thus we have

$$(\sigma_1\, t'\, (\pi\, m) \sqsubseteq \sigma_2\, t'\, (\pi\, m)) \supset (\sigma_1\, t'\, (\pi\, m)) \sqsubseteq (\sigma_2\, t'\, (\pi\, m))$$

Hence the *if* part is proved.

We will now prove the other *only if* direction. We will show that for any arbitrary π, if π is a swap, then for any arbitrary σ_1 and σ_2 the following holds:

$$((\mathit{apply}_\sigma\, \pi\, \sigma_1) \sqsubseteq_\sigma (\mathit{apply}_\sigma\, \pi\, \sigma_2)) \supset (\sigma_1 \sqsubseteq_\sigma \sigma_2)$$

Using Definition 3.6, we get

$$(\forall t\, n.\, (\mathit{apply}_\sigma\, \pi\, \sigma_1\, t\, n) \sqsubseteq (\mathit{apply}_\sigma\, \pi\, \sigma_2\, t\, n)) \supset (\sigma_1\, t'\, m \sqsubseteq \sigma_2\, t'\, m)$$

From definition of apply_σ we get

$$(\forall t\, n.\, \sigma_1\, t\, (\pi\, n) \sqsubseteq \sigma_2\, t\, (\pi\, n)) \supset (\sigma_1\, t'\, m \sqsubseteq \sigma_2\, t'\, m)$$

But since the left-hand side of implication is true for all t and n, it is true for t' and $\pi\, m$. Since π is a swap we have the following from Definition 8.2

$$(\sigma_1\, t'\, m \sqsubseteq \sigma_2\, t'\, m) \supset (\sigma_1\, t'\, m \sqsubseteq \sigma_2\, t'\, m)$$

□

Now we will define an application of permutation on a trajectory formula. This is done by providing the function apply_f that applies a given swap on node names, specified by π, onto the structure of the trajectory formula. The function $(\mathit{apply}_f\ \pi)$ can then be composed for different values of π, to give permutations of trajectory formulas. The definition of apply_f is by recursion on the structure of formulas.

Definition 8.3. *Permutation on formulas*

$$\begin{array}{lcl}
apply_f\ \pi\ (n\ \mathsf{is}\ 0) & \triangleq & (\pi\ n)\ \mathsf{is}\ 0 \\
apply_f\ \pi\ (n\ \mathsf{is}\ 1) & \triangleq & (\pi\ n)\ \mathsf{is}\ 1 \\
apply_f\ \pi\ (f\ \mathsf{and}\ g) & \triangleq & (apply_f\ \pi\ f)\ \mathsf{and}\ (apply_f\ \pi\ g) \\
apply_f\ \pi\ (f\ \mathsf{when}\ P) & \triangleq & (apply_f\ \pi\ f)\ \mathsf{when}\ P \\
apply_f\ \pi\ (\mathsf{N}\ f) & \triangleq & \mathsf{N}\,(apply_f\ \pi\ f)
\end{array}$$

We defined the permutation on a lattice sequence earlier. Now we will present an equivalence between applying a permutation on the defining sequence of a formula and computing the defining sequence of the permutation of the formula. This is a vital instrument in establishing that permuting nodes in the formulas to generate new formulas will generate an equivalent defining sequence.

Lemma 8.2. *Permutation on the defining sequence and trajectory formulas*

$$\vdash\ \forall \pi.\ self\_inverse\ \pi \supset \forall f\ \phi\ t\ n.\ (apply_\sigma\,\pi\,[f]^\phi\,t\,n\ =\ [apply_f\,\pi\,f]^\phi\,t\,n)$$

Proof. The proof takes place by induction on the structure of f. We show the steps for the first base case.

$$\begin{array}{lcll}
apply_\sigma\,\pi\,[m\ \mathsf{is}\ 0]^\phi\,t\,n & \equiv & ([m\ \mathsf{is}\ 0]^\phi)\ t\ (\pi\,n) & \text{- definition of } apply_\sigma \\
 & \equiv & if\ (m = (\pi\ n) \wedge t\ =\ 0) & \text{- definition of defining} \\
 & & then\ \text{Zero}\ else\ X & \quad\text{sequence} \\
 & & & \\
 & \equiv & [apply_f\,\pi\,(m\ \mathsf{is}\ 0)]^\phi & \text{- } \pi \text{ is a swap} \\
 & & & \quad\text{using Definition 8.2} \\
 & & & \quad\text{and definitions of} \\
 & & & \quad\text{defining sequence} \\
 & & & \quad\text{and } apply_f
\end{array}$$

The proof steps for the other base case is very similar to the above case. The case for conjunction follows from inductive hypothesis and the fact that the swap operation π distributes over join ($\sqcup$). The guarded case is proved by doing a case split on the guard P. If P is F then we get an X and so we prove that part. When P is T, we prove that case by inductive hypothesis. The last case is proved by case splitting on t. If $t \neq 0$ then we get an X and we are done, else we discharge the goal by inductive hypothesis. □

We would like to prove a similar result about defining trajectory and trajectory formulas. Defining trajectories, are sequences of states that the circuit evolves into, given a stimuli and the behaviour of the circuit model. Thus the role of circuit model becomes significant here. In order to ensure that the effect of permutation on the defining trajectory is same as the effect of applying the permutation on the formula and then computing the defining trajectory, we have to ensure that the circuit model has symmetry. The definition of symmetry over lattice models is defined in the previous chapter, page 100.

Lemma 8.3. *Permutation on the defining trajectory and trajectory formulas*

$$\begin{aligned}\vdash\ & \forall \pi.\ \mathit{self\_inverse}\ \pi\ \supset \\ & \quad \forall \mathcal{M}.\ \mathit{Sym}_\chi\ \mathcal{M}\ \pi\ \supset \\ & \quad\quad \forall f\ \phi\ t\ n.\ (\mathit{apply}_\sigma\ \pi\ [\![f]\!]^\phi\ \mathcal{M}\ t\ n\ =\ [\![\mathit{apply}_f\ \pi\ f]\!]^\phi\ \mathcal{M}\ t\ n)\end{aligned}$$

Proof. By unfolding the definition of apply_σ we get

$$\mathit{apply}_\sigma\ \pi\ [\![f]\!]^\phi\ \mathcal{M}\ t\ n\ =\ [\![f]\!]^\phi\ \mathcal{M}\ t\ (\pi\, n)$$

So the goal we will prove is

$$\forall \pi\ f\ t\ n.\ \mathit{is\_swap}\ \pi\ \supset\ \mathit{Sym}_\chi\ \mathcal{M}\ \pi\ \supset\ ([\![f]\!]^\phi\ \mathcal{M}\ t\ (\pi\, n)\ =\ [\![\mathit{apply}_f\ \pi\ f]\!]^\phi\ \mathcal{M}\ t\ n)$$

The proof of this goal is done by induction on t.

Base case: $t\ =\ 0$

We will show that $[\![f]\!]^\phi\ \mathcal{M}\ 0\ (\pi\ n)\ =\ [f]^\phi\ 0\ (\pi\ n)$. From Lemma 8.2, we know that $[f]^\phi\ t\ (\pi\ n) = [\mathit{apply}_f\ \pi\ f]^\phi\ t\ n$ for all t and n, so it is true for $t = 0$.

Step case: Let us assume that $[\![f]\!]^\phi\ \mathcal{M}\ t\ (\pi\ n)\ =\ [\![\mathit{apply}_f\ \pi\ f]\!]^\phi\ \mathcal{M}\ t\ n$.

We will show that $[\![f]\!]^\phi\ \mathcal{M}\ (t+1)\ (\pi\ n)\ =\ [\![\mathit{apply}_f\ \pi\ f]\!]^\phi\ \mathcal{M}\ (t+1)\ n$.

Unfolding the definition of defining trajectory we get

$$[\![f]\!]^\phi\ \mathcal{M}\ (t+1)\ (\pi\ n)\ =\ [f]^\phi\ (t+1)\ (\pi\ n)\ \sqcup\ \mathcal{M}\ ([\![f]\!]^\phi\ \mathcal{M}\ t)\ (\pi\ n)$$

By using the definition of apply_s we can write the following

$$[\![f]\!]^\phi\ \mathcal{M}\ (t+1)\ (\pi\ n)\ =\ [f]^\phi\ (t+1)\ (\pi\ n)\ \sqcup\ \mathcal{M}\ (\mathit{apply}_s\ \pi\ ([\![f]\!]^\phi\ \mathcal{M}\ t)\ n)$$

But from assumptions we know that the $\mathcal{M}$ preserves symmetry, so we have

$$[\![f]\!]^\phi\ \mathcal{M}\ (t+1)\ (\pi\ n)\ =\ [f]^\phi\ (t+1)\ (\pi\ n)\ \sqcup\ (\mathit{apply}_s\ \pi\ (\mathcal{M}\ ([\![f]\!]^\phi\ \mathcal{M}\ t))\ n)$$

By applying the definition of apply_s we can write the following

$$[\![f]\!]^\phi\ \mathcal{M}\ (t+1)\ (\pi\ n)\ =\ [f]^\phi\ (t+1)\ (\pi\ n)\ \sqcup\ \mathcal{M}\ ([\![f]\!]^\phi\ \mathcal{M}\ t)\ (\pi\ n)$$

From induction hypothesis we get

$$\begin{aligned}[\![f]\!]^\phi\ \mathcal{M}\ (t+1)\ (\pi\ n)\ =\ & [\mathit{apply}_f\ \pi\ f]^\phi\ (t+1)\ n) \\ & \sqcup\ \mathcal{M}\ ([\![\mathit{apply}_f\ \pi\ f]\!]^\phi\ \mathcal{M}\ t\ n)\end{aligned}$$

Applying the definition of defining trajectory we finally get

$$[\![f]\!]^\phi\ \mathcal{M}\ (t+1)\ (\pi\ n)\ =\ [\![\mathit{apply}_f\ \pi\ f]\!]^\phi\ \mathcal{M}\ (t+1)\ n$$

□

Theorem 8.1. *Soundness Theorem*

$$\begin{aligned}\vdash\ & \forall \pi.\ \mathit{self\_inverse}\ \pi\ \supset \\ & \quad \forall \mathcal{M}.\ \mathit{Sym}_\chi\ \mathcal{M}\ \pi\ \supset \\ & \quad\quad \forall A\ C.\ (\mathcal{M}\ \models\ A\ \Rightarrow\ C\ \equiv\ \mathcal{M}\ \models\ (\mathit{apply}_f\ \pi\ A)\ \Rightarrow\ (\mathit{apply}_f\ \pi\ C))\end{aligned}$$

Proof. We will show that for any arbitrary model $\mathcal{M}$, π, A and C, assuming π is a swap operation that preserves the semantics of the circuit model ($\mathit{Sym}_\chi\ \mathcal{M}\ \pi$), then the following holds:

$$(\mathcal{M}\ \models\ A\ \Rightarrow\ C) \equiv\ (\mathcal{M}\ \models\ (\mathit{apply}_f\ \pi\ A) \Rightarrow (\mathit{apply}_f\ \pi\ C))$$

By definition of STE implementation we have

$$\mathcal{M}\ \models\ A\ \Rightarrow\ C \equiv\ [C]^{\phi}\ \sqsubseteq_{\sigma}\ [\![A]\!]^{\phi}\ \mathcal{M}$$

Since the swap operation π, preserves the $\sqsubseteq_{\sigma}$ relation from Lemma 8.1, we have

$$[C]^{\phi}\ \sqsubseteq_{\sigma}\ [\![A]\!]^{\phi}\ \mathcal{M}\ \equiv\ (\mathit{apply}_{\sigma}\ \pi\ [C]^{\phi})\ \sqsubseteq_{\sigma}\ (\mathit{apply}_{\sigma}\ \pi\ ([\![A]\!]^{\phi}\ \mathcal{M}))$$

But from Lemma 8.2 and Lemma 8.3 we have

$$[C]^{\phi}\ \sqsubseteq_{\sigma}\ [\![A]\!]^{\phi}\ \mathcal{M}\ \equiv\ ([\mathit{apply}_{f}\ \pi\ C]^{\phi})\ \sqsubseteq_{\sigma}\ ([\![\mathit{apply}_{f}\ \pi\ A]\!]^{\phi}\ \mathcal{M})$$

But from the definition of STE implementation we have

$$([\mathit{apply}_{f}\ \pi\ C]^{\phi})\ \sqsubseteq\ ([\![\mathit{apply}_{f}\ \pi\ A]\!]^{\phi}\ \mathcal{M}) \equiv \mathcal{M}\ \models\ (\mathit{apply}_{f}\ \pi\ A) \Rightarrow (\mathit{apply}_{f}\ \pi\ C)$$

Hence we have shown that if π is a swap and circuit model $\mathcal{M}$ is symmetric with respect to π then

$$(\mathcal{M}\ \models\ A\ \Rightarrow\ C)\ \equiv\ (\mathcal{M}\ \models\ (\mathit{apply}_{f}\ \pi\ A) \Rightarrow (\mathit{apply}_{f}\ \pi\ C))$$

□

Manish Pandey in his PhD dissertation [82] has also proved the above theorem. However one aspect which is not clear in his work is if the above theorem holds for data symmetries as well. These are symmetries that come into being when the circuit behaviour does not change under the data complement operation of its input and output states. The symmetry property, we have used in the proof of the soundness theorem is the structural symmetry in Pandey's thesis.

We have implemented the above theory of connecting STE theory to symmetries in HOL. All the proofs shown above have been done in HOL.

8.2.2 STE inference rules

In this section, we will present the STE inference rules and their proofs. Some of these rules require the circuit model to be monotonic. We showed in Chapter 7 that the derived STE models in HOL are monotonic. The STE models built on-the-fly from the FSMs in Forte are also monotonic. We used FSMs in Forte to carry out simulation runs, and we will use inference rules presented here to first decompose the STE properties and then re-use them in the reverse direction to re-compose smaller verification results.

Many of the STE inference rules presented here have appeared in past [55, 56, 97]. These are the rules on Reflexivity, Conjunction, Transitivity, Antecedent Strengthening 1 and Consequent Weakening 1. The first of these rules that we show next is Reflexivity.

Reflexivity

$$\frac{}{\mathcal{M} \models A \Rightarrow A}$$

Proof.

$$\begin{aligned}\mathcal{M} \models A \Rightarrow A\ &\triangleq\ \forall t\ n.\ [A]^{\phi}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n \quad \text{- Theorem 3.1}\\ &\equiv\ \forall t\ n.\ [A]^{\phi}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n \quad \text{- follows from Lemma 3.9}\end{aligned}$$

□

The rule shown next provides an effective property decomposition strategy. It enables a conjunction in the antecedent and the consequent of the STE property to be decomposed into individual conjuncts, which can be verified separately by running an STE simulator. An important side condition is that the circuit models must be monotonic.

Conjunction

$$\frac{\mathcal{M} \models A_1 \Rightarrow B_1 \qquad \mathcal{M} \models A_2 \Rightarrow B_2 \quad \mathit{Monotonic}\ \mathcal{M}}{\mathcal{M} \models (A_1 \textsf{ and } A_2) \Rightarrow (B_1 \textsf{ and } B_2)}$$

Proof.

1. $\mathcal{M} \models A_1 \Rightarrow B_1 \triangleq \forall t\, n.\ [B_1]^\phi\ t\ n \sqsubseteq [\![A_1]\!]^\phi\ \mathcal{M}\ t\ n$ - Theorem 3.1
2. $\mathcal{M} \models A_2 \Rightarrow B_2 \triangleq \forall t\, n.\ [B_2]^\phi\ t\ n \sqsubseteq [\![A_2]\!]^\phi\ \mathcal{M}\ t\ n$ - Theorem 3.1
3. $\forall t\, n. [\![A_1]\!]^\phi\ \mathcal{M}\ \ t\ n \sqsubseteq [\![A_1 \textsf{ and } A_2]\!]^\phi\ \mathcal{M}\ t\ n$ - Lemma 3.13
4. $\forall t\, n. [\![A_2]\!]^\phi\ \mathcal{M}\ \ t\ n \sqsubseteq [\![A_1 \textsf{ and } A_2]\!]^\phi\ \mathcal{M}\ t\ n$ - Lemma 3.13
5. $\forall t\, n. [\![B_1]\!]^\phi\ \mathcal{M}\ \ t\ n \sqsubseteq [\![A_1 \textsf{ and } A_2]\!]^\phi\ \mathcal{M}\ t\ n$ - Lemma 3.2, 1, 3
6. $\forall t\, n. [\![B_2]\!]^\phi\ \mathcal{M}\ \ t\ n \sqsubseteq [\![A_1 \textsf{ and } A_2]\!]^\phi\ \mathcal{M}\ t\ n$ - Lemma 3.2, 2, 4
7. $\forall t\, n. [\![B_1 \textsf{ and } B_2]\!]^\phi\ \mathcal{M}\ \ t\ n \sqsubseteq [\![A_1 \textsf{ and } A_2]\!]^\phi\ \mathcal{M}\ t\ n$ - 5, 6, Lemmas 3.3 and 3.6
8. $\mathcal{M} \models (A_1 \textsf{ and } A_2) \Rightarrow (B_1 \textsf{ and } B_2)$ - Theorem 3.1 and Lemma 3.9

□

Using the transitivity rule allows us to combine together separate correctness results. It is shown below, it requires the circuit models to be monotonic.

Transitivity

$$\frac{\mathcal{M} \models A \Rightarrow B \qquad \mathcal{M} \models B \Rightarrow C \quad \mathit{Monotonic}\ \mathcal{M}}{\mathcal{M} \models A \Rightarrow C}$$

Proof.

1. $\mathit{Monotonic}\ \ \mathcal{M}$ - assumption
2. $\mathcal{M} \models A \Rightarrow B$ - assumption
3. $\mathcal{M} \models B \Rightarrow C$ - assumption
4. $\forall t\, n.\ [B]^\phi\ t\ n \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n$ - Theorem 3.1
5. $\forall t\, n.\ [C]^\phi\ t\ n \sqsubseteq [\![B]\!]^\phi\ \mathcal{M}\ t\ n$ - Theorem 3.1
6. $(\forall t\, n.\ [B]^\phi\ t\ n \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n)$
   $\supset\ (\forall t\, n. [\![B]\!]^\phi\ \mathcal{M}\ t\ n \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n)$ - 1 and Lemma 3.10
7. $\forall t\, n. [\![B]\!]^\phi\ \mathcal{M}\ t\ n \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n$ - 4 and 6
8. $\forall t\, n. [\![C]\!]^\phi\ t\ n \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n$ - Lemma 3.2 on 5 and 7
9. $\mathcal{M} \models A \Rightarrow C$ - Theorem 3.1 and Lemma 3.9

□

The following rule allows one to deduce the existence of an STE property with a stronger antecedent, if we know that a property holds for an antecedent whose defining sequence is contained in the defining sequence of the stronger antecedent.

Antecedent Strengthening 1

$$\frac{\mathcal{M} \models A' \Rightarrow C \qquad [A']^{\phi} \sqsubseteq_{\sigma} [A]^{\phi} \quad \mathit{Monotonic}\ \mathcal{M}}{\mathcal{M} \models A \Rightarrow C}$$

Proof.

1. $\mathit{Monotonic}\ \ \mathcal{M}$ - assumption
2. $\mathcal{M} \models A' \Rightarrow C$ - assumption
3. $\forall t\, n.\ [C]^{\phi}\ t\ n\ \sqsubseteq\ [\![A']\!]^{\phi}\ \mathcal{M}\ t\ n$ - Theorem 3.1
4. $\forall t\, n.\ [A']^{\phi}\ t\ n\ \sqsubseteq\ [A]^{\phi}\ t\ n$ - assumption and Definition 3.6
5. $(\forall t\, n.\ [A']^{\phi}\ t\ n\ \sqsubseteq\ [A]^{\phi}\ t\ n)$
   $\supset\ (\forall t\, n. [\![A']\!]^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n)$ - 1, Lemma 3.15
6. $\forall t\, n. [\![A']\!]^{\phi}\ \mathcal{M}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - 4 and 5
7. $\forall t\, n.\ [C]^{\phi}\ t\ n\ \sqsubseteq [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - Lemma 3.2 on 3 and 6
8. $\mathcal{M} \models A \Rightarrow C$ - Theorem 3.1

□

Dual to the antecedent strengthening, the next rule is about weakening the consequent. If a certain STE property is satisfied by a circuit model, and there is a consequent whose defining sequence is contained in the defining sequence of the consequent of the property that is satisfied by the circuit model, then the property with the weaker consequent is also satisfiable by the circuit model.

Consequent Weakening 1

$$\frac{\mathcal{M} \models A \Rightarrow C' \qquad [C]^{\phi} \sqsubseteq [C']^{\phi}}{\mathcal{M} \models A \Rightarrow C}$$

Proof.

1. $\mathcal{M} \models A \Rightarrow C'$ - assumption
2. $\forall t\, n.\ [C']^{\phi}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - Theorem 3.1
3. $\forall t\, n.\ [C]^{\phi}\ t\ n\ \sqsubseteq\ [C']^{\phi}\ t\ n$ - assumption and Definition 3.6
4. $\forall t\, n.\ [C]^{\phi}\ t\ n\ \sqsubseteq\ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - Lemma 3.2 on 2 and 3
5. $\mathcal{M} \models A \Rightarrow C$ - Theorem 3.1

□

Now we shall present the new inference rules. We have extended the language of the existing STE inference rules by adding a layer of propositional logic on top. We have done this by pushing the Boolean guards out from the language of the STE property, and use them as assumptions and asking the question: Is the verification of the STE property (without guards) true under the assumption of the Boolean guard? Thus we have lifted the notion of the formula being true if the guard holds (as is the case with STE logic), to the meta-notion outside the STE logic — Is the STE property going to be true if the Boolean guard holds?

By having the Boolean guards out and treating them as primitive objects of the propositional logic, we can develop an inference scheme ensuring sound decomposition of these Boolean formulas. Of course the challenge is in addressing how the decomposition of these Boolean formulas relates to the decomposition of the STE properties. We show how this is done in the new rules. Most of the new inference rules are derived from the known rules, however the way the new rules work in practice, exposing symmetry in properties and doing reduction, is new.

The following two Constraint Implication rules make a connection between the language of new STE inference rules and the ones that existed in the past. The rule Constraint Implication 1 states that the Boolean guard in the consequent of the trajectory formula can be taken out and turned into an implication in propositional logic. The other direction is the one in which the Boolean assumption can be pushed into the syntax of the STE property. This is given by the rule Constraint Implication 2.

Constraint Implication 1

$$\frac{\mathcal{M} \models A \Rightarrow (C \ \mathsf{when} \ G)}{G \supset (\mathcal{M} \models A \Rightarrow C)}$$

Proof. The proof takes place by a case split on the Boolean guard G.

G is true.

1. $\mathcal{M} \models A \Rightarrow (C \ \mathsf{when} \ T)$ - assumption
2. $\forall t\, n.\ [C \ \mathsf{when} \ T]^{\phi} \sqsubseteq [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - Theorem 3.1
3. $\forall t\, n.\ [C]^{\phi}\ t\ n \sqsubseteq [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - Definition 3.9
4. $\mathcal{M} \models (A \Rightarrow C)$ - Theorem 3.1

G is false.
Holds immediately. □

Constraint Implication 2

$$\frac{G \supset (\mathcal{M} \models A \Rightarrow C)}{\mathcal{M} \models A \Rightarrow (C \ \mathsf{when} \ G)}$$

Proof. The proof takes place again by a case split on G.

G is true.

1. $(\mathcal{M} \models A \Rightarrow C)$ - assumption
2. $\forall t\ n.\ [C \ \mathsf{when}\ T]^{\phi} \sqsubseteq [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - Theorem 3.1
3. $\forall t\ n.\ [C]^{\phi} \sqsubseteq [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - Definition 3.9
4. $\mathcal{M} \models (A \Rightarrow C)$ - Theorem 3.1

G is false.

1. $\mathcal{M} \models A \Rightarrow (C\ \mathsf{when}\ F)$ - to prove
2. $X \sqsubseteq [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - Theorem 3.1 and Definition 3.9
3. $X \sqsubseteq [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$ - $\forall a.X \sqsubseteq a$

□

Thus we are now able to have an extension of the antecedent strengthening and consequent weakening rules. These rules indicate that a certain STE property can be satisfied under a Boolean assumption G.

Antecedent Strengthening 2

$$\frac{G \supset (\mathcal{M} \models A' \Rightarrow C) \qquad [A']^{\phi} \sqsubseteq_{\sigma} [A]^{\phi} \quad \mathit{Monotonic}\ \mathcal{M}}{G \supset (\mathcal{M} \models A \Rightarrow C)}$$

Proof.

1. $G \supset (\mathcal{M} \models A' \Rightarrow C)$ - assumption
2. $[A']^{\phi} \sqsubseteq_{\sigma} [A]^{\phi}$ - assumption
3. G - assumption
4. $\mathcal{M} \models A' \Rightarrow C$ - 1 and 3
5. $\mathcal{M} \models A \Rightarrow C$ - using Antecedent Strengthening 1, 2 and 4

□

Consequent Weakening 2

$$\frac{G \supset (\mathcal{M} \models A \Rightarrow C') \qquad [C]^{\phi} \sqsubseteq_{\sigma} [C']^{\phi}}{G \supset (\mathcal{M} \models A \Rightarrow C)}$$

Proof.

1. $G \supset (\mathcal{M} \models A \Rightarrow C')$ - assumption
2. $[C]^{\phi} \sqsubseteq_{\sigma} [C']^{\phi}$ - assumption
3. G - assumption
4. $\mathcal{M} \models A \Rightarrow C'$ - 1 and 3.
5. $\mathcal{M} \models A \Rightarrow C$ - using Consequent Weakening 1, 2 and 4

□

The following rule allows a property with a conjunctive Boolean assumption to be decomposed into smaller properties with the smaller individual conjuncts. This rule called Guard Conjunction is very useful for proving STE properties with a Boolean guard that is a conjunction. The guards can be taken out of the STE formula by using the Constraint Implication 1 rule, and then split into smaller properties, each property being smaller in terms of having fewer guards and fewer nodes in the antecedent and the consequent. As with the Conjunction rule, this rule requires the circuit models to be monotonic.

Guard Conjunction

$$\frac{G_1 \supset (\mathcal{M} \models A \Rightarrow C) \qquad G_2 \supset (\mathcal{M} \models B \Rightarrow D) \quad \mathit{Monotonic}\ \ \mathcal{M}}{(G_1 \wedge G_2)\ \supset\ (\mathcal{M} \models (A\ \textsf{and}\ B) \Rightarrow (C\ \textsf{and}\ D))}$$

Proof.

1. $G_1\ \supset\ (\mathcal{M}\ \models\ (A\ \Rightarrow\ C))$ - assumption
2. $G_2\ \supset\ (\mathcal{M}\ \models\ (B\ \Rightarrow\ D))$ - assumption
3. G_1 - assumption
4. G_2 - assumption
5. $\mathcal{M}\ \models\ (A\ \Rightarrow\ C)$ - 1 and 3
6. $\mathcal{M}\ \models\ (B\ \Rightarrow\ D)$ - 2 and 4
7. $\mathcal{M}\ \models\ ((A\ \textsf{and}\ B)\ \Rightarrow (C\ \textsf{and}\ D))$ - using the Conjunction Rule
8. $(G_1\ \wedge\ G_2)\ \supset\ (\mathcal{M}\ \models\ (A\ \textsf{and}\ B)\ \Rightarrow (C\ \textsf{and}\ D))$ - from 3, 4, and 7

□

The next rule is the Guard Disjunction rule. What this captures is that if we can verify two STE properties against a given circuit model, with separate antecedents and separate Boolean assumptions, but identical consequent, then we can do a disjunction of the Boolean assumptions and the conjunction of the antecedents of the two STE properties to deduce that the resulting STE property is satisfied by the circuit model. This rule becomes very useful in decomposing properties that have a Boolean disjunction in the guard of the STE formula. These properties are some of the more difficult properties to verify, because of the disjunction in the Boolean guard.

Guard Disjunction

$$\frac{G_1 \supset (\mathcal{M} \models A \Rightarrow C) \qquad G_2 \supset (\mathcal{M} \models B \Rightarrow C) \quad \mathit{Monotonic}\ \ \mathcal{M}}{G_1 \vee G_2 \supset (\mathcal{M} \models (A\ \textsf{and}\ B) \Rightarrow C)}$$

Proof. Case 1
Assuming G_1 holds, we have to prove $(\mathcal{M} \models (A\ \textsf{and}\ B)\ \Rightarrow C)$ by assuming $(G_1\ \supset\ (\mathcal{M} \models A \Rightarrow C))$ and assuming $(G_2\ \supset\ (\mathcal{M} \models B \Rightarrow C))$

1. $G_1\ \supset\ (\mathcal{M} \models A \Rightarrow C)$ - assumption
2. $G_2\ \supset\ (\mathcal{M} \models B \Rightarrow C)$ - assumption
3. G_1 - assumption
4. $\mathcal{M} \models A \Rightarrow C$ - 1 and 3
5. $(\mathcal{M} \models (A\ \textsf{and}\ B)\ \Rightarrow C)$ - Using Lemma 3.13

Case 2
Assuming G_2 holds, we have to prove $(\mathcal{M} \models (A \;\mathsf{and}\; B) \;\Rightarrow C)$ by assuming $(G_1 \;\supset\; (\mathcal{M} \models A \Rightarrow C))$ and assuming $(G_2 \;\supset\; (\mathcal{M} \models B \Rightarrow C))$

1. $G_1 \;\supset\; (\mathcal{M} \models A \Rightarrow C)$ - assumption
2. $G_2 \;\supset\; (\mathcal{M} \models B \Rightarrow C)$ - assumption
3. G_2 - assumption
4. $\mathcal{M} \models B \Rightarrow C$ - 2 and 3
5. $(\mathcal{M} \models (A \;\mathsf{and}\; B) \;\Rightarrow C)$ - Using Lemma 3.13

□

The last rule we present here is known as Cut. This is another of the most useful rules for property decomposition. The reason is that, in practice, the task of verifying an STE property consisting of more than one antecedent (A_1 and A_2) and a conjunctive Boolean guard ($G_1 \wedge G_2$) can be done by decomposing the task into two smaller runs. One of the runs takes place with one of the antecedents (A_1) and a *weaker* consequent (B_1), using a part of the Boolean guard (G_1) in the assumption. Weaker formula means one with fewer symbolic variables or nodes. Then the weaker consequent is used together with the other half of the antecedent (A_2), to verify the property with the consequent C. Typically, Constraint Implication 1 rule is used first to transform the property with the conjunctive Boolean guard into a property where the Boolean conjunction appears as an implication.

Cut

$$\frac{G_1 \;\supset\; (\mathcal{M} \models A_1 \Rightarrow B_1) \qquad G_2 \;\supset\; (\mathcal{M} \models (B_1 \;\mathsf{and}\; A_2) \Rightarrow C) \quad \textit{Monotonic } \mathcal{M}}{(G_1 \wedge G_2) \;\supset\; (\mathcal{M} \models (A_1 \;\mathsf{and}\; A_2) \Rightarrow C)}$$

Proof.

1. G_1 - assumption
2. G_2 - assumption
3. $\mathcal{M} \models A_1 \Rightarrow B_1$ - assumption
4. $\mathcal{M} \models (B_1 \;\mathsf{and}\; A_2) \Rightarrow C$ - assumption
5. $\mathcal{M} \models A_2 \Rightarrow A_2$ - Reflexivity
6. $\mathcal{M} \models (A_1 \;\mathsf{and}\; B_1) \;\supset\; (\mathcal{M} \models A_2 \Rightarrow A_2)$
   $\supset\; (\mathcal{M} \models (A_1 \;\mathsf{and}\; A_2) \Rightarrow (B_1 \;\mathsf{and}\; A_2))$ - Conjunction
7. $(\mathcal{M} \models A_2 \Rightarrow A_2)$
   $\supset\; (\mathcal{M} \models (A_1 \;\mathsf{and}\; A_2) \Rightarrow (B_1 \;\mathsf{and}\; A_2))$ - 3 and 6
8. $(\mathcal{M} \models (A_1 \;\mathsf{and}\; A_2) \Rightarrow (B_1 \;\mathsf{and}\; A_2))$ - 5 and 7
9. $(\mathcal{M} \models (A_1 \;\mathsf{and}\; A_2) \Rightarrow C)$ - Transitivity on 4 and 8
10. $(G_1 \wedge G_2) \;\supset\; (\mathcal{M} \models (A_1 \;\mathsf{and}\; A_2) \Rightarrow C)$ - from 1, 2 and 9

□

There are many more derived inference rules that we have found useful in practice, but all of them can be deduced from the ones we presented in this section.

8.3 Examples

8.3.1 n bit And-gate

In this section we shall show how to use symmetries and parametric representation together with STE inference rules to verify the n-bit And gate using only one symbolic variable. Consider the following property we need to verify for an n-bit And gate. For simplicity of presentation let us consider the case for a three bit input to the gate. Let the inputs be called "a_0", "a_1" and "a_2". The output node is "*out*". Thus we need to verify

$$\begin{aligned} \mathit{And} \models (\text{“}a_0\text{”}\ \mathsf{is}\ a_0)\ \mathsf{and}\ (\text{“}a_1\text{”}\ \mathsf{is}\ a_1)\ \mathsf{and}\ (\text{“}a_2\text{”}\ \mathsf{is}\ a_2) \Rightarrow \\ ((\text{“}\mathit{out}\text{”}\ \mathsf{is}\ 1)\ \mathsf{when}\ (a_0 \wedge a_1 \wedge a_2)) \\ \mathsf{and} \\ ((\text{“}\mathit{out}\text{”}\ \mathsf{is}\ 0)\ \mathsf{when}\ (\sim a_0 \vee \sim a_1 \vee \sim a_2)) \end{aligned}$$

Decomposing the above property using the Conjunction rule, we get the following two properties to verify:

I. $\mathit{And} \models (\text{“}a_0\text{”}\ \mathsf{is}\ a_0)\ \mathsf{and}\ (\text{“}a_1\text{”}\ \mathsf{is}\ a_1)\ \mathsf{and}\ (\text{“}a_2\text{”}\ \mathsf{is}\ a_2) \Rightarrow$
$((\text{“}\mathit{out}\text{”}\ \mathsf{is}\ 1)\ \mathsf{when}\ (a_0 \wedge a_1 \wedge a_2))$

II. $\mathit{And} \models (\text{“}a_0\text{”}\ \mathsf{is}\ a_0)\ \mathsf{and}\ (\text{“}a_1\text{”}\ \mathsf{is}\ a_1)\ \mathsf{and}\ (\text{“}a_2\text{”}\ \mathsf{is}\ a_2) \Rightarrow$
$((\text{“}\mathit{out}\text{”}\ \mathsf{is}\ 0)\ \mathsf{when}\ (\sim a_0 \vee \sim a_1 \vee \sim a_2))$

Using parametric representation, we can verify the first case. By verifying the property only for the care set of values, which in this case is the set of values where all the input values are high. The Boolean formula $P(\vec{x})$ in this case is

$$a_0 \wedge a_1 \wedge a_2$$

and is satisfied when the Boolean variables a_0, a_1 and a_2 are all true.

Thus we need to verify the following scalar run to deduce the correctness property stated in (1).

$$\mathit{And} \models (\text{“}a_0\text{”}\ \mathsf{is}\ 1)\ \mathsf{and}\ (\text{“}a_1\text{”}\ \mathsf{is}\ 1)\ \mathsf{and}\ (\text{“}a_2\text{”}\ \mathsf{is}\ 1) \Rightarrow (\text{“}\mathit{out}\text{”}\ \mathsf{is}\ 1)$$

The above run can be easily done in the STE simulator. In fact the simulator Forte comes with a built-in `param` function that computes the care set automatically.

For doing (II), we will exploit the symmetry property of the And gate. First we use the Constraint Implication 1 rule to move the Boolean guard out of the consequent of the formula.

Thus we obtain the following:

$$\begin{aligned} &(\sim a_0 \vee \sim a_1 \vee \sim a_2) \supset \\ &\qquad \mathit{And} \models (\text{“}a_0\text{”}\ \mathsf{is}\ a_0)\ \mathsf{and}\ (\text{“}a_1\text{”}\ \mathsf{is}\ a_1)\ \mathsf{and}\ (\text{“}a_2\text{”}\ \mathsf{is}\ a_2) \Rightarrow (\text{“}\mathit{out}\text{”}\ \mathsf{is}\ 0) \end{aligned}$$

Using the Guard Disjunction rule, we can decompose the above property into the following:

1. $(\sim a_0) \supset (And \models (\text{“}a_0\text{”} \textsf{ is } a_0) \Rightarrow (\text{“}out\text{”} \textsf{ is } 0))$

2. $(\sim a_1) \supset (And \models (\text{“}a_1\text{”} \textsf{ is } a_1) \Rightarrow (\text{“}out\text{”} \textsf{ is } 0))$

3. $(\sim a_2) \supset (And \models (\text{“}a_2\text{”} \textsf{ is } a_2) \Rightarrow (\text{“}out\text{”} \textsf{ is } 0))$

We can the use the Constraint Implication 2 rule to push the guards back into the STE property. Thus we get:

4. $And \models (\text{“}a_0\text{”} \textsf{ is } a_0) \Rightarrow (\text{“}out\text{”} \textsf{ is } 0) \textsf{ when } (\sim a_0)$

5. $And \models (\text{“}a_1\text{”} \textsf{ is } a_1) \Rightarrow (\text{“}out\text{”} \textsf{ is } 0) \textsf{ when } (\sim a_1)$

6. $And \models (\text{“}a_2\text{”} \textsf{ is } a_2) \Rightarrow (\text{“}out\text{”} \textsf{ is } 0) \textsf{ when } (\sim a_2)$

We use the STE simulator to verify (4). This requires using only one variable. Using Theorem 8.1, with π being the permutation that obtains “a_1” from “a_0” and leaves “out” unchanged, we can deduce the correctness assertion stated in (5), which is a permutation of (4). Similarly when the π is the permutation that maps “a_0” to “a_2” and leaves “out” unchanged, we deduce the correctness assertion stated in (6).

Once we have deduced the correctness of (4), (5) and (6) by using one explicit STE run and the symmetry property, we can use the Constraint Implication 1 rule, and then Guard Disjunction Rule in the forward direction to deduce

$$(\sim a_0 \vee \sim a_1 \vee \sim a_2) \supset$$
$$And \models (\text{“}a_0\text{”} \textsf{ is } a_0) \textsf{ and } (\text{“}a_1\text{”} \textsf{ is } a_1) \textsf{ and } (\text{“}a_2\text{”} \textsf{ is } a_2) \Rightarrow (\text{“}out\text{”} \textsf{ is } 0)$$

Finally we use the Constraint Implication 2 rule to deduce the correctness of the property we wanted to have:

$$And \models (\text{“}a_0\text{”} \textsf{ is } a_0) \textsf{ and } (\text{“}a_1\text{”} \textsf{ is } a_1) \textsf{ and } (\text{“}a_2\text{”} \textsf{ is } a_2) \Rightarrow$$
$$((\text{“}out\text{”} \textsf{ is } 0) \textsf{ when } (\sim a_0 \vee \sim a_1 \vee \sim a_2))$$

We could have obtained the final correctness property from the verification results of (4), (5) and (6) directly by doing a disjunction in the guards. We could have done this by virtue of a derived inference rule, however we were very pedantic and showed that the overall process involved going through moving guards out, to doing Guard disjunction to moving them back in, all of which can be done in one go using the derived inference rule, which is what we do in practice.

Thus verification of an n-bit And gate can be done using only one symbolic variable. The cost of verification has been invested in the deductive framework and exploitation of symmetry.

The example of verifying an n-bit Or gate is completely the dual of the verification of n-bit And gate, that we have just shown. The verification of these simple gates illustrate an important point, besides showing the strength of the verification methodology based on symmetry reduction and use of inference rules. The point is that in many circuit models simple gates appear, and the Boolean values that appear at the inputs of these gates can be any Boolean expression. Having a library of verification results in the form of pre-verified results over these gates, means the Boolean variables a_0, a_1,and a_2 can be instantiated to any Boolean expression that arises out of other combinational blocks in the circuit. In a system where we turn the STE verification results into correctness theorems in a Boolean logic [47], the instantiation of free Boolean variables is direct and leads to very powerful gains, since we do not have to verify these gates over and over again when they are used as components in other bigger circuit models.

At present we do not have access to an integrated verification environment where STE verification results can be lifted automatically to theorems. So at the moment the deductive reasoning happens outside the Forte environment. Another possibility is that we already have an implementation of STE inference rules in HOL, so STE verification results from Forte can be plugged as theorems into HOL through an external scripting support, possibly using ML or some other scripting language that needs to be investigated.

Inference rules are useful because a property about n bits is in effect decomposed into n properties each talking about 1 bit. Verifying explicitly these n cases will shift the complexity of verification of a property about n bits needing a BDD built over n Boolean variables, to the verification of n properties each building a smaller BDD because of a single variable. However the time needed for verifying these n properties would without symmetry be of the order n times the complexity of verifying one of them. By exploiting symmetry properties, we are able to collapse the case of verifying n properties to the case where we only verify one representative and deduce that other symmetric properties modulo permutation on trajectory formulas have been verified as well. Notice that using symmetry based arguments for reduction, appears very similar to induction, since we deduce the correctness results of $n-1$ bits from the correctness result of 1 bit. However, it significantly differs from induction because the case of verifying $n+1$ bits does not depend on the verification of the case with n bits. The reduction argument will apply to the circuit with $n+1$ bits, because of the symmetry soundness theorem — Theorem 8.1. It is this theorem that enables the deduction of the correctness of the entire class of equivalent STE properties from verification of one representative.

8.3.2 MUX

We have shown in Chapter 6 how to model a unit-delay, 2-to-1 Mux using the combinators of FSM^*. The Mux circuit in Figure 6.6 has symmetry between the output bus and its input buses a and b. We can use the definition of the Mux given on page 85 to model a 2-to-1 Mux of any size. Let us assume the size of the input buses is 3-bits. Thus we have to verify the following property to completely characterise the behaviour of the Mux.

$$\begin{array}{l}
\quad Mux \models (\text{``}a_0\text{''} \ \mathsf{is}\ a_0)\ \mathsf{and}\ (\text{``}a_1\text{''} \ \mathsf{is}\ a_1)\ \mathsf{and}\ (\text{``}a_2\text{''} \ \mathsf{is}\ a_2)\\
\qquad\quad \mathsf{and}\ (\text{``}b_0\text{''} \ \mathsf{is}\ b_0)\ \mathsf{and}\ (\text{``}b_1\text{''} \ \mathsf{is}\ b_1)\ \mathsf{and}\ (\text{``}b_2\text{''} \ \mathsf{is}\ b_2)\\
\qquad\quad \mathsf{and}\ (\text{``}ctrl\text{''} \ \mathsf{is}\ c) \qquad \Rightarrow\\
\mathsf{N}((\text{``}out_0\text{''} \ \mathsf{is}\ a_0\ \mathsf{when}\ c)\ \mathsf{and}\ (\text{``}out_0\text{''} \ \mathsf{is}\ b_0\ \mathsf{when}\sim c))\\
\mathsf{N}((\text{``}out_1\text{''} \ \mathsf{is}\ a_1\ \mathsf{when}\ c)\ \mathsf{and}\ (\text{``}out_1\text{''} \ \mathsf{is}\ b_1\ \mathsf{when}\sim c))\\
\mathsf{N}((\text{``}out_2\text{''} \ \mathsf{is}\ a_2\ \mathsf{when}\ c)\ \mathsf{and}\ (\text{``}out_2\text{''} \ \mathsf{is}\ b_2\ \mathsf{when}\sim c))
\end{array}$$

Note that is binds tighter than when, so we have left out the parentheses and written $(\text{``}out_0\text{''}\ \mathsf{is}\ a_0\ \mathsf{when}\ c)$, instead of $((\text{``}out_0\text{''}\ \mathsf{is}\ a_0)\ \mathsf{when}\ c)$, for legibility.

Verification in the presence of symmetry

Using the Conjunction rule on the antecedent and consequent, we get the following goals:

I. $Mux \models$ (“a_0” is a_0) and (“b_0” is b_0) and (“$ctrl$” is c) $\Rightarrow$
N((“out_0” is a_0 when c) and (“out_0” is b_0 when $\sim c$))

II. $Mux \models$ (“a_1” is a_1) and (“b_1” is b_1) and (“$ctrl$” is c) $\Rightarrow$
N((“out_1” is a_1 when c) and (“out_1” is b_1 when $\sim c$))

III. $Mux \models$ (“a_2” is a_2) and (“b_2” is b_2) and (“$ctrl$” is c) $\Rightarrow$
N((“out_2” is a_2 when c) and (“out_2” is b_2 when $\sim c$))

We do an STE run to verify (I). It involves two symbolic variables a_0 and b_0 which is not a problem at all during the STE verification run. Mux exhibits symmetry; exchange the first line with the second, where π is the permutation that swaps the nodes “a_0” with “a_1”, “b_0” with “b_1”, and “out_0” with “out_1”. Similarly we can swap the first line with the third with a different π. In both the cases, Sym_χ Mux π holds, and therefore by using Theorem 8.1, we can conclude that (II) and (III) are verified as well.

Thus verifying an n-bit, 2-to-1 mux entails verifying a 1-bit mux using only two symbolic variables, and by way of using symmetry arguments, and inference rules, we can conclude that the n-bit mux is verified as well. In general by exploiting symmetry, we can verify an $n{-}to{-}1$ Mux with $m - bit$ wide input buses using n distinct symbolic variables for input buses and $(log\, n)$ variables for selecting one of the n inputs. Further if we use symbolic indexing, the requirement of n distinct variables can be reduced to $(log\, n)$. Thus instead of having to use mn number of variables without reduction; using symmetry and symbolic indexing we need $2\,(log\, n)$ variables. In a nutshell, reduction in the width of the buses is achieved by exploiting symmetry, and reduction for the number of buses is done by symbolic indexing.

This brings us to the last example of this chapter. We will show how symmetries can be exploited for an n-bit wide comparator that does equality comparison.

8.3.3 Comparator

The high level circuit model for the unit-delay comparator circuit is shown in Chapter 6. We showed that the comparator circuit in Figure 6.5, has symmetry across its two input buses and the output. We can swap “a_0” with “a_1” and “b_0” with “b_1” and the output with itself, and the behaviour of the circuit will not change. This is the property that we will exploit in the verification of the STE property.

The property we wish to verify for the comparator is given by the following:

$Comp \models$ (“a_0” is a_0) and (“b_0” is b_0) and
(“a_1” is a_1) and (“b_1” is b_1)
$\Rightarrow$
N((“out” is 1) when $((a_0 = b_0) \wedge (a_1 = b_1))$) and
N((“out” is 0) when $(\sim(a_0 = b_0) \vee (\sim(a_1 = b_1))$)))

The above property can be decomposed using the Conjunction rule into the following two cases:

I. Equality case

$$\begin{array}{lll} \mathit{Comp} & \models & (\text{``}a_0\text{''} \ \mathsf{is} \ a_0) \ \mathsf{and} \ (\text{``}b_0\text{''} \ \mathsf{is} \ b_0) \ \mathsf{and} \\ & & (\text{``}a_1\text{''} \ \mathsf{is} \ a_1) \ \mathsf{and} \ (\text{``}b_1\text{''} \ \mathsf{is} \ b_1) \\ & & \Rightarrow \ \mathsf{N}((\text{``}\mathit{out}\text{''} \ \mathsf{is} \ 1) \ \mathsf{when} \ ((a_0 = b_0) \wedge (a_1 = b_1))) \end{array}$$

II. Inequality case

$$\begin{array}{lll} \mathit{Comp} & \models & (\text{``}a_0\text{''} \ \mathsf{is} \ a_0) \ \mathsf{and} \ (\text{``}b_0\text{''} \ \mathsf{is} \ b_0) \ \mathsf{and} \\ & & (\text{``}a_1\text{''} \ \mathsf{is} \ a_1) \ \mathsf{and} \ (\text{``}b_1\text{''} \ \mathsf{is} \ b_1) \\ & & \Rightarrow \ \mathsf{N}((\text{``}\mathit{out}\text{''} \ \mathsf{is} \ 0) \ \mathsf{when} \ (\sim(a_0 = b_0) \vee (\sim(a_1 = b_1)))) \end{array}$$

We will show how to verify the equality case first.

Verification in the presence of symmetry – Equality

The goal is to show that

$$\begin{array}{lll} \mathit{Comp} & \models & (\text{``}a_0\text{''} \ \mathsf{is} \ a_0) \ \mathsf{and} \ (\text{``}b_0\text{''} \ \mathsf{is} \ b_0) \ \mathsf{and} \\ & & (\text{``}a_1\text{''} \ \mathsf{is} \ a_1) \ \mathsf{and} \ (\text{``}b_1\text{''} \ \mathsf{is} \ b_1) \\ & & \Rightarrow \mathsf{N}(\text{``}\mathit{out}\text{''} \ \mathsf{is} \ 1) \ \mathsf{when} \ ((a_0 = b_0) \wedge (a_1 = b_1)) \end{array}$$

The verification strategy proceeds as follows. We first use the Cut rule to partition the verification of a property into two cases, which rely on a weaker trajectory formula. In the sequel below these weak formulas are given by B_0 and B_1, and properties where these formulas appear are known as weak properties. Other trajectory formulas describing parts of the antecedent and the consequent of the given property are defined.

$$\begin{array}{lll} \mathit{let} \ B_0 & = & (\text{``}i_0\text{''} \ \mathsf{is} \ 1) \\ \mathit{let} \ B_1 & = & (\text{``}i_1\text{''} \ \mathsf{is} \ 1) \\ \mathit{let} \ A_0 & = & (\text{``}a_0\text{''} \ \mathsf{is} \ a_0) \ \mathsf{and} \ (\text{``}b_0\text{''} \ \mathsf{is} \ b_0) \\ \mathit{let} \ A_1 & = & (\text{``}a_1\text{''} \ \mathsf{is} \ a_1) \ \mathsf{and} \ (\text{``}b_1\text{''} \ \mathsf{is} \ b_1) \\ \mathit{let} \ G_0 & = & (a_0 = b_0) \\ \mathit{let} \ G_1 & = & (a_1 = b_1) \\ \mathit{let} \ C' & = & (\text{``}o\text{''} \ \mathsf{is} \ 1) \\ \mathit{let} \ C & = & \mathsf{N}(\text{``}\mathit{out}\text{''} \ \mathsf{is} \ 1) \end{array}$$

All the steps are outlined below. Here is how the verification will proceed once the trajectory formulas have been defined. It begins by carrying out an STE run verifying the weaker property i.e., the result of comparing two bit values (the output of the bitwise comparison of bits "a_0" and "b_0") is 1, if the values are equal. This run requires two symbolic variables. This is shown in Step 1 below.

By symmetry, using Theorem 8.1 we can conclude (Step 2) that the other part of the bitwise comparison circuitry is also correct. Then the Constraint Implication rules transform the properties (Steps 3 and 4) such that the guards are moved out of the consequent of the properties, and a Guard Conjunction rule is used to stitch the correctness results of the two bitwise comparison cases (Step 5). Then assuming that all inputs to the And gate of the comparator are 1, we check if the output is 1 (Step 6). This run basically uses the weaker formulas B_0 and B_1 in the antecedent and carries out the scalar simulation in Forte. Another straightforward scalar run (Step 7) is carried out to check if the output value appears at "*out*" one time point later than "*o*". Finally by virtue of Transitivity rule and the Cut (Steps 8 and 9), we deduce the correctness property that is finally transformed (Step 10) using the Constraint Implication 2 rule, to make it look exactly what we wanted to verify in the first instance.

1. $Comp \models A_0 \Rightarrow (B_0 \text{ when } G_0)$ (*STE run using 2 variables*)
2. $Comp \models A_1 \Rightarrow (B_1 \text{ when } G_1)$ (*Symmetry*)
3. $G_0 \supset (Comp \models A_0 \Rightarrow B_0)$ (*Constraint Implication 1*)
4. $G_1 \supset (Comp \models A_1 \Rightarrow B_1)$ (*Constraint Implication 1*)
5. $(G_0 \wedge G_1) \supset (Comp \models (A_0 \text{ and } A_1) \Rightarrow (B_0 \text{ and } B_1))$ (*Guard Conjunction*)
6. $Comp \models (B_0 \text{ and } B_1) \Rightarrow C'$ (*Scalar STE run*)
7. $Comp \models C' \Rightarrow C$ (*Scalar STE run*)
8. $Comp \models (B_0 \text{ and } B_1) \Rightarrow C$ (*Transitivity on 6 and 7*)
9. $(G_0 \wedge G_1) \supset (Comp \models (A_0 \text{ and } A_1) \Rightarrow C)$ (*Cut on 5 and 8*)
10. $Comp \models (A_0 \text{ and } A_1) \Rightarrow (C \text{ when } (G_0 \wedge G_1))$ (*Constraint Implication 2*)

Replacing the values of A_0, A_1, C, G_0 and G_1 we get

$$\begin{aligned} Comp \models\ & (\text{``}a_0\text{''} \text{ is } a_0) \text{ and } (\text{``}b_0\text{''} \text{ is } b_0) \text{ and} \\ & (\text{``}a_1\text{''} \text{ is } a_1) \text{ and } (\text{``}b_1\text{''} \text{ is } b_1) \\ & \Rightarrow \mathsf{N}(\text{``}out\text{''} \text{ is } 1) \text{ when } ((a_0 = b_0) \wedge (a_1 = b_1)) \end{aligned}$$

Verification in the presence of symmetry – Inequality

The steps for the inequality case (II) are outlined below. We begin by defining some trajectory formulas.

$$\begin{aligned} &let\ A && = (\text{``}a_0\text{''} \text{ is } a_0) \text{ and } (\text{``}b_0\text{''} \text{ is } b_0) \text{ and } (\text{``}a_1\text{''} \text{ is } a_1) \text{ and } (\text{``}b_1\text{''} \text{ is } b_1) \\ &let\ A_0 && = (\text{``}a_0\text{''} \text{ is } a_0) \text{ and } (\text{``}b_0\text{''} \text{ is } b_0) \\ &let\ A_1 && = (\text{``}a_1\text{''} \text{ is } a_1) \text{ and } (\text{``}b_1\text{''} \text{ is } b_1) \\ &let\ C' && = (''o'' \text{ is } 0) \\ &let\ C && = \mathsf{N}(\text{``}out\text{''} \text{ is } 0) \\ &let\ G_0 && = \sim(a_0 = b_0) \\ &let\ G_1 && = \sim(a_1 = b_1) \end{aligned}$$

Now we proceed as follows:

1. $Comp \models A_0 \Rightarrow (C' \text{ when } G_0)$ (*STE run using 2 variables*)
2. $Comp \models A_1 \Rightarrow (C' \text{ when } G_1)$ (*Symmetry*)
3. $G_0 \supset (Comp \models A_0 \Rightarrow C')$ (*Constraint Implication 1*)
4. $G_1 \supset (Comp \models A_1 \Rightarrow C')$ (*Constraint Implication 1*)
5. $(G_0 \vee G_1) \supset (Comp \models ((A_0 \text{ and } A_1) \Rightarrow C'))$ (*Guard Disjunction on 3 and 4*)
6. $Comp \models C' \Rightarrow C$ (*Scalar STE run*)
7. $(G_0 \vee G_1) \supset (Comp \models ((A_0 \text{ and } A_1) \Rightarrow C))$ (*Cut*)

Replacing the values of A_0, A_1, C, G_0 and G_1 we get

$$\begin{array}{rcl} \mathit{Comp} & \models & (\text{“}a_0\text{” is } a_0) \text{ and } (\text{“}b_0\text{” is } b_0) \text{ and} \\ & & (\text{“}a_1\text{” is } a_1) \text{ and } (\text{“}b_1\text{” is } b_1) \\ & & \quad \Rightarrow \mathsf{N}(\text{“}\mathit{out}\text{” is } 0) \text{ when } (\sim(a_0 = b_0) \vee (\sim(a_1 = b_1))) \end{array}$$

Thus the number of variables required in any STE run is 2. By using inference rules we are able to decompose the properties effectively, and using symmetry collapses the number of explicit cases to verify to one, and that case needs 2 variables. Thus variables required for verifying an n-bit comparator remains fixed at 2 and not $2n$. So BDDs always stay small and verification complexity remains constant with respect to BDD size.

8.4 Related work

One significant piece of research done in this area that closely resembles ours, is Manish Pandey's work on using symmetries for reduction in STE model checking [82, 83]. Pandey verified transistor level netlist representations of circuit models using STE model checking. In order to discover symmetry in these models, he had to rely on finding suitable heuristics for discovering isomorphisms in sub-graphs, a known NP-complete problem [42]. Other researchers [23, 109], have also taken the approach to discovering symmetries by having a set of heuristics. The problem with this approach is that, for each new circuit that is verified, possibly a new set of heuristics are needed. When the right heuristics are supplied, even then the time needed for discovering symmetry is proportional to the size of the circuit model [82]. Thus the total time used in verification is significantly dominated by symmetry checks.

By comparison using our approach, time required to type check circuit models for discovering symmetry is constant with respect to different sizes of a given circuit, so we are able to drastically reduce the time factor in symmetry checks.

Pandey also used the idea of property decomposition, using the known set of STE inference rules. He relied on using STE simulation to find interfaces in transistor level graphs, whereby he could partition the circuit to expose the names of input and output nodes that define the symmetry across interfaces. He would then use a strategy similar to ours which relies on verifying weaker properties (using a Cut like rule) and then use the weaker properties to build relatively stronger ones, finally verifying the original property.

Whereas Pandey had to use simulation to discover interfaces across which circuit models and properties can be partitioned to discover symmetry in models and properties, we perform better because of the way we model circuits. By recording the inputs and outputs that give rise to symmetry, at the time of circuit modelling (in FSM^*), we don't have to do much at the time of property verification to discover symmetry in models and properties. Symmetry in models is determined by type checking as explained earlier, and symmetry amongst properties is exposed by recording the names of inputs and outputs that give rise to symmetry at the high-level in FSM^*. Using the knowledge that circuits have been composed using the high-level combinators such as $\circ$, identifying symmetry inside the circuit amounts to simply identifying those circuit blocks that were composed using $\circ$, and names across those segments can be easily read off from the FSM. Those names are used to cluster the decomposed set of smaller properties into symmetric ones.

We believe, discovering symmetry information in a circuit is best accomplished when the information can be tapped before it gets lost. This means that reading symmetries in circuit descriptions at a higher level of abstraction is a much better idea, and in this manner we strongly share Ip and Dill's approach [40] of modelling circuits using special types, that explicitly announce the presence of symmetry in the circuit.

8.5 Summary

In this chapter we presented the theory behind our reduction methodology. We presented a link between the STE theory and symmetry generating swaps. We presented a very important theorem that justifies leaving out explicit verification of STE properties by running an STE simulator if they are symmetric modulo permutation. We also offered a novel set of STE inference rules that enable very effective property decomposition.

We showed several examples to illustrate our methodology of property reduction. In the next chapter we present case studies on bigger circuit models such as memories.

Chapter 9

Examples and Case Studies

We will present several examples in this chapter to illustrate the applicability of our theory we presented in the previous chapters. Examples include a steering circuit, an n-bit register, a random access memory (RAM), a content addressable memory (CAM) and finally a circuit that involves two CAMs. We will present comparisons of our work with other related work wherever applicable.

9.1 Steering circuit

Our first example in this chapter is a steering circuit. This circuit accepts three inputs, and compares the value of two of them. If the two values are equal then the output is that value, while if the values are not equal, the third one is the output. This circuit employs a multiplexer and a comparator, as shown in Figure 9.1.

There are two symmetric blocks connected together in this circuit. The first symmetry is in the comparator (*Comp*) and the second is in the multiplexer (*Mux*). The symmetry in the comparator circuit however, doesn't affect the symmetry in the multiplexer circuit. In fact the output of the comparator becomes the control input to the multiplexer. Therefore we made the design choice to group the symmetric inputs of the *Mux* together and treat them as symmetric inputs to the Steering circuit, and the symmetric inputs of the comparator circuit are grouped together as non-symmetric inputs to the Steering circuit even though they are responsible for the symmetry of the comparator. The comparator circuit itself doesn't have any non-symmetric inputs, so the first argument to *Comp* is the empty list. This is choice that we have made for all circuit modelling, where an output of one symmetric block behaves as a non-symmetric input to another symmetric block.

Circuits similar to steering circuit appear in many memory designs, where the incoming data a is compared with the stored data b, and if the values are equal then no change is made to the stored data b, else new data c is written. Of course in this circuit there is no memory, but it serves well to illustrate how to exploit the symmetries of the comparator and the multiplexer to do reduction for this circuit.

The model of the circuit in FSM^* is shown below:

$$\mathit{Steer}\ \mathit{nsym} \;=\; \mathit{Mux}(\mathit{Comp}([\,] : \mathit{bool\ list\ list})\ \mathit{nsym})$$

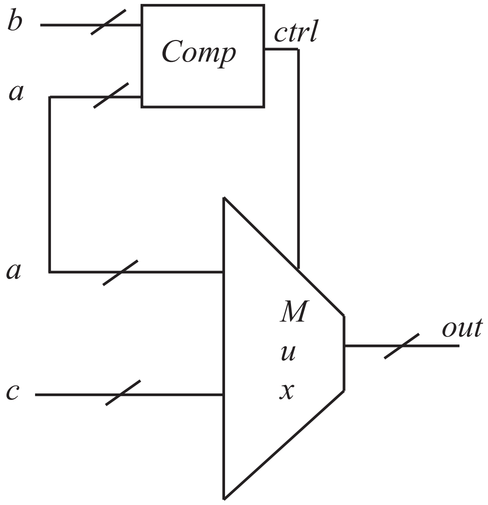


Figure 9.1: Steering circuit.

9.1.1 Property reduction

$$\begin{aligned}
\mathit{Steer} \models\ & (\text{“}a_0\text{” is } a_0) \text{ and } (\text{“}a_1\text{” is } a_1) \text{ and}\\
& (\text{“}b_0\text{” is } b_0) \text{ and } (\text{“}b_1\text{” is } b_1) \text{ and}\\
& (\text{“}c_0\text{” is } c_0) \text{ and } (\text{“}c_1\text{” is } c_1)\\
& \Rightarrow\\
& ((\text{“}out_0\text{” is } a_0) \text{ and } (\text{“}out_1\text{” is } a_1)) \text{ when } ((a_0 = b_0) \wedge (a_1 = b_1)) \text{ and}\\
& ((\text{“}out_0\text{” is } c_0) \text{ and } (\text{“}out_1\text{” is } c_1)) \text{ when } ((\sim(a_0 = b_0)) \vee (\sim(a_1 = b_1)))
\end{aligned}$$

Decompose them into

1. Case I

$$\begin{aligned}
\mathit{Steer} \models\ & (\text{“}a_0\text{” is } a_0) \text{ and } (\text{“}a_1\text{” is } a_1) \text{ and}\\
& (\text{“}b_0\text{” is } b_0) \text{ and } (\text{“}b_1\text{” is } b_1) \text{ and}\\
& \Rightarrow\\
& ((\text{“}out_0\text{” is } a_0) \text{ and } (\text{“}out_1\text{” is } a_1)) \text{ when } ((a_0 = b_0) \wedge (a_1 = b_1))
\end{aligned}$$

2. Case II

$$\begin{aligned}
\mathit{Steer} \models\ & (\text{“}a_0\text{” is } a_0) \text{ and } (\text{“}a_1\text{” is } a_1) \text{ and}\\
& (\text{“}b_0\text{” is } b_0) \text{ and } (\text{“}b_1\text{” is } b_1) \text{ and}\\
& (\text{“}c_0\text{” is } c_0) \text{ and } (\text{“}c_1\text{” is } c_1) \text{ and}\\
& \Rightarrow\\
& ((\text{“}out_0\text{” is } c_0) \text{ and } (\text{“}out_1\text{” is } c_1)) \text{ when } ((\sim(a_0 = b_0)) \vee (\sim(a_1 = b_1)))
\end{aligned}$$

Detailed solution for Case I

We need to verify the following

$$\begin{aligned} \mathit{Steer} \models\ & (\text{“}a_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}a_1\text{”}\ \textsf{is}\ a_1)\ \textsf{and} \\ & (\text{“}b_0\text{”}\ \textsf{is}\ b_0)\ \textsf{and}\ (\text{“}b_1\text{”}\ \textsf{is}\ b_1)\ \textsf{and} \\ & \quad \Rightarrow \\ & \quad ((\text{“}\mathit{out}_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}\mathit{out}_1\text{”}\ \textsf{is}\ a_1))\ \textsf{when}\ ((a_0 = b_0) \wedge (a_1 = b_1)) \end{aligned}$$

Come up with weaker properties

$$\begin{aligned} \mathit{Steer} \models\ & (\text{“}a_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}a_1\text{”}\ \textsf{is}\ a_1)\ \textsf{and} \\ & (\text{“}b_0\text{”}\ \textsf{is}\ b_0)\ \textsf{and}\ (\text{“}b_1\text{”}\ \textsf{is}\ b_1) \Rightarrow (\text{“}\mathit{ctrl}\text{”}\ \textsf{is}\ T)\ \textsf{when}\ ((a_0 = b_0) \wedge (a_1 = b_1)) \end{aligned}$$

We can use the same reasoning as for the comparator case to verify the above property. We know from the comparator verification study that we only ever need two symbolic variables to verify an n-bit comparator. Thus using the same verification flow as that for a comparator, we can verify the above property, using two variables.

Rewriting it by using pushing guards out of the consequent, we obtain (1) below.

1. $((a_0 = b_0) \wedge (a_1 = b_1)) \supset$
   $(\mathit{Steer} \models ((\text{“}a_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}a_1\text{”}\ \textsf{is}\ a_1)\ \textsf{and}$
   $(\text{“}b_0\text{”}\ \textsf{is}\ b_0)\ \textsf{and}\ (\text{“}b_1\text{”}\ \textsf{is}\ b_1) \Rightarrow (\text{“}\mathit{ctrl}\text{”}\ \textsf{is}\ T))$ (*Constraint Implication 1*)
2. $\mathit{Steer} \models (\text{“}\mathit{ctrl}\text{”}\ \textsf{is}\ T)\ \textsf{and}\ (\text{“}a_0\text{”}\ \textsf{is}\ a_0) \Rightarrow (\text{“}\mathit{out}_0\text{”}\ \textsf{is}\ a_0)$ (*STE run*)
3. $\mathit{Steer} \models (\text{“}\mathit{ctrl}\text{”}\ \textsf{is}\ T)\ \textsf{and}\ (\text{“}a_1\text{”}\ \textsf{is}\ a_1) \Rightarrow (\text{“}\mathit{out}_1\text{”}\ \textsf{is}\ a_1)$ (*Symmetry*)

4. $\mathit{Steer} \models (\text{“}a_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}a_1\text{”}\ \textsf{is}\ a_1)\ \textsf{and}\ (\text{“}\mathit{ctrl}\text{”}\ \textsf{is}\ T)$
   $\Rightarrow (\text{“}\mathit{out}_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}\mathit{out}_1\text{”}\ \textsf{is}\ a_1)$ (*Conjunction*)
5. $((a_0 = b_0) \wedge (a_1 = b_1)) \supset$
   $(\mathit{Steer} \models (\text{“}a_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}a_1\text{”}\ \textsf{is}\ a_1)\ \textsf{and}$
   $(\text{“}b_0\text{”}\ \textsf{is}\ b_0)\ \textsf{and}\ (\text{“}b_1\text{”}\ \textsf{is}\ b_1)\ \textsf{and}$
   $(\text{“}a_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}a_1\text{”}\ \textsf{is}\ a_1)$
   $\Rightarrow$
   $(\text{“}\mathit{out}_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}\mathit{out}_1\text{”}\ \textsf{is}\ a_1))$ (*Cut*)
6. $((a_0 = b_0) \wedge (a_1 = b_1)) \supset$
   $(\mathit{Steer} \models (\text{“}a_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}a_1\text{”}\ \textsf{is}\ a_1)\ \textsf{and}$
   $(\text{“}b_0\text{”}\ \textsf{is}\ b_0)\ \textsf{and}\ (\text{“}b_1\text{”}\ \textsf{is}\ b_1))$
   $\Rightarrow$
   $((\text{“}\mathit{out}_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}\mathit{out}_1\text{”}\ \textsf{is}\ a_1))$ (*Antecedent Weakening*)
7. $((a_0 = b_0) \wedge (a_1 = b_1)) \supset$
   $(\mathit{Steer} \models (\text{“}a_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}a_1\text{”}\ \textsf{is}\ a_1)\ \textsf{and}$
   $(\text{“}b_0\text{”}\ \textsf{is}\ b_0)\ \textsf{and}\ (\text{“}b_1\text{”}\ \textsf{is}\ b_1))$
   $(\text{“}c_0\text{”}\ \textsf{is}\ c_0)\ \textsf{and}\ (\text{“}c_1\text{”}\ \textsf{is}\ c_1)$
   $\Rightarrow$
   $((\text{“}\mathit{out}_0\text{”}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{“}\mathit{out}_1\text{”}\ \textsf{is}\ a_1))$ (*Antecedent Strengthening 2*)

8. $Steer \models (\text{“}a_0\text{”} \textsf{ is } a_0) \textsf{ and } (\text{“}a_1\text{”} \textsf{ is } a_1) \textsf{ and}$
   $(\text{“}b_0\text{”} \textsf{ is } b_0) \textsf{ and } (\text{“}b_1\text{”} \textsf{ is } b_1)$
   $(\text{“}c_0\text{”} \textsf{ is } c_0) \textsf{ and } (\text{“}c_1\text{”} \textsf{ is } c_1)$
   $\Rightarrow$
   $((\text{“}out_0\text{”} \textsf{ is } a_0) \textsf{ and } (\text{“}out_1\text{”} \textsf{ is } a_1))$
   $\textsf{when } ((a_0 = b_0) \wedge (a_1 = b_1))$ (*Constraint Implication 2*)

Thus we have verified Case I by actually using two symbolic variables. Two were used for the case when we said we can use the comparator checking results. One of these two were used again in the verification of the *Mux* circuitry, see step (2) above.

Detailed solution for Case II

We have to verify the following:

$Steer \models (\text{“}a_0\text{”} \textsf{ is } a_0) \textsf{ and } (\text{“}a_1\text{”} \textsf{ is } a_1) \textsf{ and}$
$(\text{“}b_0\text{”} \textsf{ is } b_0) \textsf{ and } (\text{“}b_1\text{”} \textsf{ is } b_1) \textsf{ and}$
$(\text{“}c_0\text{”} \textsf{ is } c_0) \textsf{ and } (\text{“}c_1\text{”} \textsf{ is } c_1) \textsf{ and}$
$\Rightarrow$
$((\text{“}out_0\text{”} \textsf{ is } c_0) \textsf{ and } (\text{“}out_1\text{”} \textsf{ is } c_1))$
$\textsf{when } ((\sim(a_0 = b_0)) \vee (\sim(a_1 = b_1)))$

Come up with weaker properties

$Steer \models (\text{“}a_0\text{”} \textsf{ is } a_0) \textsf{ and } (\text{“}a_1\text{”} \textsf{ is } a_1) \textsf{ and}$
$(\text{“}b_0\text{”} \textsf{ is } b_0) \textsf{ and } (\text{“}b_1\text{”} \textsf{ is } b_1)$
$\Rightarrow$
$(\text{“}ctrl\text{”} \textsf{ is } F) \textsf{ when } ((\sim(a_0 = b_0)) \vee (\sim(a_1 = b_1)))$

The above property is an instance of the inequality case of the comparator verification case. Thus we can transfer the same strategy as we used for verifying the comparator case, using only two symbolic variables.

1. $((\sim(a_0 = b_0)) \vee (\sim(a_1 = b_1))) \supset$
   $(Steer \models (\text{“}a_0\text{”} \textsf{ is } a_0) \textsf{ and } (\text{“}a_1\text{”} \textsf{ is } a_1) \textsf{ and}$
   $(\text{“}b_0\text{”} \textsf{ is } b_0) \textsf{ and } (\text{“}b_1\text{”} \textsf{ is } b_1)$
   $\Rightarrow (\text{“}ctrl\text{”} \textsf{ is } F)$ (*Constraint Implication 1*)
2. $Steer \models (\text{“}c_0\text{”} \textsf{ is } c_0) \textsf{ and } (\text{“}ctrl\text{”} \textsf{ is } F) \Rightarrow (\text{“}out_0\text{”} \textsf{ is } c_0)$ (*STE run*)
3. $Steer \models (\text{“}c_1\text{”} \textsf{ is } c_1) \textsf{ and } (\text{“}ctrl\text{”} \textsf{ is } F) \Rightarrow (\text{“}out_1\text{”} \textsf{ is } c_1)$ (*Symmetry*)
4. $Steer \models (\text{“}c_0\text{”} \textsf{ is } c_0) \textsf{ and } (\text{“}c_1\text{”} \textsf{ is } c_1) \textsf{ and } (\text{“}ctrl\text{”} \textsf{ is } F)$
   $\Rightarrow (\text{“}out_0\text{”} \textsf{ is } c_0) \textsf{ and } (\text{“}out_1\text{”} \textsf{ is } c_1)$ (*Conjunction and Ant Weakening*)
5. $((\sim(a_0 = b_0)) \vee (\sim(a_1 = b_1))) \supset$
   $(Steer \models (\text{“}a_0\text{”} \textsf{ is } a_0) \textsf{ and } (\text{“}a_1\text{”} \textsf{ is } a_1) \textsf{ and}$
   $(\text{“}b_0\text{”} \textsf{ is } b_0) \textsf{ and } (\text{“}b_1\text{”} \textsf{ is } b_1)$
   $(\text{“}c_0\text{”} \textsf{ is } c_0) \textsf{ and } (\text{“}c_1\text{”} \textsf{ is } c_1)$
   $\Rightarrow$
   $((\text{“}out_0\text{”} \textsf{ is } c_0) \textsf{ and } (\text{“}out_1\text{”} \textsf{ is } c_1)))$ (*Cut on 1 and 4*)

$$
\begin{array}{l}
6.\ \textit{Steer} \models (\text{“}a_0\text{”}\ \mathsf{is}\ a_0)\ \mathsf{and}\ (\text{“}a_1\text{”}\ \mathsf{is}\ a_1)\ \mathsf{and} \\
\qquad\qquad (\text{“}b_0\text{”}\ \mathsf{is}\ b_0)\ \mathsf{and}\ (\text{“}b_1\text{”}\ \mathsf{is}\ b_1)) \\
\qquad\qquad (\text{“}c_0\text{”}\ \mathsf{is}\ c_0)\ \mathsf{and}\ (\text{“}c_1\text{”}\ \mathsf{is}\ c_1)) \\
\qquad\qquad\qquad \Rightarrow \\
\qquad\qquad\qquad\quad ((\text{“}out_0\text{”}\ \mathsf{is}\ c_0)\ \mathsf{and}\ (\text{“}out_1\text{”}\ \mathsf{is}\ c_1)) \\
\qquad\qquad\qquad\quad \mathsf{when}\ ((\sim(a_0 = b_0)) \vee (\sim(a_1 = b_1)))\quad (\textit{Constraint Implication 2})
\end{array}
$$

We have verified Case II in this case also using two variables for comparator inequality case and one for the *Mux* circuitry. Thus the verification of an n-bit steering circuit, requires only two symbolic variables. This is no surprise since the verification of its components, namely the comparator and the multiplexer, also require two symbolic variables. Actually the multiplexer requires a minimum of just one variable as we have seen in step (2) above.

9.2 8-bit register

An 8-bit register is shown in Figure 9.2. It uses eight positive level triggered (also known as active high) latches to store the 8 bits. At time 0, the input data bits “I_0” to “I_7” get latched onto the bits “O_0” to “O_7”. If the data at bits “O_0” to “O_7” is sampled at a later time point, the bits return the stored value. The circuit is very simple indeed, but also illustrates how symmetries are exploited in a register file.

The definition of its circuit model in HOL, using high level combinators, is shown below. This definition can easily be type checked to ensure that this circuit does indeed have symmetries.

$$\textit{Reg}\,[[\,ck\,]] \;=\; map(map\ (AH\ ck))$$

Let us state the antecedents and the consequent formulas:

$$
\begin{array}{lcl}
\textit{let } A & = & (\text{“}I_0\text{”}\ \mathsf{is}\ I_0)\ \mathsf{and}\ (\text{“}I_1\text{”}\ \mathsf{is}\ I_1)\ \mathsf{and}\ (\text{“}I_2\text{”}\ \mathsf{is}\ I_2)\ \mathsf{and}\ (\text{“}I_3\text{”}\ \mathsf{is}\ I_3)\ \mathsf{and} \\
 & & (\text{“}I_4\text{”}\ \mathsf{is}\ I_4)\ \mathsf{and}\ (\text{“}I_5\text{”}\ \mathsf{is}\ I_5)\ \mathsf{and}\ (\text{“}I_6\text{”}\ \mathsf{is}\ I_6)\ \mathsf{and}\ (\text{“}I_7\text{”}\ \mathsf{is}\ I_7) \\
 & & \\
\textit{let clk} & = & (\text{“}ck\text{”}\ \mathsf{is}\ T)\ \mathsf{and}\ (\mathsf{N}\,(\text{“}ck\text{”}\ \mathsf{is}\ F)) \\
 & & \\
\textit{let } C & = & \mathsf{N}\,((\text{“}O_0\text{”}\ \mathsf{is}\ I_0)\ \mathsf{and}\ (\text{“}O_1\text{”}\ \mathsf{is}\ I_1)\ \mathsf{and}\ (\text{“}O_2\text{”}\ \mathsf{is}\ I_2)\ \mathsf{and}\ (\text{“}O_3\text{”}\ \mathsf{is}\ I_3) \\
 & & \mathsf{and}\ (\text{“}O_4\text{”}\ \mathsf{is}\ I_4)\ \mathsf{and}\ (\text{“}O_5\text{”}\ \mathsf{is}\ I_5)\ \mathsf{and}\ (\text{“}O_6\text{”}\ \mathsf{is}\ I_6)\ \mathsf{and}\ (\text{“}O_7\text{”}\ \mathsf{is}\ I_7))
\end{array}
$$

We have to verify the following property to characterise the behaviour of the register file.

$$\textit{Reg} \models (A\ \mathsf{and}\ clk)\ \Rightarrow\ C$$

The register has symmetry across the bus I and O. The bits “I_0” to “I_7” belong to a single bus in the symmetric inputs, and the bits “O_0” to “O_7” are the outputs across which the register circuit exhibits symmetry. Thus the above property can be decomposed into 8 smaller properties, using the Conjunction rule of inference, and any of the resulting smaller properties can be chosen and verified. Using the Theorem 8.1,

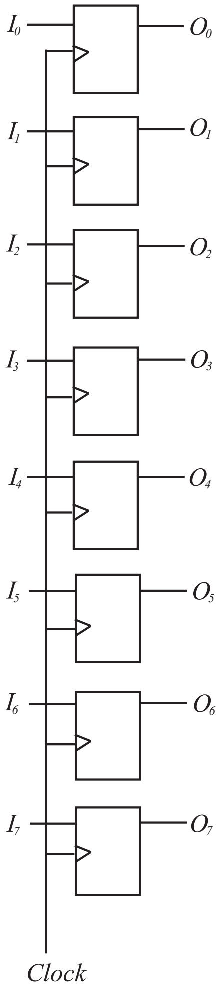


Figure 9.2: An 8-bit register.

we can deduce the correctness of the rest of the smaller properties. The Conjunction rule is used in the forward direction to combine the overall correctness property.

We can verify the following property, and conclude by symmetry arguments that the rest of the equivalent cases have been verified as well using the Theorem 8.1.

$$Reg \models ((\text{“}I_0\text{”} \textsf{ is } I_0) \textsf{ and } clk) \Rightarrow \textsf{N}(\text{“}O\text{”} \textsf{ is } I_0)$$

Thus generally an n-bit register can be verified using only one symbolic variable using symmetry based reduction.

9.3 Random Access Memories

Random access memories come in two modes, static and dynamic. The static RAM (SRAM) consists of internal latches that store the binary information. The stored

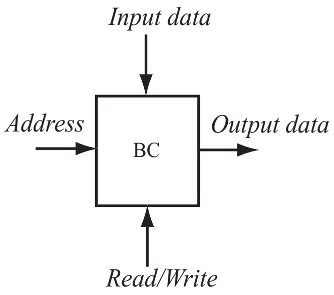


Figure 9.3: Binary storage cell (BC) of an SRAM. Taken from [72].

information remains intact as long as the power is applied to the unit. The dynamic RAM (DRAM) stores the information in the form of electric charges on capacitors. The stored charge on the capacitors tends to discharge with time and must be periodically charged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore decaying charge. DRAM offers reduced power consumption and larger storage capacity in a single memory chip, while SRAM is easier to use, it has shorter read and write cycles. SRAMs are usually costlier, and more reliable, and hence are used in several other memory based circuits, such as caches. In our approach we do not model the RAMs at this level of electrical detail, since the goal is to check the logic function of the memory, which is best done at the high level.

The internal construction of an SRAM consists of $m \times n$ binary storage cells, where m words can be stored in the memory with n bits per word. A binary storage cell (BC) is the basic building block of a memory unit, as shown in Figure 9.3. The storage part of the cell is modelled by an active high latch (*AH*). Actually the cell is an electronic circuit with 4 to 6 transistors. However for our purposes of modelling we consider it to be an active high latch with a read/write control logic surrounding the latch.

The logical construction of a small 4 word RAM is shown in Figure 9.4. This RAM stores 2 bits per word. A memory with 4 word lines needs two address inputs. The address inputs are also known as the address bus. The address bus is an input to the decoder circuit that generates the four address lines for each of the four words. With the memory select asserted high, one of the four address lines are selected depending upon the input to the decoder. Once an address line is selected, the read/write input determines whether the data is read or written from the corresponding address location.

For our purpose we choose the level of abstraction such that we leave out the decoder bit of the circuitry, and assume that the address lines are present as primary inputs to the memory. The other primary inputs to the memory are the data to be written, and the read/write signal. The output of the memory is a data port, from where the data is read out. The input and the output data bits form the symmetric inputs and outputs respectively. The address lines, and the read/write enabled signals are the

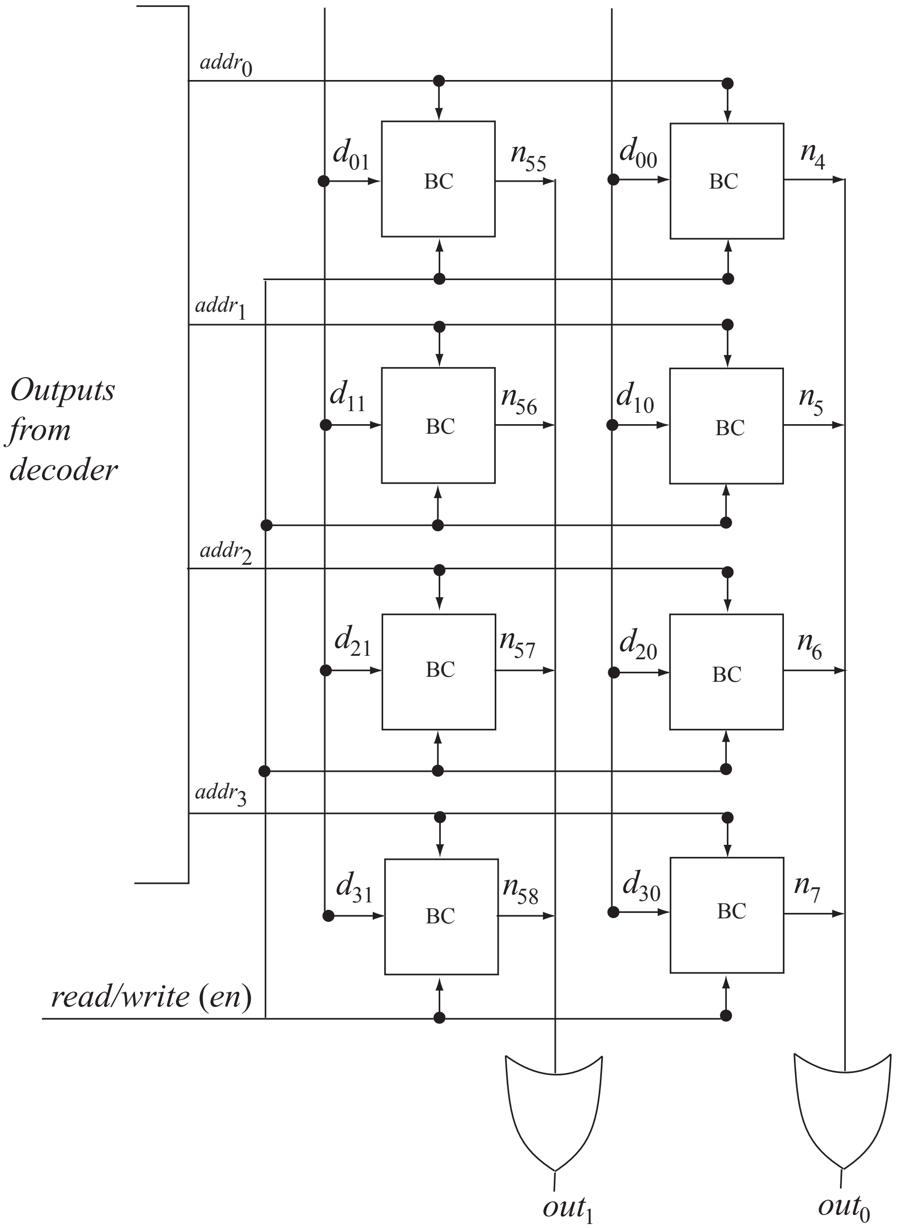


Figure 9.4: Logical construction of a 4 × 2 SRAM. The Or gates are shown according to the array logic. The state holding nodes are n_4 to n_7 and n_{55} to n_{58}. Note that we have not shown quotes around the node names for the sake of clarity in the drawing. Adapted from [72].

non-symmetric inputs to the circuit.

9.3.1 Modelling

In this section we will explain how we model the SRAM circuit shown in Figure 9.4 in FSM*. Incoming data bits are symmetric inputs, where each data bus in the input corresponds to row of data bits that would be written to a unique address. This is done by the function *Nbits* which maps each incoming data bus in the symmetric input to a unique address location specified by the first argument of the *Nbits* function. The function *Nbits* accepts an address bus of m lines, and for each of the m words, generates a line of n data bits, where n is the number of data bits stored per word. Note that there is nothing in the definition of *Nbits* function to suggest that the width of the data bus is n. The width of the data bus is determined at the time of evaluating the function *Nbits*, where the data bus is also specified as an explicit argument to the *Nbits* function.

```
(Nbits  [ ]  =  Null)
(Nbits  ([ ] :: xs)  =  Null)
(Nbits  [a :: addrlist]  =
          let cand  inp  =  map  (∧ (hd inp)) in
          let m  =  (length (a :: addrlist)  −  1) in
          (Nbits [addrlist])  ||  (map (cand [a])  ∘  (Select m Id)))
(Nbits  ((x :: y) :: xs) = Null
```

The m-word memory is defined in terms of one word of memory, which is defined by the function *oneline*. This function generates one line of memory with arbitrary number of data bits, with the read and the write control logic surrounding each storage element in that line. The storage element itself is specified by mapping an active-high latch over the incoming data bits.

```
oneline [[en]] [[addr]]  =
          let cand inp  =  map  (∧ (hd inp)) in
                         map (cand [addr])  ∘  (map (cand [en])) ∘
                                    (map (map (AH (∼en))))  ∘  Nbits [[addr]]
```

Data will be read from this line when the read/write signal (*en*) is asserted high, and will be written to the line when the *en* enable is asserted low. The line also has a unique address *addr*, specified as a second argument to the function *oneline*.

The function *Lines* generates m lines of memory with n data bits per word.

```
(Lines  en [[ ]]  =  Null)
(Lines  en [(x :: xs)]  =
          let n  =  (length  (x :: xs)  −  1) in
          (((oneline en [[x]])  ∘  (Select n Id))  ||  (Lines en [xs])))
```

Finally the function *memory* defines an $m \times n$ memory grid. The output data is read out by doing a bitwise *or* operation on the bits of each of the address words. For example the Boolean value at “out_0” in Figure 9.4, can come from any of the stored 0^{th} bit, which of course gets specified by the address location of the word.

$$memory\ \ en\ addr \ = \ (Bitwise\ \lor\ (Lines\ en\ addr))$$

The following session shows how we generate the circuit FSMs for the memory in practice. The example below shows a 4 word memory, with 2 data bits per word. Although this example is kept deliberately small in terms of word and data size to keep the presentation simple and clear, we have implemented several large sized FSMs.

```
(********** Evaluate the Structure --- obtain the netlist *********)

⊢ckt2netlist (memory [[s "en"]])[["addr3"; "addr2";"addr1"; "addr0"]]
                         [["d00"; "d01"];["d10";"d11"];
                          ["d20";"d21"];["d30";"d31"]]
                          [["out0"; "out1"]] s s' =
       (s' "out0" =
        s "addr0" ∧ s "en" ∧ AH (~s "en") (s "addr0" ∧ s "d00") ∨
        s "addr1" ∧ s "en" ∧ AH (~s "en") (s "addr1" ∧ s "d10")) ∨
        s "addr2" ∧ s "en" ∧ AH (~s "en") (s "addr2" ∧ s "d20")) ∨
        s "addr3" ∧ s "en" ∧ AH (~s "en") (s "addr3" ∧ s "d30")) ∧
       (s' "out1" =
        s "addr0" ∧ s "en" ∧ AH (~s "en") (s "addr0" ∧ s "d01") ∨
        s "addr1" ∧ s "en" ∧ AH (~s "en") (s "addr1" ∧ s "d11")) ∨
        s "addr2" ∧ s "en" ∧ AH (~s "en") (s "addr2" ∧ s "d21")) ∨
        s "addr3" ∧ s "en" ∧ AH (~s "en") (s "addr3" ∧ s "d31")) : thm

val mem_thm = it;

- netlist2exlif memory [mem_thm] "4X2mem" "";
```

9.3.2 Verification

We first illustrate that the memory functions correctly. We do this by verifying it in Forte, running simulations to check the read and the write properties. Since the example has only a few word lines and data bits, running the STE simulations is very easy and poses no significant challenge. However once we scale the number of address lines even to a moderately large number, such as 64, for a 32 bit data word, we soon realise the limit of explicit full scale simulation in STE. So we will present the symmetry based reduction strategy later to show how we can get a handle on the complexity of verification. Interestingly, symmetry reduction by itself is not sufficient to keep the size of the BDDs small, and we will subsequently show how we can leverage the symmetry reduction with symbolic indexing to get fast and efficient verification strategy for SRAM verification. We will also make comparisons to Pandey’s work on SRAM verification.

We wish to verify the following properties about the memory [26]:

1. *Read after write* is correct — if the memory is written at an address, and then read later from the same location, it should give the correct data.

2. *Read is non-destructive* — by reading the data from a certain address, we do not erase the data. In other words the state of the memory stays intact after a read operation.

We begin by stating the following antecedents and consequents, and subsequently run STE simulation runs in Forte. Note we use the convenient *from* and *to*, explained in Section 3.4, for expressing trajectory formulas that hold over several units of time.

$$
\begin{aligned}
&(*\ \texttt{symbolic address values are assigned to the address nodes}\ *)\\
&\mathit{let}\ A\ =\ ((\text{``}\mathit{addr}_0\text{''}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{``}\mathit{addr}_1\text{''}\ \textsf{is}\ a_1)\ \textsf{and}\\
&\qquad\qquad (\text{``}\mathit{addr}_2\text{''}\ \textsf{is}\ a_2)\ \textsf{and}\ (\text{``}\mathit{addr}_3\text{''}\ \textsf{is}\ a_3))\ \mathit{from}\ 0\ \mathit{to}\ 2
\end{aligned}
$$

$$
\begin{aligned}
&(*\ \texttt{formulas that state the data values to be written}\ *)\\
&\mathit{let}\ D_0\ =\ ((\text{``}d_{00}\text{''}\ \textsf{is}\ d_{00})\ \textsf{and}\ (\text{``}d_{10}\text{''}\ \textsf{is}\ d_{10})\ \textsf{and}\\
&\qquad\qquad (\text{``}d_{20}\text{''}\ \textsf{is}\ d_{20})\ \textsf{and}\ (\text{``}d_{30}\text{''}\ \textsf{is}\ d_{30}))\ \mathit{from}\ 0\ \mathit{to}\ 1
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{let}\ D_1\ =\ ((\text{``}d_{01}\text{''}\ \textsf{is}\ d_{01})\ \textsf{and}\ (\text{``}d_{11}\text{''}\ \textsf{is}\ d_{11})\ \textsf{and}\\
&\qquad\qquad (\text{``}d_{21}\text{''}\ \textsf{is}\ d_{21})\ \textsf{and}\ (\text{``}d_{31}\text{''}\ \textsf{is}\ d_{31}))\ \mathit{from}\ 0\ \mathit{to}\ 1
\end{aligned}
$$

$$
\begin{aligned}
&(*\ \texttt{read and write take place in alternate clock cycles}\ *)\\
&\mathit{let}\ \mathit{en}\ =\ (\text{``}\mathit{en}\text{''}\ \textsf{is}\ F\ \mathit{from}\ 0\ \mathit{to}\ 1)\ \textsf{and}\ (\text{``}\mathit{en}\text{''}\ \textsf{is}\ T\ \mathit{from}\ 1\ \mathit{to}\ 2)
\end{aligned}
$$

$$
\begin{aligned}
&(*\ \texttt{data written at the previous cycle appears at the output}\ *)\\
&\mathit{let}\ C_0\ =\ (\text{``}\mathit{out}_0\text{''}\ \textsf{is}\ (a_0\ \wedge\ d_{00})\ \vee\ (a_1\ \wedge\ d_{10})\ \vee\\
&\qquad\qquad\qquad\qquad (a_2\ \wedge\ d_{20})\ \vee\ (a_3\ \wedge\ d_{30}))\ \mathit{from}\ 1\ \mathit{to}\ 2
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{let}\ C_1\ =\ (\text{``}\mathit{out}_1\text{''}\ \textsf{is}\ (a_0\ \wedge\ d_{01})\ \vee\ (a_1\ \wedge\ d_{11})\ \vee\\
&\qquad\qquad\qquad\qquad (a_2\ \wedge\ d_{21})\ \vee\ (a_3\ \wedge\ d_{31}))\ \mathit{from}\ 1\ \mathit{to}\ 2
\end{aligned}
$$

$$
\begin{aligned}
&(*\ \texttt{the state of the memory is intact}\ *)\\
&\mathit{let}\ M\ =\ ((\text{``}n_4\text{''}\ \textsf{is}\ (a_0\ \wedge\ d_{00}))\ \textsf{and}\\
&\qquad\qquad (\text{``}n_5\text{''}\ \textsf{is}\ (a_1\ \wedge\ d_{10}))\ \textsf{and}\\
&\qquad\qquad (\text{``}n_6\text{''}\ \textsf{is}\ (a_2\ \wedge\ d_{20}))\ \textsf{and}\\
&\qquad\qquad (\text{``}n_7\text{''}\ \textsf{is}\ (a_3\ \wedge\ d_{30}))\ \textsf{and}\\
&\qquad\qquad (\text{``}n_{55}\text{''}\ \textsf{is}\ (a_0\ \wedge\ d_{01}))\ \textsf{and}\\
&\qquad\qquad (\text{``}n_{56}\text{''}\ \textsf{is}\ (a_1\ \wedge\ d_{11}))\ \textsf{and}\\
&\qquad\qquad (\text{``}n_{57}\text{''}\ \textsf{is}\ (a_2\ \wedge\ d_{21}))\ \textsf{and}\\
&\qquad\qquad (\text{``}n_{58}\text{''}\ \textsf{is}\ (a_3\ \wedge\ d_{31})))\ \mathit{from}\ 1\ \mathit{to}\ 2
\end{aligned}
$$

The antecedent A simply asserts distinct symbolic values at different address lines. Since we have four word lines, we need four unique BDD variables (a_0 to a_3). Similarly, we need to assert distinct data variables d_{ij} at the incoming data ports of the memory, where i denotes the word location at which the j^{th} bit is written. For this example i ranges between 0 and 3, and j is either 0 or 1. We do this by giving the formulas D_0 and D_1. The names for state nodes (“n_4”, “n_5”, “n_6”, “n_7”, “n_{55}”, “n_{56}”, “n_{57}” and “n_{58}”)

are generated internally by the *netlist2exlif* program that produces the exlif from the netlist as explained in Chapter 7.

We have paired up the 0^{th} bits of each address location together in the formula D_0, and the 1st bits of each word locations in D_1. By using the convenient *from* and *to* functions, we can say that the incoming data bits are asserted the symbolic values at time point 0. The address nodes take on the symbolic values for 2 time points, long enough to complete the read and the write operations. The enable signal is asserted low at time point 0 to allow a data write to word locations, and then asserted high to enable the data read. This is given by the trajectory formula given by en. We construct the trajectory formulas C_0 and C_1, that states the expected values to arrive at the output data ports "out_0" and "out_1", from time point 1 onwards. Finally we provide the formula M, that states that the memory state specified by the state of the internal nodes is intact.

Once we have stated the trajectory formulas, we can start the Forte simulator, and open the file "`4X2mem.exe`" for reading the circuit FSM; we assign the file pointer to the variable *memory* in Forte. Then we can run the following simulation in Forte to verify that *read after write* is correct, and the *read is non-destructive*.

$$memory \models (A \ \mathsf{and}\ D_0 \ \mathsf{and}\ D_1 \ \mathsf{and}\ en) \Rightarrow (C_0 \ \mathsf{and}\ C_1 \ \mathsf{and}\ M)$$

The output of the above run is the BDD true in Forte. Had there been an error in the run, a counterexample would have appeared at the output in the form of a BDD which will have variables mentioned at the address locations and data bits.

In total, we use 12 variables, 4 for address lines, and 8 for the data bits (2 for each address line). For an $m \times n$ memory, the number of variables required is mn, which for a memory of even modest size like an 8K bit, is 2^{10} — a large number. An attempt to verify the read operation for such a memory would result in such a big BDD, that simulation in Forte cannot complete without BDD variable orderings.

Clearly there is a strong need for a reduction strategy in order to manage the complexity of the memory verification. In the sequel below we show a symmetry based reduction methodology. This relies on observing that there is symmetry amongst data bits. What this means is that data bits within each bus (row of the memory) have symmetry. Thus if we permute bits 0 and 1 of the stored data, we will generate a permutation on the columns 0 and 1 of the memory, shown in Figure 9.4. This holds for all the bits between 0 and the bit size of the data. This is good, because it means we can verify one column of the memory and infer by way of symmetry that the other columns have been verified as well.

Verification with symmetry reduction

The symmetry based reduction strategy relies on effective property decomposition using the inference rules shown in Chapter 8. We show our methodology step by step, for the *read after write* property. The methodology is very similar for verifying other properties, and so we will not present it here. Let us begin by stating the antecedents.

$$\begin{aligned} \mathit{let}\ A \ &= \ ((\text{``}\mathit{addr}_0\text{''}\ \textsf{is}\ a_0)\ \textsf{and}\ (\text{``}\mathit{addr}_1\text{''}\ \textsf{is}\ a_1)\ \textsf{and} \\ &\quad\ (\text{``}\mathit{addr}_2\text{''}\ \textsf{is}\ a_2)\ \textsf{and}\ (\text{``}\mathit{addr}_3\text{''}\ \textsf{is}\ a_3))\ \mathit{from}\ 0\ \mathit{to}\ 2 \end{aligned}$$

$$\begin{aligned} \mathit{let}\ D_0 \ &= \ ((\text{``}d_{00}\text{''}\ \textsf{is}\ d_{00})\ \textsf{and}\ (\text{``}d_{10}\text{''}\ \textsf{is}\ d_{10})\ \textsf{and} \\ &\quad\ (\text{``}d_{20}\text{''}\ \textsf{is}\ d_{20})\ \textsf{and}\ (\text{``}d_{30}\text{''}\ \textsf{is}\ d_{30}))\ \mathit{from}\ 0\ \mathit{to}\ 1 \end{aligned}$$

$$\mathit{let}\ en \ = \ (\text{``}en\text{''}\ \textsf{is}\ F\ \mathit{from}\ 0\ \mathit{to}\ 1)\ \textsf{and}\ (\text{``}en\text{''}\ \textsf{is}\ T\ \mathit{from}\ 1\ \mathit{to}\ 2)$$

The formulas A, D_0 and en are the same as shown earlier for the *read after write* property. Data gets written in first cycle and is read in second cycle. We also devise intermediate properties so that we can decompose the overall correctness property in terms of these properties, and deduce the overall correctness statement by composing the verification runs in two steps.

$$\begin{aligned} \mathit{let}\ B_0 \ &= \ (\text{``}n_4\text{''}\ \textsf{is}\ (a_0\ \wedge\ d_{00})\ \mathit{from}\ 1\ \mathit{to}\ 2)\ \textsf{and} \\ &\quad\ (\text{``}n_5\text{''}\ \textsf{is}\ (a_1\ \wedge\ d_{10})\ \mathit{from}\ 1\ \mathit{to}\ 2)\ \textsf{and} \\ &\quad\ (\text{``}n_6\text{''}\ \textsf{is}\ (a_2\ \wedge\ d_{20})\ \mathit{from}\ 1\ \mathit{to}\ 2)\ \textsf{and} \\ &\quad\ (\text{``}n_7\text{''}\ \textsf{is}\ (a3\ \wedge\ d_{30})\ \mathit{from}\ 1\ \mathit{to}\ 2) \end{aligned}$$

$$\begin{aligned} \mathit{let}\ C_0 \ &= \ (\text{``}\mathit{out}_0\text{''}\ \textsf{is}\ (a_0\ \wedge\ d_{00})\ \vee\ (a_1\ \wedge\ d_{10})\ \vee \\ &\qquad\qquad (a_2\ \wedge\ d_{20})\ \vee\ (a_3\ \wedge\ d_{30}))\ \mathit{from}\ 1\ \mathit{to}\ 2 \end{aligned}$$

The weaker property B_0 simply identifies desirable values at intermediate points in the memory circuit. These are not difficult to identify in practice since the memory definition itself explicitly exposes the interfaces of the memory, and allows possibilities to slice the memory into several parts. The consequent C simply states the expected value at the output nodes.

Once all the formulas have been stated, the following three steps help us deduce a statement of correctness that says that the value at the output node "out_0" is either the 0^{th} bit value stored at locations "addr_0", or "addr_1" or "addr_2" or "addr_3".

1. $\mathit{memory}\ \models\ (A\ \textsf{and}\ D_0\ \textsf{and}\ en)\ \Rightarrow\ B_0$ (*STE run*)
2. $\mathit{memory}\ \models\ B_0\ \Rightarrow\ C_0$ (*STE run*)
3. $\mathit{memory}\ \models\ (A\ \textsf{and}\ D_0\ \textsf{and}\ en)\ \Rightarrow\ C_0$ (*Cut on 1 and 2*)

We have verified the property using two STE runs, and using a Cut inference rule. The total number of variables required is 8. The number of variables required to verify scales linearly with the increase in the number of address lines.

Consider the following permutation π. The permutation generate the second column of the memory from the first.

$$\begin{aligned} \pi = \{&\text{``}d_{00}\text{''}\ \rightarrow\ \text{``}d_{01}\text{''},\ \text{``}d_{10}\text{''}\ \rightarrow\ \text{``}d_{11}\text{''}, \\ &\text{``}d_{20}\text{''}\ \rightarrow\ \text{``}d_{21}\text{''},\ \text{``}d_{30}\text{''}\ \rightarrow\ \text{``}d_{31}\text{''}, \\ &\text{``}n_4\text{''}\ \rightarrow\ \text{``}n_{55}\text{''},\ \text{``}n_5\text{''}\ \rightarrow\ \text{``}n_{56}\text{''}, \\ &\text{``}n_6\text{''}\ \rightarrow\ \text{``}n_{57}\text{''},\ \text{``}n_7\text{''}\ \rightarrow\ \text{``}n_{58}\text{''}, \\ &\text{``}\mathit{out}_0\text{''}\ \rightarrow\ \text{``}\mathit{out}_1\text{''}\} \end{aligned}$$

This π when applied to D_0, B_0 and C_0 gives the properties given by D_1, B_1 and C_1 respectively. The antecedent A and en remains unchanged.

$$\begin{aligned}
let\ D_1 \ &= \ ((``d_{01}" \ \mathsf{is} \ d_{01}) \ \mathsf{and} \ (``d_{11}" \ \mathsf{is} \ d_{11}) \ \mathsf{and} \\
&\quad\ (``d_{21}" \ \mathsf{is} \ d_{21}) \ \mathsf{and} \ (``d_{31}" \ \mathsf{is} \ d_{31})) \ from\ 0\ to\ 1 \\
let\ B_1 \ &= \ (``n_{55}" \ \mathsf{is} \ (a_0 \ \wedge \ d_{01}) \ from\ 1\ to\ 2) \ \mathsf{and} \\
&\quad\ (``n_{56}" \ \mathsf{is} \ (a_1 \ \wedge \ d_{11}) \ from\ 1\ to\ 2) \ \mathsf{and} \\
&\quad\ (``n_{57}" \ \mathsf{is} \ (a_2 \ \wedge \ d_{21}) \ from\ 1\ to\ 2) \ \mathsf{and} \\
&\quad\ (``n_{58}" \ \mathsf{is} \ (a3 \ \wedge \ d_{31}) \ from\ 1\ to\ 2) \\
let\ C_1 \ &= \ (``out_1" \ \mathsf{is} \ (a_0 \ \wedge \ d_{01}) \ \vee \ (a_1 \ \wedge \ d_{11}) \ \vee \\
&\qquad\qquad (a_2 \ \wedge \ d_{21}) \ \vee \ (a_3 \ \wedge \ d_{31})) \ from\ 1\ to\ 2
\end{aligned}$$

We do not have to do the following step, but infer the overall statement of correctness shown below from symmetry based reduction argument, using the symmetry soundness theorem (Theorem 8.1).

$$memory \ \models \ (A \ \mathsf{and} \ D_1 \ \mathsf{and} \ en) \ \Rightarrow \ C_1 \quad \text{(II)}$$

Thus the overall correctness statement can be deduced by using the Conjunction rule of inference. Thus we obtain the overall correctness statement shown below:

$$memory \ \models \ (A \ \mathsf{and} \ D_0 \ \mathsf{and} \ D_1 \ \mathsf{and} \ en) \ \Rightarrow \ (C_0 \ \mathsf{and} \ C_1) \quad \text{(III)}$$

Thus, irrespective of the size of the *data bits* in the RAM, we always require a fixed number of BDD variables. However the requirement scales linearly with number of address lines. Thus for an address bus of width 16, with 64K address lines, 32 bit data bus, size of the memory is 2M bits, the number of variable required to verify the memory is $2 \ \times \ 64K$ BDD variables. This is a large number of BDD variables and in fact we cannot complete the simulation runs for this size of memory in Forte even using symmetry based reduction. We therefore need to use symbolic indexing based reduction technique on top of symmetry based reduction. Note that symmetry offers a reduction in the number of columns that need to be verified. By verifying only one column, we are able to deduce correctness statements about other columns. For each column, we need m distinct address variables for m address lines, and m distinct data variables, one for each data bit stored in each of the m locations.

By using symbolic indexing, we can collapse the number of BDD variables required in a single column verification from $2m$ to $(2\ log_2\ m)$. In the sequel below we show how to verify our memory example by exploiting symmetry reduction and symbolic indexing.

Verification with symmetry reduction and symbolic indexing

We will show how to verify the *read after write* property by using symbolic indexing with symmetry reduction. Verifying other properties is similar, so will not be shown here.

We assume the existence of new symbolic variables p, q, r and s. The four distinct address locations can be indexed by a combination of two symbolic variables p and q. The distinct variables stored at the four different locations can be indexed by the combination of r and s. The antecedent A specifies the assignment of address variables to address nodes in the usual way. Similarly data variables are assigned to the data input nodes of the first column.

$$\begin{aligned} let\ A \ = \ & ((\text{“}addr_0\text{”} \ \mathsf{is} \ (p \wedge q)) \ \mathsf{and} \ (\text{“}addr_1\text{”} \ \mathsf{is} \ (\sim p \wedge q)) \ \mathsf{and} \\ & (\text{“}addr_2\text{”} \ \mathsf{is} \ (p \wedge \sim q)) \ \mathsf{and} \ (\text{“}addr_3\text{”} \ \mathsf{is} \ (\sim p \wedge \sim q))) \ from\ 0\ to\ 2 \end{aligned}$$

$$\begin{aligned} let\ D_0 \ = \ & ((\text{“}d_{00}\text{”} \ \mathsf{is} \ (r \wedge s)) \ \mathsf{and} \ (\text{“}d_{10}\text{”} \ \mathsf{is} \ (\sim r \wedge s)) \ \mathsf{and} \\ & (\text{“}d_{20}\text{”} \ \mathsf{is} \ (r \wedge \sim s)) \ \mathsf{and} \ (\text{“}d_{30}\text{”} \ \mathsf{is} \ (\sim r \wedge \sim s))) \ from\ 0\ to\ 1 \end{aligned}$$

Read and write is enabled by stating the formula en.

$$let\ en \ = \ (\text{“}en\text{”} \ \mathsf{is} \ F\ from\ 0\ to\ 1) \ \mathsf{and} \ (\text{“}en\text{”} \ \mathsf{is} \ T\ from\ 1\ to\ 2)$$

Formula assigning symbolic vlaues to state nodes is stated next.

$$\begin{aligned} let\ B_0 \ = \ & (\text{“}n_4\text{”} \ \mathsf{is} \ ((p \wedge q) \wedge (r \wedge s))\ from\ 1\ to\ 2) \ \mathsf{and} \\ & (\text{“}n_5\text{”} \ \mathsf{is} \ ((\sim p \wedge q) \wedge (\sim r \wedge s))\ from\ 1\ to\ 2) \ \mathsf{and} \\ & (\text{“}n_6\text{”} \ \mathsf{is} \ ((p \wedge \sim q) \wedge (r \wedge \sim s))\ from\ 1\ to\ 2) \ \mathsf{and} \\ & (\text{“}n_7\text{”} \ \mathsf{is} \ ((\sim p \wedge \sim q) \wedge (\sim r \wedge \sim s))\ from\ 1\ to\ 2) \end{aligned}$$

Finally the expected output values are stated as another formula C_0.

$$\begin{aligned} let\ C_0 \ = \ (\text{“}out_0\text{”} \ \mathsf{is} \ & ((p \wedge q) \wedge (r \wedge s)) \vee \\ & ((\sim p \wedge q) \wedge (\sim r \wedge s)) \vee \\ & ((p \wedge \sim q) \wedge (r \wedge s)) \vee \\ & ((\sim p \wedge \sim q) \wedge (\sim r \wedge \sim s)))\ from\ 1\ to\ 2 \end{aligned}$$

The verification steps are outlined below.

1. $memory \models (A \ \mathsf{and} \ D_0 \ \mathsf{and} \ en) \Rightarrow B_0 \quad (STE\ run)$
2. $memory \models B_0 \Rightarrow C_0 \quad (STE\ run)$
3. $memory \models (A \ \mathsf{and} \ D_0 \ \mathsf{and} \ en) \Rightarrow C_0 \quad (Cut\ on\ 1\ and\ 2)$

Thus we have verified the following property, running two STE runs, using 4 BDD variables, rather than 8 as it was the case with the symmetry reduction approach shown above.

$$memory \models (A \ \mathsf{and} \ D_0 \ \mathsf{and} \ en) \Rightarrow C_0 \quad \text{(IV)}$$

By symmetry we can deduce the correctness statement for the second column of the memory, using $2(log_2\ m)$ BDD variables, which in this case is 4. By permutation π we obtain the following properties:

$$\begin{aligned} let\ A \ = \ & ((\text{“}addr_0\text{”} \ \mathsf{is} \ (p \wedge q)) \ \mathsf{and} \ (\text{“}addr_1\text{”} \ \mathsf{is} \ (\sim p \wedge q)) \ \mathsf{and} \\ & (\text{“}addr_2\text{”} \ \mathsf{is} \ (p \wedge \sim q)) \ \mathsf{and} \ (\text{“}addr_3\text{”} \ \mathsf{is} \ (\sim p \wedge \sim q))) \ from\ 0\ to\ 2 \end{aligned}$$

$$\begin{aligned} let\ D_1 \ = \ & ((\text{“}d_{01}\text{”} \ \mathsf{is} \ (r \wedge s)) \ \mathsf{and} \ (\text{“}d_{11}\text{”} \ \mathsf{is} \ (\sim r \wedge s)) \ \mathsf{and} \\ & (\text{“}d_{21}\text{”} \ \mathsf{is} \ (r \wedge \sim s)) \ \mathsf{and} \ (\text{“}d_{31}\text{”} \ \mathsf{is} \ (\sim r \wedge \sim s))) \ from\ 0\ to\ 1 \end{aligned}$$

$$let\ en\ =\ (\text{“}en\text{”}\ \mathsf{is}\ F\ from\ 0\ to\ 1)\ \mathsf{and}\ (\text{“}en\text{”}\ \mathsf{is}\ T\ from\ 1\ to\ 2)$$

$$\begin{aligned} let\ B_1\ =\ & (\text{“}n_{55}\text{”}\ \mathsf{is}\ ((p\ \wedge\ q)\ \wedge\ (r\ \wedge\ s))\ from\ 1\ to\ 2)\ \mathsf{and} \\ & (\text{“}n_{56}\text{”}\ \mathsf{is}\ ((\sim p\ \wedge\ q)\ \wedge\ (\sim r\ \wedge\ s))\ from\ 1\ to\ 2)\ \mathsf{and} \\ & (\text{“}n_{57}\text{”}\ \mathsf{is}\ ((p \wedge\ \sim q)\ \wedge\ (r \wedge\ \sim s))\ from\ 1\ to\ 2)\ \mathsf{and} \\ & (\text{“}n_{58}\text{”}\ \mathsf{is}\ ((\sim p \wedge\ \sim q)\ \wedge\ (\sim r \wedge\ \sim s))\ from\ 1\ to\ 2) \end{aligned}$$

$$\begin{aligned} let\ C_1\ =\ (\text{“}out_1\text{”}\ \mathsf{is}\ & ((p\ \wedge\ q)\ \wedge\ (r\ \wedge\ s)) \vee \\ & ((\sim p\ \wedge\ q)\ \wedge\ (\sim r\ \wedge\ s)) \vee \\ & ((p \wedge\ \sim q)\ \wedge\ (r\ \wedge s)) \vee \\ & ((\sim p \wedge\ \sim q)\ \wedge\ (\sim r \wedge\ \sim s)))\ from\ 1\ to\ 2 \end{aligned}$$

and using Theorem 8.1, we can obtain the following correctness statement, without having to explicitly run the property in the STE simulator.

$$memory\ \models\ (A\ \mathsf{and}\ D_1\ \mathsf{and}\ en)\ \Rightarrow\ C_1\quad \text{(V)}$$

Now by doing a Conjunction we obtain:

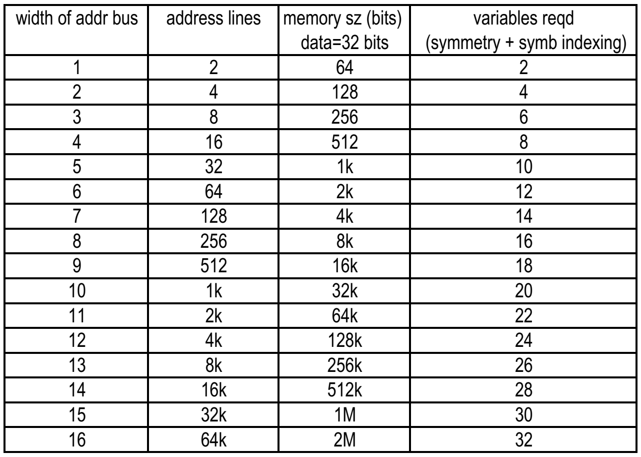
$$memory\ \models\ (A\ \mathsf{and}\ D_0\ \mathsf{and}\ D_1\ \mathsf{and}\ en)\ \Rightarrow\ (C_0\ \mathsf{and}\ C_1)\quad \text{(VI)}$$

9.3.3 Discussion

We have shown that using symbolic indexing, we are able to get a logarithmic reduction in the number of variables required to complete the verification of one column, and by way of symmetry, memory of any number of data bits requires only a fixed number of BDD variables, which is given by ($log_2\, m$), for m address lines. Thus a memory that stores 32 bit data words requires the same number of BDD variables as the one that stores 8-bit or 16-bit words. The table shown in Figure 9.5, shows the variable requirement with the increase in the number of address lines, using symmetry and symbolic indexing. Notice that we can verify a 2M bit memory using only 32 symbolic variables, which means the resulting BDDS stay very compact. Figure 9.6, shows how the BDD variable requirement scales with different size of the memory, using our approach combining symmetry and symbolic indexing.

Pandey verified SRAMS using STE and symmetry based reduction [82]. Figure 9.6 shows the variable requirement for different SRAM configurations using our approach, while Figure 9.7 shows the verification memory requirement using Pandey's approach [82]. Note that Pandey did not specify the metrics in terms of BDD variables used, but specified the memory used in verification which he refers to as verification memory. From the graph shown in Figure 9.6, Pandey's verification memory requirement seems to scale nearly linearly for smaller sizes of SRAMS, but the verification memory used, grows non-linearly as the size of SRAMS increases.

Pandey modelled SRAMS using a transistor level netlist representation, and used data symmetries besides using structural symmetries. He argued that decoder has data symmetries and some reduction can be obtained by using that. We only use structural symmetry amongst data bits, and augment it with symbolic indexing, to achieve a fast



width of addr bus	address lines	memory sz (bits) data=32 bits	variables reqd (symmetry + symb indexing)
1	2	64	2
2	4	128	4
3	8	256	6
4	16	512	8
5	32	1k	10
6	64	2k	12
7	128	4k	14
8	256	8k	16
9	512	16k	18
10	1k	32k	20
11	2k	64k	22
12	4k	128k	24
13	8k	256k	26
14	16k	512k	28
15	32k	1M	30
16	64k	2M	32

Figure 9.5: Table showing number of variables required for memory verification of 32 bit data word and varying number of word lines, using our approach, combining symmetry reduction and symbolic indexing.

and efficient verification methodology. We do rely on using deductive inference rules to handle property decomposition and composing overall correctness statements; which one can argue adds to the cost of overall verification, and this cost scales linearly with the size of SRAM. Pandey also used some deductive support in his verification case studies.

Most significant difference in our approach and that of Pandey's is in identifying symmetries in circuits. Pandey used subgraph-isomorphism to discover symmetry in flat netlists, which meant he had to rely on special heuristics to help him solve an NP-complete problem. This meant that enormous amount of time was required consuming plenty of memory and having to discover different heuristics for different circuits (see page 76-78, Chapter 3, Pandey's thesis [82]).

Our method of symmetry detection is fast since type checking is independent of the size of the RAM. Besides, structured model design, gives us a general method of circuit design which encourages re-use of circuit component definitions.

9.4 Content Addressable Memories

Most memory devices store and retrieve data by addressing specific memory locations. As a result, this path often becomes the limiting factor for systems that rely on fast memory access. The time required to find an item stored in memory can be reduced considerably if the stored data item can be identified for access by the content of the data itself rather than by its address. Memory that is accessed in this way is called content-addressable memory (CAM). CAM is ideally suited for several functions, including ethernet address lookup, data compression, pattern-recognition, cache tags, high-bandwidth

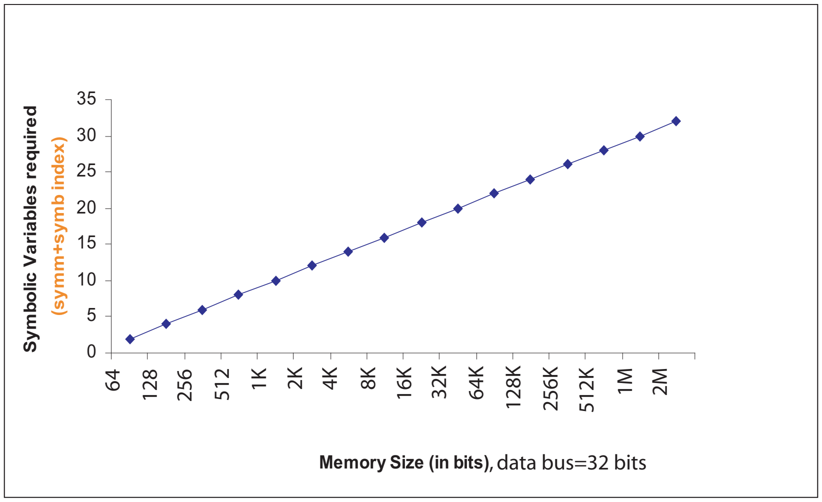


Figure 9.6: Graph showing BDD variable requirement for verifying different sizes of memory, using our approach, combining symmetry and symbolic indexing.

address filtering, fast lookup in routers. In the next section we show the modelling of a CAM in HOL using the abstract data type functions we presented in Chapter 6.

9.4.1 Modelling

At the heart of any CAM is a tag comparison unit. This is very different from any regular memory designs, since the incoming address is compared in parallel with all the all the stored addresses, which are called *tags*. As we mentioned above, in case of a CAM the tags themselves are a part of the stored data, and in fact we look for the content by looking for the stored data rather than the address in the conventional sense as in the case of a SRAM. Since the incoming tags are compared in parallel with the stored tags of n lines, we require n comparators. The stored tag themselves can be m bit wide, and the associated data to be fetched at any corresponding tag can be d bit

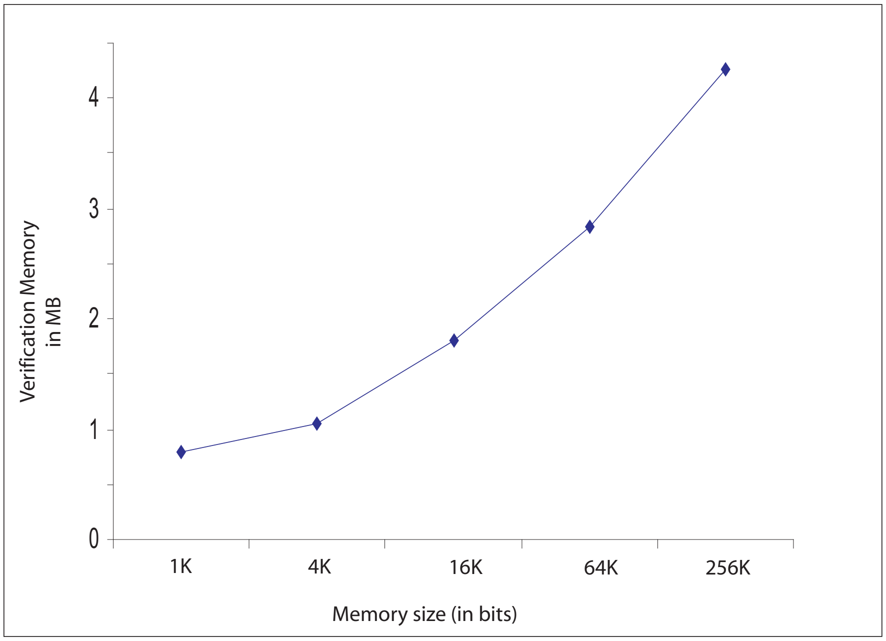


Figure 9.7: Verification memory required for verifying SRAMS of different sizes, using Pandey's approach, taken from [82].

wide. Thus CAM is three-dimensional in this sense.

$$
\begin{array}{l}
(\mathit{tcomparator}\ 0\ [\,[en]\,]\ =\ \mathit{Null})\\
(\mathit{tcomparator}\ (n+1)\ [\,[en]\,]\ =\\
\qquad\qquad \mathit{let\ cand\ \ inp}\ =\ \mathit{map}\ \ (\wedge\ (\mathit{hd\ inp}))\ \mathit{in}\\
\qquad\qquad \mathit{let\ comp}\ =\ \mathit{Bitwise\ xnor\ Id\ in}\\
\qquad\qquad \mathit{let\ intags}\ =\ (\mathit{Select}\ 0\ \mathit{Id})\ \mathit{in}\\
\qquad\qquad \mathit{let\ storedtags}\ =\ ((\mathit{map}(\mathit{cand}\ [en]))\ \circ\ (\mathit{map}(\mathit{map}(\mathit{AH}\ \ {\sim}en)))\ \circ\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Select}\ (n+1)\ \mathit{Id}))\\
\qquad\qquad\quad \mathit{in}\\
\qquad\qquad\quad (\mathit{And}\ \circ\ \mathit{comp}\ \circ\ (\mathit{intags}\ \parallel\ \mathit{storedtags}))\ \parallel\ (\mathit{tcomparator}\ n\ [\,[en]\,]))
\end{array}
$$

The function *tcomparator* builds the comparison circuitry in parallel, using the $\parallel$ combinator. It fetches the first line containing stored tags and build the circuitry that compares it with the incoming tags and then recursively walks down the list of remaining stored tags and compares it with them.

The match outcome from each line, is the logically or-ed to obtain the hit from the tags match. This is given by the function *hit*.

$$\mathit{hit}\ n\ \mathit{nsym}\ =\ \mathit{Bitwise}\ (\vee)\ (\mathit{tcomparator}\ n\ \mathit{nsym})$$

The function *camLines* generates the lines of the CAM given the address lines that are generated from the tag match circuitry.

```
(camLines[ ] = Null)
(camLines(a :: addrlist) =
          let cand  inp  =  map  (∧ (hd inp)) in
          let n  =  (length (a :: addrlist)  −  1)
                    in
                    (camLines addrlist)  ∥  (map (cand a)  ∘  (Select n Id)))
```

The CAM circuit model is given by the function *cam* below. It obtains the tag enable and data enable inputs, and given the output from the tag match circuitry, it wires the complete CAM from the data lines and tag lines.

```
cam  nsym =
          let tagen  =  (hd  ∘  hd) ((Select 0 Id) nsym)  in
          let cand  inp  =  map  (∧ (hd inp)) in
          let dataen  =  (hd  ∘  hd) ((Select 1 Id) nsym)  in
          let n  =  length (Id nsym)  −  3  in
          let match  =  tcomparator n [ [tagen] ] (((Tail  ∘  Tail) Id) nsym)  in
          let data  =  (camLines match)  ∘  (map(cand [dataen])) ∘
                                          (map(map(AH(∼dataen))))  ∘  Id
                         in  (Bitwise (∨) data)
```

In Figure 9.8 (see next page), we show an n line CAM with m bit tag and d bit data. The figure shows an abstract functional version of CAM where we have hidden the tag and data enable signals. Figure 9.9 (see next page) shows the tag comparison circuitry and shows how the hit is obtained from the match of the incoming tag with any of the stored tags.

The exlif and the exe files are obtained by using the flow explained for the SRAM circuit and other examples. The output of this flow is an exe that can be loaded in Forte for simulation.

We will now explain how we carry out the verification of CAM using symmetry reduction, and inference rules.

9.4.2 Verification

The verification session begins by declaring the symbolic (BDD) variables in a Forte session. We therefore assume the existence of BDD variables for the tag and data lines. We assume that the CAM has two lines, with two bit tag and two bit data. This is sufficient to show the principle involved in the verification using symmetry; any larger CAM can be verified using exactly the same principle.

Having declared the tag and data variables, we state the basic antecedent assertion that assigns the symbolic incoming tag values to the incoming tag nodes and defines how the tag and data enable signal waveforms determine the read and the write operations. To enable a write operation to take place, we assert the *tagen* and *dataen* low, and when they are asserted high the CAM read takes place. The CAM read operation is

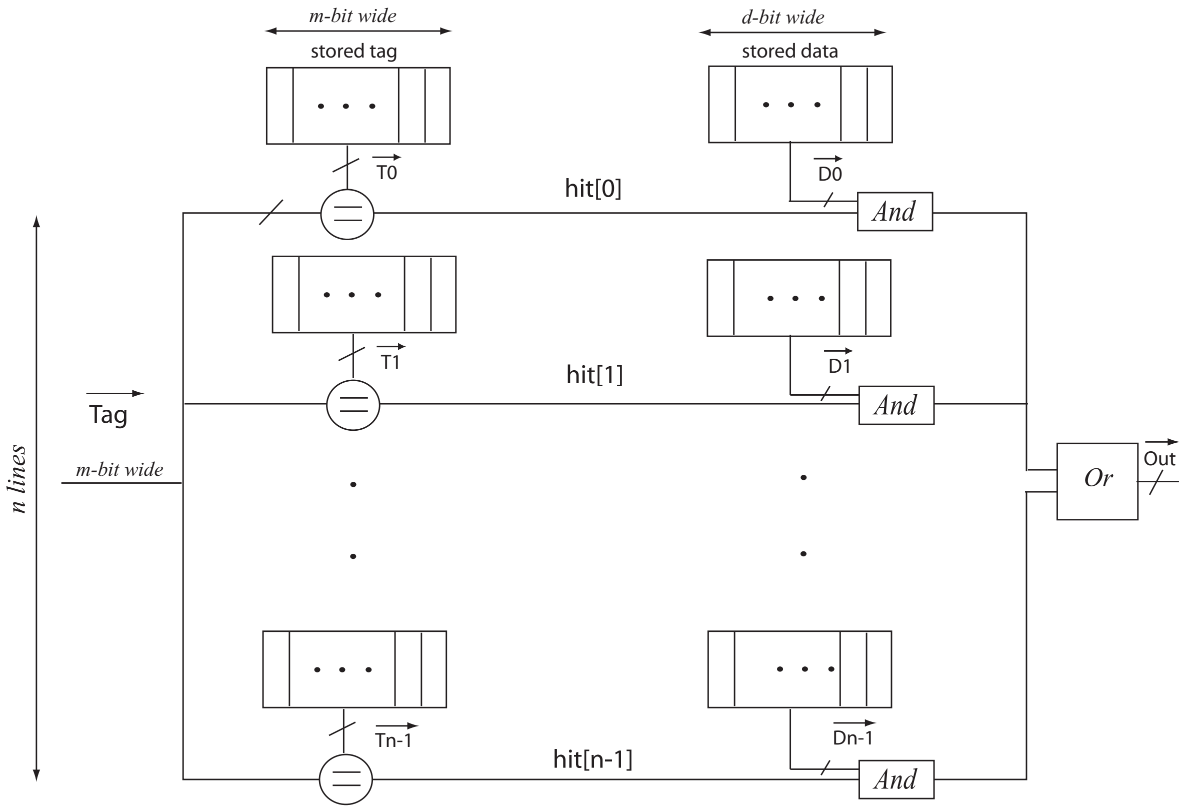


Figure 9.8: An n line CAM with m bit wide tag and d bit data.

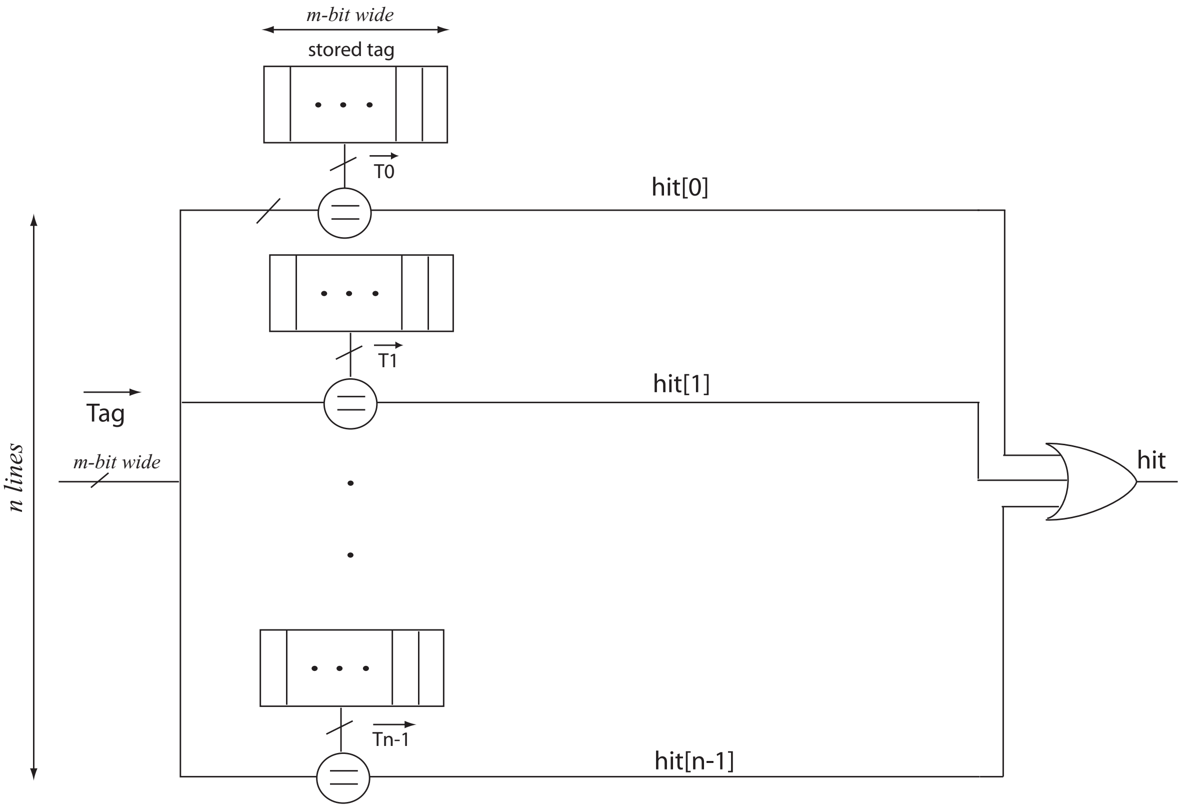


Figure 9.9: An n line CAM showing the hit circuitry.

also known as the associative read operation, since the tags are compared in parallel. We also assume that if the tags match, they match at most one line in the CAM. This is an invariant that is often desired in many CAMs [85].

$$
\begin{aligned}
&(*\ \texttt{read enabled and incoming tag takes on symbolic values}\ *)\\
&\textit{let base\_ant} \;=\; (((\text{``}Tag[0]\text{''}\ \textsf{is}\ Tag_0)\ \textsf{and}\ (\text{``}Tag[1]\text{''}\ \textsf{is}\ Tag_1))\ \textit{from}\ 0\ \textit{to}\ 2)\ \textsf{and}\\
&\qquad\qquad\quad (\text{``}tagen\text{''}\ \textsf{is}\ F\ \textit{from}\ 0\ \textit{to}\ 1)\ \textsf{and}\ (\text{``}tagen\text{''}\ \textsf{is}\ T\ \textit{from}\ 1\ \textit{to}\ 2)\ \textsf{and}\\
&\qquad\qquad\quad (\text{``}dataen\text{''}\ \textsf{is}\ F\ \textit{from}\ 0\ \textit{to}\ 1)\ \textsf{and}\ (\text{``}dataen\text{''}\ \textsf{is}\ T\ \textit{from}\ 1\ \textit{to}\ 2)
\end{aligned}
$$

We then state that the first and second lines are initialised with the symbolic values, and the expected value at the output nodes can be from any of the two lines.

$$
\begin{aligned}
&(*\ \texttt{populate the first line}\ *)\\
&\textit{let}\ A_0 \;=\; (((\text{``}T_0[0]\text{''}\ \textsf{is}\ t_{00})\ \textsf{and}\ (\text{``}T_0[1]\text{''}\ \textsf{is}\ t_{01})\ \textsf{and}\\
&\qquad\quad (\text{``}D_0[0]\text{''}\ \textsf{is}\ d_{00})\ \textsf{and}\ (\text{``}D_0[1]\text{''}\ \textsf{is}\ d_{01}))\ \textit{from}\ 0\ \textit{to}\ 1)\\
&\qquad\quad \textsf{and}\ \textit{base\_ant}
\end{aligned}
$$

$$
\begin{aligned}
&(*\ \texttt{populate the second line}\ *)\\
&\textit{let}\ A_1 \;=\; (((\text{``}T_1[0]\text{''}\ \textsf{is}\ t_{10})\ \textsf{and}\ (\text{``}T_1[1]\text{''}\ \textsf{is}\ t_{11})\ \textsf{and}\\
&\qquad\quad (\text{``}D_1[0]\text{''}\ \textsf{is}\ d_{10})\ \textsf{and}\ (\text{``}D_1[1]\text{''}\ \textsf{is}\ d_{11}))\ \textit{from}\ 0\ \textit{to}\ 1)\\
&\qquad\quad \textsf{and}\ \textit{base\_ant}
\end{aligned}
$$

Formulas stating that the stored data appears from the 1st and the 2nd line is given below.

$$
\begin{aligned}
&(*\ \texttt{data stored at the 1st line appears at the output}*\ )\\
&\textit{let}\ C_0 \;=\; ((\text{``}out[0]\text{''}\ \textsf{is}\ d_{00})\ \textsf{and}\ (\text{``}out[1]\text{''}\ \textsf{is}\ d_{01}))\ \textit{from}\ 1\ \textit{to}\ 2
\end{aligned}
$$

$$
\begin{aligned}
&(*\ \texttt{data stored at the 2nd line appears at the output}\ *)\\
&\textit{let}\ C_1 \;=\; ((\text{``}out[0]\text{''}\ \textsf{is}\ d_{10})\ \textsf{and}\ (\text{``}out[1]\text{''}\ \textsf{is}\ d_{11}))\ \textit{from}\ 1\ \textit{to}\ 2
\end{aligned}
$$

The incoming tags match at the first line, or the tags match at the second line. We state this by defining the symbolic formula below. The case when the tags do not match is similarly stated.

$$
\begin{aligned}
&(*\ \texttt{incoming tags match the tags stored at the 1st line}\ *)\\
&\textit{let}\ G_0 \;=\; (Tag_0 \;=\; t_{00})\ \wedge\ (Tag_1 \;=\; t_{01})
\end{aligned}
$$

$$
\begin{aligned}
&(*\ \texttt{incoming tags match the tags stored at the 2nd line}\ *)\\
&\textit{let}\ G_1 \;=\; (Tag_0 \;=\; t_{10})\ \wedge\ (Tag_1 \;=\; t_{11})
\end{aligned}
$$

$$
\begin{aligned}
&(*\ \texttt{incoming tags do not match the tags stored at the 1st line}\ *)\\
&\textit{let}\ nG_0 \;= \sim G_0
\end{aligned}
$$

$$
\begin{aligned}
&(*\ \texttt{incoming tags do not match the tags stored at the 2nd line}\ *)\\
&\textit{let}\ nG_1 \;= \sim G_1
\end{aligned}
$$

Then we state the final formula that the hit from each line stays low.

$(* \;\texttt{hit}[0] \;\texttt{is}\; 0 \;*)$
$let\ B_0 \ = \ \text{“}hit[0]\text{”} \ \textsf{is} \ F\ from\ 1\ to\ 2$

$(* \;\texttt{hit}[1] \;\texttt{is}\; 0 \;*)$
$let\ B_1 \ = \ \text{“}hit[1]\text{”} \ \textsf{is} \ F\ from\ 1\ to\ 2$

Correct Data is read

In this section we outline the steps involved in verifying the read property.

1. $nG_0 \supset (cam \models A_0 \Rightarrow B_0)$ (*STE run, comparator verification strategy, 2 variables for tag and 1 for data*)
2. $G_1 \supset (cam \models (B_0 \ \textsf{and}\ A_1) \Rightarrow C_1)$ (*STE run, using one data variable at a time*)
3. $(nG_0 \wedge G_1) \supset (cam \models (A_0 \ \textsf{and}\ A_1) \Rightarrow C_1)$ (*Cut on 1 and 2*)
4. $cam \models (A_0 \ \textsf{and}\ A_1) \Rightarrow (C_1 \ \textsf{when}\ (nG_0 \wedge G_1))$ (*Constraint Implication 2*)

Major reduction in the BDD size comes from identifying that the tag comparison circuitry is an instance of the comparator circuit, and we can apply the reduction strategy of the comparator to the tag comparison property stated in (1). Thus we only ever need to use 2 BDD variables as it was the case with the comparator, and because tags have symmetry, we can use the symmetry soundness theorem and inference rules to deduce the overall correctness statement stated in (1). We have not shown the micro steps involved in this since they are merely the repetition of the comparator case. Similarly the data bits have symmetry, so verifying (2) also entails verifying a property with just one data variable and using symmetry soundness theorem and inference rules to deduce (2). Using Cut, we are able to deal with only one line at a time. Thus we can prove the weaker property that if tags don't match at the first line, then hit stays low at the first line (1), and then if it is so and tags match at the second line, then the output nodes carry the data values stored at the second line (2). Constraint Implication 2 finally transforms the property into a shape we are want to verify.

We repeat the above steps for the second line (Steps 5-8, see below), now maintaining the invariant that the data only matches at the first line. This process is repeated for as many lines as there are in the CAM, and in fact a script in FL (the language of Forte) easily automates this.

5. $nG_1 \supset (cam \models A_1 \Rightarrow B_1)$ (*STE run, comparator verification strategy, 2 variables for tag and 1 for data*)
6. $G_0 \supset (cam \models (B_1 \ \textsf{and}\ A_0) \Rightarrow C_0)$ (*STE run, using one data variable at a time*)
7. $(nG_1 \wedge G_0) \supset (cam \models (A_0 \ \textsf{and}\ A_1) \Rightarrow C_0)$ (*Cut on 5 and 6*)
8. $cam \models (A_0 \ \textsf{and}\ A_1) \Rightarrow (C_0 \ \textsf{when}\ (nG_1 \wedge G_0))$ (*Constraint Implication 2*)

Finally, we use the Conjunction rule to combine the correctness statements of (4) and (8) to deduce the correct data read property of the CAM that we wish to verify.

$$\begin{aligned} &\mathit{cam} \models (A_0 \ \textsf{and} \ A_1) \Rightarrow \\ &\qquad ((C_0 \ \textsf{when} \ (nG_1 \wedge G_0)) \ \textsf{and} \\ &\qquad\qquad (C_1 \ \textsf{when} \ (nG_0 \wedge G_1))) \quad \text{(Conjunction rule on 4 and 8)} \end{aligned}$$

Hit rises if there is a match

Now we explain how we verify the property that hit rises if there is a match at any of the tag lines. The strategy is similar to the comparator verification case. We first state the assertion that hit rises.

$(*$ `hit is 1` $*)$
$\mathit{let}\ C = \text{“}\mathit{hit}\text{”} \ \textsf{is} \ T \ \mathit{from}\ 1\ \mathit{to}\ 2$

Then the steps are as follows:

9. $G_0 \supset (\mathit{cam} \models A_0 \Rightarrow C)$ (*STE run, comparator verification strategy, 2 variables for tag*)
10. $G_1 \supset (\mathit{cam} \models A_1 \Rightarrow C)$ (*STE run, comparator verification strategy, 2 variables for tag*)
11. $(G_0 \vee G_1) \supset (\mathit{cam} \models (A_0 \ \textsf{and} \ A_1) \Rightarrow C)$ (*Guard Disjunction*)
12. $(\mathit{cam} \models (A_0 \ \textsf{and} \ A_1) \Rightarrow (C \ \textsf{when} \ (G_0 \vee G_1)))$ (*Constraint Implication 2*)

Thus (9) and (10) can be verified using 2 variables for tag. Using Guard Disjunction rule we can compose the overall statement of correctness in (11). We can then transform (11) to a shape of the property (12) we wish to verify, using the Constraint Implication 2 rule. Thus hit rises if there is a match at any of the lines.

Hit stays low if there is no match

Finally, we want to establish that the hit stays low if there is no match. We begin by defining new assertions below.

$(*$ `hit[0] is 0` $*)$
$\mathit{let}\ \mathit{hit}_0 = \text{“}\mathit{hit}[0]\text{”} \ \textsf{is} \ F \ \mathit{from}\ 1\ \mathit{to}\ 2$

$(*$ `hit[1] is 0` $*)$
$\mathit{let}\ \mathit{hit}_1 = \text{“}\mathit{hit}[1]\text{”} \ \textsf{is} \ F \ \mathit{from}\ 1\ \mathit{to}\ 2$

$(*$ `hit is 0` $*)$
$\mathit{let}\ C = \text{“}\mathit{hit}\text{”} \ \textsf{is} \ F \ \mathit{from}\ 1\ \mathit{to}\ 2$

We verify the property that if the tags don't match at the first line then the corresponding hit at that line “*hit*[0]” stays low. Similar property is verified for the other hit “*hit*[1]”. Both of these require only two variables, since they are gain an instance of the comparator case. Using Guard Conjunction, we prove the weaker property that the hits from each line stays low when there is no match found at either locations. Then a scalar run stating that if the hit from each line stays low, the final outcome is that the hit signal "*hit*" stays low. This is of course very fast since it has no BDD variables.

13. $nG_0 \supset (cam \models A_0 \Rightarrow hit_0)$ (*STE run, comparator verification strategy, 2 variables for tag*)
14. $nG_1 \supset (cam \models A_1 \Rightarrow hit_1)$ (*STE run, comparator verification strategy, 2 variables for tag*)
15. $(nG_0 \wedge nG_1) \supset$
$(cam \models (A_0 \text{ and } A_1) \Rightarrow$
$(hit_0 \text{ and } hit_1))$ (*Guard Conjunction on 12 and 13*)
16. $cam \models (hit_0 \text{ and } hit_1) \Rightarrow C$ (*Scalar STE run*)
17. $(nG_0 \wedge nG_1) \supset$
$(cam \models (A_0 \text{ and } A_1) \Rightarrow C)$ (*Cut on 15 and 16*)
18. $cam \models (A_0 \text{ and } A_1) \Rightarrow$
$(C \text{ when } (nG_0 \wedge nG_1))$ (*Constraint Implication 2*)

Usage of the Cut rule allows us to combine the properties so that we can deduce that "*hit*" stays low when the tags don't match at any line. As a last step using Constraint Implication 2 rule, we deduce the property we wish to verify.

9.4.3 Discussion

Our memory and time requirement

For the *correct data read* property a CAM with n lines, with tag width t and data width d, we need to use only *two* variables at any one time for tag comparison and *one* extra variable for data bit, to verify the correct data read property. The space complexity is reduced from $n * (t + d) + t$ to 3. However, the time complexity is linear with respect to the number of CAM lines.

For verifying the *hit logic*, we exploit the symmetries amongst the tags, and we need only two variables at any point of time, for any number of CAM lines, tag entries and data entries! The time complexity is linear with respect to the number of CAM lines.

Pandey's CAM encoding requires $\log_2 n + n * \log_2 t + t + d$ variables for verification of data read and hit logic. Symmetry is not used at all, only symbolic indexing is used. For a 64 line CAM with 32 bit tags and 32 bit data, he would need 6+(64*5)+32+32=390 variables, whereas we would need 3 for correct data read property (2 for tags, and 1 for data), and 2 for just verifying the hit logic. Figure 9.10 shows the comparison of the number of BDD variables needed by Pandey, and by our approach, for varying number of CAM lines, for 4 bit wide tag and data. Figure 9.11 shows the comparison of the number of BDD variables needed by Pandey, and by our approach, when the tag size is varied, when the number of CAM lines is 4, and 4 bit wide data. Figure 9.12 shows the comparison of BDD variables needed by Pandey, and by our approach, when the data bits are varied for a CAM of 4 lines and 4 bit wide data. Notice that our variable requirement stays fixed at 3 for data read property (2 for tags, and 1 for data), and 2 for hit verification property.

The efficiency we gain through our approach comes at the cost of incorporating deduction with STE model checking. This can be seen as a limiting effect of our approach

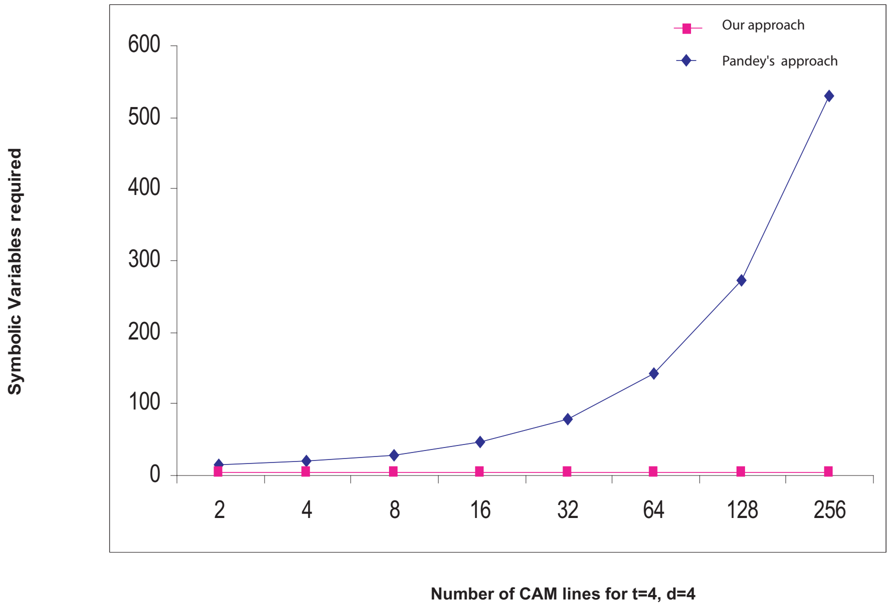


Figure 9.10: BDD variables required with respect to CAM lines. Our requirement stays fixed at 3, whereas Pandey's scales logarithmically.

since compared to fully automated model checking, our approach is an interactive one. The efficiency in our case depends on effective property decomposition. If we decompose them in a manner that does not expose the symmetric properties, then we cannot do much with the symmetry reduction method. The circuit model design often gives very useful insights into how to decompose the properties, and how to identify internal node names that are crucial in defining weaker properties, that are used when we use the Cut rule. As it was the case with the SRAM, detection of symmetry in the case of CAM was easily done by type checking the *cam* definition.

Although we are able to use property decomposition, and symmetry in tag and data bits to reduce the BDD variable requirement significantly, our time requirement has not reduced that much. Notice that we have to repeat the Steps 1-4 for the first line again for the second line in Steps 5-8. For a CAM with n lines, n repetitions are needed. This is however not a limiting factor in our approach, since automation of these steps is trivial, and the computation time and variable requirement in each repetition is low, due to symmetry exploitation in tags and data. It would be good to achieve a reduction in the number of times we have to repeat the steps for n lines. This is possible if we can record the symmetries amongst CAM lines as well, by structured modelling. At the moment our modelling framework does not support these symmetries. We call these symmetries *hierarchical* symmetries.

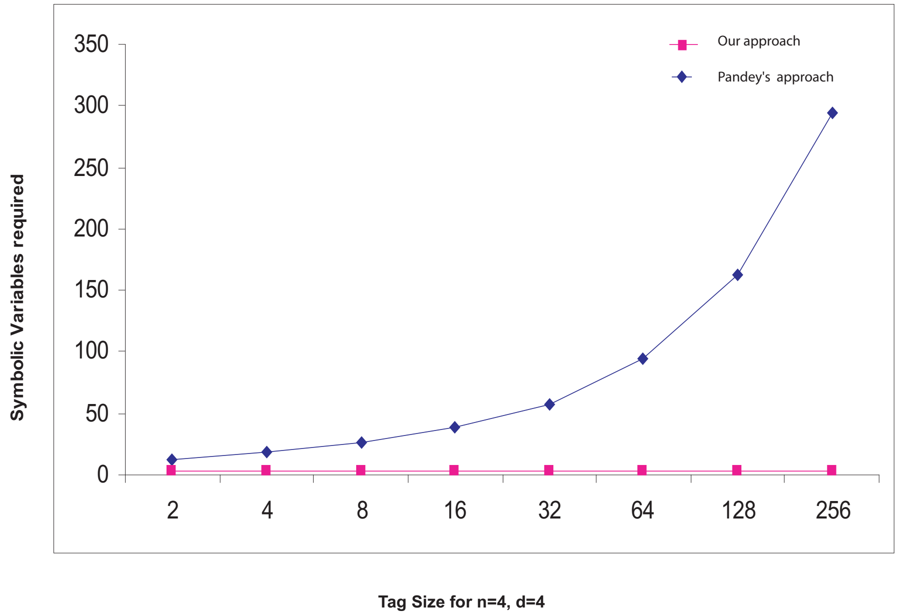


Figure 9.11: BDD variables with respect to tag size. Our requirement stays fixed at 3, whereas Pandey's scales logarithmically.

9.5 Circuit with multiple CAMs

Our final case study is that of a circuit that contains two CAMs. The property we wish to verify about this CAM is that the incoming tag can only be found in at most one CAM. If a tag is found in more than one, hit stays low. Figure 9.13 shows the circuit. Although this example is a contrived one, it illustrates an important feature — comparing incomings in parallel with multiple CAMs. Circuits with multiple CAMs are found in a cache. The associativity in a cache determines the number of CAMs used in the cache, and the number of comparators used in parallel for comparing the incoming tag address with the stored tags in different CAMs. We believe that the principle we illustrate in the verification of the circuit in this section, can be applied to a verification of a cache.

9.5.1 Modelling

The function *Hit* determines whether or not a hit is obtained from the circuit. As we said earlier, the hit rises if the tags match in at most one CAM but not both. This function uses the definition of the *cam* model we defined in the previous section, and

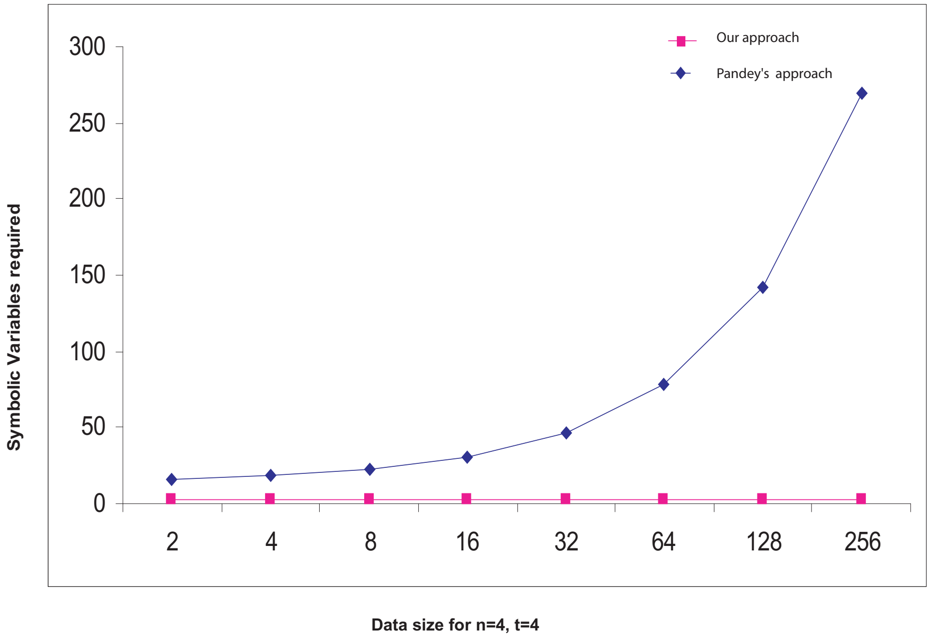


Figure 9.12: BDD variables with respect to data size. Our requirement stays fixed at 3, whereas Pandey's scales logarithmically.

employs two auxiliary functions *camTake* and *camDrop*.

$$
\begin{aligned}
&(\mathit{camTake}\ 0\ =\ (\mathit{Select}\ 0\ \mathit{Id}))\\
&(\mathit{camTake}\ (n+1)\ =\ (\mathit{camTake}\ n)\ \parallel\ (\mathit{Select}\ (n+1)\ \mathit{Id}))\\
\\
&(\mathit{camDrop}\ 0\ =\ \mathit{Id})\\
&(\mathit{camDrop}\ (n+1)\ =\ (\mathit{camDrop}\ n)\ \circ\ (\mathit{Tail}\ \mathit{Id}))
\end{aligned}
$$

The function *camTake* takes out n elements from a list of inputs that contains enable inputs for tag and data, and stored tags for both the CAMs, to give the stored tags of the first cam. The function *camDrop* similarly obtains the tags of the second CAM.

$$
\begin{aligned}
&\mathit{Hit}\ \ n\ \ \mathit{nsym}\ =\\
&\qquad \mathit{let}\ \mathit{hit}_0\ =\ \mathit{Bitwise}\ (\vee)\ ((\mathit{tcomparator}\ n\ \mathit{nsym})\ \circ\ (\mathit{camTake}\ n))\ \ \mathit{in}\\
&\qquad \mathit{let}\ \mathit{hit}_1\ =\ \mathit{Bitwise}\ (\vee)\ ((\mathit{tcomparator}\ n\ \mathit{nsym})\ \circ\\
&\qquad\qquad\qquad\qquad ((\mathit{Select}\ 0\ \mathit{Id})\ \parallel\ (\mathit{camDrop}\ (n+1))))\\
&\qquad\qquad \mathit{in}\\
&\qquad\qquad \mathit{Bitwise}\ \mathit{xor}\ (\mathit{hit}_0\ \parallel\ \mathit{hit}_1)
\end{aligned}
$$

The value of hit_0 determines if the match was found in the first CAM, and that of hit_1 determines if the tag was found in the second CAM. Note that we used the *tcomparator* function we had used in the design of the CAM model in the previous section. The circuit consisting of multiple CAMs is modelled by the function *cam_xor*. This function builds the circuit that stores the tag and data in two CAMs and is responsible for the tag and data read and write operations.

```
cam_xor nsym =
        let n = length nsym − 3 in
        let tagen = (el 0 nsym) in
        let dataen = (el 1 nsym) in
        let tagin = (el 2 nsym) in
        let tags = (tl (tl nsym)) in
        let nsym_0 = camTake n tags in
        let nsym_1 = (tagin :: (camDrop (n + 1) tags)) in
        let cam_1 = (cam (tagen :: (dataen :: nsym_0))) ∘ (camTake (n − 1)) in
        let cam_2 = (cam (tagen :: (dataen :: nsym_1))) ∘ (camDrop n)
                    in
                    Bitwise xor (cam_1 || cam_2)
```

9.5.2 Verification

In this section we show how to carry out the verification of the property that we wish to verify using symmetry based reduction and inference rules. The property of interest is that the output of the circuit (hit) rises when the query tag is found only in one of the CAMs but not in both.

We begin by initialising the BDD variables for the tags stored in both the CAMs. We choose to denote the tags stored in the first CAM by the letter T and the ones stored in the other CAM by U. For example, the node "$T_i[j]$" denotes the j^{th} bit of the tag stored in the i^{th} line of the first CAM (T). Similarly the variable t_{ij} denotes the symbolic value stored at the location "$T_i[j]$", where i denotes the line of the CAM T and the j denotes the bit, inside the line i.

Incoming tag variables are denoted by Tag_0 and Tag_1, and are assigned to nodes "$Tag[0]$" and "$Tag[1]$" respectively. We assume that we have declared symbolic variables in a Forte session for each of the lines for both the CAMs.

The base antecedent is set up to write the CAMs with the initial values in the first cycle and read them in the second cycle.

```
(* read enabled and incoming tag takes on symbolic values *)
let base_ant = ((("Tag[0]" is Tag_0) and ("Tag[1]" is Tag_1)) from 0 to 2) and
               ("tagen" is F from 0 to 1) and ("tagen" is T from 1 to 2)
```

Antecedents stating that the tag nodes take on the values in both the CAMs are shown below.

```
(* populate the 1st line of the 1st CAM *)
let A_0 = ((("T_0[0]" is t_00) and ("T_0[1]" is t_01)) from 0 to 2) and base_ant

(* populate the 2nd line of the 1st CAM *)
let A_1 = ((("T_1[0]" is t_10) and ("T_1[1]" is t_11)) from 0 to 2) and base_ant
```

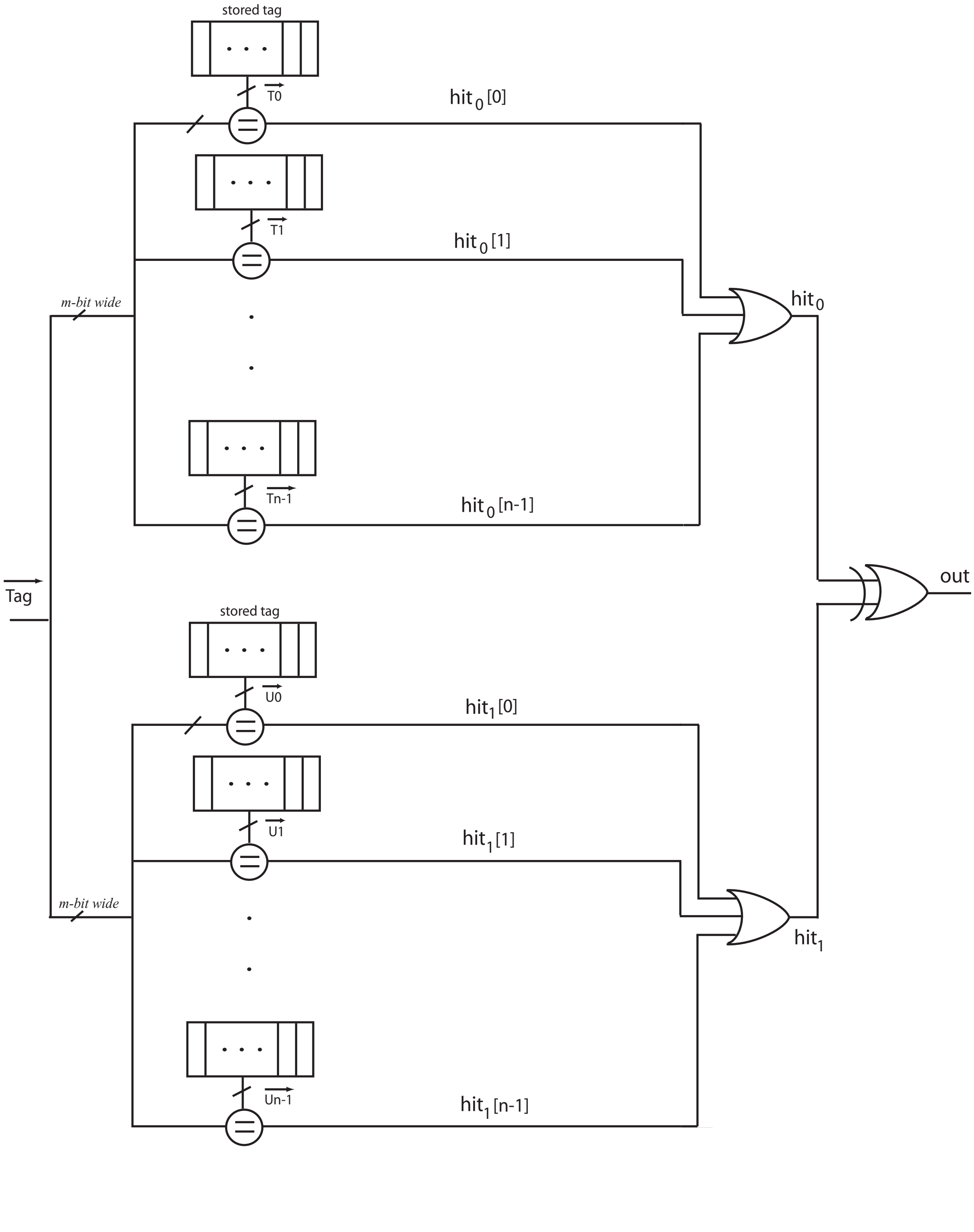


Figure 9.13: A circuit with two CAMs, hit only rises if the tag is found in at most one of the CAMs but not both.

$(*\ \texttt{populate the 1st line of the 2nd CAM}\ *)$

$\mathit{let}\ B_0\ =\ (((\text{“}U_0[0]\text{”}\ \ \mathsf{is}\ \ u_{00})\ \ \mathsf{and}\ \ (\text{“}U_0[1]\text{”}\ \ \mathsf{is}\ \ u_{01}))\ \mathit{from}\ 0\ \mathit{to}\ 2)\ \ \mathsf{and}\ \ \mathit{base\_ant}$

$(*\ \texttt{populate the 2nd line of the 2nd CAM}\ *)$

$\mathit{let}\ B_1\ =\ (((\text{“}U_1[0]\text{”}\ \ \mathsf{is}\ \ u_{10})\ \ \mathsf{and}\ \ (\text{“}U_1[1]\text{”}\ \ \mathsf{is}\ \ u_{11}))\ \mathit{from}\ 0\ \mathit{to}\ 2)\ \ \mathsf{and}\ \ \mathit{base\_ant}$

The following trajectory formulas state that the hit from each of the CAMs can be either low or high.

$(*\ \texttt{hit found in the 1st CAM}\ *)$

$\mathit{let}\ C_0\ =\ \text{“}\mathit{hit}_0\text{”}\ \ \mathsf{is}\ \ T\ \mathit{from}\ 1\ \mathit{to}\ 2$

$(*\ \texttt{hit not found in the 1st CAM}\ *)$

$\mathit{let}\ D_0\ =\ \text{“}\mathit{hit}_0\text{”}\ \ \mathsf{is}\ \ F\ \mathit{from}\ 1\ \mathit{to}\ 2$

$(*\ \texttt{hit found in the 2nd CAM}\ *)$

$\mathit{let}\ C_1\ =\ \text{“}\mathit{hit}_1\text{”}\ \ \mathsf{is}\ \ T\ \mathit{from}\ 1\ \mathit{to}\ 2$

$(*\ \texttt{hit not found in the 2nd CAM}\ *)$

$\mathit{let}\ D_1\ =\ \text{“}\mathit{hit}_1\text{”}\ \ \mathsf{is}\ \ F\ \mathit{from}\ 1\ \mathit{to}\ 2$

The following Boolean formulas state when the incoming tags match the stored tags of the CAM lines. We use the letter G to denote the Boolean formulas that state that the tags match at the lines of the first CAM, while we use the letter H to denote the Boolean formula that states that tags match at the lines of the second CAM. The suffixes distinguish where in each CAM, the tags are found.

$(*\ \texttt{tags match at 1st line of 1st CAM}\ *)$

$\mathit{let}\ G_0\ =\ (\mathit{Tag}_0\ =\ t_{00})\ \wedge\ (\mathit{Tag}_1\ =\ t_{01})$

$(*\ \texttt{tags match at 2nd line of 1st CAM}\ *)$

$\mathit{let}\ G_1\ =\ (\mathit{Tag}_0\ =\ t_{10})\ \wedge\ (\mathit{Tag}_1\ =\ t_{11})$

$(*\ \texttt{tags match at 1st line of 2nd CAM}\ *)$

$\mathit{let}\ H_0\ =\ (\mathit{Tag}_0\ =\ u_{00})\ \wedge\ (\mathit{Tag}_1\ =\ u_{01})$

$(*\ \texttt{tags match at 2nd line of 2nd CAM}\ *)$

$\mathit{let}\ H_1\ =\ (\mathit{Tag}_0\ =\ u_{10})\ \wedge\ (\mathit{Tag}_1\ =\ u_{11})$

We now outline the steps involved in verifying the property of interest. We begin by verifying properties given in (1) and (2). Property (1) states that if the incoming tag was found in the first CAM ($G_0 \vee G_1$), then the hit from the first CAM (“hit_0”) is high. This property corresponds to the property we verified in the previous section, and it can be done using two variables as we explained. Thus we have written CAM verification strategy to mean that the flow we used in verifying the property for the CAM will be used here. We do not show the steps again, since they are merely a repetition.

1. $(G_0\ \vee\ G_1)\ \supset\ (\mathit{cam\_xor}\ \models\ (A_0\ \ \mathsf{and}\ \ A_1)\ \Rightarrow\ C_0)$ (*CAM strategy, using 2 variables*)
2. $(H_0\ \vee\ H_1)\ \supset\ (\mathit{cam\_xor}\ \models\ (B_0\ \ \mathsf{and}\ \ B_1)\ \Rightarrow\ C_1)$ (*CAM strategy, using 2 variables*)

Similarly we can verify (2). Then we proceed to verify (3), which states that if the match was found in both the CAMs then the output “*out*” of the circuit with both the

CAMs is low. This weaker property is actually the verification of the xor gate, and is scalar.

3. $\mathit{cam\_xor} \models (C_0 \ \mathsf{and}\ C_1) \Rightarrow (\text{“}\mathit{out}\text{”} \ \mathsf{is}\ F)$ (*Scalar STE run*)
4. $(G_0 \vee G_1) \wedge (H_0 \vee H_1) \supset$
   $(\mathit{cam\_xor} \models (A_0 \ \mathsf{and}\ A_1) \ \mathsf{and}\ (B_0 \ \mathsf{and}\ B_1)$
   $\Rightarrow$
   $(C_0 \ \mathsf{and}\ C_1))$ (*Guard Conjunction on 1 and 2*)
5. $(G_0 \vee G_1) \wedge (H_0 \vee H_1) \supset$
   $(\mathit{cam\_xor} \models (A_0 \ \mathsf{and}\ A_1) \ \mathsf{and}\ (B_0 \ \mathsf{and}\ B_1)$
   $\Rightarrow (\text{“}\mathit{out}\text{”} \ \mathsf{is}\ F)$ (*Cut on 3 and 4*)

We then employ the Guard Conjunction rule on (1) and (2) to compose the correctness statement that states that if the tag was found in any line of the first CAM ($G_0 \vee G_1$), and in any line of the second CAM ($H_0 \vee H_1$), then the hit values from both the CAMs will be high ($C_0 \ \mathsf{and}\ C_1$). This step is done in (4). Then a Cut is done on (3) and (4) to deduce (5) which states, that if the tag was found in both the CAMs then the output (“*out*”) stays low. This is one part of the property that we wanted to verify. Now we will show how to verify the other part that states that the output becomes high only when the tags are found in at most one CAM. Property (6) and (7) state that the hits from each of the CAMs stay low if the incoming tag is not found in either of the CAMs.

6. $(nG_0 \wedge nG_1) \supset$
   $(\mathit{cam\_xor} \models (A_0 \ \mathsf{and}\ A_1) \Rightarrow D_0)$ (*CAM strategy, using two variables*)
7. $(nH_0 \wedge nH_1) \supset$
   $(\mathit{cam\_xor} \models (B_0 \ \mathsf{and}\ B_1) \Rightarrow D_1)$ (*CAM strategy, using two variables*)

Guard Conjunction on (2) and (6) gives the correctness of the property (8), that states that the hit from the second CAM is high and the hit from the first CAM is low when the tag was found in the second CAM and not in the first one. Guard Conjunction on (1) and (7) gives the correctness of the property (9), that states that the hit from the first CAM is high and the hit from the second CAM is low when the tag was found in the first CAM and not in the second one.

8. $((nG_0 \wedge nG_1) \wedge (H_0 \vee H_1)) \supset$
   $(\mathit{cam\_xor} \models (A_0 \ \mathsf{and}\ A_1) \ \mathsf{and}\ (B_0 \ \mathsf{and}\ B_1)$
   $\Rightarrow (D_0 \ \mathsf{and}\ C_1))$ (*Guard Conjunction on 2 and 6*)
9. $((nH_0 \wedge nH_1) \wedge (G_0 \vee G_1)) \supset$
   $(\mathit{cam\_xor} \models (A_0 \ \mathsf{and}\ A_1) \ \mathsf{and}\ (B_0 \ \mathsf{and}\ B_1)$
   $\Rightarrow (D_1 \ \mathsf{and}\ C_0))$ (*Guard Conjunction on 1 and 7*)

Statements (10) and (11) are scalar runs checking again a property of the xor gate, that the output is only high if only one of the inputs is low.

10. $\mathit{cam\_xor} \models (D_0 \ \mathsf{and}\ C_1) \Rightarrow (\text{“}\mathit{out}\text{”} \ \mathsf{is}\ T)$ (*Scalar STE run*)
11. $\mathit{cam\_xor} \models (D_1 \ \mathsf{and}\ C_0) \Rightarrow (\text{“}\mathit{out}\text{”} \ \mathsf{is}\ T)$ (*Scalar STE run*)

A cut on (8) and (10) gives the correctness statement (12), that when the tags are found in the second CAM, and not in the first then the output is high, and a cut on (9) and

(11), gives (13) which states that when tags are found in the first CAM and not in the second, then the output is high.

$$
\begin{array}{lll}
12. & ((nG_0 \ \wedge \ nG_1) \ \wedge \ (H_0 \ \vee \ H_1)) \supset & \\
 & \quad (cam\_xor \models (A_0 \ \mathsf{and} \ A_1) \ \mathsf{and} \ (B_0 \ \mathsf{and} \ B_1) & \\
 & \qquad \Rightarrow (\text{``}out\text{''} \ \mathsf{is} \ T)) & (\textit{Cut on 8 and 10}) \\
13. & ((nH_0 \ \wedge \ nH_1) \ \wedge \ (G_0 \ \vee \ G_1)) \supset & \\
 & \quad (cam\_xor \models (A_0 \ \mathsf{and} \ A_1) \ \mathsf{and} \ (B_0 \ \mathsf{and} \ B_1) & \\
 & \qquad \Rightarrow (\text{``}out\text{''} \ \mathsf{is} \ T)) & (\textit{Cut on 9 and 11})
\end{array}
$$

Thus overall we only ever need the same number of variables as we needed for verifying the property about the hit logic of a CAM, which is 2. The symmetry in tags means that we don't have to use $5m$ variables for the verification of the tag comparison circuit, if m is the width of the tag bits ($4m$ distinct tag variables would be required for the two lines of the two CAMs, and m for incoming tags). Thus symmetry becomes useful there, while inference rules help split the verification task into smaller ones that are verified separately and then the overall results composed from the smaller results.

9.6 Related Work

Even though Pandey invests substantial time in symmetry identification phase he successfully demonstrated [82, 83] that symmetry can be used to a very good effect in verification of memory arrays, specifically he showed that SRAMs could be verified in linear time and space using symmetry. Our work corroborates much of what he showed for SRAMS, but also offers extra reduction possibilities for memories with large number of data bits. We showed that using our argument and approach, for a given number of address lines in SRAM, one can verify different sizes of SRAM that differ in number of data bits stored in a data word, using constant number of variables that is proportional to the log of the number of address lines. In this respect our work renders an improvement over existing work done by Pandey for SRAM verification.

For CAMs, Pandey did not use symmetries, but used symbolic indexing and devised the CAM encoding to verify arbitrary sized CAMs [82, 85]. We show that by our approach we can perform significantly better in terms of variable requirement, since we only have to verify one line at a time, and during verification of a single line, usage of symmetry reduction in the tag comparison bit of circuitry, and symmetry amongst data bits, renders significant reductions in the variable requirement. However as we explained earlier, the time required to verify CAM using our approach scales linearly with the number of lines. This is due to the fact that using property decomposition, we have to repeat the verification steps for each CAM line. Although not a seriously limiting factor, because each repetition of steps exploits symmetry, and therefore does not require much time, it would be good to capture hierarchical symmetries, such as the symmetries amongst CAM lines to avoid repetition for each CAM line.

For circuit with multiple CAMs, using property decomposition and then using symmetry amongst tag bits offers a powerful reduction strategy. Without property decomposition, we conjecture, that even using symbolic indexing is not enough, since typically caches have many more multiple CAMs, and thereby the number of parallel comparisons increase beyond the scope of symbolic indexing based reduction.

9.7 Summary

In this chapter we showed several examples and case studies to illustrate our methodology and compare it to the best known work in the field, which has a strong overlap with us. Overall we believe that our approach offers significant theoretical and practical benefits because we take the approach of symmetry detection via type checking and our use of inference rules has shown that we can do very effective property reductions. Also we have shown a way to connect high level structured models to the theory of STE, offering valuable insights on how the property reduction is justifiable by designing structured models.

Chapter 10

Conclusion

This dissertation has shown a new and efficient methodology of doing symmetry based reduction for STE model checking. Efficiency comes due to two reasons. Firstly, by using structured modelling, symmetries in the circuit are recorded in the syntax of the circuit models, and are inferred by type checking. Thus we don't have to find symmetries in graph representation of models using NP-complete sub-graph isomorphism techniques. Secondly, new STE inference rules together with Theorem 8.1, provide an effective reduction strategy.

For the first contribution, we have proposed a small set of circuit construction combinators that are used in the design of circuit models. A set of typing rules that we provide, becomes the basis for judicious use of the combinators to construct symmetric circuits. We showed the proof of the type soundness theorem (Theorem 6.3) that guarantees this. A side contribution is the mechanisation of the complete theory in HOL.

The strength of our approach is that just by using such a small set of combinators and a type system based on only nine rules, we are able to model a variety of symmetric circuits, starting from the very basic combinational gates to large SRAMs, CAMs and cache like circuits. The approach is modular and encourages component re-use.

There are two perspectives on our approach. The first perspective is that for a keen functional programmer interested in circuit modelling, our implementation of combinators provides a front end to the circuit design. We have provided a system to generate netlists from these designs, and therefore one can use our functions to design circuit models in HOL, and simulate them in Forte, or for smaller designs use the STE simulator in HOL.

The second perspective is that our theory and the proof of the type soundness theorem can be used as a blue-print by other HDL designers. Most of the circuit construction combinators defined by us are simple, and if one prefers can be easily ported to other functional hardware design languages for example reFLect, Lava or even imperative ones like VHDL or Verilog. The message is that by using those combinators (in any incarnation) by a systematic way (guided by the type system), one can ensure that symmetric circuit blocks are designed — correct by construction.

The other major contribution has been the invention of new STE inference rules, that increase the expressiveness of the known STE inference system. The new STE inference rules, provide an effective property reduction strategy. The rules are used first to decompose the given original STE properties into smaller properties. Symmetry in circuit models is then used as a basis to cluster equivalent symmetric STE properties together, and a representative property from the equivalence class is verified in the STE simulator. Then the deduction of the correctness of the entire class of equivalent STE properties is made by using a soundness theorem that we have presented. Once all the smaller properties have been verified either by explicit STE runs, or by deducing their correctness by using the soundness theorem, we use the inference rules in the forward direction to compose the overall correctness statement of the original STE property.

Inference rules thus become an enabler in decomposing properties effectively so as to help expose the symmetric set. As another side contribution, the proof of the soundness theorem and the inference rules, has been completely mechanised in HOL.

We have shown several examples that illustrate the strength of our reduction strategy, and we compared our approach with Pandey's work since it has possibly the biggest overlap in terms of using STE model checking, using symmetries for reduction.

Overall we believe that our approach offers significant theoretical and practical benefits because we take the approach of symmetry detection via type checking and our use of inference rules has shown that we can do effective property reductions. Also we have shown a way to connect high level structured models to the theory of STE, offering valuable insights on how the property reduction is justifiable by designing structured models.

10.1 Future directions

We described symmetries that come into being when groups of wires, kept together as a bus are treated in a special way. These symmetries are characterised by an invariant function under permutations of the bits in each bus of the symmetric inputs and outputs.

There is however, another class of symmetries that are also interesting, and can lead to even further significant reductions. These are the symmetries that arise when one bus as a whole is swapped with another one. For example if we swapped the bus a with the bus b in a two bit comparator where the buses a and b are the symmetric inputs, the resulting circuit still has symmetry. Consider the example of the CAM. We have shown that symmetries in tag bits and data bits lead to reductions in the number of properties that have to be explicitly verified using a simulator. But if we could also characterise the symmetries in the lines of the CAM, then we wouldn't have to repeat the verification flow for every line, as we proposed in the CAM verification study in the last chapter.

Although, very visible, these symmetries pose a considerable challenge when it comes to modelling circuits. Consider the *Select* combinator which selects any one of the n buses from an input. If we swapped the i^{th} bus with the j^{th}, then the *Select i Id* will give the j^{th} input rather than the i^{th} one. This problem appears with other combinators like *Tail*, *Fork* and $\|$ as well. It was clear to us that if we needed to accommodate this symmetry, substantial changes would have to be made to the present framework, perhaps even the base type of *bool list list* → *bool list list* → *bool list list* is not adequate, we need the type to be more general:

$$((\mathit{bool\ list})\ \mathit{list})\ \mathit{list} \rightarrow ((\mathit{bool\ list})\ \mathit{list})\ \mathit{list} \rightarrow ((\mathit{bool\ list})\ \mathit{list})\ \mathit{list}$$

We could not investigate these hierarchical symmetries, since we did not have enough time.

Also it would be useful to see how data symmetries [82] in Pandey's research fit into our framework. Pandey has advocated the usage of these symmetries [82], but the connection between these symmetries and the logic of STE is not clear to us. We understand how to connect the structural symmetries presented in our work to the logic of STE in the form of symmetry soundness theorem (Theorem 8.1), but what would be the connection between data symmetries and the STE logic is a topic for further research.

We used the theorem prover to do the proof of the type soundness theorem by shallowly defining the combinators and embedding the type system using rule induction. This also paved the way for modelling circuits, and using the built-in libraries in HOL for dealing with arithmetic, strings, Booleans and combinators. We used the proof-as-computation strategy in HOL, by using conversions in HOL to advance computation of circuit's structure and behaviour. While the simulation of circuits using conversions was satisfactory in terms of speed, for small and medium sized CAMs, generating netlists for large SRAMs and CAMs was infeasible. Speed of execution became a crucial limiting factor.

The reason was the underlying proofs that needed to be done to ensure that the symmetric inputs had all the buses of equal length. Although by optimizing the conversions, some orders of magnitude speed-up were obtained, we could not proceed much further with this approach.

It would be good to investigate if by embedding the combinators in other languages such as reFLect or Lava, we can do better in terms of speed. Although these languages do not have the proof support, it is not so critical since the major proof, about the type soundness theorem has been done in HOL, and it can be used as certificate to design the combinators exactly in the same way in reFLect and Lava.

We have shown how to obtain flat STE models in HOL, and the flat FSMs in Forte from the structured circuit models. We also showed how we could simulate the structured circuit models by interpreting them by using the *toTime* function in HOL. Also we mentioned in Chapter 7 that the notion of equivalences between different models, is informal. Formalisation of this, would be a good theoretical contribution.

We showed how property decomposition together with the symmetry soundness theorem provides an effective reduction strategy. However, the reduction on the whole is interactive rather than being automatic. Future work should look into ways of automating the reduction flow.

We showed several examples involving sequential delay elements like registers, SRAMs, CAMs and so on, but we have not dealt with circuits with feedback loops. It would be good to characterise a few of those examples as well. This requires defining a loop primitive that can be used to connect some of the output buses to some of the input buses, assuming there is always a unit delay in the loop. This merits future investigation.

Appendix A

A-1 Evaluating the behaviour of comparator

We show the execution traces for evaluating the behaviour of the comparator circuit, we presented in Chapter 6. For the definition of the circuit, please refer to that chapter, here we only show the output values returned by the circuit at time points 1, 2 and 3.

```
- `` ^CompTimed ^nsym ^sym (SUC 0) `` ;
- comp_conv it;
- RIGHT_CONV_RULE(SIMP_CONV std_ss [RE_def]) it;

> val it =
    ⊢ (λnsym sym t.
          (toTime (map (map (RE (hd (hd (nsym (t - 1)))))) o
           toTime (And o comp)) sym (t - 1)) (λt. ck)
         (λt.
            (if t = 0 then
               [[a0; a1]; [b0; b1]]
             else
               (if t = SUC 0 then
                  [[T; T]; [T; F]]
                else
                  [[a0; a1]; [c0; c1]]))) (SUC 0) =
       [[(b1 ∧ a1 ∨ ~b1 ∧ ~a1) ∧ (b0 ∧ a0 ∨ ~b0 ∧ ~a0)]] : thm
```

Table A.1: Comparator output at time point 1.

```
- `` ^CompTimed ^nsym ^sym (SUC(SUC 0)) `` ;
- comp_conv it;;
- RIGHT_CONV_RULE(SIMP_CONV std_ss [RE_def]) it;;

 > val it =
    ⊢ (λnsym sym t.
          (toTime (map (map (RE (hd (hd (nsym (t - 1))))))) o
           toTime (And o comp)) sym (t - 1)) (λt. ck)
         (λt.
            (if t = 0 then
               [[a0; a1]; [b0; b1]]
             else
               (if t = SUC 0 then
                  [[T; T]; [T; F]]
                else
                  [[a0; a1]; [c0; c1]]))) (SUC (SUC 0)) =
       [[F]] : thm
```

Table A.2: Comparator output at time point 2.

```
- `` ^CompTimed ^nsym ^sym (SUC(SUC(SUC 0))) `` ;
- comp_conv it;;
- RIGHT_CONV_RULE(SIMP_CONV std_ss [RE_def]) it;;

- > val it =
    ⊢ (λnsym sym t.
          (toTime (map (map (RE (hd (hd (nsym (t - 1))))))) o
           toTime (And o comp)) sym (t - 1)) (λt. ck)
         (λt.
            (if t = 0 then
               [[a0; a1]; [b0; b1]]
             else
               (if t = SUC 0 then
                  [[T; T]; [T; F]]
                else
                  [[a0; a1]; [c0; c1]]))) (SUC (SUC (SUC 0))) =
       [[(c1 ∧ a1 ∨ ~c1 ∧ ~a1) ∧ (c0 ∧ a0 ∨ ~c0 ∧ ~a0)]] : thm
```

Table A.3: Comparator output at time point 3.

A-2 Evaluating the behaviour of Mux

```
`` ^MuxTimed  (^nsym)(^sym) ((SUC 0)) `` ;
(MUX_CONV) it;
RIGHT_CONV_RULE(mux_conv) it;
RIGHT_CONV_RULE(REWRITE_CONV [CheckLength_def, map, append]) it;
RIGHT_CONV_RULE(SIMP_CONV std_ss [ONE]) it;
RIGHT_CONV_RULE(MUX_CONV) it;
RIGHT_CONV_RULE(EVAL) it;

> val it =
    ⊢ (λnsym sym t.
          (toTime (map (map (RE (hd ck)))) ∘
           toTime (Bitwise (∨) (Auxmux (el (SUC 0) (nsym (t - 1))))))
            sym (t - 1))
         (λt. (if t = 0 then [ck; [T]] else [ck; [F]]))
         (λt. [[a0; a1]; [b0; b1]]) (SUC 0) = [[a0; a1]] : thm
```

Table A.4: Mux execution for time point 1.

```
`` ^MuxTimed  (^nsym)(^sym) ((SUC(SUC 0))) `` ;
(MUX_CONV) it;
RIGHT_CONV_RULE(mux_conv) it;
RIGHT_CONV_RULE(REWRITE_CONV [CheckLength_def, map, append]) it;
RIGHT_CONV_RULE(SIMP_CONV std_ss [ONE]) it;
RIGHT_CONV_RULE(MUX_CONV) it;
RIGHT_CONV_RULE(EVAL) it;

> val it =
    ⊢ (λnsym sym t.
          (toTime (map (map (RE (hd ck)))) ∘
           toTime (Bitwise (∨) (Auxmux (el (SUC 0) (nsym (t - 1))))))
            sym (t - 1))
         (λt. (if t = 0 then [ck; [T]] else [ck; [F]]))
         (λt. [[a0; a1]; [b0; b1]]) (SUC(SUC 0)) = [[b0; b1]] : thm
```

Table A.5: Mux execution for time point 1.

A-3 Evaluating the derived STE model of Mux

```
> val it =
    ⊢ (λs n.
          (if
             ((n = "ck") ∨ (n = "c")) ∨ ((n = "a0") ∨ (n = "a1")) ∨
             (n = "b0") ∨ (n = "b1") then X
           else (if (n = "out0") ∨ (n = "out1") then
                (if n = "out0" then
                   (if ˜c ∧ b0 ∨ c ∧ a0 then
                       One else Zero) lub s "out0"  else
                   (if n = "out1" then
               (if ˜c ∧ b1 ∨ c ∧ a1 then One else Zero) lub s "out1"
                    else X))
              else X)))
         (λn.
            (if n = "a0" then DROP a0
             else (if n = "a1" then DROP a1
                else  (if n = "b0" then  DROP b0 else
                     (if n = "b1" then
                        DROP b1   else
                        (if n = "c" then DROP c else X)))))) "out0" =
       (if ˜c ∧ b0 ∨ c ∧ a0 then (T,F) else (F,T)) : thm
```

Table A.6: Value for node “out_0”.

```
> val it =
  ⊢ (λs n.
       (if
          ((n = "ck") ∨ (n = "c")) ∨ ((n = "a0") ∨ (n = "a1")) ∨
          (n = "b0") ∨ (n = "b1") then X
        else (if (n = "out0") ∨ (n = "out1") then
               (if n = "out0" then
                  (if ~c ∧ b0 ∨ c ∧ a0 then
                     One else Zero) lub s "out0"  else
                  (if n = "out1" then
               (if ~c ∧ b1 ∨ c ∧ a1 then One else Zero) lub s "out1"
                   else X))
              else X)))
       (λn.
          (if n = "a0" then DROP a0
           else (if n = "a1" then DROP a1
              else  (if n = "b0" then  DROP b0 else
                   (if n = "b1" then
                      DROP b1   else
                      (if n = "c" then DROP c else X)))))) "out1" =
     (if ~c ∧ b1 ∨ c ∧ a1 then (T,F) else (F,T)) : thm
```

Table A.7: Value for node "*out*$_1$".

References

[1] BLIF. See http://www.bdd-portal.org/docu/blif/blif.html.

[2] The Forte Formal Verification System. See online at http://www.intel.com/cd/software/products/asmo-na/eng/219776.htm.

[3] HOL 4. Available from http://hol.sourceforge.net/.

[4] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Combining Theorem Proving and Trajectory Evaluation in an Industrial Environment. In *DAC '98: Proceedings of the 35th ACM/IEEE Design Automation Conference*, pages 538–541. ACM Press, New York, NY, USA, June 1998. ISBN 0-89791-964-5.

[5] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal Verification Using Parametric Representations of Boolean Constraints. In *DAC '99: Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 402–407. ACM Press, New York, NY, USA, July 1999. ISBN 1-58133-109-7.

[6] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLS '99: Proceedings of the 12th International Confernce on Theorem Proving in Higher-Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 323–340. Springer-Verlag, Berlin, September 1999. ISBN 3-540-66463-7.

[7] M. D. Aagaard, T. F. Melham, and J. W. O'Leary. Xs are for Trajectory Evaluation, Booleans are for Theorem Proving. In L. Pierre and T. Kropf, editors, *CHARME '99: Proceedings of the 10th IFIP WG10.5 Advanced Research Working Conference*, volume 1703 of *Lecture Notes in Computer Science*, pages 202–218. Springer-Verlag, Berlin, September 1999. ISBN 3-540-66559-5.

[8] M. D. Aagaard and C.-J. H. Seger. The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier. In *ICCAD '95: Proceedings of the 1995 ACM/IEEE International Conference on Computer-Aided Design*, pages 7–10. IEEE Computer Society, November 1995. ISBN 0-8186-7213-7.

[9] S. Aggarwal, R. P. Kurshan, and K. K. Sabnani. A Calculus for Protocol Specification and Validation. In H. Rudin and C. West, editors, *Protocol Specification,*

Testing, and Verification, III, Proceedings of the IFIP WG 6.1 Third International Workshop on Protocol Specification, Testing and Verification, volume 3, pages 19–34. Elsevier Science Publishers B.V.(North Holland), 1983. ISBN 0-444-86769-4.

[10] H. Amjad. Programming a Symbolic Model Checker in a Fully Expansive Theorem Prover. In *TPHOLS '03: Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2785/2003 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, Berlin, 2003. ISBN 0302-9743.

[11] H. Amjad. Verification of AMBA Using a Combination of Model Checking and Theorem Proving. *Electronic Notes in Theoretical Computer Science*, 145:45–61, 2006.

[12] C. M. Angelo, L. Claesen, and H. De Man. Degrees of Formality in Shallow Embedding Hardware Description Languages in HOL. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop (HUG'93)*, volume 780 of *Lecture Notes in Computer Science*, pages 89–100. Springer-Verlag, Berlin, 1994. ISBN 3-540-57826-9.

[13] M. A. Armstrong. *Groups and Symmetry.* Springer, first edition, February 1997. ISBN 0387966757.

[14] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark Challenge. In J. Hurd and T. Melham, editors, *TPHOLS '05: Proceedings of the 18th International Conference on Theorem Proving in Higher Order logics*, pages 50–65, August 2005.

[15] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *EuroSys 2006*, 2006.

[16] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213. ACM Press, New York, NY, USA, 2001. ISBN 1-58113-414-2.

[17] T. Ball and S. K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130. Springer-Verlag, Berlin, 2000. ISBN 3-540-41030-9.

[18] T. Ball and S. K. Rajamani. The SLAM Toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer-Verlag, Berlin, 2001. ISBN 3-540-42345-1.

[19] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3. ACM Press, New York, NY, USA, 2002. ISBN 1-58113-450-9.

[20] D. L. Beatty, R. E. Bryant, and C.-J. H. Seger. Synchronous circuit verification by symbolic simulation: An Illustration. In *AUSCRYPT '90: Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 98–112. MIT Press, 1990. ISBN 0-262-04109-X.

[21] R. Bird and P. Wadler. *An Introduction to Functional Programming.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1988. ISBN 0-13-484189-1.

[22] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional programming*, pages 174–184. ACM Press, New York, NY, USA, 1998. ISBN 1-58113-024-4.

[23] D. Bosnacki, D. Dams, and L. Holenderski. A Heuristic for Symmetry Reductions with Scalarsets. In J. N. Oliveira and P. Zave, editors, *FME '01: Proceedings of the International Symposium of Formal Methods for Increasing Software Productivity: International Symposium of Formal Methods Europe*, volume 2021/2001 of *Lecture Notes in Computer Science*, pages 518–533. Springer-Verlag, Berlin, 2001. ISBN 3-540-41791-5.

[24] R. Boyer and J. Moore. Proof-checking, Theorem-proving and Program Verification. In W. Bledsoe and D.W.Loveland, editors, *Contemporary Mathematics, Automated Theorem proving: After 25 years*, volume 29, pages 119–132, Providence, Rhode Island, 1984. American Mathematical Society.

[25] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[26] R. Bryant. Formal Verification of Memory Circuits by Switch-Level Simulation. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, volume 10, no. 1, pages 94–102, January 1991.

[27] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[28] O. Burkart and B. Steffen. Model checking the full Modal Mu-calculus for infinite sequential processes. *Journal of Theoretical Computer Science*, 221(1-2):251–270, 1999.

[29] S. Burris and H. Sankappanavar. *A Course in Universal Algebra.* Springer Verlag, Berlin, 1981. Graduate Texts in Mathematics.

[30] M. Calder and A. Miller. Five ways to use induction and symmetry in the verification of networks of processes by model-checking. In *Proceedings of the Second Workshop on Automated Verification of Critical Systems (AVoCS 2002)*, pages 29–42, 2002.

[31] J. Camilleri and T. Melham. Reasoning with Inductively Defined Relations in the HOL Theorem Prover. Technical Report 265, Computer Laboratory, University of Cambridge, August 1992.

[32] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. ISBN 0-201-05866-9.

[33] A. Chavan, B. W. Min, and S. Chin. *HOL2GDT, A Verification-Based Design Methodology*, March 1999. Available from http://caliper.syr.edu/ bmin/doc/hardware.html.

[34] C.-T. Chou. The Mathematical Foundation of Symbolic Trajectory Evaluation. In N. Halbwachs and D. Peled, editors, *CAV '99: Proceedings of the 11th International Conference in Computer Aided Verification*, volume 1633/1999 of *Lecture Notes in Computer Science*, pages 196–207. Springer-Verlag, Berlin, July 1999.

[35] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[36] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In E. A. Emerson and A. P. Sistla, editors, *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, Berlin, 2000. ISBN 3-540-67770-4.

[37] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[38] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8.

[39] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting Symmetry in Temporal Logic Model Checking. *Journal of Formal Methods in System Design*, 9(1-2):77–104, August 1996.

[40] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100. Elsevier Science Publishers B.V., Amsterdam, Netherland, 1993.

[41] A. Cohn. The Notion of Proof in Hardware Verification. *Journal of Automated Reasoning*, 5(2):127–139, 1989.

[42] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM Press, New York, NY, USA, 1971.

[43] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In E. M. Clarke and R. P. Kurshan, editors, *CAV '90: Proceedings of the 2nd International Conference on Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 23–32. Springer-Verlag, London, UK, 1990. ISBN 3-540-54477-1.

[44] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY, USA.

[45] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY, USA.

[46] D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[47] A. Darbari. Formalization and Execution of STE in HOL. In D. Basin and B. Wolff, editors, *Supplementary Proceedings of the 16th International Conference on Theorem Proving in Higher-Order Logics*, LNCS 2758. Springer-Verlag, Berlin, 2003.

[48] A. Darbari. Formalization and Execution of STE in HOL (Extended Version). Technical Report PRG-RR-03-17, Oxford University Computing Lab, Parks Road, Oxford, September 2003.

[49] E. W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. Hoare, editors, *Structured programming.* Academic Press, London, 1972.

[50] E. A. Emerson and A. P. Sistla. Utilizing Symmetry When Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, July 1997.

[51] F. A. Emerson and A. P. Sistla. Symmetry and Model Checking. *Journal of Formal Methods in System Design*, 9(1/2):105–131, August 1996.

[52] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. Elsevier Science Publishers, 1986.

[53] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993. ISBN 0-521-44189-7.

[54] A. Gupta. Formal Hardware Verification Methods: A Survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.

[55] S. Hazelhurst and C.-J. H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 14(4):413–422, April 1995.

[56] S. Hazelhurst and C.-J. H. Seger. Symbolic Trajectory Evaluation. In T. Kropf, editor, *Formal Hardware Verification - Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, chapter 1, pages 3–78. Springer-Verlag, Berlin, 1997. ISBN 3-540-63475-4.

[57] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, 1983.

[58] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201441241.

[59] S. Jha. *Symmetry and Induction in Model Checking*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[60] R. B. Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors*. PhD thesis, Department of Electrical Engineering, Stanford University, 1999.

[61] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham. Practical formal verification in microprocessor design. *IEEE Design & Test of Computers*, 18(4):16–25, July/August 2001.

[62] J. Joyce and C.-J. Seger. Linking BDD based symbolic evaluation to interactive theorem-proving. In *DAC '93: In Proceedings of the 30th International Conference on Design Automation*, pages 469–474. ACM Press, New York, NY, USA, June 1993. ISBN 0-89791-577-1.

[63] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[64] K. L. McMillan. A Methodology for Hardware Verification using Compositional Model Checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[65] M. Kaufmann and J. S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Trans. Softw. Eng.*, 23(4):203–213, 1997.

[66] C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2): 123–193, 1999.

[67] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes.* Princeton Series in Computer Science. Princeton University Press, 1995. ISBN 0-691-03436-2.

[68] R. S. Lazic. *A Semantic Study of Data Indenpendence with Applications to Model Checking.* PhD thesis, Oxford University, 1999.

[69] W. Luk, G. Jones, and M. Sheeran. Computer-based Tools for Regular Array Design. In J. McCanny, J. McWhirter, and E. S. Jr., editors, *Systolic Array Processors*. Prentice Hall Intl., 1989. ISBN 0-13-473422-X.

[70] M. Pandey, G. York, D. Beatty, A. Jain, S. Jain, and R. E. Bryant. Extraction of Finite State Machines from Transistor Netlists by Symbolic Simulation. In *Proceedings of the IEEE International Conference on Computer Design (ICCD '95)*, 1995.

[71] S. Maclane. Groups, Categories and Duality. *Proceedings of the National Academy of Science*, 34:263–267, June 1948.

[72] M. Mano. *Digital Design.* Prentice Hall, Upper Saddle River, NJ, USA, Third edition, August 2001. ISBN 0130621218.

[73] K. L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In L. Pierre and T. Kropf, editors, *CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–233. Springer-Verlag, Berlin, 1999. ISBN 3-540-66559-5.

[74] T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993. ISBN 0-521-41718-X.

[75] T. Melham. Integrating Model Checking and Theorem Proving in a Reflective Functional Language. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods: 4th International Conference, IFM 2004: Canterbury, UK, April 4–7, 2004: Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 36–39. Springer-Verlag, Berlin, 2004. ISBN 3-540-21377-5.

[76] T. Melham and A. Darbari. Symbolic Trajectory Evaluation in a Nutshell. Unpublished report, available on request.

[77] T. F. Melham. A Package for Inductive Relation Definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, California, August 28–30, 1991*, pages 350–357. IEEE Computer Society Press, 1992. ISBN 0-8186-2460-4.

[78] T. F. Melham and R. B. Jones. Abstraction by symbolic indexing transformations. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design: 4th International Conference, FMCAD 2002: Portland, OR, USA, November 6–8, 2002: Proceedings*, volume 2517 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, Berlin, 2002. ISBN 3-540-00116-6.

[79] N. Mokhoff. *Intel, Motorola report formal verification gains.* published at http://www.EETimes.com, June 2001.

[80] J. T. O'Donnell. Hydra: Hardware description in a functional language using recursion equations and high order combining forms. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 309–328. North-Holland, 1988.

[81] J. T. O'Donnell. Generating Netlists from Executable Circuit Specifications. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 178–194. Springer-Verlag, Berlin, 1993. ISBN 3-540-19820-2.

[82] M. Pandey. *Formal Verification of Memory Arrays.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[83] M. Pandey and R. E. Bryant. Exploiting Symmetry When Verifying Transistor-Level Circuits by Symbolic Trajectory Evaluation. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18(7):918–935, 1999.

[84] M. Pandey, R. Raimi, D. L. Beatty, and R. E. Bryant. Formal Verification of PowerPC arrays using Symbolic Trajectory Evaluation. In *DAC '96: Proceedings of the 33rd annual conference on Design Automation*, pages 649–654. ACM Press, New York, NY, USA, 1996. ISBN 0-89791-779-0.

[85] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir. Formal Verification of Content Addressable Memories Using Symbolic Trajectory Evaluation. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 167–172. ACM Press, New York, NY, USA, 1997.

[86] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, Berlin, 1981. ISBN 3-540-10576-X.

[87] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. ISBN 1-55880-069-8.

[88] L. C. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, 1983.

[89] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, second edition, June 1996. ISBN 052156543X.

[90] A. Pnueli. The Temporal Logic of Programs. In *The 18th IEEE Symposium Foundations of Computer Science*, pages 46–57, 1977.

[91] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with Embedding Hardware Description Languages in HOL. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland, 1992.

[92] J.-W. Roorda and K. Claessen. Explaining Symbolic Trajectory Evaluation by giving it a Faithful Semantics. In *Proceedings of International Computer Science Symposium in Russia*, Lecture Notes in Computer Science. Springer Verlag, June 2006. To appear.

[93] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[94] D. A. Schmidt and B. Steffen. Program Analysis as Model Checking of Abstract Interpretations. In *SAS '98: Proceedings of the 5th International Symposium on Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer-Verlag, Berlin, 1998. ISBN 3-540-65014-8.

[95] C.-J. Seger. VOSS - A Formal Hardware Verification System User's Guide. Technical Report TR-93-45, Dept. of Computing Science, Univeristy of British Columbia, Vancouver, Canada, 1993.

[96] C.-J. H. Seger and R. E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Journal of Formal Methods in System Design*, 6 (2):147–189, March 1995.

[97] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An Industrially Effective Environment for Formal Hardware Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.

[98] N. Shankar. PVS: Combining Specification, Proof Checking, and Model Checking. In *FMCAD '96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, pages 257–264. Springer-Verlag, Berlin, 1996. ISBN 3-540-61937-2.

[99] N. Shankar. Combining Theorem Proving and Model Checking through Symbolic Analysis. In *CONCUR '00: Proceedings of the 11th International Conference on*

Concurrency Theory, volume 1877 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, Berlin, 2000. ISBN 3-540-67897-2.

[100] R. Sharp. *Higher-Level Hardware Synthesis.* PhD thesis, University of Cambridge, 2002.

[101] M. Sheeran. μFP, A Language for VLSI Design. In *LISP and Functional Programming*, pages 104–112, 1984.

[102] S. Singh. Circuit Analysis by Non-Standard Interpretation. In *Proceedings of the Second IFIP WG10.2/WG10.5 Workshop on Designing Correct Circuits*, pages 119–138. North-Holland, 1992. ISBN 0-444-89335-0.

[103] A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems*, 26(4):702–734, 2004.

[104] A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 9(2):133–166, 2000.

[105] P. Stanford and P. Mancuso. *Electronic Design Interchange Format Version 2 0 0, Recommended Standard EIA-548.* Electronic Industries Association EDIF Steering Committee, 1989.

[106] C. Stirling. Bisimulation, Model Checking and Other Games. Notes for Mathfit Instructional Meeting on Games and Computation, University of Edinburgh, June 1997.

[107] T. E. Uribe. Combinations of Model Checking and Theorem Proving. In *FroCoS '00: Proceedings of the Third International Workshop on Frontiers of Combining Systems*, volume 1794 of *Lecture Notes in Computer Science*, pages 151–170. Springer-Verlag, Berlin, 2000. ISBN 3-540-67281-8.

[108] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of 1st Symposium on Logic in Computer Science*, pages 332–344, June 1986.

[109] K. M. William N.N.Hung, Adnan Aziz. Heuristic Symmetry Reduction for Invariant Verification. In *6th ACM/IEEE International Workshop on Logic Synthesis (IWLS)*, May 1997.

[110] G. Winskel. *The Formal Semantics of Programming Languages: An Antroduction.* MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-23169-7.

[111] Y. Wolfsthal and R. M. Gott. Formal Verification: Is it Real Enough? In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 670–671. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-058-2.

[112] J. Yang and C.-J. H. Seger. Introduction to Generalized Symbolic Trajectory Evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):345–353, 2003.

Index