

Concepts tool - Manual

Jonathan Beaumont

`j.r.beaumont@ncl.ac.uk`

School of Electrical and Electronic Engineering, Newcastle University, UK

Last updated September 15, 2016

Contents

1	Introduction	2
2	Installation	2
2.1	Concepts tool requirements	2
2.2	Workcraft requirements	2
2.3	Installing the tool	3
3	Using the tool from command line	3
4	Using the tool from Workcraft	4
4.1	Translating and authoring concepts . .	4
4.2	Importing concepts directly	7
4.3	Errors	8
5	Writing concepts	8
5.1	Concepts file layout	9
5.2	Built-in concepts	10
6	Features to be implemented	11

1 Introduction

Concepts are a formal method of specifying asynchronous circuits. With concepts, one can describe behaviours of a circuit and its environment at various levels, with the aim of defining the behaviours in terms of gates, protocols or simple signal interactions. Using this method, one can split a design into scenarios, based on operational modes, individual functions, or any other method a user decides. These can be specified separately, using concepts, and then combined to provide a full system specification. Through this, reuse is promoted, allowing concepts to be reused between scenarios and even entire designs. More information on the theory of concepts can be found in [1], the latest version of which can be found here.

In this document, we will discuss the associated tool for concepts. This open source tool is written in *Haskell*, and contains both the library for concepts, including abstract concepts and circuit concepts built upon the abstract, and a tool for translating concepts into *Signal Transition Graphs* (STGs) [2] [3]. These are commonly used for the specification, verification and synthesis of asynchronous control circuits in the academic community, and they are supported by multiple EDA tools, such as PETRIFY [4], MPSAT [5], VERSIFY [6], WORKCRAFT [7][8], and others. The aim of this tool is to design and debug concepts, and then translate them to STGs, where they can be used with these tools.

The latest version of this tool, and this manual, can be found in the GitHub repo. Any bugs or issues found with the tool can be reported here. This tool is also distributed as a back-end tool for WORKCRAFT. This version of the concepts tool will be the latest version that works correctly with WORKCRAFT. The latest version of WORKCRAFT, which also features tools such as PETRIFY and MPSAT, can be downloaded from <http://workcraft.org/>.

2 Installation

The concepts tool, as well as WORKCRAFT are available for *Windows*, *Linux* and *Mac OS X*. The installation instructions for all of these operating systems are the same. We will be referring to directories using the forward-slash character ('/') as a separator, however for *Windows*, replace this with a back-slash character ('\').

If choosing to use the concepts tool on its own, this must be downloaded from the GitHub repo. If you choose to use this tool as part of WORKCRAFT, download this from the WORKCRAFT website. Once downloaded, extract the contents of the folder, and move them to a directory you wish to run them from.

2.1 Concepts tool requirements

The concepts tool is written in *Haskell*, and uses *Stack* to install the necessary compiler and dependencies. If necessary, please download stack for your operating system, available from <https://haskell.org/downloads#stack>, and follow the instructions to install this.

2.2 Workcraft requirements

WORKCRAFT is written in Java, and the latest version of the *Java Runtime Environment* (JRE) needs to be installed to run. This can be downloaded from <http://java.com/en/download/>. If needed, download the JRE installer, and follow the instructions to install this.

While the concepts tool is distributed with WORKCRAFT, it still needs to be built by *stack* in order for it to be run. See Section 2.1 for requirements for the concepts tool.

2.3 Installing the tool

Once either the concepts tool or WORKCRAFT has been extracted and moved to the desired directory, using command line, navigate to the concepts tool directory, or if using WORKCRAFT, navigate to the WORKCRAFT directory, and then navigate to the concepts tool directory, found in `tools/concepts` (for *OS X*, the WORKCRAFT directory is located within the `Workcraft.app` contents folder. The concepts tool will be found at `Contents/Resources/tools/concepts`.

Now, the process of installing the tool is the same, regardless of how you aim to use the concepts tool. First of all, let's setup stack. To do this, run:

```
$ stack setup
```

This will prepare stack to install the concepts tool. Now, to build and install the concepts tool, simply run:

```
$ stack build
```

If this process completes successfully, the tool will now be installed and ready to be used.

3 Using the tool from command line

With the tool installed, we can now start to use it to translate concepts to STGs. We will begin by discussing how it is used from command line. Usage as a back-end tool in WORKCRAFT is discussed in Section 4.

The standard command for the tool is as follows:

```
$ stack runghc <path-to-translate> <path-to-concepts-file>
                                     [--stack-yaml <path-to-stack-file>]
```

The three parts of this are as follows:

- **stack** - This will ensure that the dependencies and compiler are installed when running.
- **runghc** - This runs the translation function.
- **<path-to-translate>** - This is file path pointing to the translate code file, which performs the necessary operations to translate concepts to STGs.
- **<path-to-concepts-file>** - This is the path pointing to the file containing the concepts to be translated.
- **[--stack-yaml <path-to-stack-file>]** - This is optional. If running the tool from outside of the directory, the path to the stack file needs to be given.

When running the concepts tool from command line, as long as the paths to the translate code, the concepts file and the `stack.yaml` file are correct, it doesn't matter which directory the tool is run from. The `stack.yaml` file is located in the base of the concepts directory.

Now, let's use one of the examples included with the concepts tool to show the usage. These can be found in the `examples` directory of the concepts tool. For this section, we will assume that we are currently in the concepts tool directory. The example we will use is the file titled "`Celement_with_env_1.hs`". This has the ".hs" file extension, as it is in fact a file using Haskell code, and all files containing concepts should feature this file extension. To translate this concepts file to an STG, the following command must be run:

```
$ stack runghc translate/Main.hs examples/Celement_with_env_1.hs
```

When the translation is complete, the tool will output the following:

```
.model out
.inputs A B
.outputs C
.internals
.graph
A0 A+
```

```

A+ A1
A1 A-
A- A0
B0 B+
B+ B1
B1 B-
B- B0
C0 C+
C+ C1
C1 C-
C- C0
A1 C+
C+ A1
B1 C+
C+ B1
C1 A-
A- C1
C1 B-
B- C1
A0 C-
C- A0
B0 C-
C- B0
C0 A+
A+ C0
C0 B+
B+ C0
.marking {A0 B0 C0}
.end

```

This output is the STG representation in *.g* format. *.g* files are a standard type used as input to tools, such as PETRIFY, MPSAT, and WORKCRAFT. Therefore, this output can be copy-and-pasted into a file, and saved with the file extension *.g*, and then used as input to these tools.

The concepts tool can be used in a similar way, ensuring that the file paths to the translate code file, and the concepts input file, are correct. Section 5 contains information on how to layout a concepts file, to avoid as many errors as possible.

Any errors that occur during the translation process will produce errors referring to the problematic lines of signals of the concepts that are problematic.

4 Using the tool from Workcraft

This section will discuss how to use the concepts tool from within WORKCRAFT. There are many other features of WORKCRAFT, both as part of the STG plug in, some of which I will discuss in the context of concepts here, and as part of other modelling formalisms. More information on these can be found at <http://workcraft.org/>.

4.1 Translating and authoring concepts

First of all, WORKCRAFT must be started. This can be done by running the start up script, located in the WORKCRAFT directory in *Windows* and *Linux*. In *Windows*, this script is named “**workcraft.bat**”. In *Linux*, it is simply “**workcraft**”. In *OS X*, WORKCRAFT can be started instead by double clicking the WORKCRAFT icon, which is the app container for the necessary files.

When workcraft starts, you will be greeted by blank screen, as seen in Figure 1.

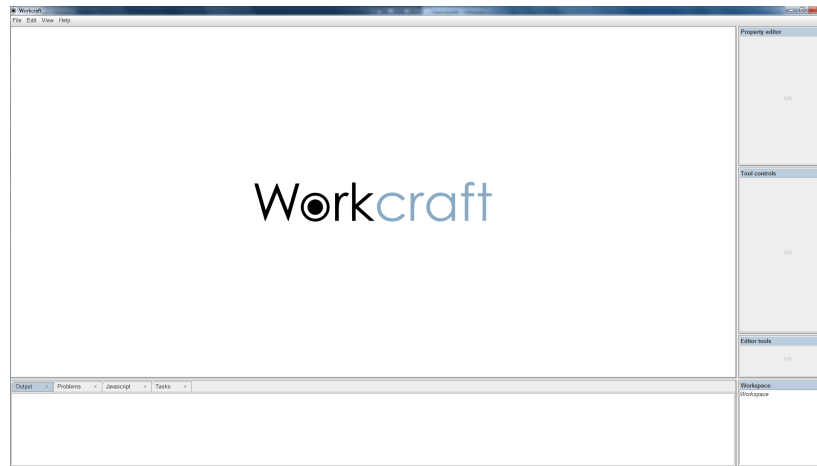


Figure 1: Workcraft immediately after starting.

Now, we need to open a new work, specifically a new STG work. Open the "New work" dialog using the menu bar, **File** -> **Create work...**, or by pressing **Ctrl-N** (CMD-N on *OS X*). This will bring up a menu as seen in Figure 2.

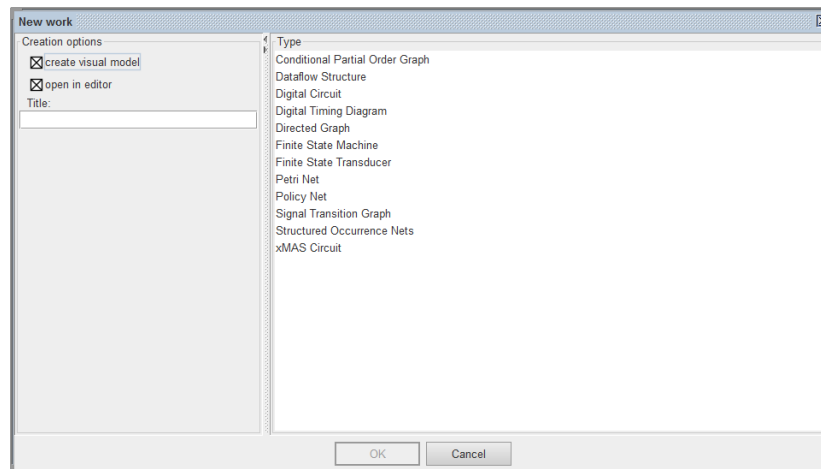


Figure 2: The create work window.

In this window, select "*Signal Transition Graph*" and click the "OK" button at the bottom of the window. This will open a blank workspace in which we can create an STG, which will look similar to Figure 3.

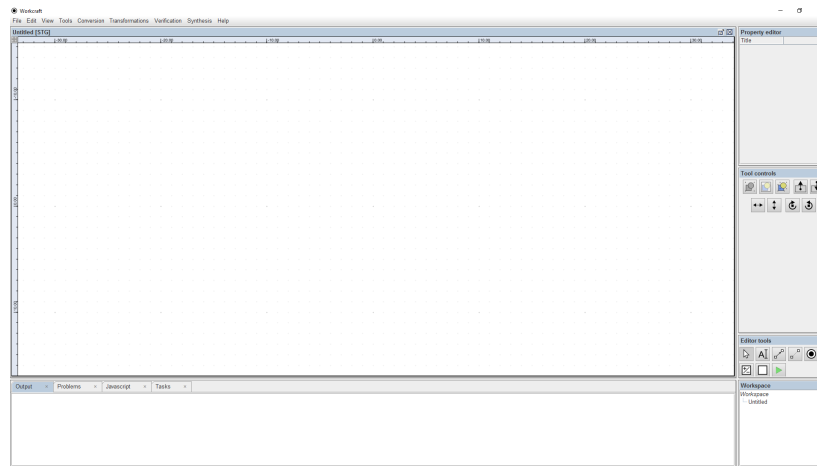


Figure 3: A new STG workspace

Now, we can start translating concepts. To do this, first we need to open the concepts dialog. This is done from the menu bar, by selecting the “*Conversion*” menu, and then the “*Translate concepts...*” option. The concepts dialog will look as shown in Figure 4.

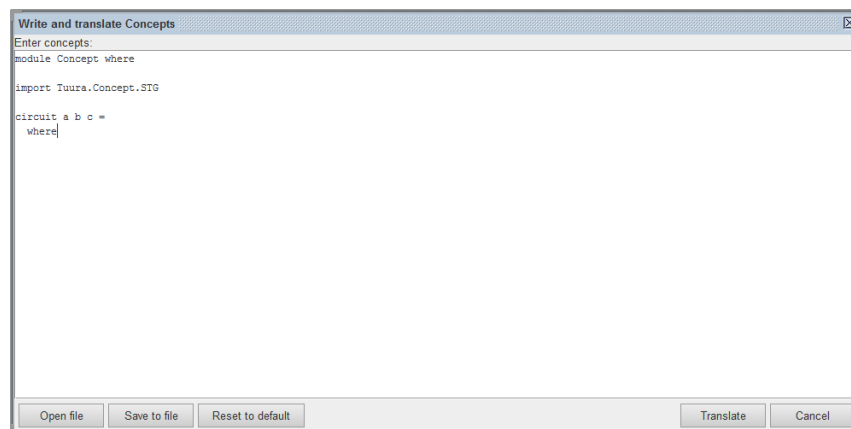


Figure 4: The concepts dialog.

From within this dialog, one can write their own concepts, from the default template as shown in Figure 4, or open an existing concepts file, with the *.hs* extension. When satisfied with the concepts written, a user can choose to save the file, if not already saved, and then translate these concepts.

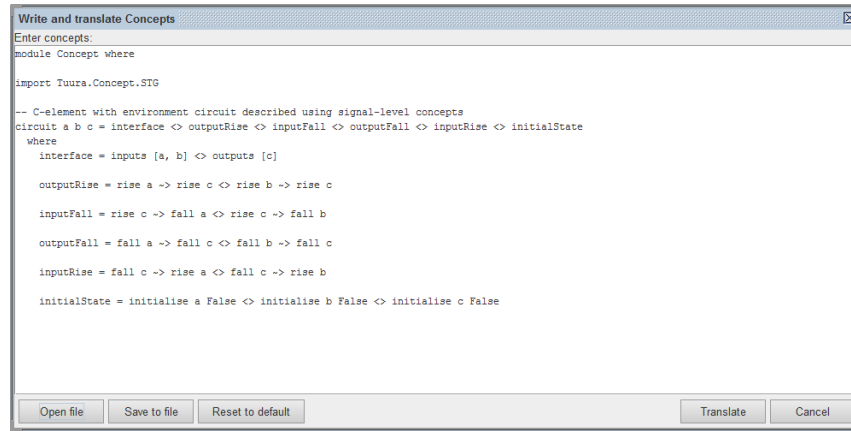


Figure 5: The concepts dialog with a concept file opened.

Figure 5 is the concepts dialog after we have opened the Celement with environment example, named “*Celement_with_env_1.hs*”, from the concepts tool examples directory. Clicking translate at this point will produce an STG representation of these concepts in the workspace.

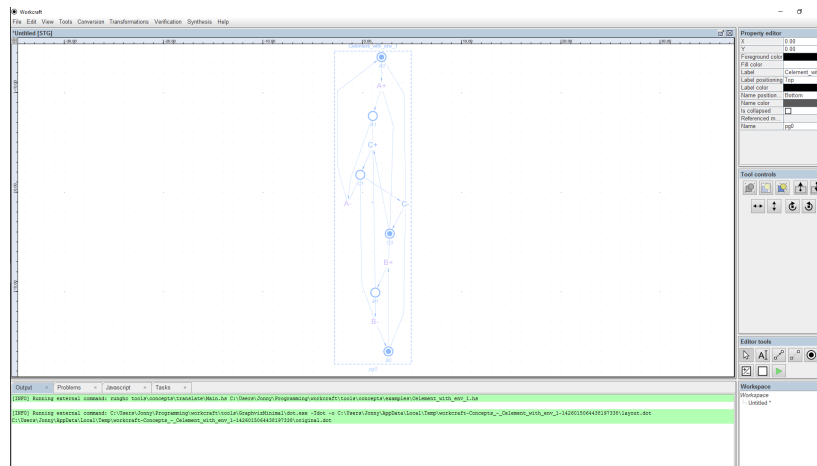


Figure 6: The STG produced from translating the concepts.

The translated concepts will look similar to Figure 6

Now, a user can choose to insert more concepts, make changes to this STG, and once they are satisfied with it, can then perform various functions on this STG. One can perform transformations, verifications, simulations and synthesis on this STG using the menus within this workspace now. Any further changes to this STG, based on the results of these operations can be made to this STG or to the concepts file.

4.2 Importing concepts directly

In WORKCRAFT it is also possible to import concepts directly from a file, without having to view the concepts first. This can be done from the “File” menu, by selecting the “Import...” option.

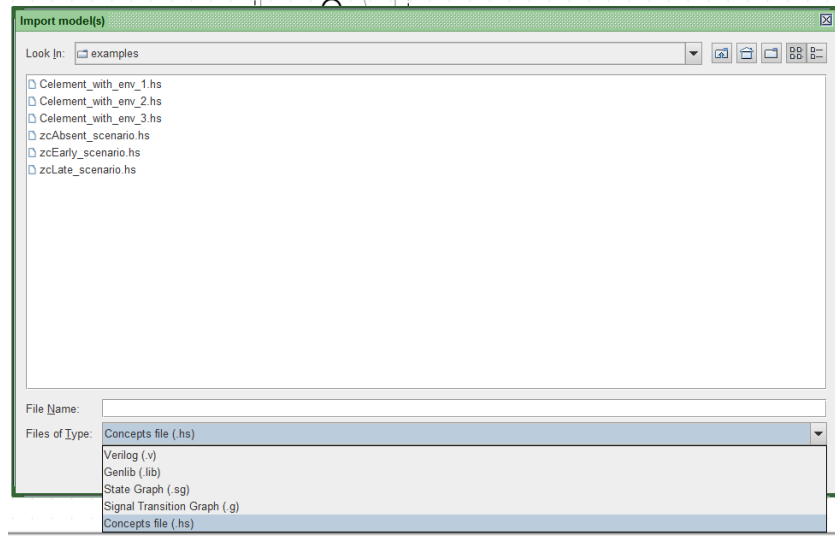


Figure 7: The STG produced from translating the concepts.

When importing concepts using this menu, ensure to set the “*Files of Type*” option to “*Concepts file (.hs)*”, as shown in Figure 7

4.3 Errors

If any errors are encountered during the translation process, WORKCRAFT will produce a helpful error message. This usually can tell you with more detail what the issue that is causing the error is, but will ask you to refer to WORKCRAFT’s console window for specific line numbers or signals which need to be corrected. These errors will include whether a signal has not been declared as an input or output, a signal has not had its initial state given, or even that the concepts tool has not been installed correctly.

5 Writing concepts

In this section, we will discuss how to write a concepts file. We will use an example and go through it based on lines, explaining what is necessary. For this example we will be using the same as example as in previous sections, from the examples included with the concepts repo. This is “*Celement_with_env1.hs*”.

5.1 Concepts file layout

```
module Concept where

import Tuura.Concept.STG

circuit abc = interface <> outputRise <> inputFall <> outputFall <> inputRise <> initialState
where

  interface = inputs [a, b] <> outputs [c]

  outputRise = rise a ~> rise c <> rise b ~> rise c

  inputFall = rise c ~> fall a <> rise c ~> fall b

  outputFall = fall a ~> fall c <> fall b ~> fall c

  inputRise = fall c ~> rise a <> fall c ~> rise b

  initialState = initialise a False <> initialise b False <> initialise c False
```

Figure 8: A concepts file

The concepts file we will discuss is found in Figure 8. The following describes important information about specific lines.

Line 1 This line must be included in all concept files, as the first line. This ensures that when translating, this is recognised as a concepts file.

Line 2 must also remain in all concept files, before any concepts begin to be defined. Importing this module means that the standard operators and existing gates/protocols can be used.

Line 3 is where a user can begin to define their concepts. “**circuit**” must begin the line, but after this, a user can choose what characters they wish to represent their signals. In this case, we use *a*, *b* and *c*. Whatever number of signals appear in the system, each one must have a character representation. Following the equals sign, “=”, we now start defining concepts. Here, any form of concepts can be used; signal-, gate- or protocol-level concepts. Or, a user can define their own concepts below, following where, and include the names of these concepts here. This can allow for a more concise and easily understood definition.

Line 4 is simply “**where**”. This is used to separate the main concept definition from the user-defined concepts. If a concept definition does not need any user defined concepts, this **where**, and all following lines can be omitted.

The lines discussed above are the basics of writing concepts. With this information, a user can write concept files, but the following lines can be used for ease-of-use, ease-of-understanding, and reuse. We will some of the following lines in the context of the *Celement_with_env1.hs* file, but the information can be applied to any concept files. More information on operators and built-in concepts can be found in Section 5.2.

Line 5 Interface is a concept defined to list the interfaces of the signals, in otherwords, their type. Signals in this can be defined as *inputs* or *outputs*. Every signal must have it’s type defined. This can be done using the functions `inputs[x, y]` and `outputs[p, q]`. Both of these functions can take a single signal, “[*a*]”, or multiple signals in a comma separated list, “[*x*, *y*, *z*]”.

Line 6 contains a composition of two signal-level concepts. This uses “**rise**” to signify a low-to-high signal transition. “~>” signifies causality between two signal transitions. “<>” is the composition operator. This is used to compose any type of concepts.

Line 7 also contains a composition of two signal-level concepts. Some are the same as in Line 11, however, this also features “**fall**”, which signifies a high-to-low transition of the signal this function applies to, in this case both **a** and **b** have included falling transitions.

Lines 8 and 9 are further concepts definitions.

Line 10 This is the definition of initial states for the signals in this system. This is done by using the “**initialise**” function. This takes a signal, and a Boolean value, and will set this signal’s initial state to 0 if the Boolean value is False, and 1 if the Boolean value is True. This is just one way of defining initial states.

It is important to note that all defined concepts after Line 8 are part of “**circuit**” definition, on Line 7. These user-defined concepts can be used both within other user-defined concepts, and within the circuit definition to be translated.

5.2 Built-in concepts

There are several built-in concepts included in the library, for simple functions, such as setting the initial state, and some standard useful gate- and protocol-level concepts included for ease-of-use. We will list the operators and concepts that are built-in to this tool in this section, and discuss their usage.

rise a is used in conjunction with a signal, in this example, **a**. This is used to show a low-to-high (0 \rightarrow 1) transition of a signal.

fall a the opposite of rise. Used in conjunction with a signal, **a** here, is used to show a high-to-low (1 \rightarrow 0) transition of a signal.

\rightsquigarrow is used to show *causality*. This is usually used in conjunction with **rise** and **fall**, to show that one signal transition, a cause, will occur before another signal transition, effect. This doesn’t force the effect to occur as soon as the cause has occurred, but simply states that the cause will happen before the effect. It does not suggest timing. One effect signal transition having multiple cause signal transitions creates *AND-causality*. This means that for the effect to occur, both causes must occur. For example:

$$\text{rise } a \rightsquigarrow \text{rise } c <> \text{rise } b \rightsquigarrow \text{rise } c$$

will form a concept where both **rise a** and **rise b** must occur before **rise c** can occur.

$\sim|\sim>$ is used to show *OR causality*. (This is *tilde* “|” *tilde* “>”. Hard to show in LaTeX). Similar to \rightsquigarrow , however the transitions on left side of this arrow can be a list of causes, e.g. [**rise a**, **rise b**]. The effect transition (on the right of the arrow) must be a single signal transition. This will imply that only one of the signal transitions in the cause list must occur in order for the effect to occur. With the previous example, the **rise a** transition alone can cause the effect.

$<>$ is the composition operator. This is used between two concepts to compose them. To compose several concepts at once, include this operator between all concepts, but not at the start and end of these concepts.

inputs [a, ...] This is used to define the interface of the included signal(s) as input. It can be used to set the type of a single signal, “[a]”, or multiple signals using a comma separated list, “[a, b, c]”.

outputs [a, ...] This is used to define the interface of the included signal(s) as output. It can be used to set the type of a single signal, “[a]”, or multiple signals using a comma separated list, “[a, b, c]”.

internals [a, ...] This is used to define the interface of the included signal(s) as internal. It can be used to set the type of a single signal, “[a]”, or multiple signals using a comma separated list, “[a, b, c]”.

initialise a Bool is used to set a single signal’s initial state. The *Bool*, if True will set the signals initial state to 1, or high. If False, the initial state of the signal will be 0, or low.

initialise0 **[a, ...]** sets multiple signals initial states to 0, or low. Passed into this concept can be a single signal, “[a]”, or several in a comma separated list, “[a, b, c]”.

initialise1 **[a, ...]** sets multiple signals initial states to 1, or high. As with the previous concept, the signals passed in to it can be a single signal, “[a]”, or several in a comma separated list, “[a, b, c]”.

buffer a b is a gate-level concept, used to show that the input signal (**a** in this case) causes the output to transition in the same way (**b** in this case).

inverter a b is a gate-level concept, used to show that the input signal (**a** in this case) causes the opposite transition in the output signal (**b** in this case).

cElement a b c is a gate-level concept, with two inputs (**a** and **b** in this case) and one output (**c**). For the output to transition from low-to-high, both inputs must have transitioned high. For the output to then transition from high-to-low, both input signals must have already transitioned low.

handshake a b is a protocol-level concept. The two signals passed into this protocol form a handshake, where the second signal (**b** in this case) follows the transitions of the first signal (**a** in this case). For example, when **a** transitions high, **b** will transition high at some point afterwards. When **a** transitions low **b** will transition low if it is not already.

handshake00 a b is another protocol-level concept. The behaviours of the signals are the same as in the standard **handshake** concepts, but this includes initial states for the signals. For this concept, both signals will be initialised to 0, or low.

handshake11 a b is another protocol-level concept. The behaviours of the signals are the same as in the standard **handshake** concepts, but this includes initial states for the signals. For this concept, both signals will be initialised to 1, or high.

me a b A protocol-level concepts. This concept defines *Mutual Exclusion* between two signals. This means that when one of these signals is high, the other cannot transition high.

meElement r1 r2 g1 g2 is a gate-level concept to apply mutual exclusion. In this case, there are four signals. Two are inputs (**r1** and **r2**), the other two are outputs (**g1** and **g2**). **r1** transitioning high will cause **g1** to go transition high, and this is the same for **r2** and **g2**, however, **g1** and **g2** are mutually exclusive. The aim of this concept is to ensure that the request signals, **r1** and **r2**, cause their respective grant signals, **g1** and **g2**, to transition high but never at the same time.

andGate a b c is a gate-level concept, using OR-causality to implement a standard AND gate. Signals **a** and **b** are inputs to the gate, and **c** is the output. Both **rise a** and **rise b** must occur for **rise c** to occur. Following this, either **fall a** or **fall b** must occur for **fall c** to occur.

orGate a b c is a gate-level concept, using OR-causality to implement a standard OR gate. Signals **a** and **b** are inputs to the gate, and **c** is the output. Either **rise a** or **rise b** must occur for **rise c** to occur. Following this, both **fall a** and **fall b** must occur for **fall c** to occur.

There are many operators and concepts. With these built-in concepts, we believe that it is possible to generate STGs of various sizes and complexities using these, and user-defined concepts.

We are always aiming to add more concepts to the library. Some that are to be added can be found in Section 6. Further suggestions can be added to the issue list in the github repo, where these can be discussed.

6 Features to be implemented

Some features which we aim to be standard functionality of the concepts tool are not implemented yet for various reasons. These will be displayed here. We will also try and explain a work-around to use until these features are implemented.

Conversations on these features can be found in the list of issues in the github repo. Any further problems or ideas for features can also be reported here.

Define signal names for translated concepts

Github issue: <https://github.com/tuura/concepts/issues/35>

Currently, when concepts are translated, regardless of the names of signals used in the concept definition, the STG translated will have signals, starting at 'A' and following the alphabet, only up to the number of signals in the system. For example, a concept containing 5 signals when translated will produce a STG containing signals 'A', 'B', 'C', 'D' and 'E'. These signals will be in the order of the signals defined in the `circuit`. For example, if the circuit definition is:

```
circuit x y z = ...
```

the signals in the translated STG will be 'A' in place of 'x', 'B' in place of 'y' and 'C' in place of 'z'. The signal names will not affect their type, or the specification in any-way.

A work-around for this when used in command line is unfortunately complicated. The `.g` file output by the tool can be edited, replacing, for example, all 'A' signals with the desired signal name. If using the tool with WORKCRAFT, this becomes somewhat easier. The STG when imported can be manipulated much more easily. The signals can have their names changed by selecting them and changing the property. Signals with the same name can be selected together (**shift-left mouse button**) and their name can be edited at once. This, however, is still not as simple as the signal names being included in the translation, and we aim to implement this feature soon.

User-defined concepts to be reused without re-definition

Github issue: <https://github.com/tuura/concepts/issues/36>

It is our aim to allow a user to define their own concepts to build a scenario STG, and then re-use the definition of some of their own concepts without having to re-define any of these concepts, to build another scenario STG, or in any other concept definition.

This feature has not been implemented in this version of the concepts tool. This may require some new form of definition of these concepts, in order to be reused. It may also require the import of a concept file including the original concept definition. When the feature has been implemented, information on how to use this feature will be included in this manual.

A work-around for this is simply to copy-and-paste the previous concept definition into the new concept file which would require this concept.

References

- [1] J Beaumont A Mokhov D Sokolov A Yakovlev. High-level asynchronous concepts at the interface between analogue and digital worlds. 2016.
- [2] T.-A. Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.
- [3] A. Yakovlev L. Rosenblum. Signal graphs: from self-timed to timed ones. *International Workshop on Timed Petri Nets*, pages 199–206.
- [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] Oriol Roig i Mansill. *Formal Verification and Testing of Asynchronous Circuits*. PhD thesis, Citeseer, 1997.
- [7] I. Poliakov, D. Sokolov, and A. Mokhov. Workcraft: a static data flow structure editing, visualisation and analysis tool. In *Petri Nets and Other Models of Concurrency*, pages 505–514. 2007.
- [8] Workcraft. www.workcraft.org.