Concurrency primitives in Haskell

Daniel Martí

Advisor: Jordi Petit (Computation)

June 27th, 2016
Computer Engineering - Computation

Barcelona School of Informatics (FIB)
Polytechnic University of Catalonia (UPC BarcelonaTech)

Receiving Insitution: University of Newcastle

Abstract

Asynchronous circuits are digital logic circuits not governed by a global clock signal. This makes them potentially more powerful, power efficient and modular in large systems. They have lots of real-world applications, although they are yet to be widely adopted in the industry. The main reason behind this is the lack of a simple way for developers to design asynchronous circuits to solve real problems. Current methods involve graphs that quickly grow in size, making development of complex systems infeasible. We were able to simplify and modernize the process by allowing developers to formally define their circuit in a way that is understandable and maintainable. This new method is also compatible with state of the art tools like Worcraft for the circuits to become real hardware.

Contents

1 Introduction

1.1 Asynchronous circuits

Asynchronous circuits are event-driven, i.e. they react to changes in a system when the changes occur [1], which may be at any point in the future. This makes them particularly useful for on-chip power management, where the ability to quickly respond to dynamically changing loads across the chip is essential for reliable operation and efficiency [2].

Signal Transition Graphs (STGs)[3] are commonly used for the specification of asynchronous circuits. To put it simply, an STG can be thought of a form of Petri Net, which is a graph with nodes (places) and vertices (transitions). An example of an STG (of a C-Element) can be seen at Figure 1. We can represent circuits using those mathematical models.

There exist tools such as Petrify [4] and Mpsat [5] that take an STG of a complete controller and produce a speed-independent circuit implementation [6]. This means that one can design entire circuits and chips on a computer, and have them become real hardware.

This approach to designing asynchronous circuits has poor scalability: as the system grows in complexity its monolithic specification becomes challenging to grasp and debug. STGs cannot be easily split into pieces and re-used, and thus each new design must be built from the ground up. This further adds to the design time, hence making asynchronous circuits undesirable for use in industry.

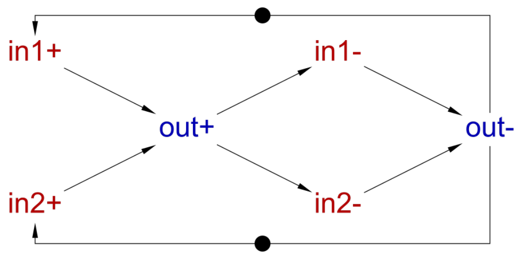


Figure 1: STG of a C-element

1.2 A new approach

To address this issue, researchers at the University of Newcastle propose a new method of asynchronous circuit design. The method splits a specification into several parts corresponding to operational modes of the circuit (scenarios). The features, constraints and requirements of each scenario (concepts), are described in a formal notation, which is being implemented as a Domain Specific Language (DSL) embedded in Haskell [7].

Concepts can be composed and one concept can be made up of multiple smaller concepts, thus supporting the design reuse at the level of system specification. Figure 2 shows a concept for a C-Element, which is equivalent to the STG in Figure 1.

A set of concepts describing the operation of a scenario is then passed to a translation program that automatically converts it into an equivalent STG, which satisfies all given concepts and can be model-checked using standard tools like Workcraft [8] as defined in [9].

When all scenarios have been translated to STGs and verified, they can be combined to produce a complete specification. All of these steps will be automatic and, ideally, transparent to the user.

This project will focus on the translation program mentioned above. This program can be thought of as the missing piece - there already are tools to work with STGs and produce complete circuit specifications, such as Workcraft. As soon as our formal specification of asynchronous circuits can be automatically transformed into an STG, developers will be able to seamlessly go from a high-level formal definition to an actual circuit usable in real life.

```haskell
cElement :: a -> a -> a -> CircuitConcept a
cElement a b c = buffer a c <> buffer b c
```

Figure 2: C-element concept

1.3 Location and context

This project is taking place in the University of Newcastle, in Northern England. With the help of teachers from UPC BarcelonaTech, the student found that a research group in England was looking for a student to help with their research. The project was agreed upon before the arrival of the student, and the entire project is being conducted in the visiting university with the help of professor Andrey Mokhov.

It should also be noted that it is the same research group that develops Workcraft. The tool is used not only in subjects and labs at Newcastle University, but also throughout many universities around Europe where circuit design is taught.

1.4 Personal motivation

The student chose the location and university because he wanted to do the entirety of the project in English, not just the paper and presentation. It was also a great opportunity to improve his English skills, living and working for half a year in the country that spawned the language.

As for the theme of the project, both functional programming and circuit design were of interest to the student. Moreover, these fields are filled with very talented people who offer many opportunities. In fact, Newcastle University is in close research collaboration with Microsoft, who are avid Haskell users and employ some of the most popular Haskell developers in the world.

2 Scope

2.1 Objectives

This section will give an overview of the objectives of this project. The temporal planning will give a more in-depth, step by step description of each of the tasks and objectives in the temporal planning.

The main objective of this project is to develop the translation program. Taking the high-level definition of an asynchronous circuit, it will generate an STG. It should be in a generic and useful format that can be imported and used by other tools.

The secondary objective is to integrate the tool that implements this program into Workcraft. That will be done in the form of a plugin for Workcraft, written in Java. It will have to integrate well with the entire program, being able to easily import a formal definition and obtain an STG directly.

2.2 Obstacles

2.2.1 Domain Specific Language (DSL)

There is plenty of design to be done, since the language is not specific enough yet. For example, signals need to be differentiated between input, output and internal signals. This is required since an STG without defined input and output signals will not be sufficient.

The DSL implementation being worked on at Newcastle University will also need work. The first implementation simply allows you to define your circuit combining concepts and then simulate it. This serves a great purpose since you can actually check that your definition is correct, but it's not what we need. The translation program needs to walk through the definition of the circuit, much like a compiler walks through an AST of a program.

2.2.2 Workcraft

Since the high-level formal definition is in a Haskell DSL, the user would need an entire Haskell compiler to make use of it. GHC weighs nearly a gigabyte, so it would be best to not have it as a hard dependency on the user's machine.

This is a rather big obstacle, as right now there is no way in Haskell to parse and evaluate Haskell source code without having to install GHC in its entirety - not even via a library.

2.3 Outside of scope

Producing a simplified version of the STG is not part of the scope of this project. This is what existing tools such as the already mentioned Petrify do. Our tool will similarly not optimize nor treat the resulting STG in any way; this is all done on the Workcraft side, with tools like Petrify.

Helping the circuit developer write better asynchronous circuits, or producing any sort of warnings if they are not well defined enough, is also not part of the project. This will be done as part of the DSL, before we grab hold of the circuit definition.

3 Methodology

The development of the project itself is going to start with the transformation of an asynchronous circuit defined using concepts into an STG by hand. This will be simply to prove that we are correct in assuming that the STGs we are going to produce are going to be valid and integrate with the rest of the workflow in designing asynchronous circuits.

The next step is going to be implementing the program, mocking the asynchronous circuit definition that the user would input and verifying that the STG outputs are valid and as expected.

Finally, we will make it possible for the developer to provide their own asynchronous circuit definition to the program, so that the program can convert it to a valid STG. This will be our first working version of the project.

3.1 Source control

Git will be at the center of our development process. We will be using GitHub, which allows for pull requests, managing of issues and milestones, releases, binary distributions and many more features.

Later in the document it will be explained how Workcraft, currently living in Launchpad, will be ported over to GitHub.

3.2 Test-driven development

Tests will be a very important part of this project. There will be tests to verify the simulation of asynchronous circuit definitions, as well as tests for the translation from such formal definitions to their STG counterparts.

We will also be using test code coverage to identify dead code paths, as well as untested code. This will greatly reduce the possibility of hidden bugs or unknown behaviour.

3.3 Continuous integration

We will integrate third party services such as Travis and Coveralls to keep the repositories on Github in shape. The former checks that the build and tests still work, and the latter checks that code coverage is either kept or improved as time goes on.

3.4 Direct feedback

We ourselves in the research group will be making use of this entire pipeline once it is finished. Aside from the feedback from our peers, there already are plenty of Workcraft users, some of which are interested in the new way to define asynchronous circuits. We will gather and act on feedback on a continuous manner.

3.5 Progress monitoring

We will be using milestones on Github to group issues that keep us from accomplishing targets in the project. One such milestone will be transforming an arbitrary asynchronous circuit into an STG, for example. Another will be to have a completely independent binary without depending on GHC, as mentioned earlier.

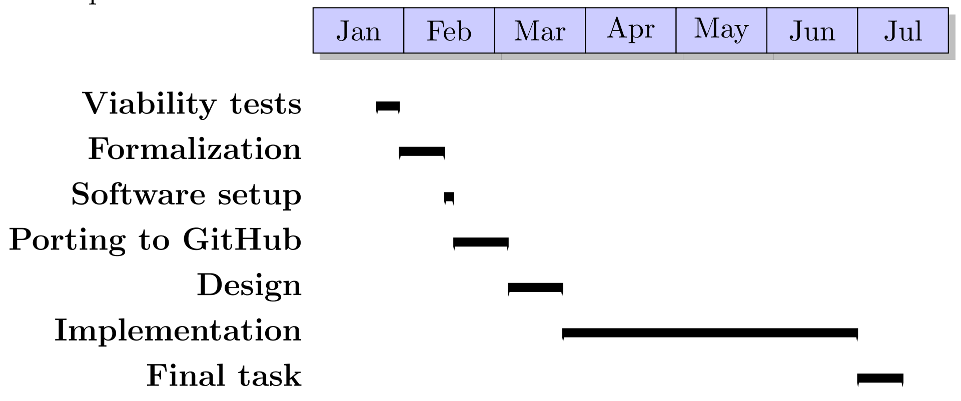
4 Temporal planning

This section concerns temporal planning and it aims to describe the tasks that are going to be executed in order to do the project. It gives an action plan that summarises the actions that have to be taken in order to finish the project in the desired time frame. However, we have to keep in mind that the following plan is subject to modifications depending on the development of the project.

The project was started on January 25th, 2016 and its deadline is June 30, 2016. At 30 hours a week, this amounts to a total of 750 hours. Therefore we have to develop our schedule given these time constraints.

4.1 Description of tasks

In this section we are going to describe the tasks that we have planned to do in order to make our project. A Gantt chart follows, showing an overview of the time spent on each section.



4.1.1 Viability tests

We spent the first week of the project confirming that the asynchronous circuits which can be defined in our Haskell DSL can all be transformed and represented as STGs. This is key for the project to be successful.

We did this simply by transforming the circuits by hand. The fact that an STG exists for each one of them - and especially that we were able to find it quite easily - proves that an program can be written.

The tests were successful.

4.1.2 Project formalization

This section was done as part of GEP. It took a total of two weeks. It is divided into these stages:

- Scope and context
- Temporal planning
- Economic management and sustainability
- Presentation

This comes before the rest of the tasks as it defines the rest of the tasks themselves.

4.1.3 Initial setup

We will need several tools. Most importantly, a development machine, which will be running Arch Linux. On it, we will need GHC version 8.0 as well as Java 1.7, which are needed by Concepts and Workcraft respectively.

We will also need several development tools, such as Git, Github and Travis, which fortunately we are already familiar with.

4.1.4 Porting from Launchpad

Before the start of this project, Workcraft lived on Launchpad. Since we plan on using Github and third party tools around it to ease collaboration and test-driven development, we need Workcraft moved over from Launchpad.

Before any coding is done, the project must be ported from Launchpad. This will be necessary to be able to integrate key parts of the development cycle such as continuous integration. This will take us two weeks.

4.1.5 Design

The next step is to design our program. We will take into account what its input will be, as well as its expected output. We also want to validate this with the rest of the research team, as they are much more familiar with the workflow of circuit design and can give valuable output.

This will take us two weeks.

4.1.6 Coding

This is the main task in our project - implementing it. It will take a total of 12 weeks. We can split it into several stages:

- Simple program (1 week): We will write a simple version of the translation program, mocking its asynchronous circuit definition input and verifying its output against the expected STG.
- Input concepts at runtime (2 weeks): Currently, the only way to use circuit concepts in the Haskell DSL in any way is to hard-code them at compile time. We will need to find a way for our program to take them as input, i.e. becoming a compiler itself.
- Taking the user's circuit (2 weeks): We will rewrite part of the Concepts software to be able to feed the user's asynchronous circuit definition to the translator program. This cannot be currently done, as the software just simulates the circuit.
- Extensive testing of the program (2 weeks): We will test our program against large inputs and all sorts of edge cases. This is mainly to make sure that any of the assumptions we may have are correct, and that our program isn't performing terribly on terms of memory or time.
- Independent binary (2 weeks): We will need to somehow produce a static, independent binary which developers can use to transform their circuit definitions in Haskell into STGs, without having to have GHC installed. This has already been mentioned as a potential obstacle, since it may not even be possible.
- Workcraft plugin (2 weeks): Finally, we will wrap up the tool in a plugin for Workcraft written in Java. This way, Workcraft users will be able to easily import a Haskell file for their circuit from within Workcraft and not worry about executing the tool and handling the STG file themselves.

The dependencies between these stages are simple - each stage depends on the one before it.

As mentioned before, we will be conducting test-driven development. So each and every one of the stages will include tests and will be measured for test coverage. Documentation will also be written.

4.1.7 Final task

In this task we are going to check that everything works as expected and we are going to prepare the delivery of the project, assuring that the documentation is correct and preparing the final presentation.

We will also ship our Workcraft plugin in a public Workcraft release. This task will take us two weeks.

4.2 Alternatives

We have stated some potential obstacles, so here we define alternatives for every one of them.

4.2.1 DSL

If we do not find a way to define which signals are input and which are output, we can make our translation program simply make assumptions, leaving those that it can't decide on up to the user to settle manually.

4.2.2 Workcraft

We have said that there is currently no way to bundle a Haskell compiler into an independent binary. At least, in a binary whose size is acceptable - at most ten or twenty megabytes. If we do not find any such way to do it, we have two options:

- The first and simplest is to require that our users install GHC on their machines.
- The second is to use a tool such as upx [10], which would let us compress most of the binary. We have run some quick tests, and this decreases size by roughly 70%, down to a size that is manageable.

4.3 Resources used

4.3.1 Hardware

- Lenovo X1 Carbon Laptop, generation 2 (2013) - used in all tasks

4.3.2 Software

- Arch Linux - used in all tasks
- GHC 8.0 - used in all tasks concerning Haskell
- OpenJDK/JRE 1.7 - used in all tasks concerning Java
- Workcraft - used to test STGs translated by hand
- Gradle - used to build Workcraft

4.3.3 Other (cloud)

- GitHub - Git repositories, issue tracking
- Travis - Continuous integration

4.4 Action plan

We already are and will continue to have weekly team meetings over lunch on Thursdays. On these, there are regular progress reports and discussion on the status of the project. They will help us keep our focus, as well as keep our mentor in the loop as to what is happening with the currently ongoing task or stage.

We have defined a clear set of tasks, which we already set expiration dates for on Github. We will use these dates and the progress in the milestones to keep track of our performance and to make sure that we finish the project in the time we have.

5 Budget and sustainability

This section treats the budget and the sustainability of the project. It contains a detailed description of the costs of the project, describing both material and human costs.

As in previous sections, this budget described is subject to modifications depending on the development of the project.

5.1 Budget estimation

In this section we estimate the budget needed in order to make our project possible. We are going to divide the budget in three sections, depending on the kind of resources that we are taking into account.

To calculate the amortisation we are going to take into account two factors, the first one being the useful life and the second one being the fact that our project is going to last for approximately six months.

5.1.1 Hardware resources

The following table contains the costs of the hardware that we are going to use in the development of the project.

Product	Price	Useful life	Amortisation
Laptop	2000€	4 years	250€
Virtual Private Server	200€	2 years	50€
Total	2200€		300€

5.1.2 Software resources

The table 2 contains the costs of the software that we are going to use in the development of the project.

Product	Price	Useful life	Amortisation
Arch Linux	0€	Indefinite	0€
GHC 8.0	0€	Indefinite	0€
OpenJDK 7	0€	Indefinite	0€
Latex	0€	Indefinite	0€
Total	0€		0€

5.1.3 Human resources

A single person will be working on the project, working on both the planning and designing aspects of it as well as all the coding and testing. All of these tasks amount to a total of 750 hours, as previously mentioned. At an hourly rate of 20€/hour, this will amount to 15.000€.

5.1.4 Total budget

Using data shown in the tables above, we can use the following table to describe the total cost of the project.

Concept	Cost
Hardware resources	300€
Software resources	0€
Human resources	15.000€
Total	15.300€

5.2 Budget control

Our budget could need a modification if the development of the project makes it impossible to follow the plan established, which is something that is likely to happen.

We could have difficulties in our project, although it is unlikely that we will need any hardware resource apart from the resources that have been already listed in the budget estimations shown in the previous section. We might need more software resources, but we will probably cover those needs with free and open source software just like we have already been doing.

Since we have no human resource costs, even if further programming or design hours are needed, they will come at no budget cost increases.

5.3 Sustainability

In this section we are going to evaluate the sustainability of our project in three different areas:

- Economic
- Social
- Environmental

5.3.1 Economic sustainability

In this document we already can find an assessment of the costs of our project, taking into account both material and human resources.

The cost stated in the Budget estimation section of this document could be the only cost spent in the project, since we aim to create a working program, so it should not need maintenance in the case that we do not create any update.

It would be difficult to do a similar project with a lower cost. We are not spending any money on software nor on human resources, so the only way to reduce costs would be to use worse hardware. Which could be accomplished, but at the expense of worse conditions for the person doing the actual development of the project, potentially worsening its quality.

We can not assure that the cost of the project would make it competitive, but we can say that it depends on how it is going to be distributed. We could make it a free application, so it would be very competitive, but then we would need a way (for example, advertisements) to get paid every time that the application is downloaded so we can recover the money invested. In the other hand, we could make it a paid application, but its price should not be high.

It is not the case, but is important to remark that this project could have been executed with collaboration of companies in the hardware manufacturing industry, since they may be interested in developing an application like this very one.

5.3.2 Social sustainability

The application that we are going to develop in this project is going to be used in the countries which are developed enough so that everyone can have access to a computer, since our goal is that it can be used by everyone who can have access to a decent computer.

Everyone desiring to design an asynchronous circuit is going to be able to use the program resulting of the development of our project. The program gives the developer a way to design it in a simple and scalable way - before its creation he or she had to use a graph manually, which would get complex quickly and not scale well.

5.3.3 Environmental sustainability

The resources used in the project have been detailed in the Budget estimation in this document.

During all the development of our project we are going to have a laptop running. The devices used and the energy spent running them are going to be the only resources used.

Knowing that, we can estimate the energy spent developing the project. We can suppose that our computer consumes an average of 40 W when we are working in the project. Since we need to use the computer in all the development of the project, which is 750 hours long, the energy we need is 30 kWh, which is equivalent to 11.3 kg of CO2. It is a moderate amount of energy, but since we need to make constant use of a computer there is no way to reduce it.

6 Design

In this section we will come up with an outline of the program, designing both how it will work on its own as well as how it will integrate with Workcraft.

6.1 Command line program

For the program to be useful as a standalone command line utility, users will have to be able to feed it input via files or standard input. That is, the circuit concepts described in the Haskell DSL.

It will also have to produce its result to an output file or standard output. If the input concept is valid Haskell that compiles, the program should work correctly without any human interaction.

Note that the program does not need to specifically run any checks to verify that the circuit is well defined, as that is already done thanks to the Haskell DSL. Any incorrect definition of a circuit will simply not compile.

This type of input/output tool follows the Unix philosophy of doing one thing and doing it well, so it will be easy to reuse it at later stages.

There are two major obstacles to face in the implementation of this program. Unfortunately, we cannot investigate them much further without diving into implementation details. Thus, these obstacles are dealt with in the implementation stage.

6.2 Workcraft plugin

As stated in the project timeline, we will be writing a Workcraft plugin that will allow users to input a circuit concept defined in the Haskell DSL directly into Workcraft, obtaining an STG that they can then test, simulate and otherwise work with.

Workcraft already has a number of plugins, adding up to 17. These are all built on top of the `WorkcraftCore` package, which provides a base and a set of interfaces to interact with the whole program and graphical interface.

Thankfully, there are already a handful of plugins that simply wrap command-line binaries. For example, there already is a plugin that wraps Petrify, which we have mentioned before. This sort of plugin is rather simple, as it does not have any complicated logic in itself.

Since Petrify also works on input and output files, our approach of writing the command-line program in this way will conjugate well with Workcraft.

7 Implementation

7.1 From Launchpad to GitHub

As described in the temporal planning, one of the first key tasks will be moving Workcraft from its Launchpad [22] repository to a Github [12] one. This will simplify the workflow in the team, as Github integrates tools such as code review better. But most importantly, it will allow for the use of key components such as continuous integration via Travis.

Launchpad and GitHub use different source control management systems - Bazaar [11] and Git [23], respectively. Luckily enough, a Bazaar repository is very similar to a Git one, and can be semi-automatically converted to one. The bigger problem lies in all the information on Launchpad outside of the source control, such as bug reports and milestones.

Over on GitHub, we register the Tuura organization [24] where all of the research group's code and products will live from this point onwards.

7.1.1 Porting information over

To port all the information over from Launchpad, scripting will be needed. Python is the language choosen for the job, as development speed is a much higher priority than runtime performance for such a one-day tool.

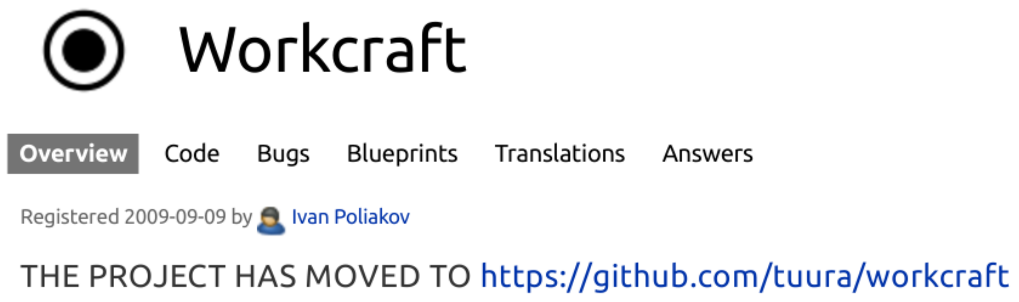
The key components that the software will do are as such:

- Download Bazaar source repository from Launchpad
- Download all bugs, blueprints and milestones from Launchpad
- Convert the Bazaar repo to a Git repo
- Fix all the commit author names and e-mail addresses to match commits with GitHub users properly
- Assign all bugs, blueprints and milestones unique ids that will be used when importing them on GitHub
- Rewrite all the Git commit messages to port all references to bugs, blueprints and milestones
- Use a bot GitHub account to push the git repository

- Use a bot GitHub account to submit all the bugs and blueprints
- Use a bot GitHub account to submit all the milestones and assign the respective issues to them

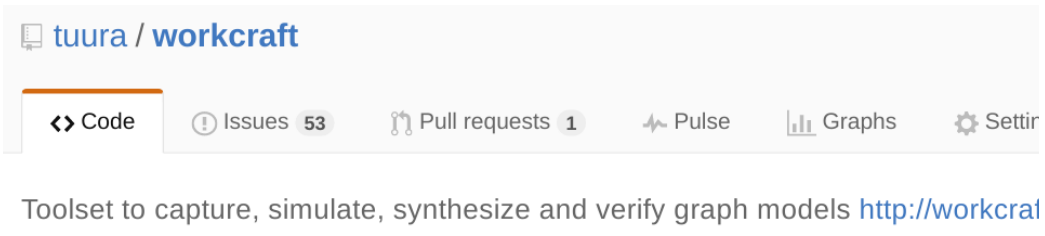
In total, it took little over a week to write the tool and finish the task. The tool itself took about an hour to do the entire thing, mainly because GitHub has rate limits in place that made the tool sit around for most of the time.

As one can expect, there were lots of little details that needed correction. After a handful of retries from scratch, the switch over to GitHub is done. The new repository can be found at tuura/workcraft [13]. The old and new repositories can be found at Figure 3 and Figure 4, respectively.



THE PROJECT HAS MOVED TO https://github.com/tuura/workcraft

Figure 3: Abandoned Launchpad repository [14]



Toolset to capture, simulate, synthesize and verify graph models http://workcraf

Figure 4: New GitHub repository [13]

We also added issue and pull request templates [15], further simplifying the process for users and developers who wish to contribute.

7.1.2 Continuous integration

As described previously in the methodology, we want to use continuous integration as part of our development process. As such, we want to set up Workcraft with Travis [16], which will let us run a number of commands or scripts at each commit and pull request.

Workcraft already has some JUnit unit tests, which Travis will run. On top of these, we add:

- Testing on multiple JDKs (OpenJDK 7, OracleJDK 7 and 8)
- Format checks via Checkstyle [18]
- Java checks via PMD [19]

Keeping a consistent code format across the codebase is very important to keep the project maintainable and easy to develop, especially since there are multiple people involved. And PMD adds many static code analysis checks that many IDEs already perform, but with the advantage that no GUI is needed and especially that they can be checked by Travis.

The Checkstyle configuration file for Workcraft can be found in the git repository [20], as can the PMD one [21].

7.2 Simple program

The main objective of this task is to get familiar with what the inputs and outputs of our program will be. The first question we should ask is how we will feed the output STG to Workcraft.

7.2.1 STG format

We will use Workcraft to draw simple STGs and observe the export formats, to see which one is the correct one for us.

The format should:

- Be in plain text, to ease debugging and development
- Be easy to generate
- Not be overly verbose
- Not contain any graphical nor UI information

Workcraft supports multiple import and export formats for STGs. One such example is the .work format, which is the default in use when exporting an STG. It is way too complicated for our needs, though.

Another format, called dotG, fits our needs. It is plaintext, only contains the STG - no graphical interface information - and it is represented in simple plain text. We can see an example of an STG in the dotG format in Figure 5.

```
.model Untitled
.inputs in1 in2
.outputs out
.graph
in1+ out+
in1- out-
in2+ out+
in2- out-
out+ in1- in2-
out- in1+ in2+
.marking {<out-,in1+> <out-,in2+>}
.end
```

Figure 5: C-element STG in the dotG format

7.2.2 Required information

Once we know what we want to produce and how, the next task is determine what information is needed to produce such output. We will need:

- A list of all the unique signals
- A list of all the unique arcs
- The initial state for each of the signals

Optionally, we could also set the type of each signal - input, output or internal - in the STG. Although that is not necessary to make a standalone circuit into a working STG.

7.2.3 Unit tests

Once we know what information is required and how to transform it into an STG, we now write a set of tests with inputs and expected outputs. The purpose of such tests is to confirm our design so far.

The program is very simple at this point, as it does little more than encoding the given information into a dotG file. Once we wire this into our real input, circuit concepts defined in the Haskell DSL, the program will become more complex.

The tests work properly, and the generated dotG files are successfully imported by Workcraft.

7.3 Input concepts at runtime

As outlined in the temporal planning, our next step is to remove the requirement that concepts be input at compile time. This is cumbersome, as the entire DSL toolset needs to be compiled with the user's concept already in place. That means that if the concept needs changing, the user must recompile everything.

We need to write a program that takes input bytes, either from standard input or a file, compiles them as a Haskell module representing the user's concept, and wires that up to the Concepts toolset [29] for it to be simulated or treated in any way.

7.3.1 Writing a compiler

The biggest challenge here is that, effectively, we need to write a compiler. Thankfully, GHC itself is a set of Haskell modules we can import. Less fortunately, its API is cumbersome to work with and changes quite drastically from release to release. Even if we got it to work with GHC 8.0, it will very probably not compile with GHC 7.8 or other relatively recent versions.

Ideally, we would want a library that wraps around the GHC API to solve both of those problems. It would simplify its usage, and provide a layer of compatibility across multiple GHC versions, allowing our program to compile and work under multiple GHC versions at once.

7.3.2 Hint library

Our research leads us to Hint [25], written by Daniel Gorin and hosted in a Darcs source code repository [26]. This library aligns very well with what we described above - it is a wrapper around the GHC API, and provides compatibility with GHC versions from 7.4 through 7.10.

Unfortunately, this library does not yet support the latest GHC release - 8.0 - and it has been seemingly abandoned since 2014, without any activity for more than two years.

As such, we e-mailed Daniel offering to take over maintenance of the package, and he agreed. As such, Hint now lives on Github [27] and has seen two major releases - 0.5 and 0.6 - since the takeover in maintenance, including contributions from many in the Open Source community.

7.3.3 Changes to Hint

A number of changes were required in Hint to allow its use in our program, and were made in the following weeks:

- GHC 8.0 support
- Do not error if GHC returns any warnings
- Allow unsafe interpretation of expressions
- Allow providing a GHC libdir at runtime

The first two are straightforward - GHC 8.0 support is required since that's the version we will be using, and we do not want our compiler to error if any warnings are found. The latter was a bug in earlier versions of Hint.

The third concerns how one can use Hint to compile modules from an array of bytes. The compilation is simple - the tricky part comes with extracting information from the compiled module. For instance, the example below will use the function `foo` as defined in the module in Figure 6, compiled at runtime in Figure 7. It has been tested to run on GHC 8.0 with Hint 0.6.0 installed.

```haskell
module Foo where

foo :: Int -> Int
foo n = n + 1
```

Figure 6: Foo.hs

```haskell
module Main (main) where

import qualified Language.Haskell.Interpreter as GHC

main = GHC.runInterpreter doInterpret

doInterpret :: GHC.Interpreter ()
doInterpret = do
    GHC.loadModules ["Foo.hs"]
    GHC.setTopLevelModules ["Foo"]
    --
     foo <- GHC.interpret "foo" (GHC.as :: Int -> Int)
     let n = 2        {- 2 -}
     let n2 = foo n {- 3 -}
     return ()
```

Figure 7: Sample Hint usage

Note that we have to know the type of the interpreted name at compile time. We need to be able to supply the type at runtime, since circuit concepts might take any number of signals to work on. As such, we added an `unsafeInterpret` variant that allows this. It is more powerful, but it is also easier to get the program crashing if the type is not formatted correctly. A usage sample can be found at Figure 8.

```haskell
module Main (main) where

import qualified Language.Haskell.Interpreter as GHC
import qualified Language.Haskell.Interpreter.Unsafe as GHC

main = GHC.runInterpreter doInterpret

doInterpret :: GHC.Interpreter ()
doInterpret = do
    GHC.loadModules ["Foo.hs"]
    GHC.setTopLevelModules ["Foo"]
    --
    foo <- GHC.unsafeInterpret "foo" "Int -> Int"
    let n = 2         {- 2 -}
    let n2 = foo n {- 3 -}
    return ()
```

Figure 8: Sample Hint usage with unsafe variant

And lastly, the fourth concerns the GHC libdir. That is a path to a directory that holds all the libraries (modules) that GHC will use when compiling modules that import others. For example, that is where Prelude will be available.

In previous releases of Hint, that would be hard-coded to the libdir path of the GHC installation that compiled Hint, via the ghc-paths library [28].

That is very inflexible, and means that a truly independent and static Haskell compiler using Hint is not possible, as the libdir path is hard-coded at compile time. As such, the hard-coding was made into a fallback, allowing the user to provide a libdir at runtime if they wish to.

7.3.4 Changes to Concepts

All of the above was integrated into the Concepts toolset, meaning that it now includes binaries that can take concepts (Haskell DSL modules) at runtime, compile and use them however they want to. To prove that this worked, we implemented the first of such binaries, that simply simulates a concept interactively. A sample of the usage of the binary can be found at Figure 9.

```
 $ runghc simulate/Main.hs examples/Concept.hs

Signal names: ["A","B","C"]
Circuit signals: 3

State: 000
Enabled: [A+,B+]
Do: A+
State: 100
Enabled: [B+]
Do: B+
State: 110
Enabled: [C+]
Do: C+
State: 111
Enabled: [A-,B-]
```

Figure 9: Simulate executable usage with a C-element

The command above can be executed from the root of the Concepts repo [29], given that both Hint and Concepts have been installed via Cabal [30].

7.4 Taking the user's circuit

Now that we know what information we need to create an STG, and we know how to use a circuit concept provided at runtime, we need to write an executable that takes a concept from a Haskell DSL module file and produces an STG in the form of a dotG file.

7.4.1 Changes to Concepts

The first step in this process is to adapt the Concepts toolset to be able to gather the required information from the concepts (circuits) that the user provides. Up to this point, the Concepts toolset can only interpret concepts to simulate them.

That means that when two concepts are combined, as in our cElement example earlier, they are collapsed into a single new concept. The behaviour is as expected and works for simulation purposes, but not for our purposes - it is no longer possible to take apart the concept in pieces, looking at each of its components separately.

As we outlined in the required input for our translation to an STG in the dotG format, we need the arcs between all the signals (nodes). Such arcs are themselves concepts, the most basic kind - they are represented with the string >. For example, we can combine two arcs to form a buffer as seen in Figure 10.

```
buffer :: a -> a -> CircuitConcept a
buffer a b = rise a ~> rise b <> fall a ~> fall b
```

Figure 10: Buffer concept

<> is used to combine the two concepts at each end, forming a new concept.

What we need to do is list all the > concepts that form any concept. As said, the simulation uses a Haskell Monoid that combines concepts by creating new concepts that AND all the components. And since it represents arcs as functions, an arbitrary concept combined with this Monoid contains a function that can simulate all the transitions and arcs, but can not list them.

As such, we will develop a separate Monoid that will behave differently. Instead of representing > (arcs) as a function, it will represent them as tuples from transition to transition as seen in Figure 11.

```
rise a ~> rise b
{- would become -}
(Transition a True, Transition b True)
{- and -}
fall b ~> fall c
{- would become -}
(Transition b False, Transition c False)
```

Figure 11: Arcs as tuples

Each concept will contain a list of such tuples. Then, when two concepts are to be combined in the Monoid, the lists will simply be appended. Were there to be any duplicates, we can deal with them at a later stage.

This simple approach will let us parse and evaluate a concept, and then obtain all the atomic causalities or arcs from it. This was one of the obstacles described, in the DSL category. We have successfully overcome this obstacle.

7.4.2 Executable binary

Now that we know how we will extract the needed information from a concept, we need to write an executable program similar to the simulate executable that we wrote with Hint earlier.

Thankfully, we can reuse much of the code. The only difference is that when using the concept, we must use our new "inspection" monad instead of the simulation one.

Inspecting the concept also gives us the initial states of each of the signals, as that is baked into a circuit concept in our Haskell DSL. And lastly, for the list of unique signals, we are already gathering them via Hint since we needed to list them for our simulation program.

Now that we have all the bits needed for our previously defined and written program, we simply have to wire it up. Figure 12 shows a sample of the usage of the program.

```
 $ runghc transform/Main.hs examples/Concept.hs
.model concept
.graph
A+ C+
A- C-
B+ C+
B- C-
C+ A-
C+ B-
C- A+
C- B+
.marking {<C-,A+> <C-,B+>}
.end
```

Figure 12: Transform executable usage with a C-element

This STG is complete with all the arcs and the initial states. Now we face the other obstacle defined in the DSL category - knowing which signals are input and which are output.

Since STGs in Workcraft can work without this information in them, we are postponing this part as it is not a crucial part in finishing our project. Once we get the rest of key components working well, we will move on to it.

7.5 Extensive testing of the program

As described in the timeline, we now need to set up a series of test cases with user inputs and desired outputs. The two goals here are to test for edge cases and to ensure that no current features are broken in the future, i.e. regression tests.

We set these up via Cabal, so that they can be easily run via `cabal test`.

One of the edge cases that we stumble upon is causalities involving more than two signals. So far we have only seen causalities involving two signals, which can be represented via a single arc from the first transition to the second transition.

But one may specify causalities from multiple transitions into a single transition, either with the AND or the OR operand. The former can be rewritten as multiple simple causalities, which we can already handle. The latter, though, is not as simple. We will need to figure out a way to represent it in terms of simple arcs.

Fortunately, though, a wide variety of circuits can be expressed in simple causalities. Only certain very specific cases need an OR causality involving more than two signals. As such, we can also delay this piece of work as it is not a key component in our tool that blocks moving forward.

7.6 Independent binary

This section concerns the remaining obstacle defined when formalising this project. Now that we have developed our programs that compile and evaluate concepts using Hint, we can investigate this further.

Our programs are compilers themselves, but they still depend on GHC being installed. What we want is for them to be truly static and independent - for them to work even if the running system is a Linux box only with the bare minimum installed programs and libraries, such as an interactive shell.

The programs can be compiled statically, and they weigh around 50 MiB. They are still not truly independent though, as they still require a path pointing at the directory holding all the Haskell libraries (modules) that any compiled program might import.

When making improvements to Hint, we already made it possible for it to take such a libdir at runtime. So we have two options:

- Use the libdir from the user's GHC setup

- Bundle a libdir along with our executable in a tarball

The first option is a bad idea on two accounts. First, the user's GHC version would have to match the GHC version to compile the binary, which is not portable at all. But most importantly, we would require that the user have GHC installed again, as having only a libdir does not make much sense.

The second option seems like a good idea at first sight. If we were to include the base library, which includes basic modules such as Prelude, the size of our distributed tarball would rise to just under 100 MiB. But even if the increased size is acceptable, this setup would be very crippled.

If the user's concept were to use any module outside of the base package, it would simply not work. This greatly limits the possibilities of writing circuits in the form of concepts in our Haskell DSL.

Having those considerations in mind, we decide to give up on the possibility to not require that the user have GHC installed. After all, if the user wants to develop circuits using Haskell, they most likely already have GHC installed or would not mind installing it.

This obstacle is not overcome simply because we have deemed the disadvantages to be greater than the advantages.

7.7 Workcraft plugin

As mentioned in the design stage, there already exist Workcraft plugins that simply wrap around command-line utilities that work on input and output files. One such plugin is the `PetrifyPlugin`.

Like it, our plugin will not have much complex logic in itself. All it will do is:

- Add an option to the "Open" menu to deal with circuit concepts (Haskell files)
- Feed the file to our standalone program and obtain the output STG in the dotG format
- Load the STG in the dotG format into memory (Workcraft already supports the format)
- Display the STG in a new work (document) on the graphical interface

There are various subtleties and possible issues to take care of, though. We describe them below.

7.7.1 GHC dependency

Since we require GHC to be installed on the user's machine, the plugin should check whether it is installed and provide a useful error message if it isn't. It should also point the user to instructions on how to install GHC properly on their machine.

The plugin should also check what GHC version is installed in the user's machine, to make sure that it's new enough for it to be able to run our translating program. In particular, our use of Hint and the Concepts libraries (our Haskell DSL) means that we require at least GHC 7.8.

7.7.2 Program compilation

Since our program is itself a compiler, as previously explained it will need access to a GHC setup to make use of any packages and libraries that the compiled module may use. As such, the program must be compiled in the user's machine, since it must be compiled for the user's platform and architecture as well as with the exact same GHC version.

Since we require GHC anyway, the simplest solution is to make the plugin compile the program on first use. It should also re-compile it if the GHC version changes.

7.7.3 Simplify STG

The STGs produced by our translation from a concept will often be verbose. They may have lots of unnecessary places and arcs, and there might be a simpler STG that is equivalent to it. This is exactly what Petrify does, which Workcraft already has support for via a plugin.

Instead of reinventing the wheel, our plugin will depend on the Petrify plugin and make use of its features to get the STGs to be simple and understandable by the time they get shown on the user's screen.

7.7.4 Shipping into a release and feedback

The main reason behind writing a plugin is for the tool to be easily accessible through a graphical interface to all Workcraft users. As such, the plugin should get shipped into a Workcraft release along with the rest of the plugins.

Once that is done, feedback can be gathered from real users, both on the Haskell DSL as well as its translation into STGs for use in Workcraft.

Since this document and the presentation of the project are all to be done three weeks prior to the end of the project, this final stage is not yet finished. As such, we will use our limited internal feedback on an unfinished version of the plugin to make our conclusions in the next section.

8 Conclusions

This project was ambitious because the student had limited experience both in Haskell and in the world of circuit design. At the same time, the objectives and timeline were realistic and the potential obstacles had been in mind since the very beginning.

Given that the project has followed its timeline closely and has reached its goals successfully, one can say that it has been a success. The fact that one of the obstacles hasn't been overcome may seem like a failure, but it is not - a decision was made balancing the pros and cons of the possible solution that was found to the obstacle.

The student has learned a great deal about Haskell and the world of circuit design thanks to the mentorship and help of the research group at the visiting organization. As such, this project has also been a personal success and a good learning experience.

Finally, the shipped product meets its initial requirements and is actually useful, as envisioned when formalising the project. It is worth keeping in mind that the time invested in the software will be of use to researchers and students using Workcraft at many universities including Newcastle university.

From our internal feedback, the tool meets its purpose correctly and does not get in the way. It is nothing too special on its own, but together with the circuit design in the Concepts DSL in Haskell it opens up new ways to design simple yet reusable circuits in the real world.

References

[1] Jens Sparsø and Stephen B Furber. *Principles of asynchronous circuit design: a systems perspective.* Springer Netherlands, 2001.

[2] Jonathan Audy. *Navigating the path to a successful IC switching regulator design.* Tutorial at IEEE International Solid-State Circuits Conference (ISSCC), 2008.

[3] A. Yakovlev, L. Lavagno,. *A unified signal transition graph model for asynchronous control circuit synthes* ICCAD, 1992.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis for Asynchronous Controllers and Interfaces.* Springer, 2002.

[5] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. *Detecting state encoding conflicts in STG unfoldings using SAT.* Fundamenta Informaticae, 62(2):221-241, 2004.

[6] W. Bartky D. Muller. *A theory of asynchronous circuits.* International Symposium of the Theory of Switching, 1959.

[7] P. Hudak. *Building domain-specific embedded languages.* ACM Computing Surveys, 28(4):196, 1996.

[8] `http://www.workcraft.org/`, March 2016

[9] Ivan Poliakov, Victor Khomenko, and Alex Yakovlev. *Workcraft - a framework for interpreted graph models.* In Applications and Theory of Petri Nets (ATPN), pages 333-342. Springer, 2009.

[10] `http://upx.sourceforge.net/`, March 2016

[11] `http://bazaar.canonical.com/en/`, April 2016

[12] `https://github.com/`, April 2016

[13] `https://github.com/tuura/workcraft`, April 2016

[14] `https://launchpad.net/workcraft`, April 2016

[15] `https://github.com/blog/2111-issue-and-pull-request-templates`, April 2016

[16] `https://travis-ci.org/`, April 2016

[17] `https://travis-ci.org/tuura/workcraft`, April 2016

[18] `https://github.com/checkstyle/checkstyle`, April 2016

[19] `https://github.com/pmd/pmd`, April 2016

[20] `https://github.com/tuura/workcraft/blob/v3.1.0/config/checkstyle/checkstyle.xml`, April 2016

[21] `https://github.com/tuura/workcraft/blob/v3.1.0/config/pmd/rules.xml`, April 2016

[22] `https://launchpad.net/workcraft`, March 2016

[23] `https://git-scm.com/`, April 2016

[24] `https://github.com/tuura`, March 2016

[25] `https://hackage.haskell.org/package/hint`, April 2016

[26] `http://hub.darcs.net/jcpetruzza/hint`, April 2016

[27] `https://github.com/mvdan/hint`, April 2016

[28] `https://hackage.haskell.org/package/ghc-paths`, April 2016

[29] `https://github.com/tuura/concepts`, April 2016

[30] `https://www.haskell.org/cabal/`, April 2016