Static and dynamic debugging techniques for the HipHop.js language

Jayanth Krishnamurthy

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Techniques de débogage statique et dynamique pour le langage HipHop.js

Jayanth Krishnamurthy

Inria centre
at Université Côte d'Azur

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur
Dirigée par : Dr. Manuel SERRANO
Soutenue le : 21/04/2023

Devant le jury, composé de :
Prof. Julien DEANTONI - Président du jury
Prof. Gérard BERRY
Prof. Elisa Gonzalez BOIX
Dr. Timothy BOURKE
Dr. Alan SCHMITT

Inria

Techniques de débogage statique et dynamique pour le langage HipHop.js

Jury:
Président du jury
Prof. Julien DEANTONI, *Professeur, Université Côte d'Azur*

Rapporteurs
Prof. Elisa Gonzalez BOIX, *Professeur, Vrije Universiteit Brussel*
Dr. Alan SCHMITT, *HDR, Senior Researcher, INRIA Rennes*

Examinateurs
Prof. Gérard BERRY, *Professeur émérite, College De France*
Dr. Timothy BOURKE, *Professeur adjoint, École polytechnique, Paris*

Résumé

Un grand nombre d'applications informatiques modernes sont interactives, réagissant au stimulus fourni par l'environnement global. HipHop.js est un DSL réactif synchrone inspiré d'Esterel pour JavaScript construit sur Hop.js dédié à la programmation de telles applications. HipHop.js peut être utilisé pour développer des applications telles que les contrôleurs IoT, des interfaces graphiques réactives, etc. Traditionnellement les systèmes réactifs ont été utilisés dans des systèmes critiques car ils apportent des garanties de sécurités au delà de ce que peuvent fournir des langages généralistes. Néanmoins, comme pour les langages classique, un usage systématique et rigoureux de procédures de tests augmentent la confiance qu'un utilisateur peut apporter dans un système réactif. Le débogage est aussi une étape nécessaire si un dysfonctionnement est détecté. Cette thèse se concentre sur le développement d'un environnement dédié pour les programmeurs HipHop.js, facilitant, l'analyse et la mise au point de programmes. Il y a deux principales contributions dans cette thèse. Le premier est le support pour le débogage d'un classe d'erreurs typiques des systèmes réactifs appelées « erreurs de causalité » dans HipHop.js et le second est de fournir une infrastructure aux programmeurs HipHop.js pour étudier et comprendre le comportement de leurs programmes réactifs, identifier ainsi toute subtilité bogues et affiner leur source.

HipHop.js suit le modèle de synchronie parfaite introduit dans le langage de programmation Esterel. Ce mémoire de thèse commence par une brève introduction à la programmation réactive à l'aide de HipHop.js afin de rendre accessible la suite du document aux lecteurs peu familiers avec le modèle de la programmation réactive. Ce modèle peut conduire à ce que cette communauté nomme des « cycles d'erreurs de causalité ». Ceux-ci sont généralement difficiles à isoler et à réparer. Dans cette thèse, nous discutons en détail la prise en charge du débogage des erreurs de causalité qui peuvent être détectées à la compilation ainsi que lors de l'exécution. Nous commençons par fournir des exemples exemples expliquant l'origine et la formation des erreurs de causalité et les difficultés rencontrées pour les déboguer. Ensuite, nous présentons les techniques basées sur des algorithmes de graphes permettant de signaliser les erreurs avec des messages précis dirigeant les programmeurs directement vers la source des erreurs. Nous illustrons l'efficacité des méthodes proposées sur un exemple concret.

Dans une seconde partie, nous proposons un ensemble d'utilitaires destinés à faciliter la compréhension des comportements « temporels » des programmes HipHop.js et cela, dès les premiers stades du développement. Ces analyseurs de programmes fournissent aux programmeurs HipHop.js des interfaces faciles à utiliser qui permettent de simplifier et d'automatiser l'identification et la localisation des

erreurs. Ce manuscrit commence par présenter les analyseurs de programme. Ensuite, des exemples illustratifs expliquant les utilisations des utilitaires et leurs intérêts sont présentés. Les détails des implémentations, y compris les outils et la théorie pertinente, constituent la dernière partie de la thèse sur l'analyseur de programmes.

Mots clés : *Techniques de débogage, HipHop.js, Erreur de causalité, Programmation réactive synchrone.*

Abstract

Many modern-day computer applications are reactive in nature, continuously reacting to the stimulus from its environment. IoT controllers for orchestration, responsive GUIs, collaborative video games are some of those reactive applications. HipHop.js, a *synchronous reactive* DSL inspired by Esterel for JavaScript can be used to build reactive applications. The objective of the thesis is to provide supporting infrastructure for building error free reactive applications using HipHop.js. There are two main contributions in this thesis. The first one is the support for debugging a special class of errors - *causal errors*, typical to synchronous reactive systems and the second one is to provide a software infrastructure for HipHop.js programmers to understand, and test the temporal behavior of their programs, aiding in fault localization and improved debugging experience of HipHop.js programs.

HipHop.js follows the model of *perfect synchrony* introduced in the Esterel programming language and this may lead to classical *causality error cycles* during execution. These are generally difficult to isolate and fix. In this thesis, the support for debugging causality errors in compile time and run time is presented in detail. First, illustrative examples explaining the origin and formation of causality errors in HipHop.js and the difficulties faced in debugging them are presented. Then, the techniques based on graph-based algorithms that are used to construct better error messages directing programmers to the source of *causal errors* are presented. The effectiveness of the proposed methods is demonstrated with a real-world example.

As a second contribution, a program analyzer with utilities that can be used to understand the temporal behavior of HipHop.js programs, right from the early stages of program development is presented. The program analyzer aims to provide HipHop.js programmers easy to use interface that can simplify and automate error identification and localization. The presentation in the thesis has an introduction to the program analyzer, detailing the motivation behind the various utilities it has. Then, illustrative examples explaining the usage of utilities and their advantages are presented. The implementation details including the tools and relevant theory is the final part of the thesis on the program analyzer.

The thesis also includes a brief introduction to reactive programming using HipHop.js to make the thesis self-contained and also provide a basic introduction to readers from non-reactive background. The illustrative examples on HipHop.js constructs include control flow visualization for better understanding. The thesis concludes with a review of related literature and future work that can be carried out on the infrastructure presented in this thesis.

Keywords: *Debugging Techniques, HipHop.js, Causality error, Synchronous reactive programming.*

Acknowledgments

I'd like to express my sincere gratitude to my advisor Manuel Serrano, Director of Research, Team Indes, Inria centre at Université Côte d'Azur for all the guidance, support, and instruction he provided me throughout my doctoral research. Without his support and guidance, the thesis would not have reached the state, it is now. I am extremely thankful to Gérard Berry for his critical insights, and guidance on this research.

I thank my team members at Indes, Inria for guiding and supporting me throughout my stay in the lab. I thank the Inria centre at Université Côte d'Azur for financial, and research infrastructure support, and the Inria staff for all the technical, logistical, and procedural support. I also thank the staff at Université Côte d'Azur for all the help they extended towards the University procedures through the years of my Ph.D work.

My parents have been an inspiration throughout my life. They have always supported my dreams and aspirations. I'd like to thank them for all they have done for me. I thank my wife Nitha Sagar.J for always being there, and for sacrificing many things throughout the period of my Ph.D. work. I thank my brother Bharath.K for wholeheartedly supporting all my academic decisions and for unflinching support to realize my dreams. I thank my sister-in-law Navya.N, parents-in-law, and brother-in-law Preetham.J for being of great help and support during my Ph.D. journey.

Last but not least, I thank all my friends and well-wishers in France and India who have been a constant support and motivation throughout this journey.

Contents

Chapter 1

Introduction

"The greatest challenge to any thinker is stating the problem in a way that will allow a solution"
— Bertrand Russell

Recent years have seen massive dissemination of computing systems and applications to every aspect of human life in the form of smart devices, triggered by the coupling of advances in communication systems, web 2.0, Big data, IoT, Cloud computing, etc. These applications can range from as simple as smart bulbs to complex virtual surgery procedures and cyber knives. Blockchain technologies, advanced signaling management systems, and highly interactive, massively distributed video games are some other applications becoming very common in our day-to-day life. These computing applications have been harnessed with efficient and critical software developed in various programming paradigms. Reactive systems belong to one such software developed using reactive programming. Reactive systems have been used extensively in applications where tasks are control intensive and are triggered by environmental events requiring timely responses [24].

Programming languages like Esterel [24], Lustre [67] are some example languages used to develop reactive systems. These are also called synchronous reactive languages, since they follow "Synchrony hypothesis" - conceptually the programs react to external events in an instantaneous and deterministic way. Functional reactive programming languages (FRPs) have modernized the data flow style of reactive programming through languages and libraries like Elm [47], Rx from Microsoft, and React maintained by Meta (formerly Facebook).

HipHop.js is a dynamic DSL inspired by Esterel for JavaScript that can be

used for the orchestration of IoT devices, development of truly responsive UIs, etc. Typically, in reactive systems the reactive program and the environment are interdependent. Debugging reactive programs brings in some intrinsic challenges that are different when compared to classical software developed using traditional programming languages. This thesis is based on the study carried out in support of programming in HipHop.js, specifically building infrastructure for debugging of HipHop.js programs. Section 1.1 elaborates on the research problem, explaining the motivation and objectives of this thesis. Section 1.2 summarizes the contribution of this thesis, and Section 1.3 presents the overview of this thesis. Now, we discuss the research problem of this thesis.

1.1 The Research Problem - Motivation and Objectives

In programming and software development as part of successful practices, testing is employed which reveals the effect of errors. These errors can be broadly classified as syntax errors, type errors, run time errors, logical errors, etc. The debugging process which then follows should help in identifying the cause of the error and henceforth fix those errors [131]. The compilers of the language usually provide debugging support with respect to syntax and type errors. Run time errors and logical errors require something extra other than the compiler support in tracing them to correct. This is where the debugging support comes into picture. Without good debugging support, programming in any of the programming languages is difficult and would make programming an arduous exercise for programmers.

Why do bugs occur? In a survey on debugging [103], it is stated that, some reasons bugs may happen is due to misconceptions about language constructs by the programmer, parallelism bug (the ability to exploit constructs in a language that can schedule parallel execution of statements), expectation and interpretation mismatch problems, and problem with code familiarity (scenarios when working with code written by others). The debugging infrastructure by definition should not only help in resolving programming errors, they should also help in finding design errors, as it helps the developer understand the problem thoroughly. The software engineering life cycle prescribes continuous verification and validation

through all the phases of the software development life cycle rather than at the end phase - where the cost of correction would be more compared to the actual development cost [131].

The typical debugging process includes examining the error symptom (bug), identifying the cause, and fixing the bug is a difficult process as seen by experienced software practitioners. The symptoms may not give clear ideas indicating the cause of the bug. Also, the symptoms can be very difficult to reproduce, as the replay is needed to better understand the problem, and reproducing the same execution trace is a known hard task when the concurrent program elements are involved.

Though there is no absolute method for fixing all the bugs, there are some useful strategies mainly focussed on the localization of the errors [131, 117]. While debugging errors, the general strategy applied is to narrow down the search and focus on the likely source of errors. The successful practices apply one or many of the following strategies to narrow down the search: bottom-up and incremental development, using stepper debugger if there is one, visualization and logging information, hypothesis based searching, backtracking and binary search, problem simplification and abstraction of non-important parts. It is also observed that "forward reasoning program order" - simulating the program's execution is an important utility in the tool kit of the programmer who intends to debug [103].

Reactive systems interact continuously with their environment and at the speed decided by the environment. Debugging reactive systems is quite different when compared to traditional systems. For debugging traditional transformational programs using a stepper, typically we need the source file and a minimum of inputs initially. We can observe the behavior of the program with no additional inputs. Whereas for reactive systems, we require agreeing inputs at each step triggering many steps, for step-by-step execution.

The typical nature of reactive systems lends us the motivation for embarking on a study providing debugging support for HipHop.js programmers. HipHop.js follows the model of perfect synchrony introduced in the Esterel programming language. This leads to classical causality error cycles which are generally difficult to isolate and to fix. This is the first motivation for this thesis - providing debugging support for causality errors in HipHop.js. As noted earlier, some logical bugs are due to a wrong understanding of the program constructs, or may be due to a mismatch between expectation and interpretation. This is even more true with respect to

reactive systems development. This motivates the study for developing debugging support that can help programmers understand and visualize the behavior of their programs, and observe any unexpected behavior in their programs, narrow down the source of logical errors, and hence, possibly prevent design errors.

The motivation and the literature help us frame the objective of the study for this thesis: to develop a simple, easy-to-use infrastructure that can help debug HipHop.js programs. This support will possibly help programmers minimize the errors as much as possible throughout the development phase, allow programmers to play with their programs by simulating the executions, and in narrowing down the source of identified errors. This is not, in traditional terms a "debugger", as that would convey a sense of being a stepper that can step through executions. It is an infrastructure, we are mooting as part of a bigger debugging support for HipHop.js programs that will also include debugging support for causality errors. In the next section, we summarize the contribution of this thesis.

1.2 Contribution of the Thesis

This thesis, as a first contribution presents the study carried out to provide debugging support for special class of errors called "causality errors" that are unique to languages following "synchrony hypotheses". This support provides a better understanding of the origin of errors by means of providing better error messages, thereby resolving those errors. The second part of the contribution of the thesis is about the study to provide various utilities as part of "HipHop.js program analyzer" that meets the objectives we arrived at in the previous section. The program analyzer allows programmers to simulate executions of their programs and observe them for any unexpected behavior, helps understand the working of HipHop.js language constructs, helps visualize the control flow in an execution, and replay recorded executions. Also, in this thesis as part of the relevant background, we present a detailed introduction to HipHop.js language constructs and reactive programming. This helps people from non-reactive programming backgrounds to understand the basics of synchronous reactive programming and the use of HipHop.js for that.

The causality error tracer provides both dynamic and static debugging support, in the sense the "error tracer" provides meaningful error messages during compile

time and during execution time. Whereas, the program analyzer is more of a static support, where analysis of HipHop.js programs is done statically, not during execution. Next, we present the overview of the chapters in this thesis.

1.3 Organization of the Thesis

Chapter 1, introduces the thesis, giving a bird's eye overview of the research topic, its background, objectives, contribution, and structure of the thesis. Chapter 2 sets the context for the thesis by reviewing the basics of synchronous reactive programming and languages. It also introduces the language features of HipHop.js with some reactive programming examples and the compiling strategy used by HipHop.js compiler. Readers who have prior exposure to synchronous reactive programming and HipHop.js can safely skip this chapter without any loss of continuity to the next chapter. Chapter 3 builds on Chapter 2 and is on support for causality error tracing in HipHop.js. The chapter introduces the effects of synchrony on compilers/runtime of reactive languages, illustrates examples of causality errors in reactive languages and shows why resolving them is a difficult problem in HipHop.js for programmers without better error messages. It has sections on the approaches taken to resolve causality errors with the help of graph theory algorithms and method of solving causality error in HipHop.js. Further, it discusses the implementation details of the causality error tracing in HipHop.js, the evaluation of the implementation with real-world examples, and conclusion with future work very specific to causality errors. The material presented in some sections of Chapter 2 and all the sections of Chapter 3 are with little or no modifications of the sections from the published paper [86], hence the chapter is self-contained with its own sections on related work and conclusion.

Chapter 4 is about the introduction to the program analyzer. It introduces various utilities provided by the HipHop.js program analyzer, the motivation behind them , and their usage. Chapter 5 is about the implementation details of the analyzer. It explains in detail the way utilities of the program analyzer are implemented. Chapter 6 introduces another utility that can be part of HipHop.js debugging infrastructure. The "RecordPlayer" utility can record the state of a HipHop.js program during each reaction and then replays it with visualization of active control flow in the program. Chapter 7 deals with related work concentrated

more on the second contribution of this thesis, about utilities of program analyzer and "RecordPlayer". Chapter 8 is about the future work and concludes the thesis. The appendix section has chapters on BLIF, AIGER and NuSMV, which are some tools and representation formats used in the design of utilities of the HipHop.js program analyzer.

With this, we end the introduction chapter. In the next chapter, we provide the relevant background to understand the contribution of this thesis.

Chapter 2

Background

"Even theories must have foundations"

— Edgar Rice Burroughs

In this chapter we cover the background required to understand the contributions of this thesis. HipHop.js [25], which is the language of discussion of this thesis is based on Esterel's semantics, a synchronous reactive language. Hence, we first introduce reactive systems and programming in section 2.1. In section 2.2 we discuss important aspects of programming in Esterel related to this thesis. In section 2.3 we introduce HipHop.js programming and language features. In section 2.4 we give an overview of the compilation process of HipHop.js programs which is also relevant to understand the contribution of this thesis. We want to once again remind the readers that section 2.1 is presented here with little modifications from the published paper [86]. Readers who are familiar in reactive programming and HipHop.js can directly skip to chapter 3 without any loss of continuity.

2.1 Introduction to Reactive Systems and Programming

Computer systems that react continuously to their environment at the rate set by the environment forms a class of the so-called *reactive systems* [21, 70]. They differ from classical computing systems which take input at the start of execution and produce output before terminating. Furthermore, they also differ from traditional

interactive systems like operating systems which endlessly interact with their environment at their own speed (in contrast to the speed determined by the environment). A reactive system can be perceived as a black box that perpetually receives some input events as external stimuli and reacts to them by producing some output events as their behavior. This output may successively affect the production of later stimuli by the environment.

Starting in the early 80s, several languages have been proposed for programming such systems. Amongst those, some opted for synchrony, making concurrent programs deterministic, which has many advantages: programs are simpler, and when analyzing, there is no need to take into account the choices that the operating system will make. Also, deterministic programs show reproducible behavior, which is advantageous for tests and validations [32]. On the flip side, handling concurrency and determinism is quite complex. Resolving the design trade-off issue between concurrency and determinism warranted special purpose languages called *synchronous languages*, designed specifically for developing reactive systems. Among these reactive languages, the *Esterel* [24] language takes imperative approach, *Lustre* [38], *Signal* [18], and *Lingua Franca* [97] are data flow based, and *SCADE* [23], *Ptolemy II* [90] are based on graphical formalisms, *SCADE 6* [45] which is both textual and graphical. Now, we introduce the programming model followed in Esterel language.

2.2 Programming with Esterel

In this section, we provide a quick introduction to the programming and execution concepts of Esterel that HipHop.js reuses.

2.2.1 Programming Model

The architecture of an Esterel program is typically made of one or several modules. Broadcast is the communication mechanism: if one wants to communicate, it broadcasts a value, so that every other module can see it instantly. A signal is a logical unit of interaction and information diffusion. Signals can be emitted as presence bits, possibly with some attached value. They can be instantly tested for presence and their value can be read instantly in any part of the program (in their

scope). The framework of signals is used to unify both internal communication (between various modules of an Esterel program) and external communication (between the Esterel program and its environment). Even if it has multiple emitters, a signal can have only one value and one status within each reaction. To keep this true, multiple emitters need to be combined before the signal can be accessed by readers.

Modules have a declared interface that interacts with the external world in the form of input signals, output signals, and input/output signals. An input signal for a module can be an output signal of another module and vice versa. If the need arises for the reactive system to receive a signal from the environment and also send the same signal back to the environment after some processing or filtering, then an input/output signal can be used, those that can be both read as well as emitted by the module. Figure 2.1 illustrates the architecture of a typical Esterel program made up of various modules with broadcast communication set up between them. The emission of signals can be simplified with a simple data bus analogy [109], the "wire" in the bus for a signal is live with the information of the signal, when that signal is emitted by any of the modules. All modules, including the one that emitted this signal can listen to this wire and read the information emitted through the signal. Once the current reaction instance is over, the bus resets, listening for any successive signals to be put by the environment on the wires. In brief, all input signals that are received from the environment as well as those signals emitted by the system as part of its behavior are available to all the modules in the same reaction instant.

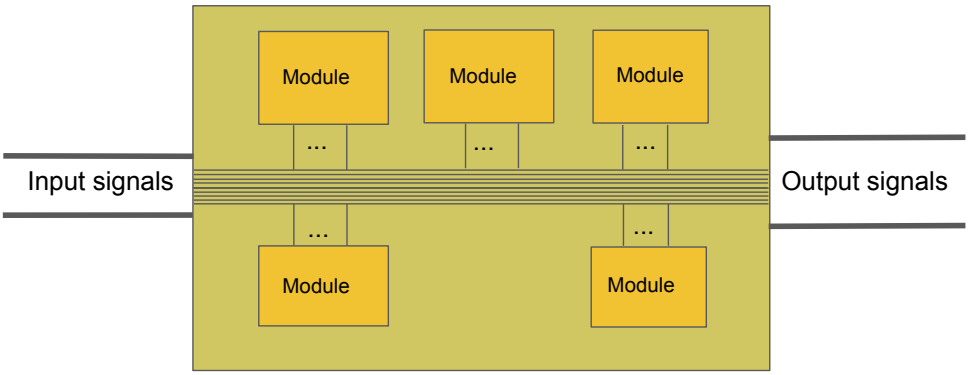


Figure 2.1: Graphical illustration of a typical Esterel program.

2.2.2 Execution model - Reaction and Instants

According to Esterel's programming model, a reactive system *reacts* when activated with input events by producing output events. In the Esterel terminology, *instants* are the moments when a reactive system *reacts*, and a collection of instants traces the system's life history [34].

Esterel instants are synchronous and deterministic (perfect synchrony hypothesis). That is, the execution of an instant takes no time, as if the computer executing it is infinitely fast. Of course, this is an intellectual illusion but it means that within an instant, the program cannot observe any temporal dependency. Within an instant, all the components of a program observe the same state and values for signals. For signals that are emitted several times during the instant, all the parallel and sequential branches of the program observe exactly the same set of emitted values. A reaction is complete when the system has reacted to all internal and external events.

The perfect synchrony hypothesis is a prominent feature of the Esterel design, which brings most of its expressiveness but also comes with a cost. It constrains the implementation that either needs static analyses (as Esterel compilers do) or dynamic detection (as HipHop.js does) to rule out programs that do not fit within this model, and, as we will see, it makes debugging more complex (discussed in detail in chapter 3). Now, we introduce the HipHop.js language features with simple programming examples.

2.3 HipHop.js

Though initially HipHop.js was developed for Hop.js [125, 126] (multitier JavaScript extension), the code generated by HipHop.js can be used in any client or server JavaScript environment like Node.js. HipHop.js combines the three traditional models of programming (computation), transformational programming, asynchronous concurrency, and synchronous reactive programming. HipHop.js can be used to develop complex web application interfaces and IoT controllers, which are dynamic in nature. HipHop.js blends Esterel's synchrony with JavaScript's asynchrony, simplifying the cooperation between synchronous and asynchronous activities that are typical in these application domains. HipHop.js differs from Esterel in hav-

ing its own syntax and programming model adapted to the web. For example, HipHop.js supports partial reconfiguration of programs between two synchronous reactions (explained in detail in section 2.2.2), while maintaining consistency of the control state. Readers who are interested for a thorough treatment including goal, design and compilation of HipHop.js are requested to refer the original literature on HipHop.js [25] and also the website[1] for recent updates.

In the following section, we incrementally introduce HipHop.js language constructs which serves two purposes. First, for the readers who are new to synchronous reactive programming, this acts as a tutorial review to reactive programming through HipHop.js. Second, it serves the purpose of setting the context of the language for the thesis contribution, thereby maintaining continuity of readability.

2.3.1 HipHop.js Language Features

The important features in HipHop.js are modules, reactive machines, signals, control flow statements and support for seamless integration of asynchronous and synchronous operations. We start with modules.

Modules

Modules in HipHop.js define a reusable behavior and are the basic execution units helping in modular development of reactive software. Modules will have an interface and a body defining their behavior. The body of a module includes declared arguments and HipHop.js statements. The arguments are generally signal declarations and is the interface between other modules and external environment of the reactive system. A module can use signals that are defined in its argument list and also local signals which are local to that particular module. The interface signals have direction, in the sense that they can be input (`in`), output (`out`) or bidirectional (`inout`) signals. Since local signals are local to a module and do not interact with the external environment, they do not have any declaration of directions. The following code listing 2.3.1 presents a typical modules in HipHop.js with interface declarations of `sig1` and `sig2` as input and output signals respectively, `localSig` declared as a local signal (`signal`) and the body containing HipHop.js statements. The `mod1` is the name of the module.

[1] `http://hop.inria.fr/home/hiphop/index.html`

```
1 hiphop module mod1() {
2     in sig1;
3     out sig2;
4     signal localSig;
5     ...;// hiphop.js statements
6     ...;// hiphop.js statements
7 }
```

A HipHop.js module is executed by loading it directly into a reactive machine. These machines are the ones generated by the HipHop.js compiler for the reactive behavior we define in the module. Once the machine is created, it can be used as JavaScript values and made to react to various inputs from the external environment based on the interface declared.

Before introducing HipHop.js programming, we present the initial boiler plate required to program with HipHop.js. To create hiphop reactive machine, as a first step we need to import the "`@hop/hiphop`" library which has the utility for doing so. The following JavaScript statement imports the "`@hop/hiphop`" DSL and makes all the bindings it exports available through the identifier "`hh`".

```
import * as hh from "@hop/hiphop";
```

Using the bindings in "`@hop/hiphop`" and HipHop.js module defined by the programmer, a new reactive machine, *e.g.*, "`machine`" can be created by using the following statement.

```
const machine = new hh.ReactiveMachine("module_name");
```

The reactive "`machine`", triggers reactions for each instants based on stimulus from the environment.

```
machine.react(["signal_name"]);
```

With this very basic introduction to modules and the creation of reactive machines, we jump to other HipHop.js statements. We proceed with statements handling *signals*.

Signals

Signals are the means to communicate with other modules and the external world. Signals are declared as part of the interface declaration of a module and these signals can be pure signals without any value or valued signals, similar to Esterel.

Further, local *signals* can be used with prior declarations within a module `M`. These are local to module `M`, its parts and submodules, and invisible to the other modules and external environment. As introduced earlier, the interface signals are declared with directions, and for local signals the keyword `signal` is used to declare inside a module. Before illustrating the declaration of signals, we present some of the HipHop.js statements used along with signals. In HipHop.js, sequential statements are separated with ';' separator. For instance, in `P; Q;` the statement `P` is run until it terminates and then `Q` is run in the same reaction instant. The absence of a signal in an instant means there is no emission of that signal in that instant. To emit a signal in an instant, `emit` statement is used. The following diagram and listing illustrates the usage of `emit` statement and signal declaration. For the sake of clear understanding of reaction instants, we also present the reaction timeline alongside. In the reaction timeline, we present the input and output signals at specific reaction instants. In the illustration, signals above the x-axis are input signals, and below are output signals.

```
1 hiphop module prgEmitAB() {
2     out A,B;
3
4     emit A();
5     emit B();
6 }
```

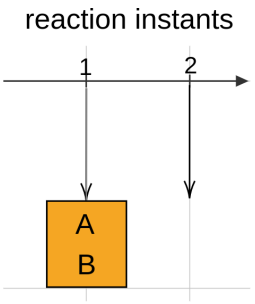


Figure 2.2: Signal declarations and emit statement usage with timeline.

In the reaction timeline, since signals `A` and `B` are output signals, they are present below the x-axis and both are emitted at the same instant. To make it more clear, we provide a snapshot of the flow of control in the reaction instant `1` as follows.

From the illustration, we see that after the control begin active at line 4, in the same instant statement it is also active at line 5. Hence, signals `A` and `B` are emitted in the instant `1`, which is highlighted in source file at line 2 also.

Signals can also be used in expressions by using `.pre`, `.now`, `.nowval`, `.preval`

```
1  hiphop module prgEmitAB() {
2      out A,B;
3
4      emit A();
5      emit B();
6  }
```

Figure 2.3: Visualization of control flow in the instant 1

attributes. The testing of signal's presence or absence at an instant can be done using the `if` construct: `if (sig_expression) {...} else {...}`. In the following listing and figure 2.4, we illustrate the above usage with reaction instants timeline.

```
1 hiphop module prgIf() {
2     in A;
3     out B;
4
5     if (A.now) emit B();
6 }
```

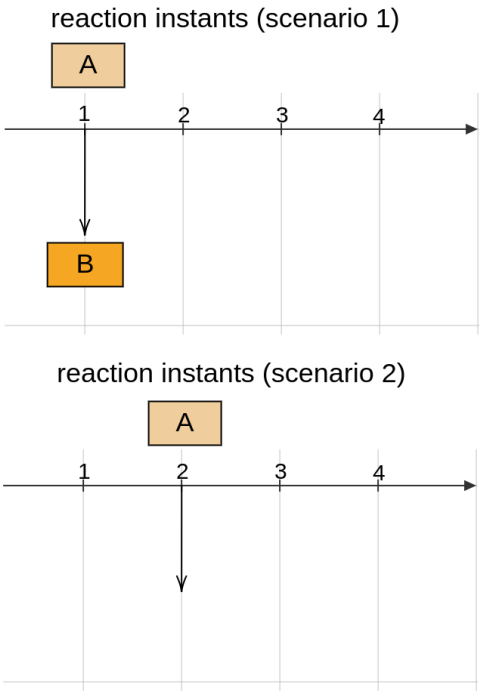


Figure 2.4: If construct example listing and timeline of two scenarios.

According to the listing the testing for presence happens in the first instant and for the value of signal `A` in that instant (`.now`). From the scenario 1 timeline, we see that since signal `A` is present, signal `B` is emitted. Whereas in scenario 2, since the testing for presence of signal `A` fails in instant `1`, there is no emission of signal `B` and the whole program is terminated and thus remains irresponsive to all

subsequent inputs, hence no emission in instant 2 even though there is presence of signal A. This can be better understood by visualizing the control flow inside the module of scenario 2 as follows:

```
1  hiphop module prgIf() {
2      in A;
3      out B;
4
5      if (A.now) emit B();
6  }
```

(a) scenario 2 - instant 1.

```
1  hiphop module prgIf() {
2      in A;
3      out B;
4
5      if (A.now) emit B();
6  }
```

(b) scenario 2 - instant 2.

Figure 2.5: Visualization of control flow for scenario 2.

We see that by instant 2, there is no active control at line 5, which was available in instant 1. The issue with scenario 2 can be improved by using simple `loop` construct which ensures the control back at the `if` construct as follows. The design of the language has a restriction that the body of the `loop` must not terminate in the same instant it was started (instantaneous loop). When using `loop` constructs care should be taken to ensure there are no "instantaneous" cycles (explained in detail in chapter 3) with the help of the `yield` statements. Executions can be suspended for an instant and resumed in the next instant by using `yield` construct. The improved example is illustrated as follows:

```
1 hiphop module prgIfLoop() {
2     in A;
3     out B;
4
5     loop{
6         if (A.now) emit B();
7         yield;
8     }
9 }
```

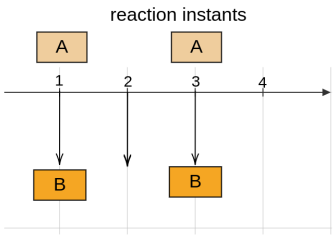


Figure 2.6: Simple looping listing and timeline

The visualization of the control flow for the above three instants is also presented here:

```
1  hiphop module prgIfLoop() {
2      in A;
3      out B;
4
5      loop{
6          if (A.now) emit B();
7          yield;
8      }
9  }
```

(a) A is **present** - instant 1.

```
1  hiphop module prgIfLoop() {
2      in A;
3      out B;
4
5      loop{
6          if (A.now) emit B();
7          yield;
8      }
9  }
```

(b) A is **absent** - instant 2.

Figure 2.7: Control flow visualization of instants 1 and 2 for the module `prgIfLoop`

```
1  hiphop module prgIfLoop() {
2      in A;
3      out B;
4
5      loop{
6          if (A.now) emit B();
7          yield;
8      }
9  }
```

Figure 2.8: A is **present** - Control flow visualization of instant 3.

In the above example, we tested for the presence of signal `A` by taking the `.now` val (value at the instant when it is tested). This can also be tested by using `.pre` val of signal `A`, wherein the value of signal `A` in the previous instant will be used for testing. It is illustrated as follows.

```
1 hiphop module prgIfPre() {
2     in A;
3     out B;
4
5     loop{
6         if (A.pre) emit B();
7         yield;
8     }
9 }
```

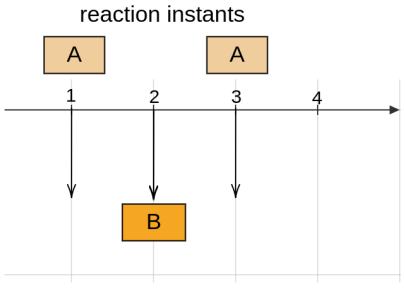


Figure 2.9: looping listing and timeline with `A.pre`

HipHop.js provides a construct using which one can wait on a signal, the `await` statement. We present it next.

await

The `await` statement is used for waiting on a signal. Typically the `await` statement checks for the presence of the signal in the next reaction instant. But, if we want to wait on signal in the immediate reaction instant, then the keyword `immediate` should be used alongside `await` construct. The value of the signal to be considered is determined by using `.now` or `.pre`. The following is the illustration of the usage of `await` construct. Here, the presence testing of signal `A` happens from instant `2` with the value of signal `A` at that instant.

```
1 hiphop module prgawaitANow() {
2     in A;
3     out B;
4
5     await(A.now);
6     emit B();
7 }
```

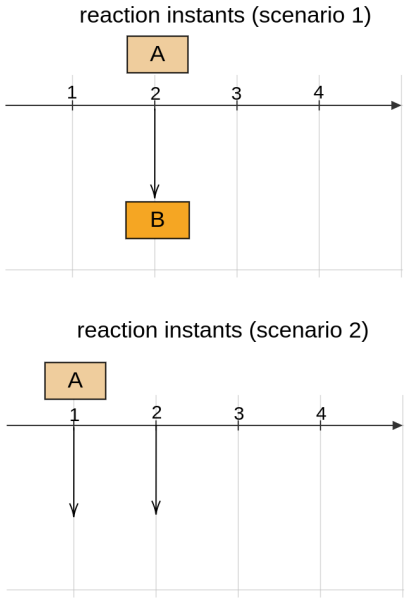


Figure 2.10: `await` statement - `[signal].now` example, Scenario 1

In scenario 1, we see that signal `A` is present in instant 2, and as expected there is emission of signal `B` (as waiting happens from instant `2`). In scenario 2, we see that signal `A` is present in instant 1, but the wait happens from instant 2 and hence the input signal is neglected in instant 1. In the second instant, there is no input `A`, hence no emission of `B`, but the wait continues. This is illustrated by visualizing the control flow as illustrated here:

```
1  hiphop module prgawaitANow() {
2      in A;
3      out B;
4
5      await(A.now);
6      emit B();
7  }
```

```
1  hiphop module prgawaitANow() {
2      in A;
3      out B;
4
5      await(A.now);
6      emit B();
7  }
```

Figure 2.11: `await` statement - `[signal].now` example, Scenario 2

For the sake of completeness, we illustrate the earlier example with `A.pre` values. The listing and corresponding timeline can be seen here:

```
1 hiphop module prgawaitAPre() {
2     in A;
3     out B;
4
5     await(A.pre);
6     emit B();
7 }
```

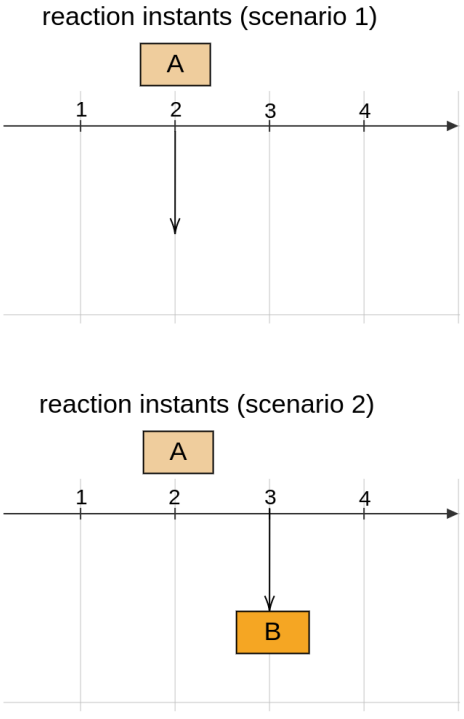


Figure 2.12: `await` statement - `[signal].pre` example

Now, if we want to make the wait on a signal from the first instant, then we can use the `immediate` keyword alongside `await`. We illustrate an example here with listing and corresponding reaction timeline.

```
1 hiphop module prgawaitImmANow() {
2     in A;
3     out B;
4
5     await immediate(A.now);
6     emit B();
7 }
```

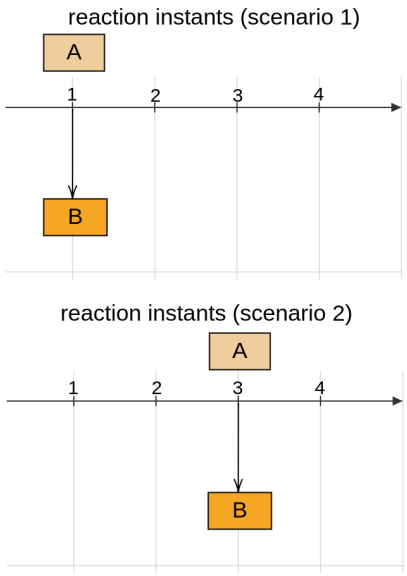


Figure 2.13: `await immediate` statement examples.

In the above example, the wait happens from the first instant on signal `A` and the value of signal `A` at the instant when it is true. For the sake of completeness, we present an example listing and timeline when `immediate` is used but the value of signal `A` used for waiting is from the previous instant to the instant when it is true.

```
1 hiphop module prgawaitImmAPre
2     in A;
3     out B;
4
5     await immediate(A.pre);
6     emit B();
7 }
```

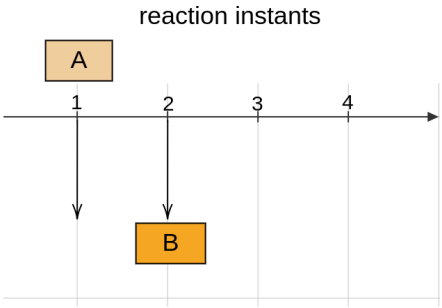


Figure 2.14: `await immediate` statement example.

From the time line, we see that, in the first instant signal `A` is present, but in the `await` construct, `A.pre` value is used which will be false, hence the control remains active in the `await` construct. In the second instant, when `await` construct checks for `A.pre` value which is true, makes the active control to flow to the next

statement (`emit B()`), hence we see the emission. Further, the `await` construct can also be used to wait on a specific number of occurrences of a signal. For this, the `count` construct should be used. In the following illustration with code listing, we see the usage of `count` keyword along with `await` and corresponding timeline in Figure 2.15.

```
1 hiphop module prgawaitCount() {
2     in A;
3     out B;
4
5     await count(3, A.now);
6     emit B();
7 }
```

Listing 2.1: `await count` construct usage example

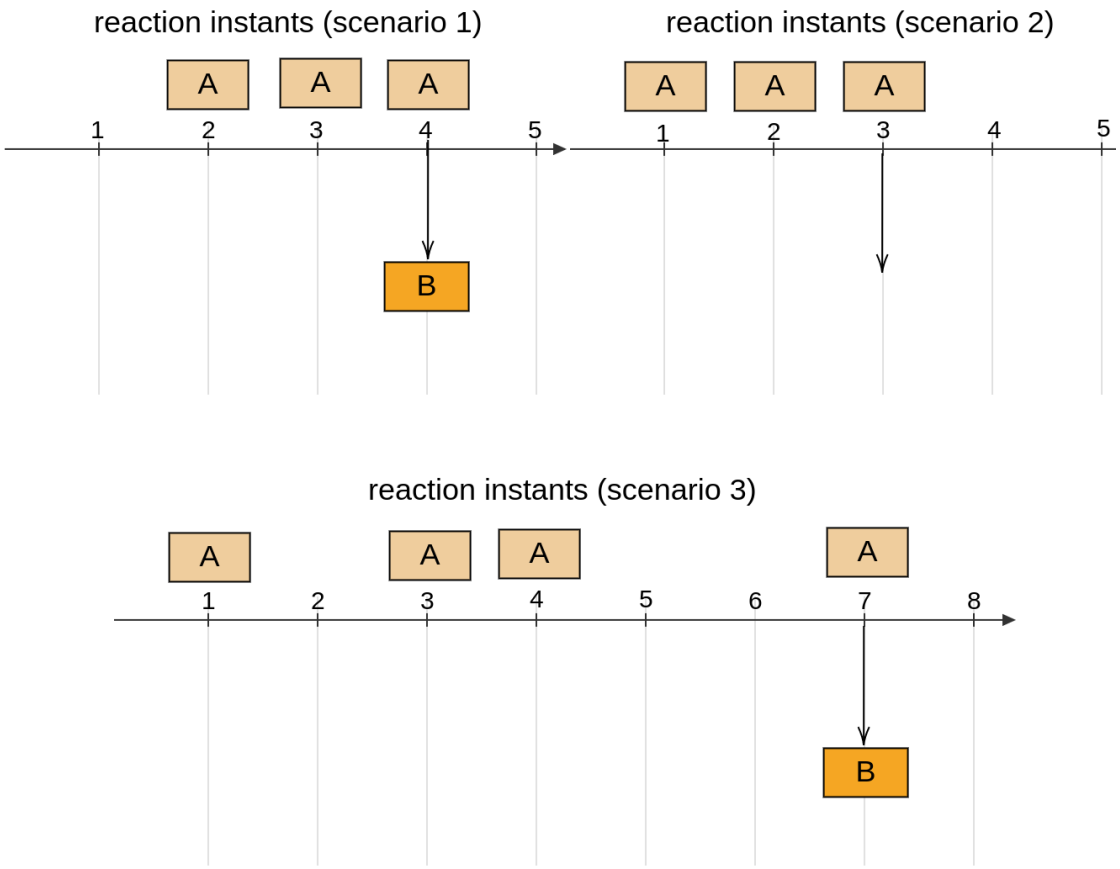


Figure 2.15: await count example timeline.

In the listing above wait happens from the second instant for occurrences of signal `A`, `3` times with the values being at those instants (`.now`). Next we present about parallel constructs `fork {...} par {...}` available in HipHop.js for parallel composition.

fork {...} par {...}

HipHop.js provides **parallel constructs** to implement parallelism and concurrency. They are within `fork {...} par {...}` constructs. There can be multiple `par` blocks, all the blocks start executing at the same instant and continue in parallel. Even though each block may not terminate at the same instant, the entire `fork/par` construct terminates only after the termination of all the blocks. In the following we provide an example listing.

```
1 hiphop module prgForkPar() {
2     in i1,i2;
3     out o1,o2,forkParDone;
4
5     fork {
6         await(i1.now);
7         emit o1();
8     } par {
9         await(i2.now);
10        emit o2();
11    }
12    emit forkParDone();
13 }
```

Listing 2.2: `fork {...} par {...}` construct usage listing

The following figure illustrates various timelines for the above listing. In scenario 1, only signal `i1` is present in the second instant, hence only emission from `fork` block - `o1`, and `par` continues waiting for signal `i2` . likewise, in scenario 2, only signal `i2` is present in the second instant, hence only emission from `par` block - `o2`, while the `fork` block continues to wait for signal `i1`. In scenario 3, both `i1` and `i2` are present, hence all the output signal emissions in the same instant and active control flows out of `fork {...} par {...}` block after this instant. Scenario 4 can be understood the same way.

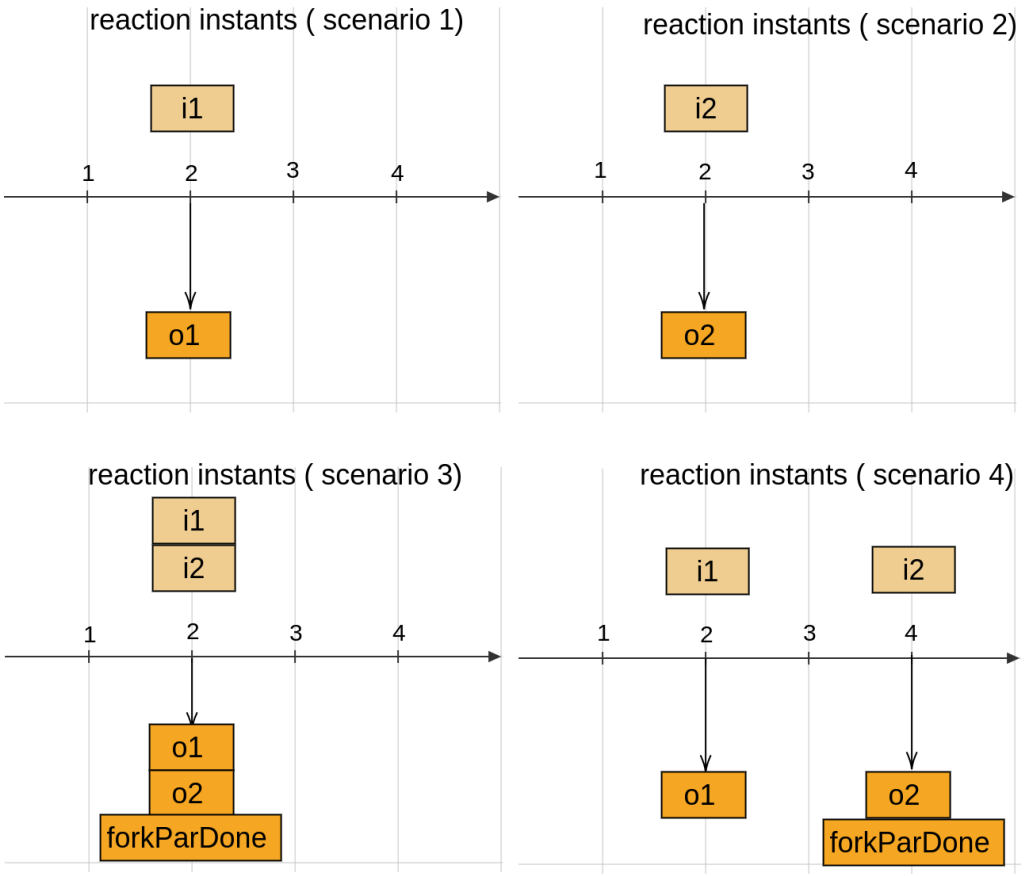


Figure 2.16: Parallel statement example.

Next we present examples on reusability of modules by composing them within one another.

Composition of HipHop.js Modules

Modules in HipHop.js promote modular programming, independent behavior can be defined inside individual modules and can be reused as with the principle "write once use many times" approach. Programmers can compose modules within one another. In HipHop.js, another method of running a module (the first one being loading directly to a reactive machine) is to run it inside another module via the "`run`" statement. It takes the following form:

```
run module() { signal_bindings }
```

We illustrate this concept with a small example as follows: In this example we are mimicking the behavior of a very simple coffee/tea vending machine in its basic form. Here, the user inputs the type of beverage they want, the machine delivers it. So, initially we develop two modules, one each for dispensing the respective beverage Coffee and Tea. In the following listing from line 1 we start with declaring a Coffee dispenser module, starting with its interface and a simple body with just an "`emit`" statement to emit a signal for dispensing the respective beverage at line 3 as follows.

```
1 hiphop module coffeeDisp() {
2    out C;
3    emit C();
4 }
```

Similarly, we declare a Tea dispenser module as follows.

```
5 hiphop module teaDisp() {
6    out T;
7    emit T();
8 }
```

Next, we use the above modules inside another module to compose a behavior of a vending machine. We call this module "`vendingMach`". This one takes input signals specifying Coffee or Tea. Based on the input signal, the request is forwarded to one of the above defined modules. In the following module line 10 and line 11 declares the input, output interface. Then it loops through waiting for input signal at each instance. Based on input signal at line 14 or line 17, the "`teaDisp`" module for dispensing Tea or "`coffeeDisp`" module for dispensing Coffee is executed using the `run` construct that we introduced earlier. In each of these modules, the arguments mapping are done using `as` keyword, which can be used for mapping arguments between various modules. Here, the output signal of `vendingMach`, which is signal `Tea_cup` is declared "`as`" the output signal `T` of the module `teaDisp` providing mapping between the arguments of modules. Also, similarly for the signal `Coffee_cup`. To prevent infinite looping in a single reaction instant, the module pauses after each output event with the help of `yield` construct at line 20.

```
9 hiphop module vendingMach() {
10   in Tea,Coffee;
11   out Coffee_cup, Tea_cup;
12
```

```
13    loop {
14       if (Tea.now) {
15          run teaDisp() { Tea_cup as T };
16       } else
17       if (Coffee.now) {
18          run coffeeDisp() { Coffee_cup as C };
19       }
20       yield;
21    }
22 }
```

Here, we see some of the reactions for the input signals in each instant.

```
23 machine.react("Coffee");// [ 'Coffee_cup' ]
24 machine.react("Tea");    // [ 'Tea_cup' ]
25 machine.react("Coffee");// [ 'Coffee_cup' ]
26 machine.react("Coffee");// [ 'Coffee_cup' ]
```

Next, we introduce the usage of preemption constructs `abort` and its variations.

abort

When an emission of some signal or signals happen, it may warrant killing of a block of statements as a requirement. HipHop.js provides `abort` construct. The abortion statement kills its body based on the occurrence of some criteria. There are *strong* and *weak* flavors of `abort` construct. In the strong abort case, the body does not get the control, while in weak abort case, the body gets the control for the last time. Based on "when to react" - present instant of the occurrence of the signal or in the next instant, and "when to kill" present instant or next instant, we have four possibilities:

1. `abort()`,
2. `abort immediate()`,
3. `weakabort()`, and
4. `weakabort immediate()`.

The following diagram 2.17 illustrates[2] the above said four options and usage pattern.

[2]adapted from Esterel Tutorial, Berry et al, 2005

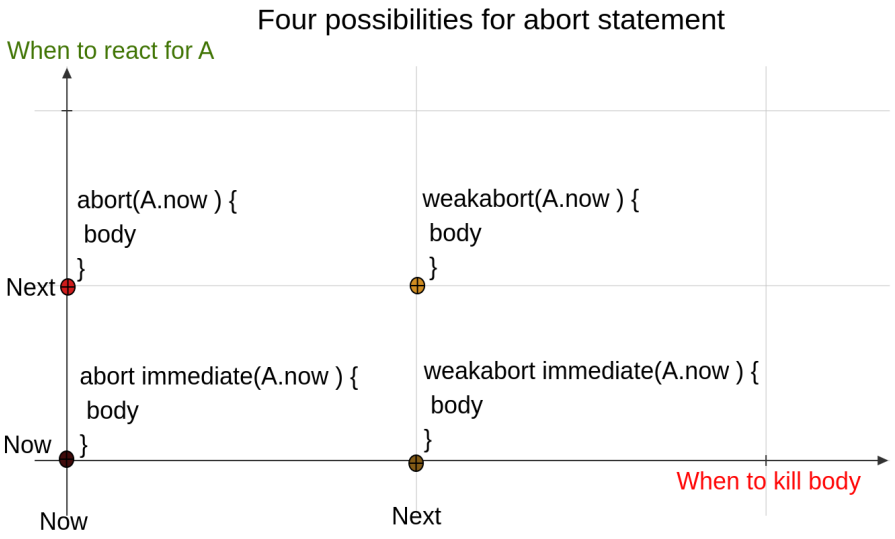


Figure 2.17: Preemption construct `abort` in HipHop.js.

In another form of preemption support, we can declare labeled **escape** blocks for trapping an execution. This can be used with the help of labeling a block along with `break label` construct. The usage and different reaction timeline is illustrated here.

```
1  hiphop module prgTrap() {
2     in B;
3     out A, C, D;
4     lb1: {
5        fork {
6           yield;
7           emit A();
8           yield;
9           break lb1;
10       } par {
11          await (B.now);
12          emit C();
13       }
14    }
15    emit D();
16 }
```

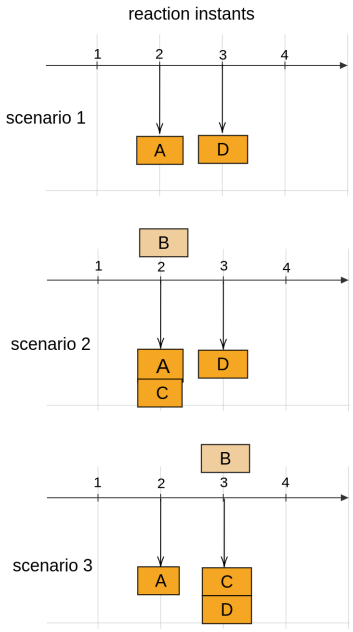


Figure 2.18: Labeled escape example.

For the sake of clarity in understanding, we present the visualization of the active control flow for **scenario 3** as follows.

```
1  hiphop module prgTrap() {
2     in B;
3     out A, C, D;
4     lb1: {
5        fork {
6           yield;
7           emit A();
8           yield;
9           break lb1;
10       } par {
11          await (B.now);
12          emit C();
13       }
14    }
15    emit D();
16 }
```

(a) instant 1

```
1  hiphop module prgTrap() {
2     in B;
3     out A, C, D;
4     lb1: {
5        fork {
6           yield;
7           emit A();
8           yield;
9           break lb1;
10       } par {
11          await (B.now);
12          emit C();
13       }
14    }
15    emit D();
16 }
```

(b) instant 2

```
1  hiphop module prgTrap() {
2     in B;
3     out A, C, D;
4     lb1: {
5        fork {
6           yield;
7           emit A();
8           yield;
9           break lb1;
10       } par {
11          await (B.now);
12          emit C();
13       }
14    }
15    emit D();
16 }
```

(c) instant 3

Figure 2.19: Active control flow visualization for labeled escape example, scenario 3.

These label blocks can be nested and the outer block takes precedence, that is if several blocks are exited by simultaneously executing breaks in parallel, only the outer break exit matters, all the others are discarded. Next, we present the usage of another construct **suspend**.

Preemption statements abort, and lexical escapes terminate some block or computation, but if there is a need to resume it later, HipHop.js has `suspend` statement. Instead of preemption, this merely suspends the block in an instant and resumes it later based on certain criteria. The following illustrates the usage

and corresponding timeline.

```
1 hiphop module prgSusp() {
2     in A;
3     out B;
4     loop {
5         suspend(A.now) {
6             emit B();
7             yield;
8         }
9
10    }
11 }
```

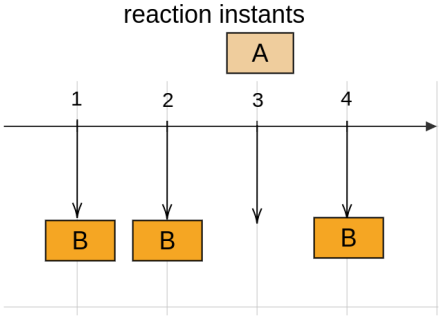


Figure 2.20: Suspension statement example.

For the sake of clarity in understanding, we present the visualization of the active control flow as follows.

```
1   hiphop module prgSusp() {
2       in A;
3       out B;
4       loop {
5           suspend(A.now) {
6               emit B();
7               yield;
8           }
9       }
10  }
```

(a) instant 1

```
1   hiphop module prgSusp() {
2       in A;
3       out B;
4       loop {
5           suspend(A.now) {
6               emit B();
7               yield;
8           }
9       }
10  }
```

(b) instant 2

```
1   hiphop module prgSusp() {
2       in A;
3       out B;
4       loop {
5           suspend(A.now) {
6               emit B();
7               yield;
8           }
9       }
10  }
```

(c) instant 3

```
1   hiphop module prgSusp() {
2       in A;
3       out B;
4       loop {
5           suspend(A.now) {
6               emit B();
7               yield;
8           }
9       }
10  }
```

(d) instant 4

Figure 2.21: Active control flow visualization for `suspend` statement example.

In the next section we present and illustrate the usage of integrating plain JavaScript code with HipHop.js.

Synchronous and asynchronous operations

One of the design goal of HipHop.js is being to integrate seamlessly with native JavaScript code. These code can be synchronous functions or asynchronous in nature. The following example illustrates the integration of plain JavaScript functions with HipHop.js reactive machine. In the following listing 2.3.1 we have a very simple JavaScript function which checks whether a number is prime or not and returns the result. This function is integrated with HipHop.js reactive machine to simulate a prime number checker in reactive programming.

```
7 function isPrimeFunc(num) {
8     for (let i = 2, n = Math.sqrt(num); i <= n; i++) {
9         if (num % i === 0) return false;
10    }
11    return num > 1;
12 }
```

The following module `prgJSync` is the HipHop.js module which will coordinate with the JavaScript function `isPrimeFun`. `N` is the input signal, with `isPrime` as the output signal. Now, in HipHop.js any plain JavaScript statements can be included within `hop {}` construct. Also, to provide arguments exchange between HipHop.js module and plain JavaScript functions, we can declare a variable with `let` and can be used for information exchange. We get the value of the number to be checked for prime, through signal `N`, specifically `N.nowval` and assign it to a variable which can be used as an argument for the JavaScript function. So, we call the function 2.3.1 inside `hop {}` construct at line 21 and also assign its return value to another variable. Finally, with the returned value we emit the output signal `isPrime` in line 22.

```
14 hiphop module prgJSync() {
15    in N;
16    out isPrime;
17
18    loop {
19        let keyin = N.nowval;
20        let keyout;
```

```
21          hop { keyout = isPrimeFunc(keyin); }
22          emit isPrime(keyout);
23          yield;
24      }
25  }
```

Through line 29 to line 31 we see various reactions on the machine in various instants with different input values.

```
29 machine.react( { N:1024 } );   // [ 'isPrime(false)' ]
30 machine.react( { N:9973 } );   // [ 'isPrime(true)' ]
31 machine.react( { N:-1 } );     // [ 'isPrime(false)' ]
```

The construct `async()` is used to integrate asynchronous operations and blocks inside HipHop.js. It provides a way to control nondeterminism. It provides integration of nondeterministic asynchronous computation with synchronous computations. The syntax is as follows:

```
async ([ident]) { ... }
[kill { ... }]
[suspend { ... }]
[resume { ... }]
```

The optional identifier in `[ident]` can be used to notify the completion of the asynchronous block as that `ident` is emitted once the asynchronous block completes its execution and returns. In the JavaScript code that triggers the HipHop.js module execution, `this.notify()` construct triggers the emission of `ident` signaling the completion of asynchronous execution. The emission will have `resolve or reject` of a promise returned by the asynchronous block. This emission can be further used by examining the values (`.nowval`). Further, the optional blocks preceded by `kill, suspend, resume` can be killed, suspended or resumed on need basis. The following is an example of the usage of a simple `async()` block in HipHop.js. We use the same JavaScript function of the previous example but asynchronous version. In the following listing line 6 through line 23 defines an asynchronous JavaScript function which checks whether a number is prime or not and returns the result after a random amount of time delay. Since it is an asynchronous function, we use the promises and resolve the result with a JSON object containing the result of checking and simulated time of delay as a time used to process.

```
6  function prime(num) {
7      let tUnit = Math.floor(Math.random() * 4);
```

```
8     return new Promise((resolve, reject) => {
9         const primCheck = num  => {
10            for (let i = 2, n = Math.sqrt(num); i <= n; i++) {
11                if (num % i === 0) {
12                    resolve({ "value": num,
13                              "test": false,
14                              "processed␣time": tUnit });
15                }
16            }
17            resolve({ "value": num,
18                      "test": num > 1,
19                      "processed␣time": tUnit });
20        }
21        setTimeout(() => primCheck(num), tUnit*1000);
22    });
23 }
```

The following listing is the HipHop.js module that will coordinate with the above function. We use the `async()` construct as introduced earlier in line 29. We have used a local signal to signal the notification of result from the asynchronous function. Using this notification, we emit the output signal `isPrime`.

```
24 hiphop module prgAsynJS(  ) {
25     in N;
26     out isPrime;
27     signal O;
28     let keyin = N.nowval;
29     async (O) {
30         this.notify(prime(keyin));
31     }
32     emit isPrime(O.nowval.val);
```

Line 37 triggers a new reaction with input number and shows the corresponding output.

```
37 machine.react({N:9973});
38 //output : isPrime({"value":9973,"test":true,"processed time":1})
```

Now, we integrate both asynchronous JavaScript functions and synchronous JavaScript functions with a single HipHop.js reactive machine and illustrate the usage as a simple orchestrator. This example is a simple implementation of a very basic air conditioner, wherein based on the user required temperature setting, the air

conditioner will manage a cooler and heater to maintain the required user set temperature. The following listing is of a plain JavaScript function which will mimic the working of a thermometer. We randomly generate a number add 20 to it to generate a temperature between 20 to 30 and return this value.

```
7  function thermometerRead() {
8      let tempSeed = Math.floor(Math.random() * 10);
9      tempSeed += 20;
10     return { "temperature" : tempSeed };
11 }
```

The following listing is of an asynchronous JavaScript function, which mimics the working of an heater. It takes the temperature setting value and present temperature sensed by the thermometer, and switches on the heater till the present temperature equals the setting value. As it is an asynchronous function, it returns a promise, which is resolved after a specific delay - here we have taken delay as equal to as many seconds as in the difference of temperature between required and present thermometer reading.

```
14 function heaterOnOff(tempRef, envTemperature) {
15     console.log("Heater␣switched␣on,␣temperature:␣", envTemperature);
16     return new Promise((resolveHeater, rejectHeater) => {
17         let count = Math.abs(envTemperature - tempRef);
18
19         function heater() {
20             while (count > 0) {
21                 envTemperature += 1;
22                 console.log("\t␣temperature␣increased␣to:␣", envTemperature);
23                 count--;
24             }
25             resolveHeater({ "temperature" : envTemperature })
26         }
27         setTimeout(heater, count*1000);
28     });
29 }
```

The following listing is also another asynchronous function but mimicking a cooler. So, instead of increasing the temperature it will reduce it based on the difference of required setting and the value read by temperature.

```
31 function acOnOff(tempRef, envTemperature) {
32     console.log("AC␣switched␣on,␣temperature:␣", envTemperature);
```

```
33      return new Promise((resolveCooler, rejectCooler) => {
34          let count = Math.abs(envTemperature - tempRef);
35
36          function cooler() {
37              while (count > 0) {
38                  envTemperature -= 1;
39                  console.log("\t␣temperature␣decreased␣to:␣", envTemperature);
40                  count--;
41              }
42              resolveCooler({ "temperature" : envTemperature })
43          }
44          setTimeout(cooler, count*1000);
45      });
46  }
```

The above three functions are integrated with a reactive machine whose behavior is defined in the module `prgAirCon`. Lines 51 through 53 declare the interface and local signals for the module. So the flow of control is as follows: when the machine is made to react by giving the temperature settings, the thermometer function returns the temperature. This function runs in a parallel branch (line 56) with three more branches and emits the read temperature if it is less or greater than the required temperature setting. The second branch in line 64 waits on the temperature emitted by the first parallel branch. Based on the difference value between the thermometer temperature and the required temperature, this branch issues a local signal to switch on either the cooler or the heater (line 67 or line 70). The branches three and four at line 73) and line 81) wait on the signal emitted by the second branch. Based on the signal, either the heater (line 77) or cooler (line 85) will be switched on to set the temperature as required. Once done, we receive the notification and emit those resolved values.

```
50  hiphop module prgAirCon() {
51      inout settingTemp;
52      out Temperature, noChangeTemperature, newTemperature;
53      signal heaterOn, coolerOn, signalCooler, signalHeater;
54      let settingTempvalue = settingTemp.nowval;
55
56      fork {
57          let envTmpval;
58          hop { envTmpval = thermometerRead(); }
59          if (envTmpval["temperature"] !== settingTempvalue) {
```

```
60              emit Temperature(envTmpval["temperature"]);
61          } else {
62              emit noChangeTemperature(settingTempvalue);
63          }
64      } par {
65          await immediate(Temperature.now)
66          if (Temperature.nowval > settingTempvalue) {
67              emit signalCooler(Temperature.nowval );
68          }
69          if (Temperature.nowval < settingTempvalue) {
70              emit signalHeater(Temperature.nowval);
71          }
72      } par {
73          await immediate(signalHeater.now)
74          let temp = signalHeater.nowval;
75
76          async(heaterOn) {
77              this.notify(heaterOnOff(settingTempvalue, temp));
78          }
79          emit newTemperature(heaterOn.nowval.val["temperature"]);
80      } par {
81          await immediate(signalCooler.now)
82          let temp = signalCooler.nowval;
83
84          async(coolerOn) {
85              this.notify(acOnOff(settingTempvalue, temp));
86          }
87          emit newTemperature(coolerOn.nowval.val["temperature"]);
88      }
89  }
```

The following is one of the example run on the reactive machine.

```
90  let machine = new hh.ReactiveMachine(prgAirCon);
```

We see the following output on different runs.

```
[ 'settingTemp(24)', 'Temperature(29)' ]
AC switched on, temperature:  29
          temperature decreased to:  28
          temperature decreased to:  27
          temperature decreased to:  26
          temperature decreased to:  25
```

```
         temperature decreased to:  24
[ 'newTemperature(24)' ]
----------****---------
[ 'settingTemp(24)', 'Temperature(21)' ]
Heater switched on, temperature:  21
         temperature increased to:  22
         temperature increased to:  23
         temperature increased to:  24
[ 'newTemperature(24)' ]
```

With the above example, we come to the end of the introduction to programming and language features of HipHop.js. We saw various examples with code listing and timeline diagrams, depicting the functionality of the HipHop.js language constructs and its temporal behavior. Next we present some internal details with respect to compilation of HipHop.js programs which is important to understand the contribution of this thesis.

2.4 Compilation of HipHop.js Programs

The HipHop.js implementation borrows from Esterel one of its implementation techniques. The HipHop.js compiler translates a program into an equivalent Boolean circuit, which is based on Esterel's hardware translation [25]. This translation is defined as a set of syntactic rules that map Esterel constructs to elementary circuits. These circuits are then *wired* together to form the executable version of the source Esterel program. We illustrate a typical organization of a circuit generated by the compiler as follows.

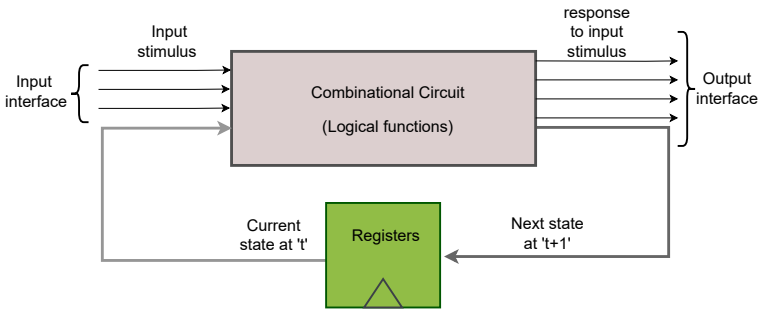


Figure 2.22: General format of HipHop.js Programs when translated to circuits

For the sake of completeness, we define some of the terms used in describing

boolean circuits as in [128]. A logic gate is a logical device with n boolean inputs and 1 boolean output, that computes a boolean function (AND, OR, NOT, etc). The register is a device which takes in a boolean data input, a boolean clock input, and provides a data output. The input value is copied to the output value during the execution of the instant. Classically, the part of the circuit (netlist) consisting of just the logic gates is called the *combinational* part. We do not further elaborate on the compilation process as it has already been fully described in previous publications [22, 116], but we present a few examples to give readers a intuitive feel of compilation of HipHop.js programs.

Let us consider the following listing, which is a simple program for unconditional emission of a signal.

```
1 hiphop module prgEmitSimple() {
2     out O;
3
4     emit O();
5 }
```

This program is compiled to a boolean circuit, whose "netlist" is illustrated with some information in Figure 2.23.

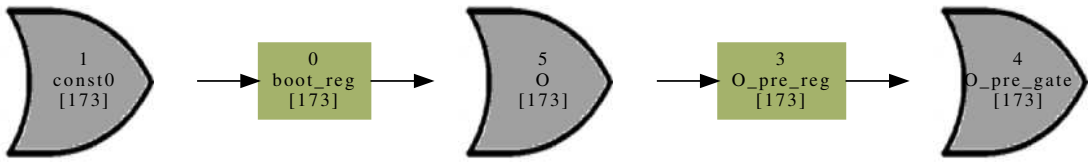


Figure 2.23: Netlist for a single signal emission.

We see a set of *OR* gates connected alongside registers. For the sake of illustration, in the figure, we have each net (logic gates/registers) named and numbered alongside the first character position in the source code that contributes to creation of the respective net (enclosed within square brackets). The actual nets generated by the HipHop.js compiler will have more information than the ones we have presented here. Number `0` is usually associated with the global boot register by the present HipHop.js compiler.

The reaction for an input event (a signal) is nothing but propagation of boolean values through the netlist that are generated for a particular program. Once the propagation of values runs through all the nets inside the netlist, we will have the

reaction (unique fixpoint) for the input signal at that reaction instant going with perfect synchrony hypotheses.

Here is a second example Figure 2.24, illustrating net list for the following listing which uses parallel execution construct `fork/par`.

```
1 hiphop module prgEmitFork() {
2     out A,B;
3
4     fork {
5         emit A();
6     } par {
7         emit B();
8     }
9 }
```

It is compiled as:

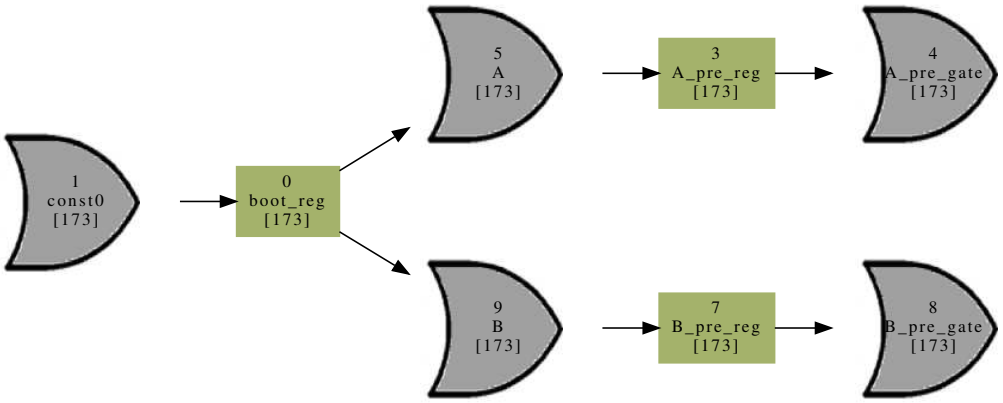


Figure 2.24: Netlist for parallel branches.

We can see as and when the program complexity increases, the number of gates in the netlist also increases. This marks the end of this section on HipHop.js and chapter on general background necessary for continuity of the next chapters. In the next chapter, we proceed with the "causality error" problem and the support HipHop.js extends to trace them.

Chapter 3

Causality Error Tracing in HipHop.js

The expressiveness and the flexibility of Esterel dialects come with a downside: the debugging support, precisely the error reporting which is difficult because errors detected by the runtime system are loosely connected with locations in the program source code. Improving these error messages is the subject of this chapter and one of the main contribution of this thesis. This work has been presented at the PPDP'21 conference [86]. We present a technique that isolates the fragments of the program that are responsible for an error when it occurs. The technique we present applies to the compilation technique HipHop.js uses to transform a source program into an equivalent electric circuit. The improved error messages are built by isolating parts in the generated circuit - minimizing the size of causality error cycles using an iterative process.

The chapter is organized as follows. Section 3.1 introduces the effect of perfect synchrony. Section 3.2 introduces and explains the classical *causality problem* typically faced in synchronous reactive languages. Section 3.3 explains in detail the method proposed to improve causality error reporting, including the theoretical basis it uses. In section 3.4, we elaborate on the strategies used to implement debugging support for causality problems. This chapter is self contained with its own related works and conclusion sections. Section 3.5 is about related work and section 3.6 concludes this chapter.

3.1 Instantaneous Reaction to Absence of a Signal

The synchronous hypothesis implies that testing and reaction to presence/absence of a signal should not take time. In the reactive synchronous language world, there are two schools of thoughts while deciding instantaneous reaction to absence of a signal. That is, for a conditional expression as the following one:

```
if (Signal.now) {
    // Signal present statement
} else {
    // Signal absent statement
}
```

if `Signal` is not emitted in the instant, then, when should the else branch, "*Signal absent statement*", be executed? Is it executed during the same instant or at the next instant? In Esterel, the first option is adopted, while other synchronous languages such as Reactive C [32], SugarCubes [35] and ReactiveML [99], go with the second option. For an unsuspecting programmer, this may seem a futile decision, but the approach chosen greatly impacts the design and programming model, bringing their own inherent intricacies to tackle with.

In the approach followed by languages like Reactive C, SugarCubes and ReactiveML, where there is no immediate reaction to absence of a signal, implicitly a "`yield`" (or "`pause`") statement is added before the "`else`" branch to avoid micro-scheduling analysis and prevent execution inconsistencies analogous to deadlocks of concurrent programming. These languages do not instantly react to signal absence, rather they delay the execution of else branch to the next instant using the aforementioned `yield`, which implies that their emission and reception of signals are also delayed. This may desynchronize communications with other concurrent modules that are often designed independently and reused. But, this approach avoids the problem of detecting and reporting synchronous inconsistencies, as is the approach taken by Esterel like languages.

Combining instantaneous reaction to absence and cyclic dependencies as Esterel supports, makes the implementation complex as a signal is known to be absent only when it can be established that it can no longer be emitted in that instant. This creates a drawback of generating so-called potential *causality problems*, which is a sort of circular dependency that may cause deadlocks during a reaction. The

compiler/runtime has to perform causality analysis to see if programs have causal cycles, then they should have a well-defined behavior [129], that is, there should be only safe combinational cycles. This is explained in detail in next section.

3.2 Causality in Esterel Family Languages

HipHop.js adopts Esterel's model of instantaneous reaction to absence of signal. HipHop.js programs can test for the absence of a signal and react in the same instant to this very absence. This leads to existence of potentially incoherent programs in which there is no way to decide, while respecting the synchrony hypothesis, if a signal is present or absent [33]. The compilers or runtime systems are responsible for detecting and rejecting these programs. The Esterel compiler detects them at compile-time. The Esterel v5 compiler accepts and generates code for combinational cycles when using option "-I", wherein correct cycles pose no problem and incorrect ones are detected at runtime. Likewise, HipHop.js generally rejects inconsistent programs at runtime whenever an attempt is made to execute inconsistent branches.

Here is an example program, where there is no possibility to determine a signal's presence status. Signal `S` is input/output signal here. The HipHop.js expression "`S.now`" tests the presence of the signal `S` and "`emit S()`" is for signal emission:

```
1 hiphop module prg() {
2     inout S;
3
4     if (!S.now) {
5         emit S();
6     }
7 }
```

In the program, signal `S` cannot be present as it is emitted only in case it is absent, also, the signal `S` cannot be absent, since it would be emitted and thus present. There is no possible interpretation of the above program that complies with the perfect synchrony hypothesis and the duality/incoherence results in deadlock at the code fragment (to be avoided by the compiler/runtime) if the environment does not provide signal `S` at a reaction instant. This situation can also arise in two parallel branches which are individually coherent as seen in the following program - `fork`/`par` construct is used to build parallel branches in HipHop.js:

```
1 hiphop module prgParallel() {
2     inout S1, S2;
3
4     fork {
5         if (!S2.now) {
6             emit S1();
7         }
8     } par {
9         if (S1.now) {
10            emit S2();
11        }
12    }
13 }
```

If we assume that `S2` is absent, then `S1` is emitted by the first parallel branch inside `fork` construct; as `S1` is present, `S2` should be emitted in the second parallel branch within `par` construct, which is contradictory with the absence of `S2` (premise for `emit` of `S1`). Now, if we assume that `S2` is present, then it should have been emitted by the `par` branch; but, this implies that `S1` is also present; which is not possible because in the fork branch, as `S2` is present, there will be no emission of `S1`. This incoherence again creates a scenario of synchronous deadlock. Generally speaking, in incoherent statements instantaneous reaction to absence of a signal leads to negating this absence, by emitting that signal and conversely.

For the user, the incoherence problem is complex to deal with as a program grows - understanding the nature and source of these inconsistencies will become difficult for non-trivial realistic programs. The aim of the study presented in this chapter is to propose tools that can help programmers in understanding and fixing these problems. In the following section, we elaborate the causality problem with respect to HipHop.js, which will set the context for the requirement of tools to mitigate from those scenarios.

Causality in HipHop.js

As we have introduced earlier in section 2.4 of previous chapter, the HipHop.js borrows from Esterel one of its implementation techniques for compilation of HipHop.js to circuits. We observed a significant amount of increase in the number of nets in the netlist of the examples we presented in section 2.4. We will see that

this growth is one of the difficulty of HipHop.js debugging.

Let us see the netlist for the following program which has "causality cycles" problem:

```
1 hiphop module prgToBeOrNotToBe() {
2     inout S;
3
4     if (S.now) {
5     } else emit S();
6 }
```

Listing 3.1: A non-constructive program.

The execution of an instant consists in propagating values through the logical gates. Values are pushed from gates to gates. When all the inputs of a gate are known, then the output is computed and propagated to the connected gates. Incorrect programs are characterized by circuits for which the value propagation does not complete. That is, after the propagation, some gates are still missing input values and output cannot be computed. This happens for circuits that contain bad cycles.

When we execute the above program, the HipHop.js compiler without causality debugging support reports the following error message, which is basically the number of nets that prevent the reaction fixpoint to be reached:

```
TypeError: causality cycle of length 5  detected
```

Listing 3.2: The error message

The netlist for the above code is illustrated in Figure 3.1. Nets having ids 5, 16, 17, 18 and 27 are not participating in the propagation of boolean values. This is the symptom of an error, because it means that the value propagation in the circuit has not completed. In other words, it means that there is an error in the program that gave rise to a *causality error* cycle.

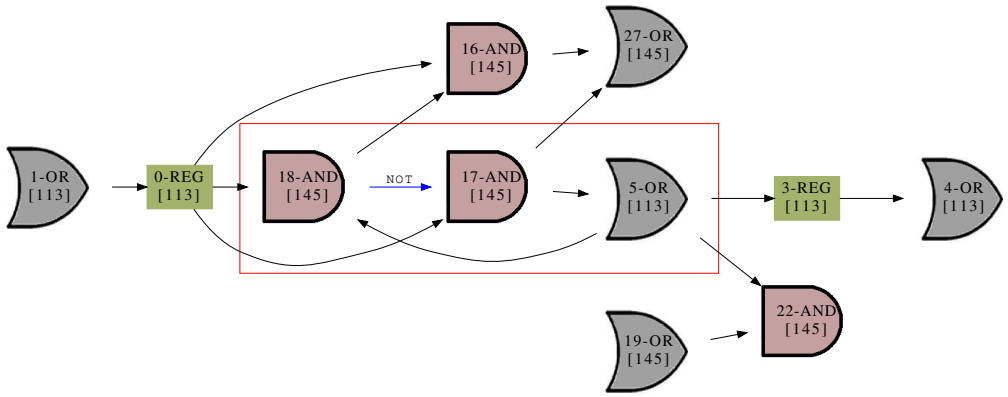


Figure 3.1: Netlist for `prgToBeOrNotToBe` program.

The above generated circuit contains a *combinational cycle* highlighted within red rectangle. The blue arrow represents negated output. In electrical engineering, circuits with combinational cycles are usually avoided, as the presence of these cycles can lead to oscillations and unpredictable behavior. This combinational cycle in the context of HipHop.js has blocked the execution (does not attain fixpoint), due to synchronous deadlock. We once again borrow the concepts of electrical engineering to explain this situation. Generally, a circuit is well-behaved, if for every input, the output stabilizes to a unique value (fixpoint) within a bounded amount of time. In circuits with combinational cycles, the circuit may not be well-behaved as they may be oscillating, giving rise to deadlocks. But, there are also useful Esterel programs that leads to perfectly correct circuits with combinational loops; these are exactly characterized by the constructive semantics [22]. For them, the semantical inconsistency can only be detected at runtime in HipHop.js (Esterel compilers detect them statically, but with an expensive Boolean or SMT based static analysis techniques).

We conclude this section with an illustration, which shows the scaling up of netlist considerably for a simple ABRO (the "Hello World" of reactive programming) program listed below.

```
1 hiphop module ABRO() {
2    in A, B, R;
3    out O;
4
5    do {
6       fork {
7          await(A.now);
```

```
8         } par {
9            await(B.now);
10        }
11        emit O();
12     } every(R.now)
13 }
```

Listing 3.3: Another simple HipHop.js program.

A program can wait for one or several signals to be emitted in HipHop.js with the `await` construct, which waits for a condition to be true 2.3. In the listing, as soon as both inputs `A` and `B` are received, `O` is emitted as the output. The behavior is reset whenever the input `R` is received. The netlist for the above program is presented in the following Figure 3.2. It shows that for a simple program, the number of nets increases considerably. Some HipHop.js compilation optimization reduces the size of this graph but for the clarity of this presentation, the optimizations have been disabled.

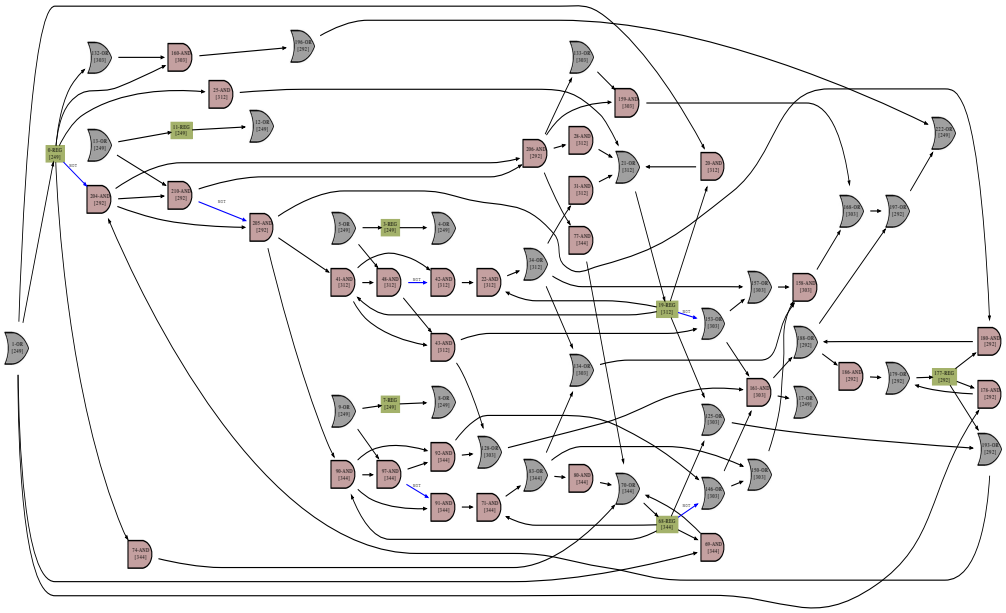


Figure 3.2: Netlist for ABRO program.

Now, with the context set, we summarize what happens when we come across causality problem in HipHop.js, which is not the case for ABRO above. In that case, the runtime while propagating the signals, will not reach fixpoint, since some of the nets will form cycles. If some of the nets do not participate in propagation,

then the HipHop.js runtime would reject the programs as non-deterministic and non-constructive due to causality error.

Debugging these errors require isolating parts of the source code that cause cycles in netlist. Realistic programs are compiled into netlist of hundreds or thousands of gates. When these netlists contain cycles, many gates are generally involved, which makes HipHop.js error messages very difficult to interpret by programmers. The purpose of this study is to improve the error messages triggered for non-constructive programs by isolating smaller cycles that actually cause these errors, and this makes it easy for programmers to localize the true source of the error cycle in their source code.

3.3 Isolating Causality Error Cycles

After presenting the problem of causality errors in the previous section, we discuss here the techniques used to isolate the causality error cycle for providing debugging support to causality problems in HipHop.js. Providing the location of the statements in the source code which form cycles in the boolean circuit to programmers is helpful in debugging the causality errors. For a simple program or small cycles (cycle size determined by number of nets participating in cycle formation), we can directly present the location that can be inspected by programmers to debug, but when the program scales or the size of the cycle becomes huge for a program, then, presenting the locations of big cycle may not be useful to programmers as it may span a large number of locations in the source code and may be humanly impossible to debug with such information. Hence, our endeavor is to provide source code locations of a smaller cycle in the program, so that programmers can comprehend and resolve the smaller cycle and then proceed further with the next bigger cycle, if any, to debug (a big cycle can contain several small cycles which are much easier to debug for the user than the bigger ones). In 3.3.1, we illustrate the application of the methodology that isolates smaller cycles and also review some more relevant graph theory concepts. In 3.3.2, we explain the algorithm that aids in isolating smaller cycles and we conclude this section in 3.3.3 with a case-study report on the application of isolating smaller causality error cycles in a real-world application that has been developed using HipHop.js.

3.3.1 Cycle Simplification - Overview of Methodology

The combinational circuits in the form of directed graphs can be analyzed with the help of graph theory algorithms to find smaller cycles in a given graph. Specifically, we will be using R. Tarjan's "Strongly Connected Components" (SCC) algorithm [136] and "Weak Topological Ordering" (WTO) algorithm proposed by F. Bourdoncle [31] to find *smaller* cycles in the graph and display it to the programmer for debugging.

In a directed graph with cycles, we may have Strongly Connected Components (SCC). We remind the reader that in a directed graph, a cycle is a directed non-empty trace of vertices (nodes) in which, the only repeated vertices are the last and first vertices, and a graph G is strongly connected if every vertex V1...Vn is reachable from every other vertex. For example, in the directed graph as in Figure 3.3, we have six vertices grouped into three SCCs, seen as clusters. These SCCs can be of varying sizes (based on number of vertices).

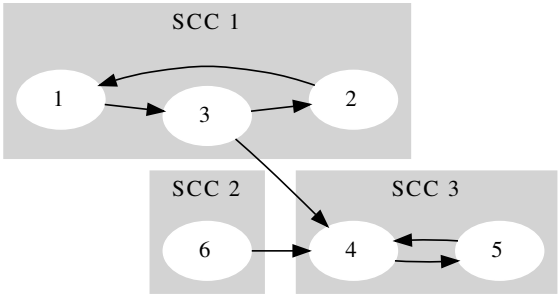


Figure 3.3: Strongly Connected Components (SCC) in a directed graph.

Now, reverting back to causality errors in HipHop.js, the initial display of error message in the form of total number of all the nets not participating in the reaction can be simplified to be useful, if we take the help of SCCs in the directed graph formed by these non-participating nets. As we see, the SCCs contain all the cycles that are formed in the graph and if their size is less than the original number of non-participating nets (number of nets in SCC being less than the total number of non participating nets in the reaction), then by presenting the source code locations of those nets in SCC, we can help programmers narrow down to a smaller cycle, that they can potentially debug.

We use Tarjan's algorithm to find SCCs in the directed graph of the netlist generated by HipHop.js compiler. We shall illustrate this application with the

following trivial program that suits our purpose for giving intuitive understanding. In HipHop.js a module can be executed inside another module by using the `run` keyword (e.g., line 14 in listing 3.4). Here in the program listing 3.4, we have causal dependency between signals `M` and `N` due to composition of the the two modules, `main` and `sub`. Moreover, on signal `J`, the problem of *to be or not to be* can be seen at line 4.

```
1 hiphop module sub() {
2    inout M,N,J;
3    if (N.now) {
4       if (!J.now) emit J();
5       emit M();
6    }
7 }
8
9 hiphop module main() {
10   inout M,N,J;
11
12   if (M.now) {
13      emit N();
14      run sub(){M,N,J};
15   }
16 }
```

Listing 3.4: More complex causality error program.

The netlist for the above listing is in Figure 3.4, where we see a large number of nets in the boolean circuit and combinational cycles.

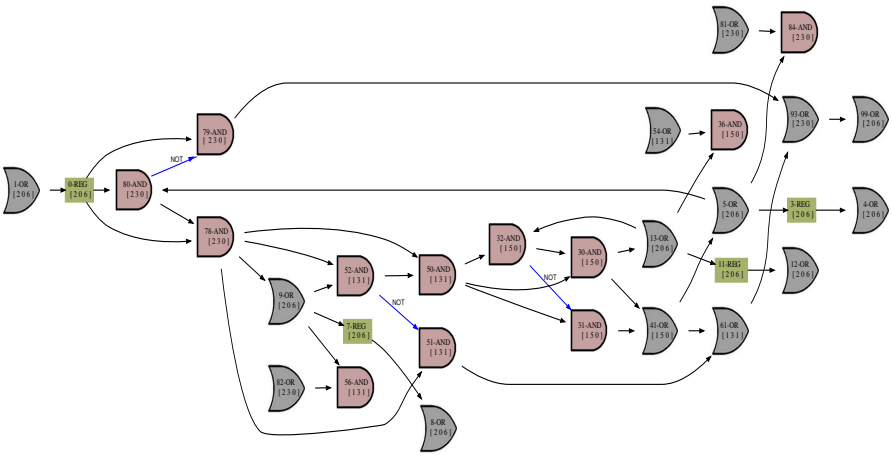


Figure 3.4: Netlist of the causality error program in listing 3.4.

When we compile the above listing 3.4 without the causality error debugging support discussed in this chapter, we see the following error message, pointing out to a specific length of the error cycle:

```
TypeError: causality cycle of length 13 detected,
involving signals J,N,M
```

This is a weakly informative message! For the same listing 3.4, using Tarjan's SCCs to build the error message improves the precision of the error report as seen here:

```
TypeError: causality cycle of length 4 detected,
involving signals J,N,M
```

This is a considerable improvement in the error trace that has been reduced at least by a factor of 3. We see in Figure 3.5, the SCC of the graph highlighted. The reason for inherent mismatch between error report (cycle of length 4) and the number of nets (11 within SCC) is due to an optimization strategy used to combine nodes formed at the same source location. That is, when two nodes originate from the same source location, they are merged in the error report, without loss of any information. We give an example of this reduction process at the end of 3.3.2.

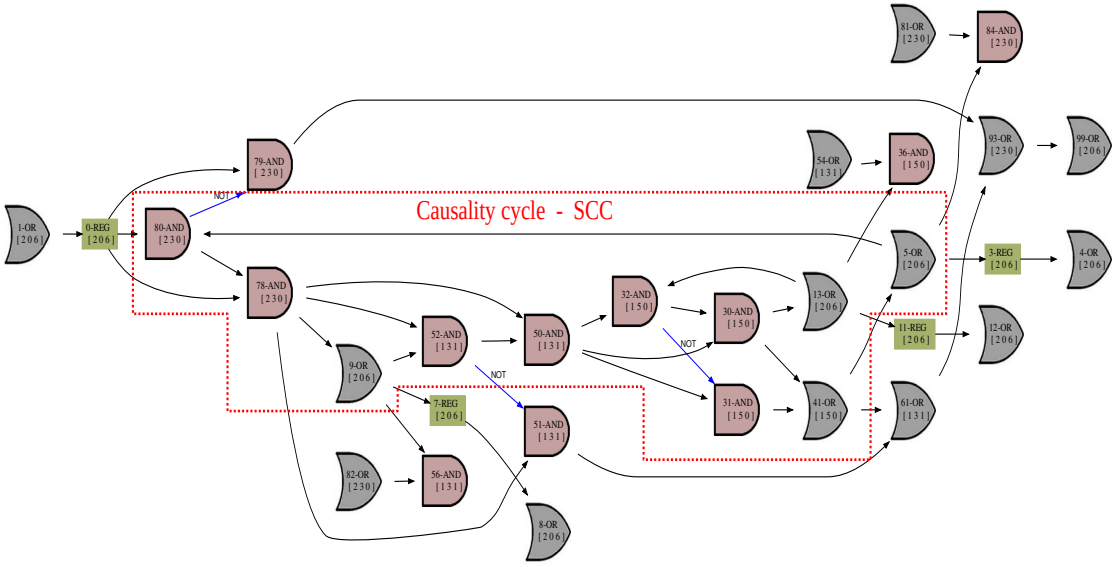


Figure 3.5: Netlist showing the causality cycle with SCC from Tarjan's SCC algorithm on listing 3.4.

Although computing SCCs helps us in narrowing down the causality problem, it may be less useful at times if the SCC itself is huge and error messages point

to a very large cycle. In graph theory, it has been observed that a SCC can have smaller sub components which are themselves SCC. We can recursively traverse through the SCC to find a smaller possible sub component which is also strongly connected inside the original SCC. If the error message displays this smaller cycle to the programmer, we believe the programmer can incrementally debug from smaller cycle to the next bigger cycle. This is where we adopt Bourdoncle's WTO algorithm [31], which is utilized to find smaller SCCs contained within larger SCCs. When the same listing 3.4 is compiled with Bourdoncle's WTO refinement (explained in next section), we see the following error message and we can see the corresponding illustration on the netlist in 3.6. In 3.7 we provide enlarged view of the smaller Bourdoncle component for better readability.

```
TypeError: causality cycle of length 2 detected,
involving signals J
```

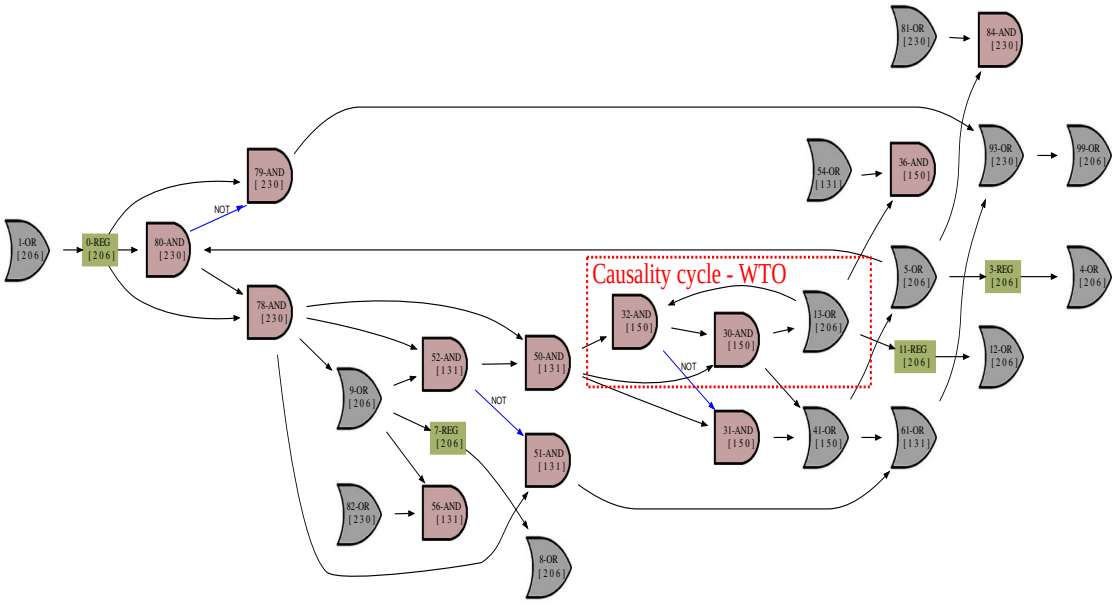


Figure 3.6: Netlist showing smaller causality cycle with Bourdoncle WTO refinement.

Compared to error messages based on SCC, we see that after WTO refinement, the quality of error message improves by another factor of 2.

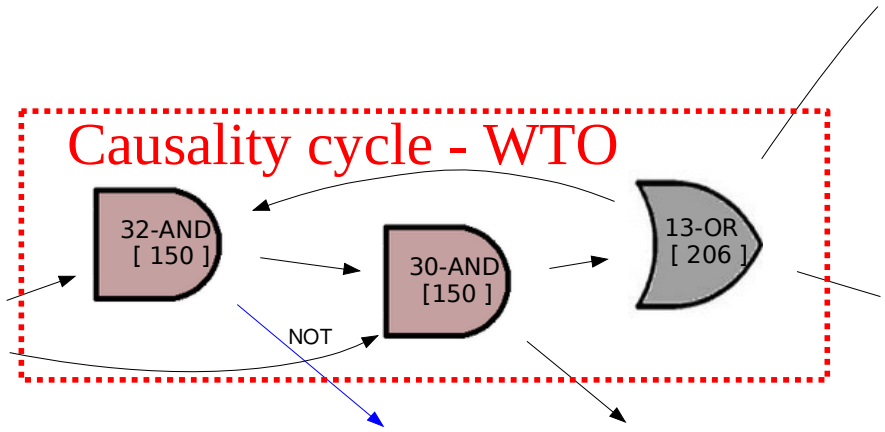


Figure 3.7: Enlarged view of the smaller causality cycle

In total an improvement by a factor of six with the initial error report without any causality error analysis support. Further, the refined error message precisely isolates the source of the problem: the pattern of line 4 in listing 3.4 is incorrect (pointing to the actual source location is explained in section 3.4).

3.3.2 Cycle Simplification - Algorithm Details

In this section, we explain in detail the WTO algorithm that we use in HipHop.js to simplify even more the cycles reported to programmers when causality errors occur. We start with some essential definitions from [31, 135, 57]. A Bourdoncle component or a component is a set of vertices within a matched pair of parentheses. The *head* of the component is the leftmost item and the vertices in the component are strongly connected. The netlist is analyzed to build WTO of the nets. WTO is defined as a well parenthesized ordering of the vertices, where in, two left parentheses are not adjacent. In a graph G, with vertices a, b, c, ... z, a trivial ordering (a (b (... (z))...)) is a valid WTO. It follows that the head of every component is in no subcomponent, and if u $\rightarrow$ v is a feedback edge, then v is the head of the component containing u.

To build a WTO, the algorithm follows the approach *iterate until stabilization*. At each iteration, Tarjan's SCC algorithm is used recursively to find a smaller SCC. It basically decomposes a directed graph hierarchically into strongly connected components and subcomponents. The following is a very high level representation of the adaptation of the WTO Algorithm [31] in our implementation.

Algorithm 1 Decomposing a directed graph into SCC and subcomponents in hierarchical manner.

```
1: WTO: list of Weak Topological Ordering for a directed graph G. Initially empty.

2: function SUBCOMPONENT (G)
3:     Compute S, the set of SCCs of G using
4:     Tarjan's SCC.
5:     for each Comp in S do
6:         append Comp in WTO
7:         if | Comp | > 1 then
8:             Identify the head H ∈ Comp.
9:             replace Comp with
10:            (H, SUBCOMPONENT ( Comp / H )) in WTO
11:        else
12:            Keep the Comp as it is in the WTO
13:        end if
14:    end for
15: end function
16: Print the WTO
```

So, WTO is an empty list initially as we see in line 1. The algorithm is executed on the given directed graph G. As said earlier, Tarjan's SCC algorithm is repeatedly used in line 3 during the recursive calls. Each component in SCC returned by the Tarjan's will be strongly connected and will have a vertex as head. These vertices are the entry points to the respective components; if sliced off, then the respective components will cease to be strongly connected. The WTO algorithm identifies the head node for each component (line 8) having more than one vertices. To identify the head, it maintains a numbering scheme of vertices (based on walk during DFS) called depth first number DFN, in a directed graph G. Parenthesis "(" are opened before every head y of edges x -> y whose tail x has a greater DFN than y and then, all the parentheses are closed ")" after the last vertex in the graph G [31]. Once the head is identified for a component, that head is sliced off from the component and the remaining vertices in the component are searched for a smaller SCC as in line 10, and this proceeds in a recursive fashion till it reaches fixpoint and hence stabilizes for the initial component. All these traversals are collected in the WTO list (line 1) and finally printed to give a topological ordering of the graph G.

We illustrate here the Bourdoncle's WTO approach for the listing 3.4. Initially, Bourdoncle's refinement starts with the nets identified by the Tarjan's SCC as in

line 0 below and Figure 3.8 illustrates the isolated Tarjan's SCC. Net 31 will have the minimum DFN compared to all other nets and hence it will be identified as the initial head and accordingly we see the parentheses.

```
0 (31, 41, 5, 80, 78, 9, 52, 50, 32, 30, 13)
```

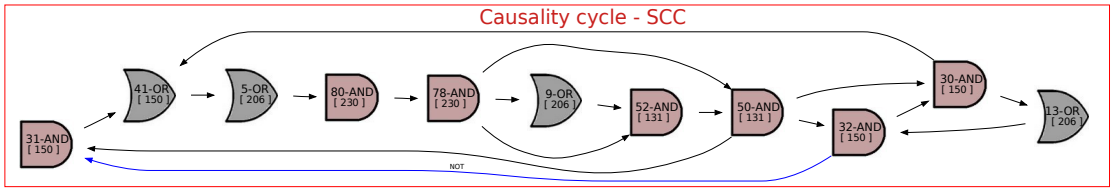
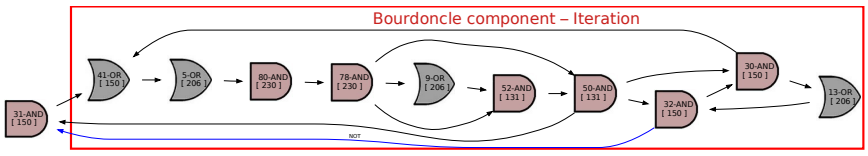


Figure 3.8: Isolated Tarjan SCC.

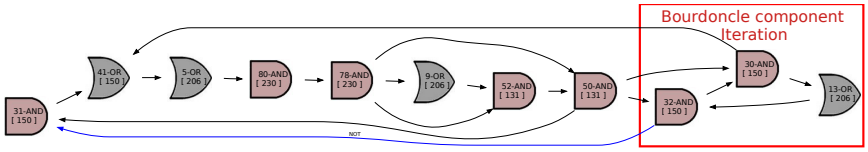
Accordingly, we see the following steps while going through the WTO evolution, the process of *iterate until stabilization*.

```
1 (31, (41, 5, 80, 78, 9, 52, 50, 32, 30, 13))
2 (31, (41, 5, 80, 78, 9, 52, 50, (32, 30, 13)))
3 (31, (41, 5, 80, 78, 9, 52, 50, (32, 30, 13))) WTO (stabilisation)
```

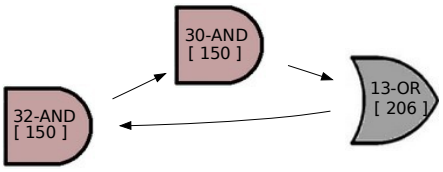
The following figures 3.9a, and 3.9b illustrate the respective iteration stages. Figure 3.9c gives an enlarged view (for better readability) of the smaller cycle we obtain.



(a) Iteration 1



(b) Iteration 2 and stabilization



(c) smaller cycle

Figure 3.9: Bourdoncle component evolution.

The Figure 3.9c illustrates the result of the error message which is displayed by HipHop.js runtime when rejecting the program. As mentioned earlier, the final optimization merges the nets formed at the same source location, ultimately reducing the cycle to its minimum. In Figure 3.9c, the 3 nets highlighted are (32: {pos:150}, 30: {pos:150}, 13: {pos: 206}). Nets 32 and 30 are considered as one, because they both originate from the same source code location 150, yielding a cycle of length 2 as was reported in the earlier error message. Next, we present a case study report of using the causality error message reporting in a real world application developed using HipHop.js.

3.3.3 Case Study: Skini

Skini is an interactive music system where a composer programs musical scores in HipHop.js, which are then played in live concerts, in interaction with the audience [112] [113]. The size and complexity of HipHop.js programs representing scores depend on the duration of the music, the number of involved musical instruments, and the granularity of the interactions with the audience. Typically, these programs span over several thousand lines of HipHop.js code, which are compiled into netlist of several thousand gates. For instance, *opus2* [1], which is a classical music piece that lasts 2 minutes and that has been played several time in concerts, spans over 20000 LOC (spanning multiple source files), uses 111 signals, and when compiled, counts 15053 of nets. For programs of this complexity simplifying as much as possible the error messages is critical, especially because Skini is supposed to be programmed by music composers, not by computer science experts!

During the creative process of composing *opus2*, several "causality errors" were reported. The last one we observed before the integration within HipHop.js of the "causality error" tracing techniques presented in this chapter had an error report, a causality cycle of length 11393. This spanned through multiple source files and included all the signals declared in the program. This was un-trackable for the composer. Improving on these error messages has been one of the motivation for this work.

Using Tarjan's SCC algorithm, we had the error report reduced to a length of

[1] www.hedelin.fr

276 and 111 signals. These locations and signals span over multiple source code files which is again difficult to debug. After applying the WTO refinement, we reduced the error report even more drastically: the causality cycle reduced to a length of 13 within a single source code file and based on a signal `endReservoir` (reduction by a factor of 21× compared to SCC based error message):

```
TypeError: causality cycle of length 13 detected,
involving signals endReservoir
"filename":"autoOpus2-2.js",
"locations":[2602,3033,3047,3109,3399,3417,3712,
             3729,3798,12935,13849,13894,16149]
```

The character positions in the error message correspond to actual lines in the source code file, 13 positions are mapped to 12 lines: *e.g.*, position 2602 is in line number 90, 3033 and 3047 in line 107, 3109 line 110, and so on. The actual source code of `autoOpus2-2.js` looks as represented in the listing 3.5; due to paucity of space we have skipped many parts of the code and overly simplified it, as our main intention is to illustrate how WTO refinement helps in pointing to specific positions of the source code which are contributing a comparatively smaller causality error cycle that can be manageably debugged.

In the source code, we see the musical notes translated to plain JavaScript code. We have multiple modules providing various musical sessions and orchestrating parallel interactive musical events. Each module has many parallel constructs that read and generate various signals representing various musical instruments based on temporal dependencies as specified by the music composer. Before we proceed further, we will explain in short about two constructs. In HipHop.js, a body is allowed to run even when the preemption condition holds, but is terminated at that precise time. This is implemented using `weakabort` statement in HipHop.js. Statements enclosed within "`hop {...}`" are plain JavaScript statements.

```
90  hiphop module resevoirTrompettesRouge(tick, endReservoir) {
...     ...
107     abort immediate(endReservoir.now) {
...         ...
110         hop { hop.broadcast('startTank', trompettesRouge) }
...         ...
115         emit trompettesRouge+"OUT"([true,255]);
116         fork {
...             ...
```

```
127           await(trompettesRougeIN.now);
128           emit trompettesRougeOUT([false,255]);
129        }
...        ...
132     }
133     hop { hop.broadcast('killTank', trompettesRouge ) }
...     ...
153 }
... ...
391 hiphop module sessionRouge(tick, in abortSessionRouge) {
392     signal endReservoir, abortTheSession, stopEveryAbort;
...     ...
406     fork {
407        every(abortSessionRouge.now) {
408           emit endReservoir();
...           ...
425        }
426      } par {
427         weakabort immediate(abortTheSession.now) {
428            fork {
429               await count(2, violonsRougeIN.now);
430            }
...            ...
517            fork {
518               run resevoirTrompettesRouge(...);
519            }
...            ...
633        }
634     }
635 }
636 hiphop module sessionNoire(...inout endReservoir) {
...    ...
705 }
```

Listing 3.5: `autoOpus2-2.js`

In the code snippet, we see the composition of two modules: `resevoirTrompettesRouge` and `sessionRouge` are composed within one another. The module `resevoirTrompettesRouge` is run inside the module `sessionRouge` at line 518. The `endReservoir` signal is used to abort some specific patterns stored in a tank (repository) of musical patterns. These patterns, for example, consists of those waiting for

specific musical events like the ones we see in lines 127 and 128. The code block "`abort immediate (endReservoir.now)`" (from line 107) instantly *kills* all these patterns with the help of broadcast (line 133) whenever there is emit of `endReservoir` signal, from the other module. So, when we analyze the error report generated based on WTO refinement, we see that the causality error cycle spans through these composed modules - when the module `resevoirTrompettesRouge` is executed inside module `sessionRouge`, which has emissions of signal `endReservoir` in various parallel branches. A causality error occurs due to a read-write inconsistency on `endReservoir` signal.

The above error, for example, can be resolved by insertion of simple delay statements (`yield`); for instance, after emissions of signal `endReservoir` in module `sessionRouge`, which can break the combinational cycle as it will introduce a register net in between the cycle. Once this error is corrected, and after recompiling, the runtime points to another smaller causality cycle in the module `sessionNoire` (at the end of the listing in line 636), involving signals `endReservoir` and `abortSessionNoire`. Identifying the previous error would have been very difficult without the help of WTO refinement, as the causality error cycle would have spanned numerous positions in multiple source files, warranting a complete mental model by the composer of all the parallel executions, reaction by reaction, to resolve it. Likewise, programmers can embark upon debugging smaller causality cycles incrementally and hence solve big causality errors in complex or lengthy programs. We experimented on various musical patterns of the *opus2* musical piece which had causality error cycles at various locations. Table 3.1 summarizes the effect of Tarjan's SCC and Bourdoncle's WTO refinement on causality error message reporting.

We observe that the initial number of nets in the netlist varied for each piece (modified versions of *opus2*), whereas the SCCs in each piece were of the same size at 281 nets spanning multiple source files. The corresponding causality error locations based on WTOs were within a single source file and were much smaller in number. Furthermore, we also experimented on some of the other musical pieces that are available in Skini like *opus1*, *opus2-3*, *opus3-1*, and *opus5*. We observe varying sizes of SCC and relatively smaller WTOs. Table 3.2 summarizes our observation.

WTO refinement throws open the error accordingly to much smaller cycles.

Sl no	Piece name	netlist size	Initial error size	SCC size	WTO size
1	opus2Ex1	24355	18619	281	22
2	opus2Ex2	12555	9475	281	16
3	opus2Ex3	15043	11477	281	22
4	opus2Ex4	19369	14718	281	22
5	opus2Ex5	26325	20516	281	13
6	opus2Ex6	21850	16705	281	22
7	opus2Ex7	17547	13393	281	13

Table 3.1: Causality cycle size in various music patterns based on opus2 with induced causality errors.

Sl no	Piece name	netlist size	Initial error size	SCC size	WTO size
1	opus1	17253	16098	352	10
2	opus2-3	8351	6326	263	14
3	opus3-1	17326	15418	294	10
4	opus5	5460	4538	111	3

Table 3.2: Causality cycle size in other musical pieces of *Skini* with induced causality errors.

We see tremendous improvement in error reporting with WTO support, on an average of 21× over SCC and even more when compared to initial error size in real life projects like *Skini* as observed here. The HipHop.js programmers can reason out and debug causality errors with the help of error messages based on WTO refinement, as we have shown. The error messages can point to specific locations (relatively smaller number) of the source code of a specific module with relevant signals participating in causality error. In the following section, we explain the implementation aspects of causality error analysis in HipHop.js, the process of mapping nodes to source code locations and also an elaborate example debugging process of causality errors.

3.4 Implementation of Causality Error Tracer

In this section, we elaborate on the implementation of the aforementioned algorithms of section 3.3 to debug the causality errors and the process of error reporting done by HipHop.js runtime and compiler.

For each reaction of the program, the reactive machine propagates the values through each net in an orderly manner. All the nets participating in a reaction are pushed into a FIFO data structure called `known_list` by the HipHop.js compiler and when a reaction happens, the FIFO structure has to be empty. If at the end of propagation, there are nets which are found to be not participating in a reaction, then, it means the machine's netlist has some nets which are participating in a combinational cycle and hence are not part of the reaction. We have a causality error and the program is rejected. When this occurs, the nets that are still pending are analyzed by building the SCC and WTO. From the reduced graph, HipHop.js builds its error report. In this section, we show how the actual error messages are built and how the error reporting integrates within various programming environments.

Each net's data structure is self-contained with all the necessary information, including the source code location, which contributes to the respective nets creation. The WTO of the directed graph in the form of components returned by the algorithm is further processed to identify smaller subcomponents, pick the position of the nets in that subcomponent, perform redundancy checks on positions, and then present those positions to the programmer as the locations contributing to causality error in the source code. For one of the previous example introduced in Section 3.3, listing 3.4, from the WTO returned by the algorithm: (31, (41, 5, 80, 78, 9, 52, 50, (32, 30, 13))), HipHop.js extracts the deepest component, here (32, 30, 13), and then processes it for source code locations (as explained at the end of section 3.3.2). Components involving only one source location are ignored because they do not bring any useful information of their own.

The generated source code locations (*e.g.*, 150 and 206 for nets 32, 30, and 13 as above) are stored in a JSON object. To see the locations of the causality error cycle in the source file, the source program has to be compiled with -g option, which is normally used for debugging: `hop -g somefilename.js`. The locations in JSON format, will be stored in a file `hiphop.causality.json`, which is created just for this purpose. The popular editors like Emacs, VScode, and Atom are programmed

to use the contents of this file to read the positions and highlight the respective source code position (the complete word starting from the given location) as visual markers to help programmers locate the position easily.

For Emacs, the programmer can see the locations by opening the source file in Emacs editor and typing `[C-x C-h]` to highlight the locations in the source file. As an illustration of causality error cycle's visualization for Emacs editor, we refer back to the same example of our discussion in Section 3.3, listing 3.4. When we use Emacs editor to visualize the causality error cycle, we see the following visual markings in the following Figure 3.10 (source location contributing to causality highlighted in green color). We remind the reader of the error message the HipHop.js generated when using WTO approach for the above program:

```
TypeError: causality cycle of length 2 detected,
involving signals J
```

```
hiphop module sub(M,N,J) {
    if (N.now) {
       if (!J.now) emit J();
       emit M();
    }
}

hiphop module main(M,N,J) {
    if (M.now) {
       emit N();
       run sub(...)
    }
}
```

Figure 3.10: Visual markings of causality error in Emacs.

With the above information, the programmer can easily deduce that the causality error is due to signal `J` being checked for presence (`if`) to emit itself (`J`) and hence it can be easily debugged. We elaborate on the cycle debugging with another example first presented for the Quartz synchronous programming language [123] that we have adapted to HipHop.js:

```
1 hiphop module prog() {
2    in J;
3    inout M,N;
4
5    weakabort immediate(N.now) {
6       fork{
7          if (!J.now) yield;
8          emit M();
9       } par {
10         if (M.now) emit N();
11      }
12   }
13   emit M();
14 }
```

Listing 3.6: Complex causality error program.

The program in listing 3.6 displays two different behaviors based on the value of `J`. When `J` is false, the first branch (lines 6-8) of the `abort` statement stops at the `yield` statement (line 7), so that "`emit M()`" at line 8 is not executed. The second branch (lines 9-11) cannot decide yet whether `M` is *true* or *false* at line 10.

If we assume `M` as false, "`emit N()`" does not get executed, hence `M` and `N` are false (as initially assumed for `M`). On the other hand, if `M` is assumed to be true, then `N` is emitted in line 10, and hence both `M` and `N` are true. Since `N` is present, the weak abortion at line 5 takes place from the same instant. With `J` being absent, we have "two" logically consistent behaviors where `M` and `N` are logically equivalent, giving rise to a non-deterministic program. When `J` is true, we see that the program is constructive during runtime emitting `M` and `N` from the two parallel branches (since weak abort, the emissions will remain valid). When we execute the program with `J` and `M` being absent, we get the following error message with WTO refinement:

```
TypeError: causality cycle of length 4 detected,
involving signals N,M
```

In the Emacs editor, we see the visual markings of the causality error cycle as shown in Figure 3.11. These visual markings are very much precise and are specific to the exact locations that are giving rise to non-determinism, matching the explanation we discussed earlier.

```
hiphop module prog(in J,M,N) {
   weakabort immediate (N.now) {
      fork {
         if (!J.now) yield;
         emit M();
      } par {
         if (M.now) emit N();
      }
   }
   emit M();
}
```

Figure 3.11: Visual markings of causality error locations in Emacs for listing 3.6.

As discussed earlier, to aid HipHop.js programmers in debugging causality error cycles when using VScode and Atom editors, extensions are developed to pinpoint error cycles visually in the source code. This is similar to the way we display error cycles in Emacs editor. If these extensions are activated at the command palette while programming after compiling HipHop.js programs, and if there are causality errors at runtime, they can be visualized. We call our extensions as "causality-decorators".

The `hiphop.causality.json` introduced earlier provides us the character positions of the source code where causality cycles happen. Using the positions in the file, we identify the text area (first character position) where the character is present, we then take the help of VScode's word definition to identify the whole word area that should be highlighted, and then push that range of location for highlighting with a particular color. In Atom editor, there is a slight change in the way we identify the position of the source text, as it is dependent on the various APIs support provided to access the editor properties.

The Screen shot in Figure 3.12 shows how the source localization is displayed in VScode editor for a sample causality error program after activating our causality-decorator extension in the command palette. The code decorator for Atom editor also provides similar graphical highlighting for causality error locations and this can be extended to a few more popular IDEs.

```
1  hiphop module prog() {
2     in J;
3     inout M,N;
4
5     weakabort immediate(N.now) {
6        fork{
7           if (!J.now) yield;
8           emit M();
9        } par {
10          if (M.now) emit N();
11       }
12    }
13    emit M();
14 }
```

Figure 3.12: Causal error cycles displayed in VScode editor.

As illustrated with examples above, we see the advantages of Bourdoncle's WTO approach in providing smaller error trace to the programmer to aid in debugging causality errors in HipHop.js programs and also the visual extensions for popular IDEs.

3.5 Related Work

In the following part, we discuss some of the approaches used for debugging reactive programs by other languages and systems.

In Esterel, a static analysis technique [129] is used before execution to ensure that Esterel programs are constructive. Two approaches are proposed for building this analyzer, with one of them using Bourdoncle's WTO. In HipHop.js we propose to use WTO for a totally different purpose and in a totally different context. HipHop.js uses this approach to provide debugging support for causality error programs in both compile and runtime. HipHop.js provides compile time warnings of any causality error cycles after generation of the nets. It uses the same implementation as we presented earlier at compile time, but here instead of analyzing the non participating nets of a reaction, all the nets that are generated by the HipHop.js compiler are analyzed. If we find a cycle during this process in the WTO returned, then we flag a warning to the programmer, that there is a potential causality

cycle. This causality cycle of compile time detection may or may not be resolved at reaction time.

Causality errors also appear in another language of the synchronous family Lustre. In Lustre [67], it is not allowed for a variable to instantly depend on itself. Furthermore, a static criterion is used for rejecting causality error programs: the system of equations should be cycle free. A system of equations which is cyclic is rejected, even if it has a unique solution. In Argos [69], incorrect compositions contributing to causality are not given any meaning, similar to an Esterel based approach. A Binary Decision Diagrams based procedure is proposed to check the consistency of combinational loops and whenever possible, these loops are removed without affecting the semantics of the circuit. The synchronous reactive approach of the languages Reactive C [32] and ReactiveML [99] differs from Esterel and HipHop.js as they avoid causality error by only allowing instantaneous reaction to presence and forbids instantaneous reaction to absence. SugarCubes [35] is to Java what Reactive C is to C and as expected SugarCubes also avoids causality errors by forbidding instantaneous reaction to absence.

Functional reactive programming (FRP) is generally supported as DSL for stateful logic that can replace the widely used observer pattern (callbacks/listeners). FRP is used for programming reactive applications like graphical user interfaces (GUIs), games, robotics, and music by explicit modeling of time. Generally an FRP program gets executed in two stages; in the first stage, FRP code segments are converted into a directed graph, and in the second stage for the input fed, the FRP engine produces the output based on the dependencies stipulated by the directed graph [29]. These directed graphs can also be modified dynamically during the second stage. Reflex-FRP is an Haskell based reactive ecosystem to develop user interfaces and web applications. In FRP, inadvertent cyclic dependency errors can happen and the support to programmers is quite minimal such as "Maximum call stack size exceeded". Debugging causality error loops has been discussed as quite time intensive and taxing on programmers; they are advised to keep programs causal safe and avoid constructs which may contribute to unresolved causal error loops. Here, we foresee the application of Bourdoncle's WTO refinement on directed graphs having causality errors. We believe one can provide better error reporting to FRP programmers, as we do in HipHop.js.

ReactiveX (Rx) is a reactive library developed for many programming languages

like Java, JavaScript, Scala, C++, C#, etc, and used by companies like Netflix, GitHub and many others. According to the developer's[2] claim, Rx differs from original FRP as FRP operates on values that continuously change over time, while Rx operates on values that are discrete and are emitted over a period of time. Rx introduces two basic types called Observable and Observers. Observables are used to define the data flow and to produce the data, while observers consume the data and move it further down the stream. Rxfiddle is a visual debugger for the Rx [15]. It provides an overview of dependencies in dataflow and timing of individual events. The Rxfiddle debugger provides two diagrams, the data flow graph and dynamic marble diagram as a means of visualizing reactive programs for debugging. The "marble diagrams" contain one or more timeline containing the events that enter and leave observables. Consider the following code listing:

```
Observable.just(2, 4, 6, 8)
  .map(x => 20 * x)
  .subscribe(x => console.log("item:␣" x)));
```

The source Observable stream contains four elements with numeric values 2, 4, 6 and 8. The `just` operator is used to convert an item into an Observable that can emit that item. The items emitted by an Observable are transformed by the `map` operator, which applies a function on each item emitted. Here, it takes numeric values of each item (`x`) and multiplies it by `20`. The resulting Observable contains items with numeric values 40 (20 * 2), 80 (20 * 4), 120 (20 * 6) and 160 (20 * 8). A simple marble diagram for the above is as shown here.

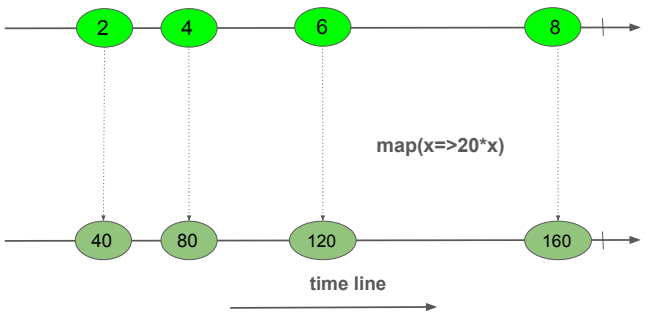


Figure 3.13: Marble diagram.

Next events are represented as circles, error events as crosses and completed events with vertical lines. The Marble diagrams are read from left to right. In

[2] `http://reactivex.io/intro.html`

[15], the authors extend the original Marble diagram by introducing animation to make it dynamic. For example, the dynamic marble diagrams update are live, when new events occur and are stacked showing the complete data flow. The authors claim that as part of their future work, they want to show combination of Observable streams in Marble Diagrams, visualize explicitly the causality of events and scale up the marble diagrams for larger programs. One of the challenging factors with visual support for debugging large reactive programs is scalability of observable graphs and marble diagrams. Rendering them at real time may be very difficult, one reason being visual display space being restricted. So, while displaying marble diagrams involving causality of events, we believe the HipHop.js approach to narrow down causality cycles can be used to visually display the smallest of causality relationship to the programmer in Rx programs and also for better error messaging .

In [121], the problem of debugging reactive programs is highlighted. They propose a reactive debugger called RP Debugging for Eclipse/Scala IDE. The debugger uses dependency graph as the runtime model to understand the progress of reactive applications. The programmer is visually presented with a dependency graph while debugging. At the definition position of signals, new nodes are created for the dependency graph and dependencies among reactive values are established. These graphs help programmers in rectifying any of the incorrect dependencies (causality error) through the nodes and their dependencies. One of the disadvantages in this method is scalability with respect to huge applications and large number of signals. We believe the usage of WTO here can narrow down to a smaller cycle and actual signals when facing causality errors in complex systems. In the next section we conclude this chapter along with a brief discussion about proposed future work.

3.6 Conclusion and Future Work

We presented the method of causality error analysis and debugging process by building on SCC and WTO approaches, which help HipHop.js programmers narrow down to smaller error positions in source code with supporting examples and illustrations. Before presenting our approach, we also provide introduction and relevant theoretical basis to understand the genesis of causality error in synchronous reactive languages. We show the results and advantages of application of our

debugging approach in a real life project developed using HipHop.js. We also present the visual debugging support for the same in Emacs and VScode editors.

We believe that in some of the recent approaches to reactive programming based GUI design and game programming, SCC and WTO based approaches can be of great help in debugging causality problems as many of these approaches are using directed graphs to build dependencies, and when graphs become bigger and complex, SCC and WTO will help in narrowing down to a smaller causality error cycle.

As part of future work, we plan to investigate the process of providing causality error debugging support for graphical programming environments, such as the Blockly tool kit. Blockly[3] is a JavaScript library, which provides ready made UI for creating a visual language [110] that emits syntactically correct user generated code. The visual approach to programming is gaining lot of traction and will be more attractive and fun to introduce programming to non-cs background programmers. It will be interesting to provide debugging support for HipHop.js in these sort of visual programming environments.

With this we end this chapter on Causality error tracing, which is a part of HipHop.js debugging infrastructure. This complements the HipHop.js program analyzer which we present in the next chapter.

[3] `https://developers.google.com/blockly`

Chapter 4

HipHop.js Program Analyzer

> *"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."*
>
> — Maurice Wilkes

In the previous chapter, we presented about the support to HipHop.js compiler in debugging "causality errors". This chapter introduces about HipHop.js program analyzer, part of HipHop.js debugging infrastructure. Section 4.1 introduces about HipHop.js program analyzer, including the motivation and objective for having one. Section 4.2 will detail the various utilities provided by the HipHop.js analyzer. Section 4.3 explains the usage of the HipHop.js analyzer utilities.

4.1 Introduction and Objective

Software have bugs, and programmers generally encounter lengthy and laborious correction process to remove bugs. Human errors, ever-increasing complexity of software, and lack of debugging knowledge are some reasons attributed for Software failures in the software engineering literature. One of the survey on successful practices for debugging [103] notes that the difficulty of debugging is more in the earlier stages of software development, rather than in the later stages of repairing

errors. Questions [81] and query based debugging [91, 93, 60] have been effective in software fault localization and is practiced in providing interactive support to debugging software throughout its life cycle. In [100] on designing a debugger for Lustre reactive programming language, the authors argue that the debugger in reactive systems can also be a support system for reasoning and understanding about the system throughout the software development phase, and it should allow a programmer to trace the execution of their program by choosing inputs, and observe the evolution of its states by playing with the program, with simple easy to use tools.

In reactive systems, when debugging, step by step execution is on the basic clock of the reactive system, where every clock tick is a step. Breakpoints in reactive systems are generally set through identifying a goal point. Here, a programmer can choose a specific goal point (output or a combination of outputs) and check what input leads to such a goal and at which instant [59]. When we talk about goal points, these can be based on the expectation of the programmer that something bad (not expected signals) should never happen, or the expected output should happen in near future. Our objective is to provide a simple, easy to use program analyzer infrastructure which can be part of a bigger debugging infrastructure that can help HipHop.js programmers minimize the bugs as much as possible throughout the development phase by helping them in understanding the behavior of their programs, play with programs by simulating the executions, and in narrowing down to the source of errors when confronted with. The idea of "step" or instant brings "temporal" behavior of a program into context, and we keep this in mind in the design of the HipHop.js program analyzer.

The HipHop.js program analyzer we propose aims to provide a library of utilities which can help programmers analyze and understand the temporal behavior of their programs. They can search for strange, expected, or unexpected behaviors that may exhibit programming or design errors and also in parallel improve their understanding of the intended program to be developed.

These analyzer utilities can be used by the HipHop.js programmers in a simple fashion, with each of them having a specific usage context and these utilities, we believe panders to most of the needs of HipHop.js programmers. In the next section, we introduce the utilities provided by the program analyzer.

4.2 HipHop.js Program Analyzer

The utilities we propose are guided by the objectives stated earlier and based on experience of HipHop.js programmers including ourselves. The utilities automate the part of the work which programmers had to do manually for observing the temporal behavior of their program. The design of the utilities is inspired in part by questions and query based debugging. Each of the utilities help HipHop.js programmers to raise a specific question (can also be constraints) on the behavior of the program, validated by HipHop.js program analyzer. These utilities are designed to raise questions on the temporal relationship between input and output signal events.

Any HipHop.js programmer or any reactive system programmer will be curious on the emission of the signals from the reactive systems being designed. For a program or a module which has inputs and outputs, to know its behavior, the likely question that a programmer may have is: *At what reaction instant, a specific output signal may be emitted?* or *what are the input stimulus that are required from the environment for a specific output signal to be emitted?* The answer to above questions, if the proposed program analyzer can predict, without manual intervention from the programmers, by the way of providing inputs or deciding the order of the inputs, we believe will help programmers clearly understand what is happening in their program, with respect to the temporal behavior. By removing the manual intervention of providing inputs, the analyzer can remove any unintended biases programmers may have which may hide the actual behavior of the program while providing inputs, especially in the early days of developing reactive systems. Talking about biases on the inputs, for example, there may be a scenario where input signals `A` and `B` should never be present at the same instant, or there may be a scenario where some input signals should be present at a given reaction instant. Then, the likely question can be: *If availability of inputs have some constraints on their timing, what will be the status of the output signals henceforth?* These answers from the analyzer will help programmers understand a different scenario with constraints on input signals than the previous one.

When dealing with composition of multiple modules, wherein multiple output signals and inputs come into context, the concern of the programmers may be tilted towards undesirable output emissions in a given instant. For example, in an air

condition system, if the "raise_temperature" to heater and "reduce_temperature" to cooler are emitted at the same instant, it will surely defeat the design objectives of an air conditioner. In these scenarios, the programmers will be interested to know: *Will two output signals emit at the same instant?* This may be a desired or not so desired event for the programmers, like integration errors when combining two perfectly behaving modules. This will help programmer see through some of the new behavior that is not exposed when dealing individual modules. Further, when talking about the temporal behavior of a program, there may be instances where the programmers want a specific behavior at a certain instant in terms of output signal - *Can a signal emit between specific reaction instants?* The response from the analyzer, can help assure the programmers on what to expect and hence can ensure design of their systems in a deterministic way.

As part of analyzer, we propose a combination of utilities that to support programmers to query on the temporal behavior of programs and get meaningful response by the analyzer. The analyzer we provide is in the form of a library of utilities. For the questions to be framed on their programs and interact with utilities by the programmers, we propose easy to use and intuitive interface through utilities.

Now, we introduce the interface to the utilities HipHop.js program analyzer provides. They are listed as follows:

1. `checkOutput(reactivMachine, [sigArray_out])` - The programmer can query the analyzer for checking the output signals emission. The arguments for the utility includes the reactive machine generated by HipHop.js compiler and an optional list of output signals (`sigArray_out`) to be checked for emissions. If it is empty, then all the output signals in the module will be considered to check for emissions.

2. The following are the interfaces where constraints (relations) can be specified on the timing of input signals at each instant to the HipHop.js program analyzer. Other than the reactive machine from the HipHop.js compiler, the arguments to the utility should include a list of input signals (`sigArray_in`) which have constraints and an optional output signal list(`sigArray_out`):

   (a) `inputImplies(reactivMachine, sigArray_in, [sigArray_out])` - here the input signals in the argument list can be made available by the analyzer

at the same instant and then can be checked for the emission of output signals. The output from the analyzer will show when a particular signal will be emitted, if at all, and the condition under which it happens.

(b) `inputExclusive(reactivMachine, sigArray_in, [sigArray_out])` - here the input signals in the argument list can be made mutually exclusive to one another by the analyzer and then can be checked for the emission of output signals. The output from the analyzer will show when a particular signal will be emitted, if at all, and the condition under which it happens.

3. `sameInstance(reactivMachine, sigArray_out)` - programmers can check with the analyzer, if the output signals in the `sigArray_out` list can be emitted at the same reaction instant. The analyzer checks for possible concurrent emission of those signals and informs the programmers if it happens so. Also, the analyzer provides a scenario of how this can happen in terms of statuses of inputs signals.

4. `outputRange(reactivMachine, sigArray_out, timeRange)` - programmers can check if there are output signals emission in the specified reaction instants range. The arguments include list of output signals (`sigArray_out`) programmers are interested and another list (`timeRange`) of size two, containing the reaction instant range in which the programmers are expecting the output signals emission. If this is possible, the analyzer returns the result of input signals and their timing, for all the output signals whose emission can possibly happen in the specified range of instants.

The HipHop.js program analyzer with the above introduced utilities to begin with, can help programmers understand the temporal behavior of their programs, play with their programs, and we believe this helps in fault localization by narrowing down any of the unintended design or logical errors. Also, based on the infrastructure for the above introduced utilities, some more utilities can be built, providing better testing and debugging experience to HipHop.js programmers. We discuss a couple of them in the next chapter.

In the next section, we illustrate the usage of the above listed utilities in actual HipHop.js programs and the way the program analyzer generates meaningful response to programmers.

4.3 Interfacing with Utilities

This section elaborates on previous sections with details on the usage of the HipHop.js program analyzer utilities introduced earlier. First, we present the way the HipHop.js program analyzer can be integrated with normal HipHop.js programming and then present the usage of each of the utilities we propose.

The analyzer utilities are provided as part of a library which needs to be imported into the source file of HipHop.js program the programmers want to play with. The HipHop.js compiler generates a reactive machine for each module and this reactive machine is used by all the utilities provided by HipHop.js program analyzer. For each of the utilities, the arguments will include the reactive machine and specific inputs based on the type of utility used. The following is an example which illustrates the generic usage of the proposed HipHop.js program analyzer.

```
1 "use␣@hop/hiphop";
2 "use␣hopscript";
3
4 import * as hh from "@hop/hiphop";
5 import * as analyzer from "hhAnalyzer";
6
7 hiphop module prg() {
8    out B,C;
9    ...
10 }
11 const machine = new hh.ReactiveMachine(prg);
12 analyzer.checkOutput(machine, ["B"]);
```

Listing 4.1: General usage of Analyzer utility.

The library is imported in line 5 and the HipHop.js compiler compiles the module `prg` to a reactive machine (line 11). This is passed as an argument to the program analyzer utility, `checkOutput` which is part of the imported library namespace `analyzer` as in line 12. When this program is executed, programmers can get to know when (in terms of reactive instants) will the output signal `B` be emitted, if it is possible. In the subsequent sections, we will elaborate on the usage of all the utilities we propose that are part of HipHop.js program analyzer.

The first utility we present is `checkOutput`, which can be used to check for the emissions of all the output signals.

Checking Output Emissions - `checkOutput`

The primary utility of the program analyzer is `checkOutput`. It analyzes the HipHop.js program and returns one of the possible scenarios when emissions of the output signals can happen. Listing 4.2 is a simple example with the utility. As we have presented earlier in the background section on HipHop.js (section 2.3), `await(Sig)` waits from the next instant for "`Sig`" to be present. In line 16, `analyzer.checkOutput(reactivMachine)`, the analyzer utility is used to check for emission of output signals. Since the argument to the utility does not have the optional output signals list, the analyzer returns results for possible emission scenarios of all the output signals in the module.

```
1  "use␣@hop/hiphop";
2  "use␣hopscript";
3
4  import * as hh from "@hop/hiphop";
5  import * as analyzer from "hhAnalyzer";
6
7  hiphop module prg() {
8     in A, B;
9     out O;
10
11    await(A.now);
12    await(B.now);
13    emit O();
14 }
15 const machine = new hh.ReactiveMachine(prg,{ sweep:true });
16 analyzer.checkOutput(machine);
```

Listing 4.2: `utility 1`, checking emission of all the outputs.

The utility will return one of the possible trace of input events for the output signal emission of `O`, if it can happen. The output is presented as JSON file. The output in listing 4.3 is a JSON object, the output signals which are being checked for will be the "keys" and "values" will be in the form of another JSON object. The JSON object representing the values will have number of reaction instants as its keys and inputs signals as values for those reaction instants.

```
1 {
2         "O": {
3                 "1": [],
```

```
4                   "2": ["A"],
5                   "3": ["B"]
6           }
7   }
```

Listing 4.3: output trace, with inputs and output after certain reaction instants

From the output listing, we see that signal `O` can be emitted and one of the possible scenarios for that is, when the emission happens at instant `3`, with the inputs specified as in the value (array of input signals) for particular reaction instant. In instant `1` at line 3 - no inputs, in instant `2` - input signal `A`, and in instant `3` - input signal `B`.

Here is another example (listing 4.4) wherein there is `await` (section 2.3) with a counter to count the instances of input signal `I` (3 counts) at line 5, after which signal `O` should be emitted as in line 6. The argument list to the utility does not have the optional output list to consider, meaning which the analyzer will return the result for all the output signals in the module, and in this example `O` is the only output signal.

```
1  hiphop module prg() {
2      in I;
3      out O;
4
5      await count(3, I.now) {
6          emit O();
7      }
8  }
9  const machine = new hh.ReactiveMachine(prg );
10 output.checkOutput(machine);
```

Listing 4.4: checking outputs with a counter on input.

The output with one of the possible scenario is as follows in output lisiting 4.5. It can be seen that output signal `O` is emitted at reaction instant `4`. For each of the four reaction instants, the corresponding input signal that should be true is also presented, stressing the fact tha signal `O` should be emitted at the third instance of occurrence of signal `I`, and since it is `await` without `immediate` construct, the counting happens from second instant.

```
1   {
2           "O": {
```

```
3                   "1": [],
4                   "2": ["I"],
5                   "3": ["I"],
6                   "4": ["I"]
7           }
8   }
```

Listing 4.5: output trace, with inputs bound by counter and output after certain reaction instants.

Further, the programmers can also check with the program analyzer about the emission of a specific output signal emission. This is illustrated in the listing 4.6. Here, the interest is in the emission of signal `P`. This can be specified as - `checkOutput(reactivMachine, ["P"])` at line 12.

```
1  hiphop module prg() {
2     in A,B;
3     out O,P;
4
5     await(A.now);
6     emit P();
7     await(B.now);
8     emit O();
9  }
10
11 const machine = new hh.ReactiveMachine( prg,{ sweep:true } );
12 output.checkOutput(machine,["P"]);
```

Listing 4.6: `Utility 1`, checking a specific output signal emission.

The following output listing 4.7 illustrates one of the possible scenario for the emission of signal `P`.

```
1  {
2          "P": {
3                  "1": [],
4                  "2": ["A"]
5          }
6  }
```

Listing 4.7: output trace, specific signal emission and its relation with inputs.

In the above utility, there is no restriction or constraint on the order input signals were provided. Now, we present utilities that can enforce certain constraints on the order of the input signals.

Constraint on Input Signal Presence

The HipHop.js program analyzer provides two utilities using which programmers can specify constraints on inputs signals. These can be viewed as a relationship between input signals, as follows:

1. Implies - if one input signal is present at a instant then all the other signals specified are also to be present at the same instant.

2. Exclusivity - no two or more input signals specified can be present at the same instant.

These constraints can be set on a subset of the input signals. We provide utilities with interfaces `inputImplies` and `inputExclusive` to let the program analyzer enforce respective constraints while analyzing the programs.

We illustrate the usage with examples as follows. In the listing 4.8, with the help of parallel constructs `fork-par`, the reactive machine is waiting in parallel for signals `A`, `B` and `C` starting from the instant `2`. Once they occur in any order, signal `O` is emitted. The behavior resets when signal `R` is present. In the listing at line 18, we can check with the program analyzer if there will be emission of `O`, provided the constraint set on input signals, such that they are only available at the same instant. This is specified by the list of input signals `["A", "B", "C"]` in the argument list.

```
1 hiphop module prg() {
2     in A,B,C,R;
3     out O;
4
5     do {
6         fork {
7             await(A.now);
8         } par {
9             await(B.now);
10        } par {
```

```
11              await(C.now);
12          }
13          emit O();
14      } every (R.now);
15  }
16
17  let machine = new hh.ReactiveMachine(prg);
18  analyzer.inputImplies(machine, ["A", "B", "C" ]);
```

Listing 4.8: `Utility 2`, checking outputs with a constraint on the input.

The analyzer returns the result of one possible scenario for emission, and it is as illustrated in the following listing 4.9. We see here that, the analyzer has identified the trace for emission of signal `O` with inputs `A`, `B` and `C`, all provided at the same instant.

```
1  {
2          "O": {
3                  "1": [],
4                  "2": ["A","B","C"]
5          }
6  }
```

Listing 4.9: output trace, signal emission based on constraints on input signals.

For the same source listing 4.8, we illustrate another constraint on inputs, the exclusivity of input signal occurrence as in listing 4.10, checked with program analyzer.

```
20 analyzer.inputExclusive(machine, ["A", "B", "C" ]);
```

Listing 4.10: `utility 2`, checking outputs with constraints on the input.

The result by the program analyzer is illustrated in the following listing 4.11. The analyzer predicts that with inputs `A`, `B` and `C` at different instants (one possible combination), signal `O` emission is possible at instant `4`.

```
1  {
2          "O": {
3                  "1": [],
4                  "2": ["C"],
5                  "3": ["A"],
6                  "4": ["B"]
7          }
```

```
8 }
```

Listing 4.11: output trace, signal emission based on constraints on input signals.

We illustrate another combination of exclusivity constraint on inputs as in listing 4.12, checked with program analyzer.

```
20 analyzer.inputExclusive(machine, ["A", "B"]);
```

Listing 4.12: `utility 2`, checking outputs with constraints on the input.

The result by the program analyzer is illustrated in the following listing 4.13. The analyzer predicts that with inputs `A`, `B` at different instants (one possible combination), signal `O` emission is possible at instant `3`.

```
{
        "O": {
                "1": [],
                "2": [
                        "A",
                        "C"
                ],
                "3": [
                        "B"
                ]
        }
}
```

Listing 4.13: output trace, signal emission based on constraints on input signals.

The programmers may be interested to execute the HipHop.js reactive machine manually for a few instants with their own set of inputs (by the means of `machine.react()`) and from then on, they may be interested to check emission of output signals using the program analyzer. By executing manually for a few instants, the programmers get the flexibility of specifying their own inputs for a few instants and then checking for expected or unexpected behavior of their program with the help of the analyzer. This adds on to the effort of providing interactive debugging experience for the HipHop.js programmers. We illustrate the above scenario with following examples. Let us consider the listing 4.14, the reactive machine is executed manually for two instants at lines 16 (no input) and 17 (input `B`) and then it is checked with the program analyzer for output emissions after the two reaction instants.

```
1  hiphop module prg() {
2      in A, B,  R;
3      out O;
4
5      do{
6          fork{
7              await(A.now);
8          } par {
9              await(B.now);
10         }
11         emit O();
12     } every(R.now);
13 }
14
15 let machine = new hh.ReactiveMachine(prg);
16 machine.react();
17 machine.react("B")
18 analyzer.checkOutput(machine);
```

Listing 4.14: `Checking outputs after some reaction instants.`

In the listing, in line 16, we see that the machine is executed with no input in the first reaction instant, and in line 17 the machine is executed with input `B` as part of second reaction instant. Then in line 18, the program analyzer is asked to check for possible emissions. The program analyzer returns as result, one of the scenario for possible emission of signal `O`. The output is as illustrated here in listing 4.15.

```
{
        "O": {
                "1": ["A"]
        }
}
```

Listing 4.15: output trace, signal emission after manual execution for few instants.

Since the machine has already reacted for two instants with some inputs before calling the analyzer, the result shows that in the instant `1`, signal `O` can be emitted if input `A` is provided at that instant as input from the environment, Here the analyzer has taken into account that in the previous instant signal `B` has been present.

Another combination of manual execution for the above source is illustrated here in the following listing, wherein the analyzer is checking, if after three instants of manual execution (with emission of signal `O` at instant 3), can there be emission of signal `O`.

```
machine.react();
machine.react("B")
machine.react("A")
analyzer.checkOutput(machine);
```

Listing 4.16: `utility 3`, checking outputs after some reaction instants.

The result provided by the analyzer in listing 4.17 tells the programmer that to get the emission of the output signal `O`, one possible scenario henceforth the manual execution is that in the first instant, the input should be `R`, which will reset the behavior of the above reactive machine and then provide the other inputs to get emission of signal `O`.

```
{
        "O": {
                "1": ["R"],
                "2": ["A","B"]
        }
}
```

Listing 4.17: output trace, signal emission after manual execution for few instants.

At this juncture, we also want to stress that the utilities which help in setting constraints on the inputs signals ordering can also be used in interactive manner, i.e, after executing the reactive machine for a few instants, utilities `inputImplies` and `inputExclusive` can be used to interact with program analyzer to check for output emissions. The program analyzer, takes note of the manual execution of the reactive machine with user inputs, and the constraint set on the inputs, and then provides the possible scenarios of output emission then on. Here we illustrate the usage of `inputExclusive` as follows for the program (listing 4.14) as in the following listing 4.18. We see that after three instants of manual execution, the program analyzer is asked at line 4 to check for emission of output signals with the constraint that input signals `A` and `B` are mutually exclusive.

```
machine.react();
machine.react("B");
```

```
3 machine.react("A")
4 analyzer.inputExclusive(machine,["AO","B"]);
```

Listing 4.18: `Combination of utilities`, checking outputs after some reaction instants with constraints on input

The output is listed as follows.

```
{
        "O": {
                "1": ["R"],
                "2": ["B"],
                "3": ["A"]
        }
}
```

Listing 4.19: output trace, signal emission after manual execution for few instants with constraint on inputs.

The output gives one possible scenario for the emission of signal `O` taking into consideration the constraints in input signals ordering and execution for three instants manually. As before, to get signal `O` emission, the input signal at first instant should be `R`, which will reset the behavior of the reactive machine and then signals `B` and `A` in instants `2` and `3` as they should be mutually exclusive of one another in an instant. Next, we present the usage of utility which checks if two or more output signals are emitted at the same instant.

Checking for Emission at the Same Instant - sameInstance

Here, the programmers can ascertain if there are any conflicting (unexpected) or desired combination of emissions happening at the same instant. We illustrate the usage with a very simple example, to show the usage rather than stressing on the functionality, as follows.

In listing 4.20, we see that there is a wait on signals `A` and `B` and then emissions of two signals `O` and `P` happening at the same instant. This can be checked by the program analyzer when the utility `sameInstance` with the signal list `["O","P"]` is used as in line 12. This utility will check for emissions of those signals at the same instant and produces an output based a possible scenario of those emissions happening.

```
hiphop module prg() {
   in A, B;
   out O, P;

   await(A.now);
   await(B.now);
   emit O();
   emit P();
}
const machine = new hh.ReactiveMachine (prg);
module.exports = machine;
analyzer.sameInstance(machine,["O","P"]);
```

Listing 4.20: observer utility usage for emissons at the same instant.

The output from the program analyzer, tells that whether the signals emissions as specified in the argument list for the utility happen at the same instant or not. The following line tells us if the test is successful or not.

```
-----verifying-----
[ 'O','P' ] are emitted at the same instant.
```

Listing 4.21: output for sameInstance utility.

The resulting output scenario is as in the following listing 4.22

```
{
        "O": {
                "1": [],
                "2": [
                        "A"
                ],
                "3": [
                        "B"
                ]
        },
        "P": {
                "1": [],
                "2": [
                        "A"
                ],
                "3": [
                        "B"
                ]
```

```
19          }
20  }
```

Listing 4.22: output trace for two signals emitting at the same instant

Both the signals in one possible scenario are emitted at the same instant - 3. As an illustration for failure of the test, we present the following example listing 4.23.

```
1  hiphop module prg() {
2     in A, B;
3     out O, P;
4
5     await(A.now);
6     emit O();
7     await(B.now);
8     emit P();
9  }
10 const machine = new hh.ReactiveMachine (prg);
11 module.exports = machine;
12 analyzer.sameInstance(machine,["O","P"]);
```

Listing 4.23: observer utility usage for emissons at the same instant.

The program analyzer provides the following output, implying the failure of the test and also there will be no output showing the possible scenarios.

```
-----verifying-----
[ 'O','P' ] are not emitted at the same instant.
```

Listing 4.24: output for sameInstance utility.

Next, we present the utility which help programmers to determine if a particular signal can be emitted at some particular time instants.

Checking Output Emissions at a Specific Range of Instants

We provide an utility which can check with the program analyzer if a particular signal emission happens between a range of time instants. The analyzer utility `outputRange` will check if an output signal gets emitted within a specific range of instants. The output signal in question and the range of instants are given as arguments to the utility. The range of instants should follow the order $[L, H]$, where $L \geq 0$ and $H \geq L$.

The following listing illustrates the usage of utility `outputRange` at line 11 to check if signal `O` can be emitted within a range of [1,4] instants.

```
1 hiphop module prg() {
2     in I;
3     out O;
4
5     loop {
6         await count(3, I.now);
7         emit O();
8     }
9 }
10 const machine = new hh.ReactiveMachine(prg);
11 analyzer.outputRange(machine, ["O"],[1,4]);
```

Listing 4.25: `outputs within range of time instants.`

The output for the above program from the analyzer provides a possible scenario for the emission of output signal in the specified range of instants.

```
{
        "O": {
                "1": [],
                "2": ["I"],
                "3": ["I"],
                "4": ["I"]
        }
}
```

Listing 4.26: output trace, signal emission within specified range of instants.

We run the same code with a different range as follows.

```
analyzer.outputRange(machine, ["O"],[1,2]);
```

Listing 4.27: output for another run with different criterion

The analyzer returns a failure status by means of empty output, to convey that the output signal `O` cannot be emitted within the specified range of instants `[1,2]`. With this we come to the end of this section on the usage of various program analyzer utilities. These utilities help in program comprehension and fault localization by providing interactive query based debugging support for HipHop.js programmers. Though these utilities are not exhaustive, even new additions can never cover all the questions that may arise in the programmers mind. We believe these utilities,

without overwhelming, provide HipHop.js programmers simple intuitive tool to understand reactive programs temporal behavior and when faced with design or logical errors, help with narrowing down to the source of error by improving program comprehension. These utilities provide a basic level of support to the HipHop.js programmers as these revolve around input-output signals relationships crucial to working of reactive systems. In the next chapter, we detail the implementation of each of the utilities of the program analyzer.

Chapter 5

Implementation of Program Analyzer

"We argue that proof construction is unnecessary in the case of finite state concurrent systems and can be replaced by a model-theoretic approach which will mechanically determine if the system meets a specification expressed in propositional temporal logic"
– Edmund M. Clarke and E. Allen Emerson

In this chapter, the implementation details of the program analyzer is presented. This chapter presents and explains the building blocks of the program analyzer, the approach taken in implementing those building blocks, the tools integrated with the analyzer and a brief introduction to theory of temporal logic used by model checkers. Overview of the implementation process is presented in section 5.1. In section 5.2, the tools that are used by the program analyzer and relevant theory about temporal logic is presented. Finally, in section 5.3, the implementation method is presented in detail.

5.1 Overview

This section presents an overview of the design methodology of the utilities available in HipHop.js program analyzer. The objective of the utilities is to help programmers understand the temporal behavior of the HipHop.js program, thereby minimizing design errors and also have a better understanding of their programs. The program analyzer helps the programmers automate questions relating to the temporal

behavior of their programs; it takes input, an HipHop.js program, based on the utility used, provides the corresponding output to the programmers which are readily understandable and can be used further.

The reactive machine generated by HipHop.js compiler has states and by design is deterministic. The net list (set of logical gates and latches) generated by the HipHop.js compiler of the corresponding reactive machine provides a Finite State Machine (FSM) representation of the reactive machine. Using one of the strategies like explicit state searching, or some of the advanced techniques used in the field of model checking, states can be searched to answer the questions raised through the utilities by the programmers. The objective of the implementation process is, getting all the states of the reactive machine and then search for a particular state which satisfies the condition set by the programmers through the utilities. If explicit state checking strategies are used, one may encounter classical state explosion problem for bigger and complex programs. Nowadays, tools like model checkers which use new techniques to counter the state explosion problem are of industrial strength and found to be relatively efficient in the job of state searching. These model checkers are used as back end in the design of HipHop.js program analyzer utilities for state searching. What to search, and providing FSM representation of the program in an understandable form to the model checker is the responsibility of utilities in the program analyzer.

A model checker operates on a model of the program and the queries raised on the program. The model checker performs an efficient search and returns the result. The model checker tools stipulate a particular input format for the model of the program and the queries. Intermediate layers are designed inside HipHop.js program analyzer which translates the net list generated by HipHop.js compiler into a format that can be used by the model checkers.

The program analyzer utilizes two tools in its backend for state searching. The XEVE [30] verification tool (introduced in next section) which has been used with Esterel, is one of the backend used in HipHop.js program analyzer. It requires the model of the reactive program to be in Berkeley logic interchange format (BLIF) [20]. NuSMV model checker is the second tool in the backend which requires the model of the program to be in SMV format and the queries are to be in standard notations, used to represent "temporal logic".

The program analyzer translates HipHop.js programs to BLIF format that can

be used with XEVE and then utilizes the readily available tools like AIGER [27] to translate BLIF representation into SMV format which can be used with NuSMV model checker. The queries raised by the utilities are converted to temporal logic by the program analyzer and then given to model checker. The model checker searches the model based on the search criterion defined by temporal logic. In the next section, a brief introduction to all the tools and temporal logic is presented.

5.2 Theory and Tools

As HipHop.js program analyzer uses model checkers as back end, a brief overview on model checking is presented and as HipHop.js program analyzer uses XEVE, AIGER, and NuSMV tools, a brief introduction to them is also presented in later part of the section.

5.2.1 Model Checking and Temporal Logics

The goal of model checking is automatic verification of systems. In the initial days of computing systems, for the transformational systems, total correctness was described as a process of proving partial correctness and termination. The model used to represent the systems was the input - output relation represented in formal semantics and proposition logic was used as a specification language for proving correctness.

In the early 1960s the advent of reactive systems happened. In reactive systems, termination is not a desirable virtue. Here the total correctness is described as a combination of safety, progress and fairness, etc. Programs have to be represented in a way that is verifiable and also abstracted with many of the finer details which are not crucial for verification, called modeling. Modeling of the programs is one of the crucial step and also specifying what needs to be searched on these models is as important as modeling the systems or the programs in question. Usually "safety" and "liveness" are the interesting properties one is concerned about. "Safety" ensures nothing bad will never happen and "liveness" ensures something good will eventually happen. Fair transition systems and temporal logics are used for the above said modeling of systems and properties.

A fair transition system is basically represented as set of states (with an initial

state or set of states), with transitions in between them and labels associated with states and transitions. Fair transition systems can be used to model a system, specifically reactive systems, and can be used for verification. These transition systems can be represented using first order logic. Temporal logic can be used to specify the property which needs to be checked. It all started with Linear Temporal Logic (LTL), then Computation Tree Logic (CTL) and then a combination of both called CTL*. These specifications are represented as first order logic and used by model checkers. The model used to represent FTS is Kripke structures and the specification language is temporal logic.

The model checkers initially followed explicit state model checking, going through all the states and checking for any bad states. This proved to be fine for small programs or programs with very less number of states. With large programs or with those having many number of states, the popular "State explosion problem" became an issue and this approach was prohibitive. Binary Decision Diagrams (BDD), specifically Ordered Binary Decision Diagrams (OBDD) was a big step in advancing the field of model checking by providing initial solution for the state explosion problem which easily solved programs with up to 10^{20} states. This approach was further supported with other advancements like partial order reduction strategy. Later, propositional satisfiability played important role in the progress of the model checking and increased the capability of model checkers. Bounded model checking is based on using SAT solvers in the problem of model checking; here, based on the value of some 'k', the state machine representing the system is unrolled, in the sense that the maximum path distance of 'k' is considered, and all the states in that diameter is considered while model checking. This is further refined with approaches called "K-induction", "invariant interpolation" and other approaches, each refining on the previous methods. As of now SMT solvers are occupying the scene of model checking, due to advances in theories of various domains.

The evolution of temporal logic as a specification language dates back at least to the beginning of Middle Ages, where temporal logic was used by philosophers as modal and temporal inferences in natural languages. In the beginning of 20^{th} century temporal logic went through more formalization, new primitives like `always`, `sometime`, `until`, `since`, etc., were added to the existing ones which included `past`, `present`, `future`. A. Pnueli in 1977 suggested the use of temporal logic as

specification language [114]. In the lingua franca of model checking community, it is defined as *verifying whether a Kripke structure of a system is the model of a temporal formula.* The steps in model checking are as follows [105]:

- Model abstraction of system under investigation.
- Validate the model.
- Run model checker for properties of interest.
  - If a property holds "true", then property holds for model and possibly for the system too.
  - If the property holds "false", counterexample helps debugging of model and system.
  - If there is a timeout, then the model building has to be fine-tuned.

Kripke Structure is a basic model of computation represented by K = *(S,R,L)*, where S is the system states, $R \subseteq S \times S$. and L is the labeling function. With AP has a set of atomic propositions over states, L: $S \rightarrow \mathcal{P}(AP)$ is the set of atomic propositions true in each state. A path π in K is an infinite sequence of states: $\pi = s_0, s_1, s_2...$ such that, if $i \geq 0$ then $(s_i, s_{i+1}) \in R$.

Temporal logics typically differ based on the way the branches are handled in the computation tree. In linear temporal logic (LTL), the operators describe the events on a single computation path, whereas in branching-time logic, also called as Computation Tree Logic (CTL), the temporal operators are provided to quantify all possible paths from a given state. The computation tree logic CTL* combines both linear-time and branching-time operators.

Linear-time temporal logic is used to express time-dependent properties of system runs. It is evaluated over infinite sequence of labels. LTL is built over the set of atomic propositions with the logical operators ($\neg$ and $\vee$), and the temporal operators `X` (next), and `U` (until). Formally, the set of LTL formulas over atomic proposition (AP) is inductively defined as follows:

- if $ap \in$ AP then, ap is a valid LTL formula.
- if ψ, ϕ are LTL formulas which are valid, then $\neg\psi$, $\psi \vee \phi$, Xψ, ψUϕ are also valid LTL formulas.

The temporal operators G(Globally), F(Future), R(Release) in LTL can be defined in terms of X and U. Examples: if P, Q are valid LTL formulas, then

- invariant can be expressed as (GP),
- response, recurrence as (G(P→ FQ)),
- reactivity as (GFP → GFQ), and
- precedence as G(P_1 U P_2 ...P_n).

Branching-time temporal logic includes assertions about branching behavior. The general format is QT. Where Q can be E - *at least one path* (**possibly**), A - *for all paths* (**inevitably**). Whereas T can be one of the X, F, G, U, or W operators as introduced in LTL. These are the basic CTL operators:

- AX (all next) and EX (exists next),
- AG (all global) and EG (exists global),
- AF (all future) and EF (exists future),
- AU (all until) and EU (exists until).

An example CTL formula is as follows - EF(begin $\wedge$ $\neg$ready), which is true for states where begin holds but ready does not hold.
Difference between LTL and CTL: LTL cannot express CTL formula AG(EF p), while CTL cannot express A(FG p). More expressive logics like CTL* and μ-calculus [101] can be used further. In CTL* logic, a *path quantifier* is used to **prefix** an assertion composed of arbitrary combination of the *LTL operators*. Model checking algorithms and techniques are not presented here as they are out of scope of this thesis. Interested readers can refer any of the standard literatures available like [44, 114]. Next, the BLIF format is introduced.

5.2.2 BLIF

Berkeley logic interchange format (BLIF) [20] is a form of describing logic-level hierarchical circuits in "textual" form. A logic circuit can be a combination of combinational and sequential circuits. Each circuit is seen as a directed graph of

combinational and sequential logic elements where, a two-level, single-output logic function is associated with each node. A detailed presentation on BLIF is given in appendix A.

5.2.3 XEVE

XEVE [30] is a tool for verifying Esterel programs. The Esterel compiler, as one of the compilation methods, compiles Esterel programs into Finite state Machines where boolean equations and latches are used to enumerate states and transitions into boolean circuits. The equations generated are in BLIF format. Binary Decision Diagram (BDD) library called `TIGER` is used internally by XEVE. `TIGER` takes input an FSM described in BLIF and provides functionalities to the programmers, like checking the emission status of output signals. XEVE is elaborated further in related works section. AIGER format and tool set is presented in the next section.

5.2.4 AIGER

The AIGER file format is another way of representing combinational and sequential logic files using AIG format, the `AND INVERT GRAPH` format. It was introduced by Biere et al. [27] and has been used in model checking competitions since 2007. The format uses both ASCII and binary representations. The ASCII format is helpful as a human-readable format, while binary format is more compact and can be used by software applications. The AIGER tool set includes routines converting BLIF format files to AIG format and from AIG format to SMV format which can be utilized by NuSMV model checker. The HipHop.js program analyzer uses AIGER tool set for the above intermediate translations. For more details on AIGER, readers are requested to refer appendix B.

5.2.5 NuSMV

NuSMV [42] is a symbolic model checker, it is a re-implemented, extended version of CMU SMV [104]- a BDD (ROBDD) based Symbolic model checker. Finite-state systems are described in a specialized language and specifications to be verified are given as LTL-CTL formulas. It verifies specification or produces a counterexample in case of failure. NuSMV has evolved over versions. Version 1

implemented BDD-based symbolic model checking, while the version 2 of NuSMV extends version 1 with many other features, notable being SAT based model checking techniques (Bounded model checking). BDD and SAT based model checking have often been complementing each other as they are generally being used to solve different classes of problems. NuSMV processes files written in an extended "SMV" language. "SMV" language is used to describe finite state machines (FSM) by declaring and instantiating modules and processes corresponding to synchronous and asynchronous compositions (the present version does not support directly asynchronous compositions). Specifications are expressed in LTL and CTL notations supported by extended version of the SMV language. The language supports modularized and hierarchical descriptions. More details are presented in appendix C. In the next section, the building blocks of the program analyzer and their implementation details are presented.

5.3 Implementation Details

In this section the details of program analyzer design and implementation is presented. Initially, the details of building blocks of the program analyzer is presented including the actual steps followed in the source code. Then, the nuances of implementation strategies used in each utility is presented. In this process, following are the steps:

1. BLIF generation,
2. Verification using XEVE,
3. Intermediate translation,
4. State searching using NuSMV.

An HipHop.js program of interest is initially translated to BLIF format by a primary translator. The BLIF file generated can then be used with XEVE similar to the way programmers used it with Esterel programs, to verify simple output emissions. With XEVE, simple output emissions using utilities `checkOutput(machine)`, `inputImplies(machine)`, `inputExclusive(machine)`, `sameInstance` can be checked. For anything more complex like the utility `outputRange`, we interface with the

advanced NuSMV tool. Also, the infrastructure which we are building with tools like NuSMV, helps us to provide model checking support on HipHop.js programs. We present the details in later parts of this chapter. The program analyzer proceeds further by converting the BLIF file to AIG format and then to SMV format using AIGER tool set. Also, the specifications according to each utility are generated in LTL, CTL notations and added to the SMV source file. Along with SMV source file of an HipHop.js program, the analyzer also generates a command file that is used with the NuSMV tool. This file is generated to interact with NuSMV shell. The output generated by the NuSMV is processed further and stored in JSON format. This processing is necessary since the result provided as output by the NuSMV tool has a lot of information in terms of many variables which may not be of interest to the programmers. Only, the relevant information with respect to inputs and outputs of a HipHop.js program are presented.

The process followed by HipHop.js program analyzer is illustrated in the following figure 5.1.

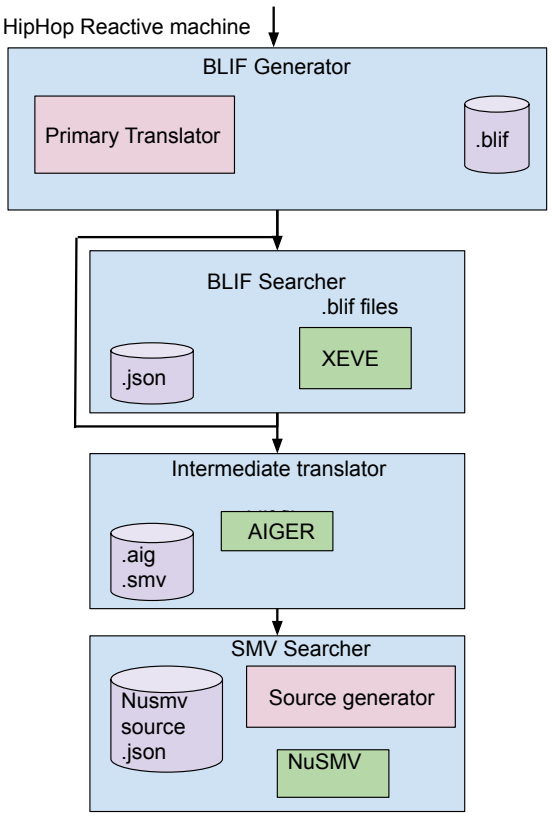


Figure 5.1: General Process Flow.

The above process is elaborated on in the following sections individually. In the next section, details about the primary translator that generates BLIF equivalent of an HipHop.js reactive machine is presented.

5.3.1 BLIF Generation

The primary step is to generate BLIF equivalent of the HipHop.js reactive machine. This is important since, the circuit hence generated is used with XEVE verification tool and also feeds on to subsequent steps. The reactive machine has net lists which is a combination of `AND`, `OR` and `Register` nets. The BLIF generator generates their equivalent BLIF nets based on the definitions for each type of nets as explained in appendix A. The translation process involves in generating three types of BLIF circuits:

1. **primary** circuit,
2. **counter** circuits, and
3. input **relation** circuit.

The **primary** circuit is the one which will have the translations (BLIF circuit) for all the HipHop.js constructs in a HipHop.js source file. In these constructs, the programmers may have used `count` keyword as in `await`, `abort`, etc. HipHop.js is a DSL for JavaScript, explicit circuits for counting is not generated by the HipHop.js compiler in the netlist, rather it is implemented as a plain JavaScript function which counts to `0` from a set initial value. For example, in `count(4, I.now)`, the decrement function will decrement to `0` from an initial value of `4` whenever signal `I` is present in various instants. To simulate this behavior, the program analyzer needs to explicitly generate the **counter** circuit in terms of `AND`, `OR` and `Register` nets. With respect to **relation** circuit, the programmers through the utility may have specified some constraints on the availability of inputs, through relations. If it is so, then BLIF circuits have to be generated which represent the specified constraints on the input signals timing through an input **relation** circuit.

Process of generating BLIF source from HipHop.js is presented in detail here in the form of a series of high level steps, with each step elaborated further with micro steps. The process presented here is an abstracted version of the actual process

carried out by the HipHop.js program analyzer and is based on the naming scheme used in the present version of the compiler as of writing this thesis.

High level steps for BLIF file generation

1. From the HipHop.js source, generate the input, ouput signal list.
2. If signals are declared `inout` (bidirectional), generate the corresponding input, output signal names and include in the respective list.
3. If counters are in the circuit, then generate the data for corresponding counter circuits including the input for the counters and respective counting values.
4. Generate the input and output BLIF source from the lists prepared from the earlier steps 1, 2, 3.
5. If there are counters in the circuit, generate BLIF source of the respective **counter** circuits.
6. Generate the **primary** circuit of the module.
   (a) If `register` net, generate and store the BLIF source for "Register" nets.
   (b) If `AND` net, generate and store the BLIF source for "AND" gates.
   (c) If `OR` net, generate and store the BLIF source for "OR" gates.
7. If inputs have relation, generate and store **relation** circuit in another BLIF source file (`.rel.blif` extension).

The preliminary step is to generate a placeholder for the BLIF source of the primary circuit. Since this BLIF file will be used by other utilities, the source file is created with a unique name (`.blif` extension) borrowed from the file name of HipHop.js program and also the first line (`.model` "name") based on the requirement of BLIF format. The following are the high level steps that are done to generate a complete BLIF source file for an HipHop.js program. The initial three steps are preparatory in nature, wherein the BLIF generator collects all the necessary information in terms of input, output signals, and counter circuits information. The subsequent steps are concerned with the generation of actual BLIF circuits.

The first step is related to generating BLIF source for the input and output signals of the reactive machine, the BLIF generator gathers the input and output signal list. The micro steps are as follows:

Micro steps for Step 1

1. From one of the nets of the HipHop.js machine, get the input signal list from `.input_signal_map`.
2. From one of the nets of the HipHop.js machine, get the output signal list from `.output_signal_map`.

In the gathered list, there may be some signals declared as bidirectional, *i.e.,* `inout`. These signals can participate in both input and output events. The program analyzer differentiates signals in events where a bidirectional signal is participating. For example, if a signal `A` is declared as `inout`, then in the case of input events signal `A` will be marked as `A_IO_I` and in case of signal `A` emission as output, it will be marked as `A_IO_O` to provide meaningful results to the programmers. If the emission of signal `A` (`A_IO_O`) happens at instant 3 and is dependent on itself as input (`A_IO_I`) in instant 1, then the analyzer would like to convey this resulting relationship as follows.

```
A_IO_O = ["A_IO_I","", ""]
```

To facilitate the above illustrated information, the BLIF generator adds the marker `_IO_I` and `_IO_O` to signals identified in both input and output lists. The micro steps are presented here.

Micro steps for Step 2

1. An `inout` list of signals is generated by filtering the common signals from step 1.
2. For each signal in `inout` list, two signals are generated with markers ‘‘`_IO_I`’’ to specify input variant of the signal and ‘‘`_IO_O`’’ to specify the output variant of the signal and are stored in corresponding input and output signal lists of step 3.

The next step is to gather information for creating counter circuits in BLIF format. In the HipHop.js machine net list, there are some specific nets generated for counter circuits for information keeping whenever the keyword `count` is used in HipHop.js source. One of them is to store the initial value for the “step down” counter which is implemented using a plain JavaScript function. The BLIF generator

gathers information from the machine nets to build counter circuits in BLIF. For each counter, nets generated by HipHop.js compiler will have an ‘‘`init_counter`’’ net among other nets. From this net, the BLIF generator fetches ‘‘`countFunc`’’ which will provide the initial value of the “step down” counter. It also gathers the input signal attached to counter nets and the net which gets triggered when the counter counts down to `0`. The micro steps are as follows.

Micro steps for Step 3

1. For each of ‘‘`init_counter`’’ net:
   (a) Gather the values for each of the counter from `.countFunc()`.
   (b) Gather the signal name from `.ast_node.accessor_list[0].signame` - the signal whose number of occurrences should drive the counter circuit (the input signal for the counter).
   (c) Gather the output gate name which will be triggered on the counter reaching the specified number of counts.
   (d) Create a reset input for the counter which when set, will clear the respective counter.

Now, with all the information collected, the BLIF generator starts laying out the BLIF source in the file. First, it generates the BLIF source for all the input signals, and then BLIF source for all the output signals. An example format for generating them is as follows:

```
.inputs sig_a sig_b sig_c
.outputs sig_d sig_e sig_f
```

The micro steps are detailed here.

Micro steps for Step 4

1. In the new line, generate the input signal list line, to the string `.inputs`, concatenate each signal name in the input list, separated by a blank space.
2. In the new line, generate the output signal list line, to the string `.outputs` concatenate each signal name in the output list, separated by a blank space.

Next, the BLIF counter circuits are generated before generating the primary circuit of the main HipHop.js module. Here, there is a small difference with

respect to the way counter circuit works in BLIF when compared to HipHop.js. In HipHop.js, plain JavaScript function will count down to `0`, whereas in BLIF, the counter circuits, count upwards from `0` to the respective counter values gathered in the earlier steps. The detailed steps in generating the BLIF source for counter circuits is presented:

Micro steps for Step 5

1. For each counter circuit, with corresponding counter `initial` value, `signal` name, `reset` name, and the name of the `output` gate from step 5- generate the counter circuit.
   (a) Generate a synchronous counter circuit in BLIF format, using three bit adder as building blocks that counts upwards to the `initial` value:
      i. The initial carry in will be the truth value of the `signal` name.
      ii. The truth value of the final `over flow` will be fed to the `output` gate.
      iii. The reset input value generated earlier will be fed to the `reset` signal name, which when `true` will reset that particular counter.
2. store the generated BLIF source for each counter in the original BLIF source file.

A counter circuit is built with a three bit adder as a building block for each occurrence of the `count` keyword. The three bit adder is generated using `XOR` gates. For the sake of continuity to the reader, we present the BLIF formats for `AND`, `OR` and `Register` nets. For example, a three input `AND` gate, with input names as `A`, `B`, `C` and output name as `andOut`, can be described (`1` for `truth`, `0` for `false` and `-` for `dont-care`) as follows.

```
.names A B C andOut
111 1
```

For the same input, we can describe a three input `OR` gate, with output `orOut` as follows.

```
.names A B C orOut
1-- 1
-1- 1
--1 1
```

An example register net is generated in the following format.

```
.latch I reg_I 0
```

Here `I` is the input to the register, `reg_I` is the name of the register (also the output name), and `0` is the initial value of the register, whose next value will be based upon its input. For an example statement `count(1, I.now)`, the following circuit is generated.

```
.names I decr_counter_34
1 1
```

It is seen that the counter's initial value is set to `1`. The presence of `I` will trigger the gate that sets the truth value of the output of the counter (`decr_counter_34`). For a little more complex example statement `count(2, I.now)`, the following circuit is generated.

```
.names I sum0I sumxorcin0I
01 1
10 1
.names I sumxorcin0I sum0I resetelse0I
11- 1
0-1 1
.names resetI resetelse0I nextsum0I
00 0
.latch nextsum0I sum0I 0
.names I sum0I enablethen0I
11 1
.names I enablethen0I overflowI coutandarg0I
11- 1
0-1 1
.names resetI coutandarg0I nextcout0I
01 1
.latch nextcout0I overflowI 0
.names I overflowI decr_counter_34
11 1
```

In the circuit, whenever `decr_counter_34` is set to true, means the output of the counter circuit is true, and it has successfully counted as many occurrences (specified by initial value) of the associated input signal. We see that the above circuit is also a combination of `OR`, `AND` and `register` nets. Once the BLIF circuit is generated for all the counters, the BLIF source for the primary circuit is generated. The detailed steps for generation of primary circuit is in the next section.

Generation of primary circuit

The primary circuit is generated based on the nets that are available in the net list of the main module. Each net from the net list is read and based on the type of net, the following steps will be followed.

If the net is a register net, then the following procedure is followed:

Micro steps for Step 6a

1. For a register net, generate a BLIF source line:
   - With `.latch` keyword in the beginning of the line.
   - Get the input net name for the register net, and attach it to the string.
   - Attach the name of the register net.
   - Set the initial value to `0` for all the registers, except for `boot_register`, whose initial value is set to 1 initially.

If the net is an `AND` net, then the following procedure is followed:

Micro steps for Step 6b

1. For an `AND` net, generate a BLIF source line:
   (a) Create a temporary string `tempStr1` = `.names`; `.names` is the keyword to begin the line for an `AND` net in BLIF source.
   (b) To the tempStr, attach the list of inputs to the `AND` net separated by blank space which are fetched from the `.fanin_list` of the net.
   (c) Attach to `tempStr1`, the net name and store this `tempStr1` to BLIF source file in a new line.
   (d) Create another empty temporary string - `tempStr2`, to this string, from the fan in list of the `AND` net, get the polarity of each net (`.polarity`) and based on that attach the values - `1` for `true` and `0` for `false` to the `tempStr2`, without any separators.
   (e) To the `tempStr2`, after a blank space attach a value `1`, specifying the output of the present `AND` net.
   (f) Write this `tempStr2` to the BLIF source file.

If the net is an `OR` net, then the following procedure is followed:

Micro steps for Step 6c

1. For an `OR` net,
   (a) Create a temporary string `tempStr1` = `.names`; `.names` is the keyword to begin the line for an `OR` net in BLIF source.
   (b) To the tempStr, attach the list of inputs to the `OR` net separated by blank space which are fetched from the `.fanin_list` of the net.
   (c) Attach to `tempStr1` the net name and store this `tempStr1` to BLIF source file in a new line.
   (d) For each input net, `innet` in the `.fanin_list` list of the `OR` net,
      i. Create another empty temporary string - `tempStr2`,
      ii. Based on the position of the `innet` in the `.fanin_list` - `pos(innet)`, attach (`pos(innet)` - 1) number of "`-`" characters to `tempStr2`.
      iii. Get the polarity of that `innet` from (`.polarity`) and based on that attach the values - `1` for `true` and `0` for `false` to the `tempStr2`, followed by as many remaining nets in the `.fanin_list`. "`-`" specifies "dont care" condition for the other nets in the `.fanin_list`, as we are building multi input `OR` net.
      iv. To the `tempStr2`, after a blank space attach a value of `1`, specifying the output of the present `OR` net.
      v. Write this `tempStr2` to the BLIF source file in the new line.

Next, if the programmer has specified some constraints or relationship on the timing of the input signals, then another BLIF source file with `.rel.blif` extension is generated to be used with the main circuit generated earlier. This relationship circuit will ensure constraints on the way input signals are fed to the main circuit when used with XEVE.

As of now, the program analyzer generates two types of relations on input signals as follows:

- Implicit - the input signals specified under this relationship should be present at the same instant, if they are present.
- Exclusive - the input signals specified under this relationship are to be present exclusive of each other.

The signal list on whom these constraints are to be generated are taken and two versions of the same signals are generated. The first version will be the input

signal, the second version will be the output signal generated based on relationship. For example, if two signal `A` and `B` are input signals with implicit relationship, then in the relation ship circuit, the output signals will be `A` and `B`. The input list will be generated with special markers on signals `A` and `B`. The output signals will be generated based on the relationship filter. The steps are detailed here:

Micro steps for Step 7

1. Create a separate BLIF source file with `.rel.blif` extension.
2. Based on the signal lists which are required to be in a relationship, a new input and output list is created - the output list is similar to the list of input signals which are in a relationship, the input list includes the same input signals which are in a relationship with extra markers to differentiate between input and output versions of the same signal.
3. The initial `.model` line is created with name `relation`.
4. The input and output signal list is used to generate the BLIF source lines as explained earlier.
5. A filter circuit is generated which ensures the generation of the output signals as specified by the relationship that are used by the main circuit as the input signals.
6. If exclusive relationship is specified between inputs:
   (a) Create an exclusive filter based on the working of `exclusive-or` relationship which generates truth value for a particular output signal only when its input version is true and all other input signals are false.
   (b) Generate BLIF source in a new line.
7. If implicit relationship between inputs:
   (a) Create an implicit filter based on the working of `AND` relationship which is true only when all the inputs are high and false if one of the inputs are false.
   (b) Generate BLIF source in a new line.

We illustrate with an example, the format of the relationship file generated with two inputs `A` and `B` in "exclusive" relationship.

```
.model relation
```

```
.inputs A_A B_A
.outputs A B
.names A_A B_A EXCL_0
10 1
01 1
00 1
.names EXCL_0 _REL_FILTER
1 1
.names A_A _REL_FILTER A
11 1
.names B_A _REL_FILTER B
11 1
.end
```

The signals listed in `.outputs` line will be fed to the primary circuit, which ensures constraints on the way input `A` and `B` are input to the primary BLIF circuit when used with XEVE. Next, We provide a complete listing of BLIF generated for a trivial HipHop.js program.

Example BLIF listing

The following is a very simple example HipHop.js program for which the generated BLIF file for the source code and also for the relation specified on inputs is presented.

```
1  hiphop module prg( ) {
2      in A, B, R;
3      out O;
4
5      fork {
6          await (A.now);
7      } par {
8          await (B.now);
9      }
10     emit O();
11 }
12
13 let machine = new hh.ReactiveMachine(prg);
14 analyzer.inputExclusive( machine, ["A", "B" ] );
```

Listing 5.1: Example listing to illustrate BLIF generation.

The following BLIF listing is for the above source file 5.1.

```
1 .model prg
2 .inputs A0 B R
3 .outputs O
4 .latch global_const0_1 global_boot_register_0 1
5 .names global_const0_1
6 0
7 .latch A0 A0_pre_reg_3 0
8 .names A0_pre_reg_3  A0_pre_gate_4
9 1 1
10 .latch B B_pre_reg_7 0
11 .names B_pre_reg_7  B_pre_gate_8
12 1 1
13 .latch O O_pre_reg_15 0
14 .names O_pre_reg_15  O_pre_gate_16
15 1 1
16 .names and_k0_123 and_k0_122  O
17 1- 1
18 -1 1
19 .latch or_to_reg_21 reg_19 0
20 .names reg_19 global_const0_1  and_to_reg_20
21 11 1
22 .names and_to_reg_20 global_boot_register_0 and_to_k0_22  or_to_reg_21
23 1-- 1
24 -1- 1
25 --1 1
26 .names reg_19 and2_negtest_and1_36  and_to_k0_22
27 11 1
28 .names reg_19 global_boot_register_0  and1_sel_res_35
29 10 1
30 .names testexpr_41 and1_sel_res_35  and2_negtest_and1_36
31 01 1
32 .names testexpr_41 and1_sel_res_35  and3_test_and1_37
33 11 1
34 .names A0 and1_sel_res_35  testexpr_41
35 11 1
36 .latch or_to_reg_57 reg_55 0
37 .names reg_55 global_const0_1  and_to_reg_56
38 11 1
39 .names and_to_reg_56 global_boot_register_0 and_to_k0_58  or_to_reg_57
40 1-- 1
41 -1- 1
```

```
42 --1 1
43 .names reg_55 and2_negtest_and1_72  and_to_k0_58
44 11 1
45 .names reg_55 global_boot_register_0  and1_sel_res_71
46 10 1
47 .names testexpr_77 and1_sel_res_71  and2_negtest_and1_72
48 01 1
49 .names testexpr_77 and1_sel_res_71  and3_test_and1_73
50 11 1
51 .names B and1_sel_res_71  testexpr_77
52 11 1
53 .names union_k0_98
54 0
55 .names and3_test_and1_37 and3_test_and1_73  union_k0_99
56 1- 1
57 -1 1
58 .names or_min_k0_child1_110
59 0
60 .names and3_test_and1_73 reg_55  or_min_k0_child1_111
61 1- 1
62 -0 1
63 .names or_min_k0_child0_115
64 0
65 .names and3_test_and1_37 reg_19  or_min_k0_child0_116
66 1- 1
67 -0 1
68 .names union_k0_99 or_min_k0_child1_111 or_min_k0_child0_116  and_k0_122
69 111 1
70 .names union_k0_98 or_min_k0_child1_110 or_min_k0_child0_115  and_k0_123
71 111 1
72 .end
```

Listing 5.2: BLIF listing for the source file 5.1.

The following BLIF listing is for the "exclusive" relation constraint on input signals `A` and `B`.

```
1 .model relation
2 .inputs A0_A0 B_A0
3 .outputs A0 B
4 .names A0_A0 B_A0 EXCL_0
5 10 1
6 01 1
```

```
00 1
.names EXCL_0 _REL_FILTER
1 1
.names A0_A0 _REL_FILTER A0
11 1
.names B_A0 _REL_FILTER B
11 1
.end
```

Listing 5.3: BLIF listing for relation constraint on inputs in listing 5.1.

With this, we come to the end of the implementation details of BLIF generator in the HipHop.js program analyzer. In the next section we present about verifying the generated BLIF source with `XEVE` toolkit.

5.3.2 BLIF Checker - Verification using XEVE

For the utilities, `checkOutput`, `inputImplies`, `inputExclusive`, `sameInstance`, the program analyzer uses XEVE tool to verify output emissions. Once the BLIF circuits are generated, the BLIF checker module uses the `checkblif` utility provided by the XEVE tool kit to check for output emissions as specified by the utilities. The `checkblif` returns results when it encounters the first state a particular output signal is true. The result, if not empty includes the input conditions for which output signal emission happens. If the emission of an output signal is not possible, then the output will be empty.

The output is generated as `.esi` file for each of the output signals. It is then processed by a routine in the BLIF checker module to aggregate all the output signals into a single JSON object and is stored in a JSON file. This can be used by other utilities. The following listing is an example illustration of an typical JSON file which BLIF checker module generates.

```
{
    "O": {
        "1": [],
        "2": [
            "A",
            "B"
        ]
    },
```

```
9         "P": {
10            "1": ["A"],
11            "2": ["B" ]
12        }
13    }
```

Here, `O`, `P` are output signals and `A`, `B` are input signals. It says that `O` is emitted at the `2` instant provided it has `A` and `B` signals present in the `2` instant. Whereas w.r.t `P` signal, though the emission happens at instant `2`, it requires signal `A` in `1` instant and signal `B` in `2` instant. This is not the only condition, but this result conveys one of the way to get the emissions of signals `O` and `P`.

In the next step, the process of converting BLIF source to other intermediate formats to be used by more powerful tools is presented.

5.3.3 SMV Generator and Manipulator - Intermediate Translation

The generated BLIF file, the one which includes main module circuit and the counters circuits is then translated to intermediate format AIG and then to SMV format from AIG format. This is because the tool kit provided by the AIGER requires converting BLIF source to AIG format first, there by also structurally hashing the nets. This translation reduces the number of nets considerably and also converts all the `OR` nets to AIG format. The utility `bliftoaig` converts a BLIF file to AIG format. The following is the ASCII format representations of AIG file for source file 5.1

```
1 aag 24 3 6 1 15
2 2
3 4
4 6
5 8 0 1
6 10 2
7 12 4
8 14 36
9 16 43
10 18 49
11 36
12 20 16 9
```

```
22 20 2
24 18 9
26 24 4
28 27 23
30 27 18
32 31 29
34 23 16
36 35 32
38 20 3
40 38 16
42 41 9
44 24 5
46 44 18
48 47 9
i0 A0
i1 B
i2 R
l0 global_boot_register_0
l1 A0_pre_reg_3
l2 B_pre_reg_7
l3 O_pre_reg_15
l4 reg_19
l5 reg_55
o0 O
c
```

Listing 5.4: AIG (ASCII) listing for the source file 5.1.

The above AIG file is further translated to SMV format. The utility `aigtosmv` converts the AIG file to SMV format. The following is the SMV format of the source file 5.1

```
MODULE main
VAR
--inputs
A0 : boolean;
B : boolean;
R : boolean;
--latches
global_boot_register_0 : boolean;
A0_pre_reg_3 : boolean;
B_pre_reg_7 : boolean;
```

```
11 O_pre_reg_15 : boolean;
12 reg_19 : boolean;
13 reg_55 : boolean;
14 ASSIGN
15 init(global_boot_register_0) := TRUE;
16 next(global_boot_register_0) := FALSE;
17 init(A0_pre_reg_3) := FALSE;
18 next(A0_pre_reg_3) := A0;
19 init(B_pre_reg_7) := FALSE;
20 next(B_pre_reg_7) := B;
21 init(O_pre_reg_15) := FALSE;
22 next(O_pre_reg_15) := a36;
23 init(reg_19) := FALSE;
24 next(reg_19) := !a42;
25 init(reg_55) := FALSE;
26 next(reg_55) := !a48;
27 DEFINE
28 --ands
29 a20 := reg_19 & !global_boot_register_0;
30 a22 := a20 & A0;
31 a24 := reg_55 & !global_boot_register_0;
32 a26 := a24 & B;
33 a28 := !a26 & !a22;
34 a30 := !a26 & reg_55;
35 a32 := !a30 & !a28;
36 a34 := !a22 & reg_19;
37 a36 := !a34 & a32;
38 a38 := a20 & !A0;
39 a40 := a38 & reg_19;
40 a42 := !a40 & !global_boot_register_0;
41 a44 := a24 & !B;
42 a46 := a44 & reg_55;
43 a48 := !a46 & !global_boot_register_0;
44 --outputs
45 o0 := a36;
```

Listing 5.5: SMV listing for the source file 5.1.

These files are used in the later steps. Next, the steps for manipulating the generated SMV file and hence forth generating a source file that can interact with NuSMV model checker is presented.

Source Generator

Once the SMV file is generated, the analyzer manipulates this file to include the search condition programmers want the NuSMV model checker to search for. It generates these search criterions based on the utilities and store them as `LTL` or `CTL` commands at the end of the file. Also, for specifying constraints on the input timing, the analyzer proceeds with a new approach to generate those constraints on the input timing, rather than using the relation BLIF file to generate the constraints on timing of the input signals. The following listing illustrates the generation of LTL specification and also the constraint on inputs (exclusive relationship).

```
1  MODULE main
2  VAR
3  --inputs
4  A0 : boolean;
5  B : boolean;
6  R : boolean;
7  --latches
8  global_boot_register_0 : boolean;
9  A0_pre_reg_3 : boolean;
10 B_pre_reg_7 : boolean;
11 O_pre_reg_15 : boolean;
12 reg_19 : boolean;
13 reg_55 : boolean;
14 ASSIGN
15 init(global_boot_register_0) := TRUE;
16 next(global_boot_register_0) := FALSE;
17 init(A0_pre_reg_3) := FALSE;
18 next(A0_pre_reg_3) := A01;
19 init(B_pre_reg_7) := FALSE;
20 next(B_pre_reg_7) := B1;
21 init(O_pre_reg_15) := FALSE;
22 next(O_pre_reg_15) := a36;
23 init(reg_19) := FALSE;
24 next(reg_19) := !a42;
25 init(reg_55) := FALSE;
26 next(reg_55) := !a48;
27 DEFINE
28 --ands
29 a20 := reg_19 & !global_boot_register_0;
```

```
30 a22 := a20 & A01;
31 a24 := reg_55 & !global_boot_register_0;
32 a26 := a24 & B1;
33 a28 := !a26 & !a22;
34 a30 := !a26 & reg_55;
35 a32 := !a30 & !a28;
36 a34 := !a22 & reg_19;
37 a36 := !a34 & a32;
38 a38 := a20 & !A01;
39 a40 := a38 & reg_19;
40 a42 := !a40 & !global_boot_register_0;
41 a44 := a24 & !B1;
42 a46 := a44 & reg_55;
43 a48 := !a46 & !global_boot_register_0;
44 --outputs
45 o0 := a36;
46
47 --Exclusive relation modification
48 A01 := A0  & !B ;
49 B1 := !A0  & B ;
50
51 --output model checking
52 LTLSPEC G ! (o0)
```

Listing 5.6: SMV listing with added notations, for the source file 5.1.

We can see that in listing 5.6, for example lines 18, 20, 30, are different compared to the listing 5.5, lines 18, 20, 30 of the initially generated SMV file. This process is explained in detail when discussing each of the specific utility implementation. The analyzer also generates a source file which will have commands to be executed by the NuSMV tool. These commands are specific to NuSMV tool and are generated for each utility based on its requirement. As an example, the following listing is provided of source file that has commands to be executed by the NuSMV model checker.

```
1 set on_failure_script_quits
2 read_model -i abro.smv
3 go
4 check_ltlspec -n 0
5 show_traces -o abro_0.xml  -p 4 1
```

```
6 quit
```

Listing 5.7: Source file to interact with NuSMV for an example program.

line 1 specifies that on failure of any command, quit from the NuSMV shell and return the control to analyzer routine. In line 2, it is asking the NuSMV tool to read SMV format file specified. In line 3, it is instructing the NuSMV tool to build model suitable for checking based on the source file specified in SMV format. Once the model is generated, then the model checker is asked to check for states satisfying a condition in line 4. In line 5 the tool is instructed to generate results in XML format and store it in a specific file before quitting from the NuSMV shell.

In the next section, the process of interactively executing source file with NuSMV model checker is presented.

5.3.4 SMV Checker - State Searching using NuSMV

This routine utilizes the generated SMV file and the command file to interact with NuSMV tool. Based on the commands specified in the source file interaction happens. The NuSMV tool generates outputs in XML file which is further processed to provide meaningful result to programmers. The program analyzer provides a JSON file similar to the one which is generated during checking with BLIF using XEVE. This file can be utilized by other utilities.

5.4 Utilities specific implementation

In the previous section we presented the common process to be followed to implement the utilities provided in the HipHop.js program analyzer. Now we provide the details which are specific to implementation of a particular utility.

5.4.1 Implementation of `checkOutput` utility

Programmers use the `checkOutput` analyzer utility when they what to know a possible scenario that yield to emitting some output signals. This is a search based on the goal point and this goal point is used as the criterion to search for states in the SMV model of the reactive machine by the NuSMV model checker. The utility after generating the SMV file through the common process, these search

criterions are added to the SMV source file using `LTL` or `CTL` notations. For example, in a program there are two input signals `A` and `B`, and one output signal `O` as in a module `prgABRO`, the SMV file will have input signals as `A0` (to avoid clashing with `CTL` construct `A`) and `B`, and `o0` (outputs are identified as `o0`, `o1`, etc). The utility generates the CTL formula `CTLSPEC AG !o0` and asks the model checker to check if in all the states of the machine `o0` is false. If the NuSMV model checker returns a result, then it means there is a state where `o0` is true and also in the model checking jargon, whenever a test fails, the model checker is bounded to provide that path which led to the failure of the search criterion, which is called as counter example. This counter example contains all the information including the state of the input signals, which ensured the failure of the required condition. This is stored in an XML file, processed further by utilities to provide a meaningful feedback to programmers.

For each output signal identified in the utility, the utility generates the condition to be searched, and based on the results obtained, the utility presents a meaningful result to the programmer.

5.4.2 Implementation of `inputImplies` and `inputExclusive` utilities

These utilities are concerned with putting constraints on the way inputs should be presented by the debugger. So, for XEVE other than the BLIF file on the HipHop.js source, the utility generates another file which is a relational file specifying relationships between input signals. For example, if there are two inputs `A0` and `B`, and if the programmer wants both the inputs to be present at the same instance (`inputImplies`), then a relation file is generated as follows in BLIF format.

```
.model relation
.inputs A0_A0   B_A0
.outputs A0 B
.names A0_A0 B_A0 IMPL_0
10 0
.names IMPL_0 _REL_FILTER
1 1
.names A0_A0 _REL_FILTER A0
11 1
.names B_A0 _REL_FILTER B
```

```
11 1
.end
```

This file is generated based on the simple observation on the relation that if one input is present in an instant then another input is also present at the same instant. This particular relation file along with BLIF file of HipHop.js program is fed to XEVE, to ensure the inputs are presented at the same instants. For use with NuSMV, the utility does not generate another file, rather it modifies the SMV file generated. The utility creates temporary variables and use them instead of the input variables in SMV source, for the previous example, `A01` and `B1` are created for two input signals `A0` and `B`, and they are defined as follows in the SMV source file.

```
filter := A0  & B  ;
A01:= filter ;
B1 :=  filter ;
```

This does the same job as the relational file for XEVE. As with `checkOutput` utility, the utility generates temporal formula, for example if there is one output: `CTLSPEC AG !o0`, and use it for searching for states satisfying the condition. Similarly, if the programmer wants both the inputs to be present at different instants, then the utility `inputExclusive` produces a relation file describing the above said relation. For the previous example with two inputs `A0` and `B`, the following relational file is generated in BLIF format.

```
.model relation
.inputs A0_A0 B_A0
.outputs A0 B
.names A0_A0 B_A0 EXCL_0
10 1
01 1
00 1
.names EXCL_0 _REL_FILTER
1 1
.names A0_A0 _REL_FILTER A0
11 1
.names B_A0 _REL_FILTER B
11 1
.end
```

Interaction with NuSMV is similar to `inputImplies` utility, the `inputExclusive` utility modifies the SMV file generated as follows. It creates temporary variables

and use them instead of input signals, for example `A01` and `B1` for two inputs `A0` and `B` for the previous example and then provide definitions of those temporary variables as follows.

```
A01 := A0  & !B ;
B1 := !A0  & B ;
```

If there is a single output `o0`, the temporal formula `CTLSPEC AG !o0` is generated as a search criterion with the model checker. The counter example returned is then further processed to provide relevant information to the programmer. Next, the idea of observers and the implementation of `sameInstance utility` is presented.

5.4.3 Implementation of `sameInstance` utility

In program verification for reactive programs, it is generally advised to verify *simple logical* safety properties. *Simple logical* means logical dependence between events rather than between numerical values [68]. In synchronous reactive programming, parallel composition is synchronous and hence the desired verification can be expressed by means of another (second) program which observes the behavior of the first one and decides if it is behaving correctly. The second program is called "observer" and is typically written in the same language as the first one. The verification process consists in checking if the parallel composition of the program and observer makes the observer complain. As an illustration of this, the HipHop.js program analyzer provides `sameInstance` utility.

The idea is illustrated as follows: consider a reactive module `foo`, then another module `obs_foo` is created which observes the output of the module `foo` without influencing the behavior of the module `foo`. For the example listing 4.20 (page 81) in previous chapter, we checked with the program analyzer, if the emissions of two signals `O` and `P` happen at the same instant. What the program analyzer does is it creates synchronous observer module as follows, taking into account the signals which need to be observed for same instant emission. A signal `sameInstant` is used to signal possibility of a scenario where the signals can be emitted at the same instant.

```
1 hiphop module observer(){
2     out O, P, sameInstant;
3
4     loop {
```

```
5         if (O.now && P.now){
6             emit sameInstant();
7         }
8         yield;
9     }
10 }
```

This observer is run in parallel with the original source module (listing 4.20) as given in the following listing. Also, the earlier analyzer utility to check emission of output signals is used:

```
1 hiphop module main() {
2     in A, B;
3     out O, P, sameInstant;
4
5     fork {
6         run prg() {*}
7     } par {
8         run observer() {*}
9     }
10 }
11
12 const machine = new hh.ReactiveMachine (main);
13 analyzer.checkOutput(machine);
```

The ability to build observers, that is programs running beside the original one, is the strength coming from module composition ability of HipHop.js that it inherits from Esterel. Next, we detail the implementation of emissions at a specific range of instants utility.

5.4.4 Implementation of `outputRange` utility

In the listing 4.25 (page 83), we check with the program analyzer if signal `O` can be emitted within a range of [1,4] instants. The implementation is once again straightforward, we follow the common process to generate SMV equivalent and then generate a specific formula that can help the underlying model checker to search for states satisfying specific conditions. In this context the utility generates the CTL formula, `CTLSPEC ABG 1..4 (!o0)` which means, within a bound of 1 to 4 instants, search for all the states where output `o0` is false, here `o0` represents signal

`O`. If the model checker returns a result, it means there is a state where the `oO` is true and that means within the range of 1 to 4 instants, signal `O` can be emitted.

With the infrastructure to implement the HipHop.js program analyzer, we can extend it with few more functionalities that can be utilized by programmers who are experienced in model checking. We introduce one such utility in the following section.

5.4.5 LTL-CTL Formula Checker

The HipHop.js program analyzer provides a utility to experienced programmers who can specify complex LTL, CTL formulas using the utility `checkFormula(arg_list)` and perform traditional model checking on their program supported by NuSMV model checker. The `arg_list` can have any LTL or CTL formulas as devised by the programmers. The utility will check for syntax correctness of the formula forwards the SMV model and temporal formula to the NuSMV model checker and processes the result returned by the model checker. This utility can be used by power users who want to check their programs with advanced properties. The utility initially performs some information processing. In the sense, it modifies input signal names which may clash with `LTL` and `CTL` notations like `G`, `A`, etc. Also, the utility employs a parser to check for syntax correctness before forwarding it to NuSMV model checker. The result returned by the model checker is processed by the utility to provide meaningful information to the programmers. This utility allows experienced programmers to play with their HipHop.js program and model check based on their specifications. Also, a utility which can predict the output emissions for the user given inputs without executing the reactive machine can be built. This utility is based on the basic utilities provided in HipHop.js program analyzer.

5.5 Summary

This chapter provides details of the implementation of the HipHop.js program analyzer, starting with the overview of methodology, brief introduction to the tools used and to the theory on model checking, and in depth presentation of steps that are carried out in implementing the program analyzer. With this we come to

the end of this chapter which detailed the implementation of HipHop.js program analyzer utilities.

Chapter 6

Replayer

One of the objective of the work in this thesis is to study various ways to support HipHop.js programmers with analyzing and understanding temporal behavior of their HipHop.js programs. When building complex systems, collaborative development is a part of software engineering process, programmers should be able to read and understand code written by others usually during integration and testing. As a cognitive task while reading programs written by others, "Program Comprehension" is hard involving building of various mental models of the programs to reconstruct thinking of the original programmer. Program comprehension can become easier with tools visualizing various aspects of the program like its behavior and structure. Program visualization and program animations are seen as important techniques for tools to create visual representations of the program. This allows programmers to relate execution of the program statements with results produced by them, thereby improving understanding of the behavior of programs [111] Continuing with the earlier introduced program analyzer utilities in our effort to support HipHop.js program comprehension, we introduce a utility *RecordPlayer*, with the aim to improve programmers understanding of the programs they are writing and also understand HipHop.js programs written by others. It is introduced in section 6.1 and in section 6.2, the implementation details are presented.

6.1 Introduction

The *RecordPlayer* is a utility which can help programmers visualize the control flow of a program at an instant. The *RecordPlayer* highlights specific programming

constructs where the control flows in an instant. Consider the following HipHop.js program.

```
1 hiphop module prgABC() {
2     in A,B;
3     out C;
4
5     fork {
6         await count(2, A.now);
7     } par {
8         await(B.now);
9     }
10    emit C();
11 }
12
13 let machine = new hh.ReactiveMachine(prgABC);
14 machine.debug_emitted_func = console.log;
```

Listing 6.1: A simple HipHop.js program

How to understand the temporal behavior of this program? One approach is to use the earlier presented utilities which will predict scenarios when the output signals may be emitted. Another approach, typically programmers use is to experiment by checking outputs for specific inputs the programmer has in mind. Let us assume the following set of inputs, the programmer might give as input to check the behavior in each instant. We observe the emission of signal `C` in instant 5.

```
1 machine.react();     // output:
2 machine.react("A"); // output:
3 machine.react("B"); // output:
4 machine.react();     // output:
5 machine.react("A"); // output: C
```

Listing 6.2: HipHop.js program reation instants.

Why the emission of output signal happened only in instant 5 for this input sequence? What happens to control flow in each of the earlier instants? These may be questions that may linger in programmers who are new to reactive programming. We want to help HipHop.js programmers answering these questions by providing visualization of control flow in a HipHop.js module during a reaction instant, so that they can build relationship mental model between flow of control and output emissions.

Using *RecordPlayer*, programmers can record the status of the machine in each instant for the specified inputs, and can replay the flow of control. In our example they can replay the flow of control in each of the 5 instants. The snapshots of each of the instants are illustrated here. First, the snapshots of control flow in instant 1 and 2 are presented in figure 6.1

```
1   hiphop module prgABC() {
2       in A,B;
3       out C;
4
5       fork {
6           await count(2, A.now);
7       } par {
8           await(B.now);
9       }
10      emit C();
11  }
```

(a) Instant 1 - `machine.react()`

```
1   hiphop module prgABC() {
2       in A,B;
3       out C;
4
5       fork {
6           await count(2, A.now);
7       } par {
8           await(B.now);
9       }
10      emit C();
11  }
```

(b) Instant 2 - `machine.react("A")`

Figure 6.1: Control flow visualization for an example program.

In instant 1, the control is present within "`fork {} par {}`" constructs and does not go further to line 10 (hence no emission in instant 1). This tells the programmer that control is active at those constructs through which they are waiting for respective signals starting from the next instant. In instant 2, input `A` is present as input, hence it is highlighted in line 2, and the control still stays within "`fork {} par {}`" construct, hence no emission of `C`. The visualization tells

the programmer that control is active in those constructs since they are waiting for the occurrences of signals `A` and `B`. Though, only signal `A` is present, the control continues to stay active in the "`fork {} par {}`" construct. Now, in the following illustration 6.2 the snapshots of flow of control in instant 3 and instant 4 are presented.

```
1  hiphop module prgABC() {
2      in A,B;
3      out C;
4
5      fork {
6          await count(2, A.now);
7      } par {
8          await(B.now);
9      }
10     emit C();
11 }
```

(a) Instant 3 - `machine.react("B")`

```
1  hiphop module prgABC() {
2      in A,B;
3      out C;
4
5      fork {
6          await count(2, A.now);
7      } par {
8          await(B.now);
9      }
10     emit C();
11 }
```

(b) Instant 4 - `machine.react()`

Figure 6.2: Control flow visualization for an example program.

In instant 3, input `B` is present, hence it is highlighted in line 1, and the control flow visualization says that the control stays with "`fork {} par {}`" construct. Inside "`fork {}`" waiting for another occurrence of signal `A`. Inside "`par {}`" as it is waiting for occurrence of signal B. Next, in instant 4 there are no inputs, and the fork construct has active control waiting for another occurrence of signal `A`; inside the par construct there is no active control, indicating that control is not

active inside "`par {}`" construct. Now for control to be active at signal emission of `C`, there should be no active control inside "`fork {}`" construct. This is seen happening in instant 5 which is illustrated in the following illustration 6.4.

```
1  hiphop module prgABC() {
2      in A,B;
3      out C;
4
5      fork {
6          await count(2, A.now);
7      } par {
8          await(B.now);
9      }
10     emit C();
11 }
```

Figure 6.3: Instant 5 - `machine.react("A")`

Figure 6.4: Control flow visualization for an example program

In instant 4, the active control flow was active inside "`fork {}`" construct waiting for presence of signal `A` second time (since counter value is set to 2). In the instant 5, signal `A` is present as input and hence the control is active outside of "`fork {} par {}`" construct at the "`emit C()`" statement, thereby emitting it and hence it is highlighted at line 3 and 10.

With this sort of visual support for control flow in HipHop.js reactive programs, we believe it will meet the requirement of the program comprehension tools as suggested in the related literature. Next we present the implementation and usage details of the *RecordPlayer*.

6.2 Implementation and Usage of the Replayer

The usage of the *RecordPlayer* utility is presented first, and then the implementation details are presented. The API `reactRecord(reactiveMachine, sigArray_in)` instantiates the execution of the *RecordPlayer*. The argument list includes the reactive machine `reactiveMachine` and a list of input signals `sigArray_in`.

```
1 hiphop module prgABC() {
2     in A,B;
3     out C;
```

```
4
5       fork {
6           await count(2, A.now);
7       } par {
8           await(B.now);
9       }
10      emit C();
11  }
12
13  let machine = new hh.ReactiveMachine(prgABC);
14  machine.debug_emitted_func = console.log;
15  rec.reactRecord(machine, ["","A","B","","A",""]);
```

Listing 6.3: HipHop.js "RecordPlayer" utility usage.

The visualization support that is provided by the *RecordPlayer* is based on syntax highlighting. Syntax highlighting has been shown to improve Program Comprehension [122]. The utility captures the state of each reaction instant and then replays them inside the textual editor. For now, our implementation relies on VSCode extensions, and only that editor is supported. This can be extended to all the popular editors and also can be made web based. The building blocks of the utility is illustrated in the following figure 6.5.

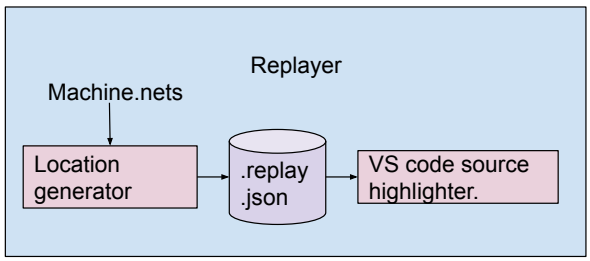


Figure 6.5: Replayer Building blocks.

For each reaction instant, the state of the nets are recorded. Specifically, the nets whose value are true. Once that is done, the *location generator* generates the locations from those nets using the location information of nets provided by the HipHop.js compiler. The *location generator*, further process the information from the source file and to generate all the relevant positions and stores it in a JSON file, with `.replay.json` extension as illustrated follows in an example file.

```
{
    "reaction1":{"filename":"forkpar.js",
    "locations":[152,213,198,256,242,219],
    "interface":{"A":[[152,false]],"B":[[152,false]],"C":[[152,false]]},
    "localSignals":{}}
}
```

Listing 6.4: An example JSON file generated and used by "RecordPlayer"

Further processing is required because, in the HipHop.js compiler the input and output signals are instantiated at the beginning of the module. To visually highlight the signals when they are emitted at the positions where they are declared the *location generator* routine performs text processing to get the actual position where the signals are declared. So to highlight the position whenever there is an input or output of a signal, the *RecordPlayer* checks for those signals which are high at a particular instant, locates their position in the text file. Those positions are stored in the JSON file. The VSCode extension animates the execution. The routine reads the locations, and the number of reaction instants in the JSON file and based on the location info, highlights those positions with a specific color. All the positions are highlighted and displayed. If there are more than one instant, then it is animated with a specific amount of time delay between each reaction instants.

The *RecordPlayer* aims to provide visual support for HipHop.js program comprehension. Since, the mental grasp of humans are limited, for a better understanding of module behavior it is not prudent to record large number of reactions before playing. Also, it is advised to run the *RecordPlayer* routine on programs which are of size that can be visually grasped. Visualization where control flow spans hundreds of lines of code may not be very advantageous when building mental models for program comprehension. Large programs can be modularized and then used with visualization tools. With this, we end this chapters devoted to the presentation of the tools helping programmers understand the behaviors of their HipHop programs. The next chapter presents the related work.

Chapter 7

Related Work

This chapter presents literature review which is closely related to our work. Software professionals spend a large amount of time and energy testing software to ascertain whether it is behaving the right way, and to find and fix bugs (debug). Software testing as a technique used in Verification and Validation of software, ensures adherence to the desired specifications, thereby increasing confidence in the software. Debugging software has remained an art [66] for a long time. Software engineering community and scientists continue to innovate new practices and methodologies to standardize the practice of software testing and debugging, in pursuit of minimizing the development cost and time.

Programmers have generally observed that they experience lengthy and difficulty testing and correction process which is attributed to ever-increasing complexity of software [66] being developed. The important challenge for preventing software failures is detection and localization of the required information from the source code [53]. A major requirement in testing is developing a set of test cases that are effective enough to cover all of the source code, and finding inputs that leads to trigger subtle corner case bugs. Programmers are required to have understanding of the fault chains in smaller software parts, as the cause of failure are typically distributed throughout the software and also due to the dynamic context in which error occurs. Furthermore, advances in programming technologies and varied programming paradigms brings in a requirement of specialized domain specific knowledge, making testing and debugging even more difficult. Software fault localization [85], program comprehension [134, 130] have opened up new frontiers of research and practices in software debugging. Likewise, automated

testing techniques capable of performing testing activities, including test data generation without manual intervention are being increasingly used to build reliable software [55, 7]. Our work is closely related to techniques used in automated testing techniques, fault localization and behavior comprehension in the bigger realm of software testing and debugging.

Automated testing techniques like random testing [8], differential testing [64], Concolic testing [140], etc. are matured enough to be used in building medium to very large reliable software systems. Though traditional fault localization techniques are in practice even today, like program logging [56], assertions [119, 120] and breakpoints [133, 46, 71, 1], newer strategies have been employed in coming up with new strategies for fault localization. The research in the field of program comprehension as part of debugging activity has seen various shifts in paradigms in the last few decades contributing to rich and varied literature. In the following sections, we review some of the automated testing techniques, approaches in "software fault localization" and "program comprehension".

A brief history of testing and some approaches in automated test data generation techniques are presented in section 7.1. We review some of the techniques that have evolved in software fault localization in section 7.2 and then on program comprehension methodologies used for debugging in section 7.3. Later, in section 7.4 we present debugging approaches carried out in other reactive languages. First, we present about synchronous reactive languages Esterel, Lustre, and then we present the debugging strategies employed in other reactive programming paradigms and libraries like Reactive Scala, RX, etc. Finally, in section 7.5, we present a review of literature on a new framework for verification of programs, especially for programming languages like HipHop.js which provides facilities for composition of synchronous and asynchronous computations.

7.1 Software Testing

Testing is generally used to detect presence of faults in the software but does not guarantee absence of bugs. The testing process has been an integral part of the Software life cycle, starting from the software requirement phase and continuing throughout the lifetime of the software.

As a practice, a test plan is developed defining the scope, goal, methods,

resources and timeline of testing. The test plan also has details explaining who, when, why and how testing should be performed. Managers make the choices on forms of testing like black box testing, white box testing and gray box testing. As in other branches of software engineering, newer technologies and methods are being developed in the domain of testing thereby continuously increasing the confidence on the reliability of the software. In a simplified testing process, test input data is selected from the "input" domain and is used to generate test cases. The test case generation may be manual, or automated. The test cases hence generated are executed on the software which is under testing. The output obtained is then declared as pass or fail based on the "expected" result defined by the specification of the software. The "input" domain includes all possible inputs for the software. Test data generation is the process of identifying and selecting input data that satisfies the given objective of testing. Various aspects of the software are considered to generate the test data, like requirements, model, code, etc. The objective of testing dictates the type of test data generators to be used, for example, there are test data generators to check the coverage of all the control paths in the software of interest.

Random testing is a technique, wherein the test data is randomly generated (automated) based on the test objectives or specifications of the software. This approach is seen to be very effective in discovering bugs and also found to be economical - requiring less intellectual and computational efforts [43]. The simplicity and cost-effectiveness makes random testing preferable in running large numbers of test cases, as opposed to other techniques that are seen to require considerable time and resources for generation of test cases and execution. It has been observed that random testing is as effective as other methods in detecting subtle faults due to large test cases employed in testing [54]. Since the test data generation happens without any background information (thereby sometimes violating pre and post conditions of the software), Random testing is criticized. Further, many of the generated test cases fail at the same state of software which is also seen as a drawback of the approach.

Differential testing, a form of random testing requires availability of two or more comparable systems to the tester. An exhaustive series of automatically generated test cases are presented to the above systems. If there are differences in the results, or one of the systems indefinitely loops, then the tester has a candidate

for debugging. To have effective differential testing, the quality of testing is a very important issue, programmers must evolve test cases that drive deep into the software getting tested. The problem of **false positives** - though the results of two tested programs differ, they are found to be correct, is another problem that has to be addressed, further, the amount of noise for generating test cases, in differential testing is huge.

Symbolic execution provides a better approach to the problem, by exploring many possible execution paths systematically at the same time without requiring concrete inputs, rather by using symbols, thereby reducing noise considerably. Execution by a symbolic execution engine [13] is by building a first-order Boolean formula using symbols for all the explored control paths, along with a symbol memory storing the variables - symbols mapping. Using SMT based model checker, violations of any required property along each explored path is verified.

In concrete executions, the program is run on specific inputs and a single control flow path is explored. In contrast, symbolic execution explores simultaneously multiple paths, a program can take under different inputs. Modeling all possible runs provides very interesting analysis, but is unfeasible on real-world software. The problem is of scaling up, due to memory constraints and state explosion problem arising out of large number of variables. To mitigate the above scenario, dynamic symbolic execution strategies like Concolic testing mixes both concrete and symbolic execution strategies as a middle path between exploring too less or too many paths.

The programs in Concolic testing, are initially executed with some concrete input. Based on that input, the program execution branches into different paths at branching statements. The Concolic tester in parallel constructs a "symbolic constraint" which describes the possible input values that can cause the program to take either the true or false branch at a statement. The conjunction of the above "symbolic constraints" is the "path constraint" and by solving them using SMT solvers, test inputs can be generated for subsequent runs. Though Concolic testing is very promising, there are some issues which is preventing easy adaptation like expensive constraint solving, issues with symbolic representations, handling of multithreaded programs, scalability issues, path explosion, and ways to handling native calls or system calls [78]. Newer techniques are being discovered in the pursuit of truly automated testing. We end the review of testing strategies here.

From the perspective of testing, the HipHop.js program analyzer provides a semi-automated testing infrastructure. The input can be generated without any manual intervention or the programmers can set constraints on the way input should be generated in the form of input relationships. Also, the flexibility provided by the program analyzer to test manually by the programmers for a few instants and then checking for output emissions through program analyzer, helps in exploring lots of various temporal behaviors. But, as highlighted in drawbacks of the many testing strategies, the HipHop.js program analyzer cannot guarantee absence of bugs and also provide explorations of all the paths. The goal based searching as in HipHop.js program analyzer provides an efficient trade off with respect to exploring too less or too many possible control paths. The programmers can verify whether their expected goal is reached or not with the help of program analyzer, improving their confidence on the software. With this we end this section. From the next section we review the literature on debugging.

7.2 Software Fault Localization

Software fault localization is one among expensive activities in software development life cycle. In practice, programmers tend to localize fault manually; they observe test cases which fail and search faults in the source code. The typical debugging approach includes inserting logging statements and breakpoints, inspecting the stack trace, etc. All these ad-hoc manual processes are generally costlier and time-consuming. Recent advances in fault localization tends to automate, standardize the techniques and have been found to be relatively cost-effective in software development [48]. The advanced fault localization techniques can be categorized in many ways and in one of the groupings based on the approaches to isolate the faults, we present some of these techniques [145] and then correlate with fault localization approaches used in our work. One of the basic fault localization technique used is slicing.

7.2.1 Slice-based Localization

Programs are abstracted into a reduced form by slicing off the parts which do not affect the behavior of the program with respect to certain specifications [137, 28].

Slicing reduces the searching domain for the programmers to locate the bugs in their programs. There are various slicing techniques like static slicing [142, 87], dynamic slicing [5, 4, 141] and execution slicing [6]. Although slicing reduces the amount of code to be inspected, the code that still remains is excessive in actual production environment [48] prompting further refinements and combination with other localization techniques. The next technique is based on program states.

7.2.2 Program State-based Localization

The variables in a program provide the snapshot of states of a program through their values, during program execution. Faults in the "development" version are located by a comparison at the runtime of the internal states to a "reference" version of the program which is also called as relative debugging [2]. Further, the values of some variables can be modified to narrow down on the ones which causes erroneous program execution. The delta debugging [151, 152] contrasts the program states between successful test and failed test executions using their memory graphs. This technique can be used in automated debugging with various other improvised approaches on delta debugging. The flip side of this technique has been state space explosion as and when the number of program variables increase. The next approach is based on modeling.

7.2.3 Model-based Localization

This method has been initially used in electronic digital circuits and later incorporated into software domain. In model based localization [147, 148], a logical model is generated from the source code, and then by using logical reasoning, a minimal set of statements explaining the existing faults are obtained. Modeling can be:

- dependency based - dependencies between statements,
- abstraction based - using abstract interpretation of loops, recursive procedures and heap data structures,
- value based - dataflow information in programs are used to locate components that may contain bugs.

Value based modeling is seen to be costlier and generally applied to smaller programs [102], whereas for bigger programs, dependency and abstraction based modeling is preferred. This method has been augmented with the use of model checkers. Here, a model checker is used to show (localize faults) how specifications are violated with the help of counter-examples whenever a model of the software does not satisfy the corresponding program specifications. A counter example may not help directly in identifying the parts of a model that are associated with a given bug, but, it can be viewed as a failed test case to identify reasons for the bug [61, 62, 84]. Refining further in [14], a model checker is used again to explore all the paths in the program except those paths of counter examples. The paths which do not cause failures are recorded. From these, transitions to counter example are identified and all these are the ones contributing to bug. Since all the paths except the counter example one can be numerous, it is generally cost intensive and is further refined. In [63], it is proposed to generate less number of executions by going backwards from the counterexample using a model checker. These executions which may or may not cause a failure are further analyzed to localize the source of bugs.

The HipHop.js program analyzer uses the model based localization strategy along with a model checker in helping programmers localize the fault based on the results returned. This technique has been used quite vividly. In [9] a dynamic test generation technique is presented which combines both symbolic execution and explicit state model checking. The test cases are generated with constraints on inputs and those tests which fail are documented and a bug report is developed based on that. In [106] using model checking, a testing infrastructure which can give feedback when dealing with design of distributed systems is presented. In [83], symbolic execution is used for fault localization in imperative programs. The classical literature on modeling software identifies the short coming in this approach. A wrong modeling process may miss out some of the details and may give a false result for fault localization, or may include too many details and contribute to state explosion problem. The spectrum based localization techniques presented next can be used alongside model based localization further refining or independently in software fault localization.

7.2.4 Spectrum-based Localization

This method utilizes the information of program entities executed by test cases to localize entities that are more likely to be faulty. Code coverage, testing data, dynamic information, execution trace, execution path, path profile, and execution profile are some of the similar terms that are used [48]. Various techniques have been in practice like the ones based on failed test cases, successful test cases, combination of failed and successful test cases, coefficient based, Program Invariants Hit Spectrum (PIHS) based, Predicate Count Spectrum (PRCS) based, etc [3]. We will not elaborate on the techniques here, curious readers can refer any of the cited literature for further information.

Even though there are so many techniques, none of the techniques has been able to outperform all others in every possible scenarios [149]. The usage of a particular technique depends on the context and application in consideration. It has been observed that large proportion of bugs cannot be directly detected with the above techniques, requiring inputs from multiple failed test cases and inspection of large number of lines of codes [80]. The next category exploits advancements in the field of AI like machine learning and data mining techniques.

7.2.5 Machine Learning and Data Mining based Localization

Artificial Intelligence techniques like machine learning and data mining are being applied for localization of faults, with program spectrum data as input for classifying faulting program elements. Machine learning techniques can produce models based on data. With fault localization problem, models can be produced which learn and can deduce the location of a fault, based on training input data of past errors. Various approaches from machine learning are used in fault localization, like back propagation neural network learner [146, 10], Radial basis Function neural networks [144], and decision tree algorithm [36]. Although, a program's complete execution trace is a valuable resource for fault localization, the high volume of data makes it humanly difficult for usage in practice. Therefore, some approaches have applied data mining techniques on execution traces [40, 108, 41, 150], one of them being graph mining. The AI application to software fault localization brings in both the advantages and disadvantages of respective methods to fault localization domain and may require special training to use in debugging activity which may

drive away programmers.

Other than the above localization techniques, there are other techniques which we briefly review as part of completeness of this section. Algorithmic debugging technique (declarative debugging) as part of fault localization decomposes a complex computation into a sequential sub-computations thereby helping in localization of program bugs [127, 37]. Each sub-computation's outcome is checked for its correctness with respect to given input values. Then, on those sub-computations that has faults, the algorithmic debugger is used again to further narrow down the fault. In formula based fault localization techniques [76, 77] failed execution traces are encoded into a "error trace formula". The error trace formula is then proved for non-satisfiability using certain tools or algorithms like SAT/SMT solvers. By doing so, the programmers can capture the relevant statements causing the failure.

In the context of our work, the work on causality error tracing is also a part of software fault localization technique, wherein a bigger causality cycle is chosen and searched for relatively smaller cycles by slicing off parts of bigger cycle using advanced graph theory algorithms. With respect to HipHop.js program analyzer, a combination of techniques presented above like program state based and model based localization along with SAT/SMT solvers are utilized to give better results in fault localization to HipHop.js programmers. In the next section, we present the related work on another aspect of HipHop.js program analyzer, supporting HipHop.js program comprehension.

7.3 Program Comprehension

Supporting understanding of programs has been a major topic in software engineering research, initially as part of software maintenance and now in a supporting role for all the software engineering activities, including debugging. Code and program behavior understanding as part of program comprehension [124] forms the cornerstone of research in program comprehension. Programmers build *mental* model of a program with the help of information structures of the program. The mental model of a program is the programmer's mental representation of their program. The information of the program to build the mental model can be gathered by following various approaches, it is observed that programmers try to understand their programs through collecting runtime information by executing the programs

using a debugger quite frequently [98]. Programmers build a meta-model of the program [94] which is a combination of various models to aid in cognition process. According to [134], the tools that are developed to aid in comprehension should support browsing, searching (querying), present multiple views in the form of visualization (graphical views). We concentrate on querying, software visualization and dynamic analysis techniques utilized in tools for program comprehension which are related to our work in HipHop.js program analyzer.

7.3.1 Query-based Program Comprehension Tool

Query-based debugging tools have been designed to aid program comprehension. A debugging tool with support for program comprehension [82] enables programmers to select from a set of standardized "why did" and "why did not" questions deduced from the source code. The tool uses call graphs and a mix of both static and dynamic slicing (fault localization techniques) to find possible explanation to failures. To understand program behavior, programmers can translate their query into code-related questions, which can speculate about the causes of faults. In [74] time traveling queries are proposed which can be used to explore programs as part of program comprehension. These queries collect execution data and present it to programmers. Further, in the same paper based on a survey on programmers, it is concluded that the queries help in better comprehension of programs. Efficient searches are made in large object spaces and their relationships whenever a program stops as part of query based debugging tools [92, 88, 118]. A specialized query language is suggested in "snap shot" query based debugging [115] with its own user-friendly syntax.

Are the queries provided sufficient to cover all the behavioral patterns of a program? This is something that cannot be answered directly and also scientifically. Probably a survey with programmers can help in noting down the efficiency of individual queries, thereby trimming queries or adding up after starting from a basic set of queries. HipHop.js program analyzer also provides a basic set of standardized queries in the form of utilities that can be used in understanding the temporal behavior of the programs. We believe these set of queries can be the starting set and can be added upon based on the feedback from programmers, which can be collected by a survey. Next, we present related work on software visualization.

7.3.2 Software Visualization

Software visualization improves program comprehension, thereby making debugging of programs easier [12]. The visualization support provided can be based on structure, behavior and evolution of software [52]. The visualization of a software structure helps in analyzing the structure of the source code and the differences in their organization based on the domain of usage. The visualization of behavior helps in understanding complex programs and its performance, whereas the visualization on evolution helps in understanding the versioning of software. As one of the guideline [89] for developing visualizing tools in software, it is advised to have tools which can visualize steps of a function, thereby helping in improving program understanding. Execution traces have been used to present the behavior of programs to help in program comprehension and debugging. There are various methods of presenting execution traces like call-graph, scenario diagrams, sequence diagrams etc.

In [96], the *Enlighten* approach uses fault localization techniques along with queries on suspicious invocations which are expressed in terms of inputs and outputs as part of visualization support. In [51], a visual instrumentation platform for reactive programs is presented, wherein users can dynamically wire into a running reactive program and get to know the flow of values. In [122, 11], the visualization support in program comprehension is provided using syntax highlighting. It has been observed that rich code visualization helps in understanding the code features without overloading visually. In [139], a tool based on *Topic* models for program comprehension in JAVA domain is proposed. This tool mines information from Java Source code and presents a project overview to the programmers to understand the program. Without actually running the program, all possible execution paths are displayed in visual symbolic debugger [65] helping in understanding smaller parts of the code.

Though there are many methods, some common drawbacks as in visualization tools as of now are with respect to scalability with big programs and tools which cannot be generalized for all the scenarios and programming domains. It has also been observed that the visualization tools usage is somewhat limited, reasons being visualization tools not part of IDEs and ease of adaptation of the tools. HipHop.js program analyzer's utility not only helps in building mental model of the

language constructs of HipHop.js, it also helps in understanding complex behaviors by providing syntax highlighting of presence of control in a particular reaction instant. The highlighting of respective inputs and outputs also helps programmers note input-output relationships. Next we present about dynamic analysis, which lately has been an important approach to program comprehension.

7.3.3 Dynamic Analysis

Dynamic analysis is used in understanding the behavior and properties of a running program. This is fundamental to debugging activities as it provides programmers a better perspective on the corrupted program behavior. By augmenting the code with logging code, run time information is collected, this is quite famous and one of the oldest approaches. These approaches run the risk of slowing the code. Recent approaches like efficient path profiling, encoding program executions, dynamic instrumentation have been seen to optimize the runtime overhead. This is done by compressing the trace data, or considering only subset of data, and dynamically inserting and removing logging instructions by a smart observer without manual intervention. Further, programming languages which support meta programming features offers some more features like a middle layer which performs some task on behalf - "Behavioral reflection" [50] and "Spy framework" [19] provides an abstracted middle layer hiding many of the actual implementation details that can be worked on. These metaobjects implement a transparent tracing mechanism that captures run time behavior. Also, aspect oriented programming tends to extend the support of metaobject to programming languages which do not have meta features. Here tracing mechanism is chosen based on the level of details by the programmers. All the above approaches typically have performance issues.

As part of dynamic analysis, time travel debugging is proposed. Wherein programmers can step front and step back through their debuggers. Many tools have been proposed. In [95] backward debugging is proposed - here state changes are recorded through the run of a program and is presented back to the programmers. The issue with time travel debugging is the amount of memory that is required to record the executions. Various tools that optimize the memory usage and a considerable performance are in practice. TARDIS [16], JARDIS [17] and MCFLY [138] are some implementations of time travel debugging for languages like

Java and JavaScript. In HipHop.js program analyzer, programmers can execute for a few reaction instants and then use the utilities to analyze the temporal behavior from then on. The results speculate possible input output relations from any reaction instant. In the next section, we present the related work more specific to debugging approaches in other reactive languages.

7.4 Debugging in other Reactive Languages and Libraries

HipHop.js program analyzer has been in parts inspired by the verification and debugging approach taken in Esterel. XEVE [30] is one of the tool that is used for verifying and understanding Esterel programs in terms of input - output relations. The Esterel compiler compiles programs into Finite state machines, where states and transitions are enumerated into boolean circuits with equations and latches. These equations are generated in BLIF format. XEVE is built on a Binary Decision Diagram (BDD) library called `TIGER`, which takes FSMs described in BLIF as input and provides two functionality to the programmers: minimization of the number of states in FSM, and checking the emission status of selected output signals. The minimization is based on symbolic bisimulation (states that are not distinguishable are equated while walking the FSM graph). The output signal status checking can be used to check if there is a possibility of emission of signals. Users can decide the status of the input signals, and based on that they can verify the emission of output signals. If emission is possible, then the path will be available in a file saved with `.esi` extension. The functions which perform minimisation (`BLIFFC2`) and output status checking (`CHECKBLIF`) are available as binaries that can be used in other applications. The output status checking can be used along with "synchronous observers" - these are Esterel programs, that are designed to catch misbehavior and property violations from the program being verified.

The HipHop.js program analyzer is inspired by the idea of checking output emissions and "synchronous observers" as used in Esterel. Program analyzer provides an interface for HipHop.js programs to be used with XEVE. Further, the idea of checking output emissions and observers are also implemented with backend model checker using SAT formulas. HipHop.js program analyzer provides few more

standard queries other than checking outputs to understand temporal behavior of HipHop.js programs.

Lustre is another synchronous reactive language. Ludic [100] the debugger designed for the Lustre reactive programming language, combines the idea of imperative language debuggers with algorithmic debugging technique. Step by step execution with breakpoints is implemented with the idea of observers, which observes module, taking into account the inputs and outputs of the original module. This approach lends traditional debugging approach to Lustre due to incorporating stepper. Though HipHop.js program analyzer does not support a stepper as of now, it provides standardized query based searching to help programmers understand and debug their HipHop.js programs. Building on Ludic, in [59], another tool for debugging Lustre is proposed. This debugger takes the approach of finding out the inputs for a given state to be reached, in contrast to the approach taken by typical debuggers as to reach a particular state for a given set of inputs which is similar to verification of safety properties. The HipHop.js program analyzer also follows the approach of reaching a possible goal state as defined by the programmers through queries and then identifying the input conditions responsible for reaching the goal states.

There are many other reactive programming paradigms other than the synchronous reactive approach that are in practice as of today mainly used in game development and design of reactive GUIs. Functional reactive programming (FRP) combines functional and reactive programming paradigms, specifically time flow and compositional events are integrated into functional programming [75]. Languages like Haskell and Scala have various FRP libraries, the ReactiveX (Reactive Extensions) library initially for Microsoft platforms has been extended to many of the languages in the form of RxJava for Java, RxScala for Scala, RxJS for JavaScript and so on. In [121], about designing debuggers for reactive programs written in Reactive Scala, it is suggested to use "dependency graph" of values as runtime model in place of "execution stack" as in traditional programming. The programmers can step through the construction of the graph, seeing the creation of nodes and new dependencies among reactive values as soon as it is established. Since understanding reactive programs is complex, providing visualization tools of these sort will be more helpful to programmers as they can build mental model of the program behavior. Also, for Reactive Scala in [107], another debugger is

designed based on data flow graphs, where the graph nodes represent the input variables and transitions from one value to another represent the edges of the graph. Here, time traveling is supported by playing with the graph. Programmers can go to a specific node and assign a new value at the node to check the data flow. In HipHop.js program analyzer, the dependency between input and output is exploited to provide tools for understanding temporal behavior. The programmers can also play with program analyzer to understand if an output emission can happen at a particular time instant and if so what are its dependencies on input signals. The code replayer provides visualization support which will help programmers build mental models of the program.

LINGUA FRANCA [49] is a language designed for coordination and composition among reactive components built in various languages (C, C++, Python, TypeScript, Esterel). The components can be concurrent, time sensitive and distributed. As part of debugging support, an interactive debugger and a trace debugger is proposed through a graphical IDE different from the way HipHop.js program analyzer provides interfacing with its utilities. In the interactive debugger which is more related to work on HipHop.js program analyzer, the runtime state of the model can be viewed as a graphical representation. Users can time travel on the graphical representation and explore various possible execution paths. The verification of temporal properties can be done by specifying constraints as assertions checked during debugging sessions, or for finite state space programs, similar to HipHop.js program analyzer, model checking support is provided. The difference in approach with respect to HipHop.js program analyzer is in the usage of CADP [58] model checker in place of NuSMV and Model Checking Language (MCL), an enhanced version of modal mu-calculus used for specifying verification properties, in place of LTL, CTL.

The paper[73] is about a debugger for multi-tier version of ELM programming language. Along with time traveling, it also provides interactive stepper with graphical interface. Selecting on the events or on the graphical timeline can step back to a previous point in the timeline. The paper [143] presents dynamic analysis and visualization of reactive program behavior. The premise is that it will help programmers to observe any unwanted or unusual behavior in the program and help in locating the defects in the code. The sequence of reactions of the program are analyzed to build high-level models, and are mined for recurring patterns of

execution, to classify the patterns into similar and diverging behavior, and to identify unusual behavior. The above approach uses a combination of some techniques earlier discussed on software fault localization. In [72], to make debugging easier and supportive to programmers in reactive functional programming, the authors vouch for the visualization aspect of debugging, which will help in probing the state, visualize relationships between input and outputs, and inspect transitions.

Building software tools for languages is getting automated to some degree. Debuggers are being developed in a semi automated way using generic patterns and metalanguages. The generic approach may not be quite productive when understanding states of the program written in DSLs during debugging sessions. To help build better debugging environment for DSLs, the ongoing work [79] proposes a metalanguage which can help language designers specify runtime state of programs, in the best possible way suiting the DSL.

7.5 Another Framework for Verification

Some challenges to property verification are the idea of concurrent executions with preemption, combination of synchrony and preemption, and the combination of synchronous constructs with asynchronous constructs as in HipHop.js (*async*). Modern temporal verification techniques using traditional model checking are yet to handle the combined model of *synchronous* and *asynchronous* computations effectively. The temporal logics LTL and CTL are not complete enough to specify properties for verification based on these compositions in model checkers.

Term rewriting is a computational model based on repeated application of *simplification* rules. It has been advantageous for symbolic computation, program analysis and transformation [26]. The systems built on Term rewriting (TRS) have been found to be relatively efficient in verifying systems made up of large number of components, as the approach does not involve translation process of generating FSM, thereby mitigating potential problem of stat space explosion. A new verification framework using extended regular expressions is proposed as part of a PhD thesis [132]. This framework provides compositional verification via a Hoare-style forward verifier in the front-end, and a Term Rewriting Systems (TRS) in the back-end.

The work proposes a new **"Effect logic"** for specification of properties in various

programming domains, in place of generic temporal logic notations LTL, CTL. *DependentEffs* for general effectful programs, *ASyncEffs* for mixed synchronous and asynchronous reactive programs, *TimEffs* for time-critical distributed programs and *ContEffs* for programs with user-defined effects and handlers. The language *ASyncEffs* for specification of verification properties on programs written in languages like Esterel, HipHop.js, can also be used to generate notations equivalent to LTL and CTL notations. The traditional temporal operators `X (Next)`, `F (Future)`, `G (Global)` and `U (Until)` can be translated to "Effect logic" as follows.

- $X\texttt{p} = \{\}.\{p\}$
- $\texttt{Fp} = \{\}^*.\{p\}$
- $\texttt{Gp} = \{p\}^*$,
- $\texttt{p U q} = \{p\}^*.\{q\}$

We see that the "Effect logic" notations follow Hoare-style triples: $\{\Phi_{pre}\}\ e\ \{\Phi_{post}\}$, where $\{\Phi_{pre}\}$ is the effects prior to the execution of e and $\{\Phi_{post}\}$ is the resulting effect after executing e. According to the author, the advantages of the new verification framework are: (i) efficiency due to usage of TRS in the backend rather than model checkers, (ii) it can address the issues in verification of composition of synchronous and asynchronous computations as in HipHop.js programs, and (iii) provide finer-grained, modular verification for big multi-module systems.

The framework presented here helps in proving logical correctness of programs coming from various domains, whereas the HipHop.js program analyzer aims to provide infrastructure for program comprehension, semi-automated testing and fault localization for programs written in HipHop.js, which represents one of the programming domain of the many for which the framework presented here can be used. Model checkers at the back end provide state space searching services for the goals identified by the analyzer utilities, whereas in the framework the backend TRS, with the help of forward verifier provides verification support. The HipHop.js program analyzer as of now provides support only to the programs having synchronous constructs of HipHop.js. The framework presented here can be incorporated as part of the future work to deal with temporal verification of

composition of *synchronous* and *asynchronous* HipHop.js constructs. Further, the HipHop.js program analyzer aims to increase the adaptation of analyzer utilities by programmers, with easy to use and intuitive interfaces that can be readily used within their programs.

This chapter provides an overview of the related work, specifically concentrating on advances in automated testing, software fault localization and program comprehension techniques which is related to the contribution of this thesis. With this we end this chapter. The next chapter concludes this thesis along with presenting some of the future work that can be done on the infrastructure provided by the work in this thesis.

Chapter 8

Future Work and Conclusion

This chapter summarizes our work and presents directions for future work. Section 8.1 sums up contributions answering the initial research objectives of this thesis. Section 8.2 concludes this thesis by presenting additional work that can extend our HipHop.js program analyzer in order to support debugging even better.

8.1 Summary

Debugging software has continued to be an art and new practices are being adopted to standardize the process. Even before debugging, finding subtle bugs has been trickier. Any programming language is as much attractive as the supporting infrastructure the language has, including IDEs, debugging infrastructure and so on. The objective of this dissertation is to present the debugging infrastructure, we have built for the HipHop.js DSL and also to demonstrate its advantages to HipHop.js programmers.

In HipHop.js, there is a special class of errors called causality errors. To debug these errors, fault localization techniques are required to narrow down source of errors. Also, to debug logical errors, special infrastructure support is required which is different from traditional programming due to the inherent nature of synchronous reactive systems. To debug causality errors, a new technique based on graph theory algorithms is proposed to generate better error messages. The technique is shown to be effective by experiments conducted on real world software written in HipHop.js. This technique can be used in other reactive languages to build better error messages for causality errors. Also, visualization of the error

source is supported in popular IDEs making it easier to locate in bigger programs.

Questions, query based debugging strategies are being increasingly used in programming languages supporting infrastructure to identify subtle errors and also in fault localization. The proposed HipHop.js program analyzer is built on those strategies, providing utilities which help in goal based searching. These utilities are easy to use and can be integrated within HipHop.js programs. Since the learning curve to use HipHop.js program analyzer is very minimal, the adaptation of the utilities by the programmers will be more. This thesis presents the objectives, usage, advantages of having a HipHop.js program analyzer, improving HipHop.js debugging infrastructure.

Further, to help understand programs better, a tool supporting visualization has been proposed in the form of *RecordPlayer*. This tool utilizes syntax highlighting while replaying the executions. The usage, implementation and advantages of the visualization tool in program comprehension is demonstrated in this thesis. Our contribution in this thesis is towards the endeavor of providing various aspects of HipHop.js debugging infrastructure including program understanding. We believe, the infrastructure presented in this thesis can improve debugging experience for HipHop.js programmers. In the next section we provide an outlook on work that can be carried out further on the contributions of this thesis.

8.2 Future Work

The future work that can be carried out on the first contribution of this thesis, "causality error tracer" is already presented in section 3.6. Here, we introduce some of the work that can be carried out on the present debugging infrastructure. The utilities that we have presented can be tested with HipHop.js programmers to gain feedback on ease of usability and its effectiveness, henceforth modify the utilities accordingly. As in *LINGUA FRANCA*, CADP model checker and MCL property specification language can be explored for any comparable advantages. The infrastructure provided through *RecordPlayer* and the HipHop.js program analyzer can be integrated onto a web platform. Also, interactive simulation of execution of HipHop.js programs can be provided through the web interface. As of now, the utilities are using NuSMV model checker, which can be upgraded to more recent upgraded version, NuXMV. Since, HipHop.js incorporates extends support

to asynchronous communication, program analyzer utilities can be extended to handle those constructs.

While working on the program analyzer infrastructure, we observed that plain JavaScript code can be generated from the simplified models as in SMV/AIGER format. To begin with, the plain JavaScript code can be easier to understand for people getting introduced to reactive programming. Further work should be done to see if the generated plain JavaScript code can replace the original programs written in HipHop.js DSL, to provide the same functionality in operations. An outlook is provided in the following section. Here, we want to stress that these simplification techniques have been used in Esterel since 2003, including Esterel V7 industrial compiler.

8.2.1 Simplified Code Generator

When we use the earlier presented analyzer utilities, the first step is to translate HipHop.js reactive machine to equivalent AIG/SMV format. In this format, the HipHop.js program can be represented as a graph with topologically sorted nodes. This graph, can be translated to plain JavaScript code and can be used to give programmers another perspective about their programs. We illustrate this process with an example and then detail the implementation details.

Consider a simple HipHop.js program as in the following listing 8.1.

```
1 hiphop module prgIf() {
2     in I;
3     out O;
4
5     if (I.now) {
6         emit O();
7     }
8 }
```

Listing 8.1: A simple HipHop.js program

In this module, it is tested for the presence of input signal `I` in the present instant and if true, emits signal `O` in the same instant. When we convert this program to BLIF and then to AIG/SMV and based on those format, we can represent the same translated HipHop.js program into a graph as follows in the figure 8.1

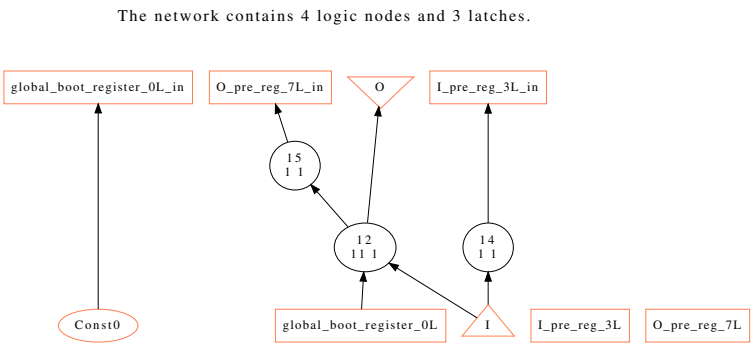


Figure 8.1: Directed Graph representation of the program 8.1

The graph should be read in bottom-up fashion and it is topologically sorted. From this graph, we can generate plain JavaScript code reaction function. This is given in the following listing 8.2. The following code is a direct translation of the above graph. We can see that the nodes are presented in the same way as in the graph.

```
1  // HipHop.js reaction function
2  // global register variable declarations
3  let global_boot_register_0L = true;
4  let I_pre_reg_3L = false;
5  let O_pre_reg_7L = false;
6
7  function reaction( I) {
8      // topologically sorted nodes
9          // level 0 nodes
10
11         let Node1 = I;
12         let Node5 = global_boot_register_0L;
13         let Node8 = I_pre_reg_3L;
14         let Node11 = O_pre_reg_7L;
15         let Node16 = false;
16         // level 1 nodes
17         let Node12 = Node1 && Node5;
18         let Node14 = Node1;
19         // level 2 nodes
20         let Node15 = Node12;
21         // level 3 nodes
22         let Node2 = Node12;
23         let O = Node2;
```

```
24          let Node4 = Node16;
25          let global_boot_register_0L_in = Node4;          // for next state
26              global_boot_register_0L = global_boot_register_0L_in;
27          let Node7 = Node14;
28          let I_pre_reg_3L_in = Node7;              // for next state
29              I_pre_reg_3L = I_pre_reg_3L_in;
30          let Node10 = Node15;
31          let O_pre_reg_7L_in = Node10;             // for next state
32              O_pre_reg_7L = O_pre_reg_7L_in;
33          //output signals
34          return { "O" : O }
35  }
36  console.log(reaction(true));
37  console.log(reaction(true));
```

Listing 8.2: Equivalent plaint JS function for the above HipHop.js program

The above reaction function can be used in similar fashion as previously the HipHop.js reactive machine are used. This code can be further optimized and it is as below in the listing 8.3.

```
1  // HipHop.js reaction function
2  // global register variable declarations
3  let global_boot_register_0L = true;
4  let I_pre_reg_3L = false;
5  let O_pre_reg_7L = false;
6
7  function if_reaction( I) {
8
9          let O = I && global_boot_register_0L;
10
11         global_boot_register_0L = false; // for next state
12         I_pre_reg_3L = I; // for next state
13         O_pre_reg_7L = O; // for next state
14
15         //output signals
16         return { "O" : O }
17 }
```

Listing 8.3: Equivalent plaint JS function for the above HipHop.js program

The above code is a simplified plain JavaScript reaction function that can be used to understand the behavior of their reactive program. In the following section, we

present the implementation details for the simplified code generator.

Implementation Details

The BLIF generator generates the BLIF equivalent of an HipHop.js source. This file is then converted into AIG format, which performs "Strashing" - structural Hashing. This process makes sure that there will be no subgraphs which are structurally equivalent in the network of nodes (nets). From this AIG file, AIG graph is generated using ABC[1] tool, another tool for synthesis and verification. The ABC generates the DOT format for the graph. The nodes in this graph are topologically sorted, and we use this graph as input to translate into the plain JavaScript functions we saw earlier. We use a DOT parser which process the information from the DOT format of the graph and based upon the data we generate the JavaScript equivalent code, which is further optimized. We present the block diagram of the code generator in the following figure 8.2:

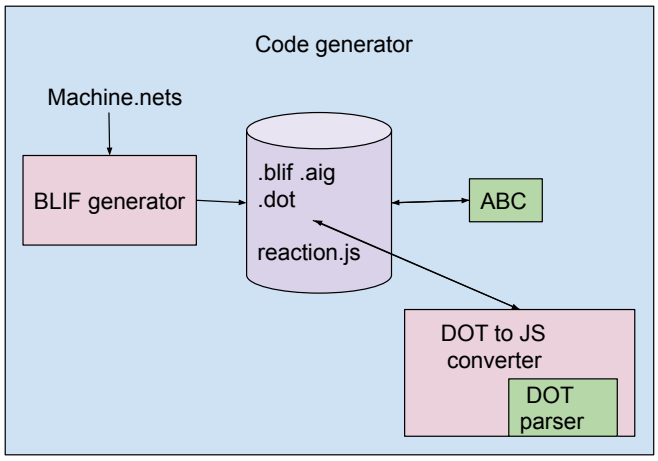


Figure 8.2: Building blocks of the Code generator

Since the size of the code is reduced, and also it is plain JavaScript the approach may reduce the execution time. This approach can be explored further to check for its effectiveness in reactive environment. With this we conclude this chapter and the thesis.

[1]https://people.eecs.berkeley.edu/ alanmi/abc/

Bibliography

[1] First look at the debugger - visual studio (windows). `https://learn.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour?view=vs-2022`. Online: 2022-10-05.

[2] David Abramson, Ian Foster, John Michalakes, and Rok Sosic. Relative debugging and its application to the development of large numerical models. In *Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, pages 51–51. IEEE, 1995.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[4] Hiralal Agrawal, Richard A DeMillo, and Eugene H Spafford. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616, 1993.

[5] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[6] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151. IEEE, 1995.

[7] Mian Asbat Ahmad. *New Strategies for Automated Random Testing*. PhD thesis, University of York, 2014.

[8] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results and practical implications. *IEEE transactions on Software Engineering*, 38(2):258–277, 2011.

[9] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, 2010.

[10] Luciano C Ascari, Lucilia Y Araki, Aurora RT Pozo, and Silvia R Vergilio. Exploring machine learning techniques for fault localization. In *2009 10th Latin American Test Workshop*, pages 1–6. IEEE, 2009.

[11] Dimitar Asenov, Otmar Hilliges, and Peter Müller. The effect of richer visualizations on code comprehension. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5040–5045, 2016.

[12] Ron Baecker, Chris DiGiano, and Aaron Marcus. Software visualization for debugging. *Communications of the ACM*, 40(4):44–54, 1997.

[13] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[14] Thomas Ball, Mayur Naik, and Sriram K Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–105, 2003.

[15] Herman Banken, Erik Meijer, and Georgios Gousios. Debugging data flows in reactive programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 752–763. IEEE, 2018.

[16] Earl T Barr and Mark Marron. Tardis: Affordable time-travel debugging in managed runtimes. *ACM SIGPLAN Notices*, 49(10):67–82, 2014.

[17] Earl T Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. Time-travel debugging for javascript/node. js. In *Proceedings of the 2016*

24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 1003–1007, 2016.

[18] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of computer programming*, 16(2):103–149, 1991.

[19] Alexandre Bergel, Felipe Banados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Computer Languages, Systems & Structures*, 38(1):16–28, 2012.

[20] UoC Berkeley. Berkeley logic interchange format (blif). *Oct Tools Distribution*, 2:197–247, 1992.

[21] Gérard Berry. Real time programming: Special purpose or general purpose languages. Technical report, INRIA, 1989.

[22] Gerard Berry. The constructive semantics of pure esterel. draft version 3. *Draft Version*, 3, 1999.

[23] Gérard Berry. Scade: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer, 2007.

[24] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[25] Gérard Berry and Manuel Serrano. Hiphop.js:(a) synchronous reactive web programming. In *PLDI*, pages 533–545, 2020.

[26] Marc Bezem, JW Klop, and Roel de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003.

[27] Armin Biere. The aiger and-inverter graph (aig) format version 20071012. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Austria, 2007.

[28] David W Binkley and Mark Harman. A survey of empirical results on program slicing. *Adv. Comput.*, 62(105178):105–178, 2004.

[29] Stephen Blackheath and Anthony Jones. *Functional reactive programming*. Manning Publications Co., 2016.

[30] Amar Bouali. Xeve, an esterel verification environment. In *International Conference on Computer Aided Verification*, pages 500–504. Springer, 1998.

[31] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993.

[32] Frédéric Boussinot. Reactive c: An extension of c to program reactive systems. *Software: Practice and Experience*, 21(4):401–428, 1991.

[33] Frédéric Boussinot. Sugarcubes implementation of causality. Technical report, Inria, 1998.

[34] Frédéric Boussinot and Robert De Simone. The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[35] Frédéric Boussinot and Jean-Ferdy Susini. The sugarcubes tool box: a reactive java framework. *Software: Practice and Experience*, 28(14):1531–1550, 1998.

[36] Lionel C Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, pages 137–146. IEEE, 2007.

[37] Rafael Caballero, Adrián Riesco, and Josep Silva. A survey of algorithmic debugging. *ACM Computing Surveys (CSUR)*, 50(4):1–35, 2017.

[38] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, 1987.

[39] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.

[40] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Dellis: A data mining process for fault localization. In *SEKE*, pages 432–437, 2009.

[41] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *SEKE*, pages 238–243, 2011.

[42] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International conference on computer aided verification*, pages 359–364. Springer, 2002.

[43] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, pages 71–80, 2008.

[44] Edmund M Clarke. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

[45] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–11. IEEE, 2017.

[46] Deborah S Coutant, Sue Meloy, and Michelle Ruscetta. Doc: A practical approach to source-level debugging of globally optimized code. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation*, pages 125–134, 1988.

[47] Evan Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 30, 2012.

[48] Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[49] Julien Deantoni, João Cambeiro, Soroush Bateni, Shaokai Lin, and Marten Lohstroh. Debugging and verification tools for lingua franca in gemoc studio.

In *2021 Forum on specification & Design Languages (FDL)*, pages 01–08. IEEE, 2021.

[50] Marcus Denker, Orla Greevy, and Michele Lanza. Higher abstractions for dynamic analysis. In *2nd international workshop on program comprehension through dynamic analysis (PCODA 2006)*, pages 32–38. Universiteit Antwerpen, 2006.

[51] Cloé Descheemaeker, Sam Van den Vonder, Thierry Renaux, and Wolfgang De Meuter. Poker: visual instrumentation of reactive programs with programmable probes. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pages 14–26, 2021.

[52] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[53] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of the 22nd international conference on Software engineering*, pages 467–476, 2000.

[54] Joe W Duran and Simeon Ntafos. A report on random testing. In *ICSE*, volume 81, pages 179–183. Citeseer, 1981.

[55] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[56] Jermaine Charles Edwards. Method, system, and program for logging statements to monitor execution of a program, March 25 2003. US Patent 6,539,501.

[57] Matt Elder. Bourdoncle components, 2010.

[58] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[59] Fabien Gaucher, Erwan Jahier, Florence Maraninchi, and Bertrand Jeannet. Automatic state reaching for debugging reactive programs. In *the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, 2003.

[60] Michael Gorbovitski, K Tuncay Tekle, Tom Rothamel, Scott D Stoller, and Yanhong A Liu. Analysis and transformations for efficient query-based debugging. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 174–183. IEEE, 2008.

[61] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. *Electronic Notes in Theoretical Computer Science*, 174(4):95–111, 2007.

[62] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Fault localization using a model checker. *Software Testing, Verification and Reliability*, 20(2):149–173, 2010.

[63] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *International SPIN Workshop on Model Checking of Software*, pages 121–136. Springer, 2003.

[64] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. Perception and practices of differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 71–80. IEEE, 2019.

[65] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A visual interactive debugger based on symbolic execution. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 143–146, 2010.

[66] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[67] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[68] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST'93)*, pages 83–96. Springer, 1994.

[69] Nicolas Halbwachs and Florence Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs, 1995.

[70] David Harel, Amir Pnueli, and KR Apt. Logics and models of concurrent systems. *NATO Advanced Study Institute. chapter On the development of reactive systems*, 471:498, 1985.

[71] John Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):323–344, 1982.

[72] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. Visual debugging techniques for reactive data visualization. In *Computer Graphics Forum*, volume 35, pages 271–280. Wiley Online Library, 2016.

[73] Jeff Horemans, Bob Reynders, Dominique Devriese, and Frank Piessens. Elmsvuur: A multi-tier version of elm and its time-traveling debugger. In *International Symposium on Trends in Functional Programming*, pages 79–97. Springer, 2017.

[74] Willembrinck Santander Maximilian Ignacio, Steven Costiou, Anne Etien, and Stéphane Ducasse. Time-traveling queries for faster debugging and program comprehension. In *Journées Nationales du Génie de la Programmation et du Logiciel 2022*, 2022.

[75] Wolfgang Jeltsch. Fucntional reactive programming. `https://wiki.haskell.org/Functional_Reactive_Programming`. Online: 2022-10-18.

[76] Manu Jose and Rupak Majumdar. Bug-assist: assisting fault localization in ansi-c programs. In *International conference on computer aided verification*, pages 504–509. Springer, 2011.

[77] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, 46(6):437–446, 2011.

[78] Raghudeep Kannavara, Christopher J Havlicek, Bo Chen, Mark R Tuttle, Kai Cong, Sandip Ray, and Fei Xie. Challenges and opportunities with concolic testing. In *2015 National Aerospace and Electronics Conference (NAECON)*, pages 374–378. IEEE, 2015.

[79] Ryana Karaki, Ludovic Marti, and Julien Deantoni. Wip: Domain specific debugging by using runstar. In *Work in Progress of the 2021 Forum on specification & Design Languages (FDL)*, 2021.

[80] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125. IEEE, 2017.

[81] Amy J Ko and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, 2004.

[82] Amy J Ko and Brad A Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, pages 301–310, 2008.

[83] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 91–100. IEEE, 2011.

[84] Bogdan Korel. Pelas-program error-locating assistant system. *IEEE Transactions on Software Engineering*, 14(9):1253–1260, 1988.

[85] Bogdan Korel and Janusz Laski. Algorithmic software fault localization. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume 2, pages 246–252. IEEE, 1991.

[86] Jayanth Krishnamurthy and Manuel Serrano. Causality error tracing in hiphop. js. In *23rd International Symposium on Principles and Practice of Declarative Programming*, pages 1–13, 2021.

[87] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, 2002.

[88] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[89] Robert S Laramee. Using visualization to debug visualization software. *IEEE computer graphics and applications*, 30(6):67–73, 2010.

[90] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, EMSOFT '07, pages 114–123, New York, NY, USA, 2007. Association for Computing Machinery.

[91] Raimondas Lencevicius. *Query-based debugging*. University of California, Santa Barbara, 1999.

[92] Raimondas Lencevicius, Urs Hölzle, and Ambuj K Singh. Query-based debugging of object-oriented programs. *ACM SIGPLAN Notices*, 32(10):304–317, 1997.

[93] Raimondas Lencevicius, Urs Hölzle, and Ambuj K Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engineering*, 10(1):39–74, 2003.

[94] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[95] Bil Lewis. Debugging backwards in time. *arXiv preprint cs/0310016*, 2003.

[96] Xiangyu Li, Shaowei Zhu, Marcelo d'Amorim, and Alessandro Orso. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering*, pages 82–92, 2018.

[97] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee. A language for deterministic

coordination across multiple timelines. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8, 2020.

[98] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–37, 2014.

[99] Louis Mandel and Marc Pouzet. Reactiveml: a reactive extension to ml. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 82–93, 2005.

[100] Florence Maraninchi and Fabien Gaucher. Step-wise+ algorithmic debugging for reactive programs: Ludic, a debugger for lustre. In *AADEBUG*. Citeseer, 2000.

[101] Radu Mateescu and Damien Thivolle. A model checking language for concurrent value-passing systems. In *International Symposium on Formal Methods*, pages 148–164. Springer, 2008.

[102] Wolfgang Mayer and Markus Stumptner. Evaluating models for model-based debugging. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 128–137. IEEE, 2008.

[103] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, 2008.

[104] Kenneth L McMillan and Symbolic Model Checking. *An approach to the state explosion problem*. PhD thesis, Ph. D. Thesis, Carnegie Mellon University, 1992, CMU-CS-92-131, 1993.

[105] Stephan Merz. Elements of model checking. *Actes de l'Ecole d'Eté Temps Réel 2005-ETR'2005*, page 55, 2005.

[106] Ellis Michael, Doug Woos, Thomas Anderson, Michael D Ernst, and Zachary Tatlock. Teaching rigorous distributed systems with efficient model checking. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[107] Ragnar Mogk, Pascal Weisenburger, Julian Haas, David Richter, Guido Salvaneschi, and Mira Mezini. From debugging towards live tuning of reactive applications. In *2018 LIVE Programming Workshop. LIVE*, volume 18, 2018.

[108] S Nessa, M Abedin, W Eric Wong, L Khan, and Y Qi. Fault localization using n-gram analysis. In *Proceedings of the 3rd International Conference on Wireless Algorithms, Systems, and Applications*, pages 548–559, 2009.

[109] Girish Keshav Palshikar. An introduction to esterel. *Embedded Systems Programming*, 14(11), 2001.

[110] Erik Pasternak, Rachel Fenichel, and Andrew N Marshall. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B&B)*, pages 21–24. IEEE, 2017.

[111] Maria João Varanda Pereira, Marjan Mernik, Pedro Rangel Henriques, et al. Program comprehension for domain-specific languages. *Computer Science and Information Systems*, 5(2):1–17, 2008.

[112] Bertrand Petit and Manuel Serrano. Composing and performing interactive music using the hiphop. js language. In *NIME 2019-New Interfaces for Musical Expression*, 2019.

[113] Bertrand Petit and Manuel Serrano. Interactive music and synchronous reactive programming. *arXiv preprint arXiv:2006.03102*, 2020.

[114] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee, 1977.

[115] Alex Potanin, James Noble, and Robert Biddle. Snapshot query-based debugging. In *2004 Australian Software Engineering Conference. Proceedings.*, pages 251–259. IEEE, 2004.

[116] Dumitru Potop-Butucaru, Stephen A Edwards, and Gérard Berry. *Compiling esterel*, volume 86. Springer Science & Business Media, 2007.

[117] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

[118] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 485–495. IEEE, 2012.

[119] David S Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th international conference on Software engineering*, pages 92–104, 1992.

[120] David S. Rosenblum. A practical approach to programming with assertions. *IEEE transactions on Software Engineering*, 21(1):19–31, 1995.

[121] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 796–807. IEEE, 2016.

[122] Advait Sarkar. The impact of syntax colouring on program comprehension. In *PPIG*, page 8, 2015.

[123] Klaus Schneider. The synchronous programming language quartz. Technical report, Internal Report 375, Department of Computer Science, University of Kaiserslautern, 2009.

[124] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 308–311. IEEE, 2017.

[125] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2. 0. In *OOPSLA companion*, pages 975–985, 2006.

[126] Manuel Serrano and Vincent Prunet. A glimpse of hop.js. *ACM SIGPLAN Notices*, 51(9):180–192, 2016.

[127] Ehud Yehuda Shapiro. *Algorithmic program debugging*. Yale University, 1982.

[128] Thomas R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, EECS Department, University of California, Berkeley, Dec 1996.

[129] Thomas R Shiple, Gérard Berry, and Herve Touati. Constructive analysis of cyclic circuits. In *Proceedings ED&TC European Design and Test Conference*, pages 328–333. IEEE, 1996.

[130] Janet Siegmund. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 13–20. IEEE, 2016.

[131] Ian Sommerville. *Software Engineering, 9/E*. Pearson Education India, 2011.

[132] Yahui Song. *AUTOMATED TEMPORAL VERIFICATION WITH EXTENDED REGULAR EXPRESSIONS*. PhD thesis, National University of Singapore, 2022.

[133] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.

[134] M-A Storey. Theories, methods and tools in program comprehension: past, present and future. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 181–191. IEEE, 2005.

[135] Thibault Suzanne. Weak topological. `http://ocamlgraph.lri.fr/doc/WeakTopological.html`. Online: 2022-09-18.

[136] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[137] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[138] John Vilk, Emery D Berger, James Mickens, and Mark Marron. Mcfly: Time-travel debugging for the web. *arXiv preprint arXiv:1810.11865*, 2018.

[139] Tianxia Wang and Yan Liu. Jsea: A program comprehension tool adopting lda-based topic modeling. *International Journal of Advanced Computer Science and Applications*, 8(3), 2017.

[140] Weiguang Wang and Qingkai Zeng. Evaluating initial inputs for concolic testing. In *2015 International Symposium on Theoretical Aspects of Software Engineering*, pages 47–54. IEEE, 2015.

[141] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of annual IEEE/ACM international symposium on code generation and optimization*, pages 98–108, 2014.

[142] Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* University of Michigan, 1979.

[143] Christian Wirth, Herbert Prahofer, and Roland Schatz. A multi-level approach for visualization and exploration of reactive program behavior. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–4. IEEE, 2011.

[144] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2011.

[145] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[146] W Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009.

[147] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 746–757. Springer, 2002.

[148] Franz Wotawa, Jörg Weber, Mihai Nica, and Rafael Ceballos. On the complexity of program debugging using constraints for modeling the program's syntax and semantics. In *Conference of the Spanish Association for Artificial Intelligence*, pages 22–31. Springer, 2009.

[149] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark

Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. *RN*, 14(14):14, 2014.

[150] Zhongxing Yu, Hai Hu, Chenggang Bai, Kai-Yuan Cai, and W Eric Wong. Gui software fault localization using n-gram analysis. In *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*, pages 325–332. IEEE, 2011.

[151] Andreas Zeller. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):1–10, 2002.

[152] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

Appendices

The appendices presented here include details about the input formats that are utilized in HipHop.js program analyzer. Readers who wish to have a brief introduction to the various input formats and tools used in HipHop.js program analyzer can refer the following appendices before reading the implementation parts. The material presented here is a brief summary of the original literature, adapted to our work. In appendix A, the BLIF format is introduced detailing the notations used and examples. In appendix B, input format AIGER is presented with illustrations on sample circuits. In appendix C, the details about the model checker NuSMV and the extended SMV language notations are presented. As part of ready reference, the HipHop.js language grammar is presented in appendix D.

A BLIF

We present a brief overview of the Berkeley logic interchange format (BLIF) [20] which is a textual form of describing a logic-level hierarchical circuit. We use this format as one of the intermediate format for transformation of HipHop.js programs that will be used by HipHop.js program analyzer. The input format is as follows. A logic circuit can be any combinational or sequential interconnections of logic functions. A circuit is generally viewed as a directed graph of combinational and sequential logic elements. A general BLIF file will have many "models" described. A model will have declarations of flattened hierarchical logical circuits. A model declaration inside a BLIF file has the following format and order:

- **.model <decl-model-name>**
- **.inputs <decl-input-list>**
- **.outputs <decl-output-list>**
- **.clock <decl-clock-list>**
- **<command>** - This section can declare one of the following:
    1. <logic-gate>
    2. <generic-latch>
    3. <library-gate>

4. <model-reference>
5. <subfile-reference>
6. <fsm-description>
7. <clock-constraint>
8. <delay-constraint>

- **.end** - signals the end of declaration of the model to the parser.

For example, a two input "AND" gate, with input names as `in1`, `in2` and corresponding output name as `and1`, can be described as follows.

```
.names in1 in2 and1
11 1
```

In the above listing `.names` is the keyword to specify the definition of a logical relation. `in1`, `in2` are inputs and `and1` is output. In the next line, the truth table for "AND" gate is represented. As we know the truth value of "AND" gate is true when both the inputs are true, in the above listing `''11 1''` represents the same with `1` meaning true, `0` meaning false, and `-` meaning don't care. For the same input signals, we can describe a two input "OR" gate, with output signal `or1`, based on the truth table for "OR" gate as follows.

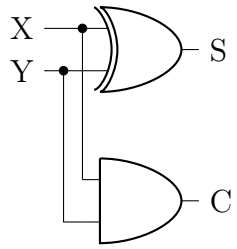
```
.names in1 in2 or1
1- 1
-1 1
```

Now, we illustrate an example latch representation.

```
.latch in reg 0
```

In the above listing, `.latch` is the keyword used to denote a generic latch. `in` is a single input to latch, of name `reg`, with `0` being the initial value of the latch. The next state value is fed from the input `in`. We describe a BLIF combinational logic circuit, the half adder now. The following table and circuit diagram illustrates the truth table and logic diagram of half adder with input signals `X`, `Y` and output signals `S,C`. `S` denotes the Sum and `C` denotes the carry output.

X	Y	S (SUM)	C (Carry)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



The BLIF representation for the above circuit is as follows:

```
.model halfadder
.inputs X Y
.outputs S C
.names X Y C
11 1
.names X Y S
10 1
01 1
.end
```

We illustrate an example translation from a HipHop.js program to BLIF format. The following listing is a simple HipHop.js program which we introduced in the previous section.

```
1 hiphop module prg() {
2     in I;
3     out O;
4     if (I.now) {
5        emit O();
6     }
7 }
8 let machine = new hh.ReactiveMachine(prg);
9 machine.debug_emitted_func = console.log;
```

```
10 machine.react();
```

Listing 4: A sample HipHop.js program, If.js

The following listing is the BLIF format representation of the above program.

```
.model prg
.inputs I
.outputs O
.latch global_const0_1 global_boot_register_0 1
.names global_const0_1
0
.latch I I_pre_reg_3 0
.names I_pre_reg_3  I_pre_gate_4
1 1
.latch O O_pre_reg_7 0
.names O_pre_reg_7  O_pre_gate_8
1 1
.names and_then_18 and_then_22  O
1- 1
-1 1
.names testexpr_20 global_boot_register_0  and_then_18
11 1
.names I global_boot_register_0  testexpr_20
11 1
.names go_21
0
.names testexpr_24 go_21  and_then_22
11 1
.names I go_21  testexpr_24
11 1
.end
```

Listing 5: BLIF representation of the If.js program

The above listing is generated based on the netlist generated by HipHop.js compiler. Typically, the netlist includes only "AND" and "OR" gate representations with latches. The netlist for the circuit generated by HipHop.js compiler is as shown in the following figure.

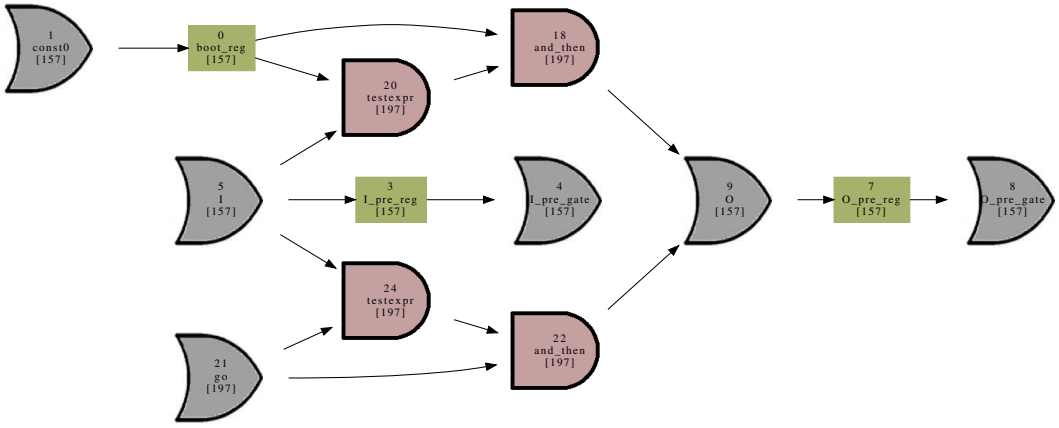


Figure 3: Graphical illustration of the nets in If.js program.

This is the end of appendix on BLIF format. In the next part of appendices, the AIGER format and tool is introduced.

B AIGER

The following appendix helps understand the AIGER format, and the way it can be generated. The AIGER file format is another way of representing combinational and sequential logic files using AIG format, the *And Invert Graph* format. It was introduced by Biere et al., [27] and has been used in model checking competitions since 2007. The format uses both ASCII and binary representations. The ASCII format is helpful as a human-readable format, while binary format is compact and can be used by other software applications. We provide a brief overview of the ASCII format representation that is helpful in our HipHop.js program transformation. The following notations are as in the original literature cited above.

AIGER format - ASCII version

A file in AIGER format consists of the following parts:

- **Header** - The header consists of a single line with text `aag` to indicate ASCII format (`AIG` for binary format), followed by non-negative integers represented in the form `M I L O A`. `aag M I L O A`. Here, `M` represents the maximum variable index. `I` specifies the number of inputs, `L`, the number of latches, `O` the number of outputs and `A` the total number of "AND" gates. If all the variables are used and there are no unused "AND" gates in a circuit, then `M = I + L + A`. Each of these non-negative numbers (including `0`) are written in ASCII representation of numbers and are separated by a space character.

- **Input definitions** - inputs are defined by "even" number literals. Literals are basically constants or signed variables which are represented by unsigned integers.

- **Latch definitions** - every latch definition consists of an "even" number literal followed by a number that specifies the variable used to update the latch in each step. Initial value of a latch is assumed to be zero.

- **Output definitions** - defined as a single literal, represents one of the inputs (or negated input), a latch, or an "AND" gate.

- **And-gate definitions** - Each "AND" gate definition consists of three numbers, an even number literal representing the output of the "AND" gate, followed by two literals representing the inputs.

In the unsigned word encoding of a literal, the least significant bit is the sign bit, while the remaining bits represent the variable index. For example, the literal `2`, whose two bit binary representation is `10` represents a non-negated **first** (1) variable. Since least significant bit is `0`. Literal `3 (11)` represents a negated **first** (1) variable. Likewise, `4 (100)` and `5 (101)` literals represent non-negated, negated **second** (`10`) variable. The input variable indices can range as `1, 2, 3 ... I`. The latch variable indices range as `I+1, I+2, I+3 ... L`. Finally, the "AND" variable indices range as `I+L+1, I+L+2, I+L+3 ... A`. To get the value of a literal from a variable, the variable index is multiplied by 2 and optionally `1` is added if the variable should be negated. Accordingly, the corresponding unsigned input literals range as `2, 4, ... , 2*I`. The latch literals are in the range `2*I+2, 2*I+4, .., 2*(I+L)`. And finally, the "AND" literals `2*(I+L)+2, 2*(I+L)+4, ... , 2*(I+L+A) == 2*M`. AIGER library models only cycle-accurate circuits. We shall see some example representations.

Empty Circuit The empty circuit consists of a single line, which is just the header:

```
    M I L O A
aag 0 0 0 0 0
```

TRUE constant can be represented as follows - a single `1` in the header in the `O` field specifies that the number of outputs is one. Then the header is followed by a single line which contains the literal of the single output. Here, it is `1`.

```
aag 0 0 0 1 0
1
```

FALSE constant is similar to **TRUE**, except that the literal following the header will be "0" as follows.

```
aag 0 0 0 1 0
0
```

A simple **Buffer**, which forwards the data has one input and one output variable, with zero latches and zero "AND" gates. Accordingly, it is represented with `1` at `I`, `1` at `O` and `0` at `L` and `A` fields in the header. A `1` at `M` field is deduced based on the

earlier constraint `M = I+L+A`. The input variables are numbered starting from value `1`. Since there is only one input variable, we then list the definition of the input field by multiplying the `first` input variable numbered `1` with `2` to get the literal `2`, defining the input variable. Since it is a forwarding buffer without inversion, another single line defines what should be the `output`. Here it will be same as input `2`.

```
aag 1 1 0 1 0
2
2
```

Now, to represent an inverting buffer, we take the previous header and just change the output definition line accordingly. Since, an inverting buffer will have one input and one output, the input is represented by literal `2` and the output will be represented by literal `3`, which means that the input represented by literal `2` has to be inverted before feeding the output. The number `3` is arrived at by flipping the LSB of binary encoding of `2` as follows:

```
aag 1 1 0 1 0
2
3
```

AND gate: An **AND** gate header will have two inputs, one output, with zero latches and a single **AND** gate. The maximum variable index `M` will be `3` as follows.

```
aag 3 2 0 1 1
```

Now the header should be followed by two lines representing the input variables. These are numbered `1` and `2` respectively. Each one will be multiplied by value `2` to get the respective literals representing the input variables (`2, 4`). After the input definitions, we proceed with output definition. We see that the value of `M` is `3`. As of now, we have used two variables to stipulate the input variables, Now the third value can be used to define the output variable (`6`) as follows.

```
2
4
6
```

Now, we need to define the inputs and output of the "AND" gate. The output should be an even number, while the two inputs should be specified based on previous definitions (`I, L or A`). For the present "AND" gate example, `6` will be

the output, with `2` and `4` as inputs. The final representation of "AND" gate looks as follows.

```
aag 3 2 0 1 1
2
4
6
6 2 4
```

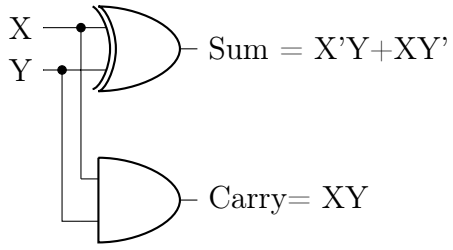
Now the **OR** gate representation. We know that a two input "OR" gate with two inputs `A` and `B` can be written as `!(!A AND !B)`. This insight is used in defining the AIGER format of "OR" gate.

```
aag 3 2 0 1 1
2
4
7
6 3 5
```

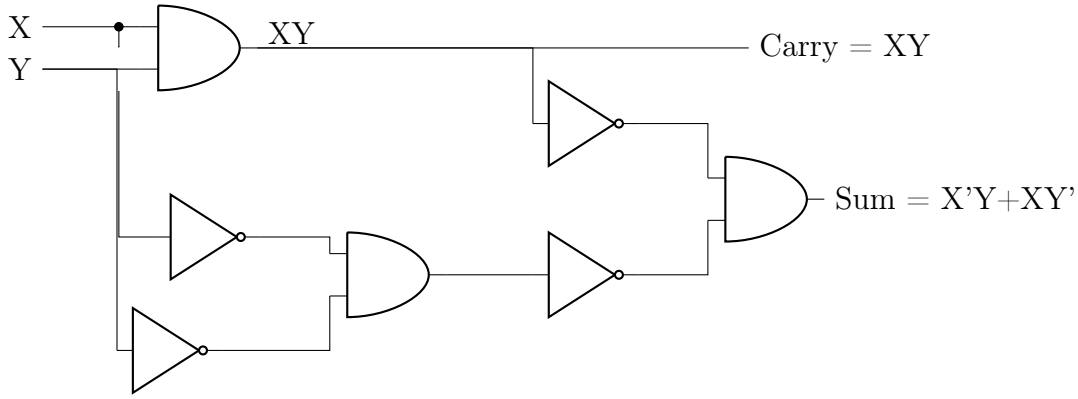
The output definition is `7` to specify that output of the "AND" gate `6` has to be inverted which is specified by flipping LSB of binary representation of `6`. The **latches** can be used to build sequential circuits. Here is an example circuit of a toggle flip-flop with zero input, one latch and two outputs (0 and its negation 1).

```
aag 1 0 1 2 0
2 3
2
3
```

output `2` defines `0`, while output `3` defines negation of `0` in the next state, `1`. Now, we can take an example of representing a more complex circuit, a half adder circuit. This circuit uses 2 inputs, and produces two outputs, sum and carry.



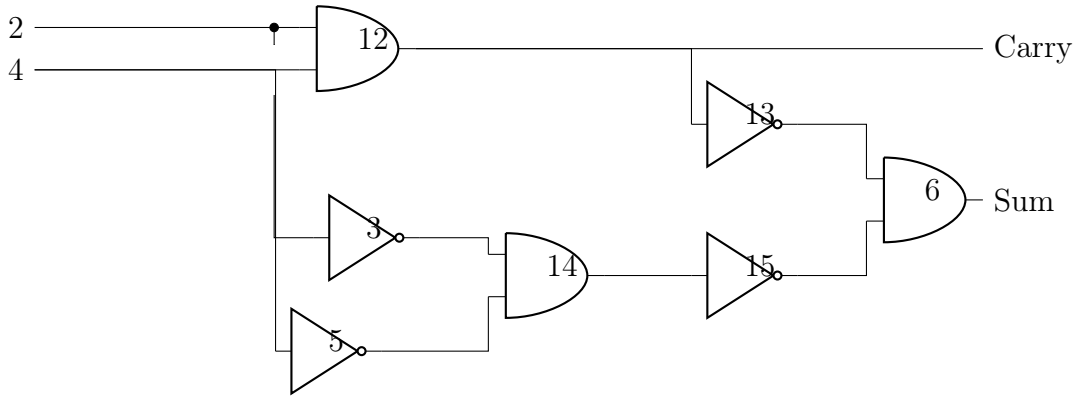
This circuit can be re-written as follows, with just the `AND` gates and `NOT` gates as follows.



The above circuit can be replicated in AIGER format as follows: It uses three "AND" gates to specify the half adder operations. The two inputs are 2 and 4 and the two outputs are 6 and 12 representing sum and carry respectively.

```
aag 7 2 0 2 3
2
4
6
12
6 13 15
12 2 4
14 3 5
```

The above encoding in AIGER list follows the following circuit representation presented earlier.



We end this section with a small encoding example of a simple HipHop.js program. In the following listing the signal O is emitted based on the presence of signal R at that instant.

```
1 hiphop module prg() {
2     in I;
```

```
3     out O;
4     if (I.now) {
5        emit O();
6     }
7  }
8  let machine = new hh.ReactiveMachine(prg);
9  machine.debug_emitted_func = console.log;
10 machine.react();
```

Listing 6: A sample HipHop.js program, If.js

We see the following circuit representation based on the nets generated by the HipHop.js compiler for the program.

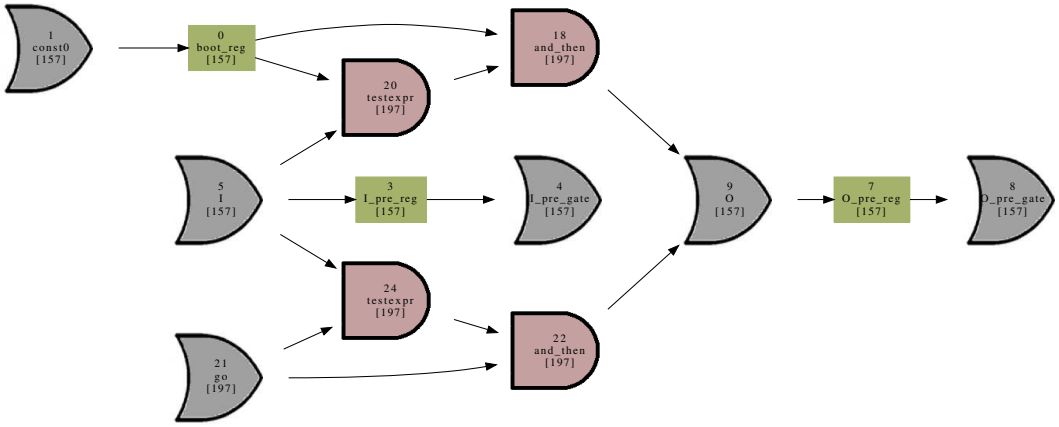


Figure 4: Graphical illustration of the nets in HipHop.js program.

The AIGER's ASCII representation will be more optimized and is as follows. It has single input, of `I`, three latches as we see in the number of registers in the previous circuit diagram, one output for `O` and a single `AND` gate which has input from `R` and boot register.

```
aag 5 1 3 1 1
2
4 0 1
6 2
8 10
10
10 4 2
```

After the "AND" gates definition, symbol table may follow, which are optional. The optional symbol table can be used to attach symbols, one per input, latch

and output literals. Each entry is one line and consists of a symbol type specifier which can be either “i”, “l”, or “o”, in the first character position. This is followed by a position value, not separated by a space. After a space, the symbol name (an arbitrary ASCII string) starts and continues until before the next new line character. A symbol table entry looks as follows:

```
[ilo]<pos> <string>
```

So, for the previous example, with one input, three latches and one output variables, the AIGER file with symbol table is as follows:

```
aag 5 1 3 1 1
2
4 0 1
6 2
8 10
10
10 4 2
i0 I
l0 global_boot_register_0
l1 I_pre_reg_3
l2 O_pre_reg_7
o0 O
```

Listing 7: AIGER represenation of If.js

The inclusion of symbol table is advantageous for us, as it will help the HipHop.js program analyzer with the actual names of the input and output variables. In the ASCII format both, checking for cyclic dependencies has to be done explicitly, whereas in binary format, based on strict order requirement on the literals ensures no explicit checking as the representation disallows cyclic dependencies. For more information, the eager reader is required to refer to AIGER manual cited earlier. In HipHop.js program analyzer, the cyclic dependency check is already taken care by HipHop.js compiler and for us ASCII representation of AIGER file is sufficient. Further, there are utilities available inside AIGER library for transformation between ASCII and binary formats, which can be utilized based on the need. In the next appendix section, the model checker NuSMV is introduced.

C NuSMV

NuSMV [42] is a symbolic model checker. It processes files written in an extended SMV language. Using this language, one can describe finite state machines (FSM) by declaring and instantiating modules and processes corresponding to synchronous compositions. Specification requirements can be expressed in CTL and LTL in the same extended SMV language.

FSMs in NuSMV are described in terms of variables, transitions and Fairness conditions (constraints on the valid paths of the execution of the FSM). A module declaration encapsulates declarations, constraints and specifications. The specifications to be checked on the FSM is expressed in temporal logics like CTL, LTL and Property Specification Language (PSL) and real-time CTL specifications. These specifications are identified with the keywords `CTLSPEC`, `LTLSPEC`, and `PSLSPEC`. Following is an example illustration of the specifications.

```
LTLSPEC F  !(o0)
CTLSPEC AF !(o0)
```

CTL and LTL specifications are evaluated by NuSMV to determine their truth or falsity in the FSM. If a specification is false, then NuSMV outputs a counterexample which is a trace of the FSM that fails the property. Verification of PSL specifications in NuSMV is limited and is supported only for the specs having the operators X, G, F, A, and E, and for a subset of PSL which can be translated to LTL. For full usage of PSL, it is beneficial to use the extended version of NuSMV called NuXMV. NuSMV also supports specifications that have real time CTL (RTCTL) specifications. In NuSMV each transition is assumed to take a unit time for execution, which facilitates RTCTL specifications. `CTLSPEC` path expressions are extended by RTCTL. In specifications like `EBF`, symbol `B` identifies RTCTL specifications for NuSMV. The user can apply BDD-based or SAT-based model checking in NuSMV. BDD-based model checking, uses BDD based representation of the FSM, while in SAT based model checking, FSM is represented as "Reduced Boolean Circuits (RBC)", a form of representing propositional formulae. The RBC is then converted into CNF form and input to SAT solvers by NuSMV.

In bounded model checking (BMC), NuSMV iterates inside a loop until the maximum bound specified is reached or a solution if found, whichever is smaller. Different verification properties can be checked on an FSM and these are independent

of the model checking engine used for the verification (BDD/SAT). The programmer can decide what engine to adopt for each property and the traces of counterexample generated by NuSMV can be exported to various formats including XML format. Next, NuSMV representation of a sample HipHop.js program, `If.js` is presented.

An example HipHop.js program represented in NuSMV notation

We repeat the program in the following listing.

```
1 hiphop module prg() {
2     in I;
3     out O;
4     if (I.now) {
5        emit O();
6     }
7 }
8 let machine = new hh.ReactiveMachine(prg);
9 machine.debug_emitted_func = console.log;
10 machine.react();
```

Listing 8: A sample HipHop.js program, If.js

Its equivalent NuSMV representation is as follows.

```
MODULE main
    VAR
    --inputs
        I : boolean;
    --latches
        global_boot_register_0 : boolean;
        I_pre_reg_3 : boolean;
        O_pre_reg_7 : boolean;
    ASSIGN
        init(global_boot_register_0) := TRUE;
        next(global_boot_register_0) := FALSE;
        init(I_pre_reg_3) := FALSE;
        next(I_pre_reg_3) := I;
        init(O_pre_reg_7) := FALSE;
        next(O_pre_reg_7) := a10;
    DEFINE
    --ands
        a10 := global_boot_register_0 & I;
    --outputs
```

```
o0 := a10;
```

Listing 9: NuSMV representation of If.js

NuXMV

NuXMV [39] inherits NuSMV and extends it across both Finite-state and Infinite-state systems. It complements basic verification techniques in NuSMV with a family of various new verification algorithms including k-induction, IC3 and k-liveness algorithm. Other than SAT solvers, NuXMV uses SMT solvers with support for various theories. NuXMV supports AIGER format directly. For HipHop.js program analyzer, NuSMV is quite sufficient, but to speed up the processes the later versions of the analyzer can use NuXMV. Other than the direct support for model checking on AIGER format and ability to use SMT solvers, the output trace generated by NuXMV can be more easily integrated with the analyzer. This marks the end of appendices on tools used in HipHop.js program analyzer. In the next part of the appendices, HipHop.js language grammar is presented.

D HipHop.js grammar

HipHop.js Grammar

```
<Expression> --> ... | <HHExprStatement>

<HHExprStatement> --> hiphop <HHStatement>

<HHStatement> --> <HHHost>
  | <HHMachine>
  | <HHModule>
  | <HHInterface>
  | <HHSeq>
  | <HHLet>
  | <HHSignal>
  | <HHHalt>
  | <HHFork>
  | <HHEmit>
  | <HHSustain>
  | <HHAbort>
  | <HHWeakabort>
  | <HHSuspend>
  | <HHLoop>
  | <HHAsync>
  | <HHRun>
  | <HHEvery>
  | <HHDo>
  | <HHIf>
  | <HHTrap>
  | <HHBreak>
  | <HHYield>
  | <HHAwait>
  | <HHDollarExpression>
  | <HHExpression> ;

<HHHost> --> host <HHStatement>

<HHBlock> --> {}
  | { <HHStatement> }
  | { <HHStatement> ... <HHStatement> }
```

```
<HHMachine> --> machine <HHMachineModule>

<HHModule> --> module <HHMachineModule>

<HHMachineModule> --> [ <Identifier> ][implements <MirrorIntfList>]( <FormalVarList> )
                      { <FormalSignalList><HHStatement>+ }

<HHInterface> --> interface [ <Identifier> ] [extends <IntfList>]
                  { <FormalSignalList> }

<MirrorIntfList> --> [mirror] <Intf>, ... [mirror] <Intf>

<IntfList> --> <Intf>, ... <Intf>

<Intf> --> <HHDollarIdent>

<FormalVarList> --> | <FormalVar>, ... <FormalVar>

<FormalVar> --> <Identifier> | <Identifier> = <Expression>

<FormalSignalList> --> | <GlobalSignal>, ... <GlobalSignal>

<GlobalSignal> --> <Direction> <Signal>, ... <Signal>

<Direction> --> | in | out | inout

<Signal> --> <Identifier> <Combine>
  | <Identifier> = <HHExpression> <Combine>

<Combine> --> | combine <Expression>

<HHAwait> --> await <HHDelay>

<HHExpression> --> <Expression>
  | <Identifier>.now
  | <Identifier>.pre
  | <Identifier>.nowval
  | <Identifier>.preval
  | <Identifier>.signame

<HHDelay> --> ( <HHExpression> )
```

```
  | count( <HHExpression>, <HHExpression> )
  | immediate( <HHExpression> )

<HHLet> --> let <Declaration> ... <Declaration>

<Declaration> --> <Identifier> | <Identifier> = <HHExpression>

<HHSignal> --> signal <Signal> | signal [mirror] <Intf>

<HHalt> --> halt

<HHSeq> --> [ <String> ] <HHBlock>

<HHFork> --> fork [ <String> ] <HHBlock> [ par <HHBlock> ... par <HHBlock> ]

<HHEmit> --> emit <Identifier>()
  | emit <Identifier>( <HHExpression> )

<HHSustain> --> sustain <Identifier>()
  | sustain <Identifier>( <HHExpression> )

<HHAbort> --> abort <HHDelay> <HHBlock>

<HHWeakabort> --> weakabort <HHDelay> <HHBlock>

<HHSuspend> --> suspend <HHDelay> <HHBlock>
  | suspend from <HHDelay> to <HHDelay> <HHBlock>
  | suspend from <HHDelay> to <HHDelay> emit <Identifier>() <HHBlock>
  | suspend toggle <HHDelay> <HHBlock>
  | suspend toggle <HHDelay> emit <Identifier>() <HHBlock>

<HHLoop> --> loop <HHBlock>

<HHAsync> --> async ( [ <Identifier> ] ) HHBLock <HHAsyncKill>
              <HHAsyncSuspend> <HHAsyncResume>

<HHAsyncKill> --> | kill <HHBlock>

<HHAsyncSuspend> --> | suspend <HHBlock>

<HHAsyncResume> --> | resume <HHBlock>
```

```
<HHRun> --> run <HHDollarIdent> ( <HHExpression>, ... <HHExpression> ) { <HHSigRun> }

<HHSigRun> --> <Identifier>
  | <Identifier> as <Identifier>
  | <Identifier> from <Identifier>
  | <Identifier> to <Identifier>
  | *

<HHEvery> --> every <HHDelay> <HHBlock>

<HHDo> --> do <HHBlock> every <HHDelay>

<HHIf> --> if( <HHExpression> ) <HHStatement>
  | if( <HHExpression> ) <HHStatement> else <HHStatement>

<HHTrap> --> <HHLabel> : <HHStatement>

<HHbreak> --> break <HHLabel>

<HHYield> --> yield

<HHDollarExpression> --> ${ <Expression> }

<HHDollarIdent> --> <HHDollarExpression> | <Identifier>
```