# Plato: a tool for behavioural specification of asynchronous circuits

Jonathan Beaumont
j.r.beaumont@ncl.ac.uk
*School of Electrical and Electronic Engineering, Newcastle University, UK*

***Abstract*—Asynchronous circuits are becoming increasingly important in system design, where they orchestrate the interface between synchronous computation components and the analogue environment. However, wide adoption of asynchronous circuits by industrial users is hindered by a steep learning curve for asynchronous control models, such as Signal Transition Graphs, that are developed by the academic community for specification, verification and synthesis of asynchronous circuits.**

**Previously, we have introduced a novel high-level description language for asynchronous circuits, which is based on behavioural *concepts* – high-level descriptions of asynchronous circuit requirements. In this paper we will discuss our open-source tool, PLATO which allows the specification of asynchronous circuits using concepts, and features the ability to automatically translate these to Signal Transition Graphs for further processing by conventional asynchronous EDA tools, such as PETRIFY and MPSAT.**

## I. INTRODUCTION

*Concepts* have been presented in order to provide a more compact and intuitive method of designing asynchronous circuits, using a fully compositional description method. Composition is an important part of concepts and the use of composition in some algebraic representations, such as with DI algebra [1], Conditional Signal Graphs [2] and Algebra of Parameterised Graphs [3] helped to inspire this.

As discussed in [4], concepts are a useful language for specifying the behaviours of asynchronous circuits, in the preferred form of the user. This can be as low-level signal-level concepts, or higher-level gate- or protocol-level concepts. It also allows the definition of their own concepts, which can be reused within the same or any other specification that they wish to increase the speed of designing a system, and future systems.

With concepts, we aim to solve the problems that can arise from the more commonly used monolithic approach with Signal Transition Graphs (STGs) [5][6], where a user must design each system in the form of an STG from a blank page. The scalability of this is poor: as the system grows in complexity its monolithic specification becomes challenging to comprehend and debug. When the number of signals increases, the gates and protocols describing the interactions between some signals become difficult to include with each occurrence of these signals.

STGs are supported by multiple electronic design automation (EDA) tools for verification and synthesis of asynchronous control circuits, such as PETRIFY [7], MPSAT [8], VERSIFY [9], WORKCRAFT [10][11], and others. These tools take an STG specification and can formally verify its correctness, as well as synthesize an asynchronous circuit implementation that is *speed-independent*, i.e. guaranteed to work correctly regardless of component delays [12].

Concepts are not supported by these tools, and rather than authoring tools to verify and synthesize concepts directly, which is a time consuming process, we can *translate* concepts to STGs, for use with these existing tools.

In this paper, we will discuss the implementation of the domain specific language and translation tool for concepts, implemented in HASKELL, called PLATO [13]. If STGs are the assembly language of asynchronous circuits, PLATO is a compiler from a higher-level language, concepts, to this assembler. PLATO is integrated into WORKCRAFT [11], an open-source EDA tool which also features some verification and synthesis tools for STGs.

The contributions of this paper are:

- We detail the notations of concepts within PLATO and the built in concepts and their uses in Section II.
- We present the implemented algorithm for translating concepts to STGs in Section III.
- We explain how to use the tool from within WORKCRAFT to translate concepts in Section IV.

## II. FEATURES OF PLATO

The abstract base of concepts, on which asynchronous circuit specifications is founded, is discussed in [4].

### A. Concept notation

Concepts can be composed of other concepts, and this applies to the behaviours of signals, as well as the specification of the initial states and the signal types. This allows there to be different levels of concepts, each level of which will feature a composition of concepts from a lower level. In this section we will explain these levels, and the standard notations of concepts for these.

*1) Signal-level concepts:* Asynchronous circuit specifications are mainly composed of signal transitions, and interactions between these, to show causal relationships. Signal transitions are denoted as $a^+$ and $a^-$, where $a$ is any signal name, and the $+$ or $-$ indicates which way this signal transitions, $+$ denoting a low-to-high or 0 to 1 transition, and $-$ denoting a high-to-low, 1 to 0 transition.

PLATO is written in Haskell, a functional programming language. The domain specific language which we have implemented therefore uses Haskell notations, and this means

some notations are different to standard signal transition specifications. $a^+$ and $a^-$ are in post-fix notation, where the operator is stated after the operand. Haskell does not support post-fix notation, and as such, we have to use a differing signal transition notation.

In PLATO, signal transitions $a^+$ and $a^-$ are noted as `rise a` and `fall a` respectively. We will use the tool notation in code examples, but describe these using standard notation in the text

Signal-level concepts are the base level of concepts, and are the type all other concepts are built on. Here we display the standard concepts available at this level.

A key concept in asynchronous circuits is *causality*: one signal transition *causes* another signal transition, a cause and an effect. This is denoted in the form:

```
rise a ~> rise c
```

This is read as $a^+$ causes $c^+$, meaning that for the $c^+$ transition to occur, $a^+$ must have occurred previously. The operator `~>` is used to show causal relationships between signals.

One can compose any concepts using the operator `<>`, and this applies to concepts of any level, whether predefined or user-defined. For example, two causality concepts can be composed.

```
rise a ~> rise c <> rise b ~> rise c
```

In words, $a^+$ *and* $b^+$ must occur before $c^+$ can occur. This corresponds to so-called AND-causality in the fact that several cause transitions *must all* have occurred before an event can occur. AND-causality is commonly used to imply behaviours in circuits, for specific requirements of effect transitions.

The above notation can cause long-winded specifications when lots of AND-causality is involved. To solve this we provide a listing option. The following will achieve the same results:

```
[rise a, rise b] ~&~> rise c
```

This form of concept can be composed as usual with any other concepts.

A less common, but still useful form of causality is OR-causality. This is where an effect transition can have several possible cause transitions. Only one cause transition is required to occur to allow the effect transition to occur.

With OR-causality, the notation used lists all possible causes for the stated effect:

```
[rise x, rise y] ~|~> rise z
```

This is, *either* $x^+$ *or* $y^+$ must occur in order for $z^+$ to occur. The interactions when an effect transition is included in both AND- and OR-causality are interesting, and an example of when this occurs can be found in Section III.

*2) Gate-level concepts :* Using the causality concept we can express the behaviour of gates in asynchronous circuits. For example, a *buffer* is a gate with one input signal and one output signal, whose output transitions causally depend on the input ones:

```
buffer a b = rise a ~> rise b
           <> fall a ~> fall b
```

An *inverter* has a similar conceptual specification, but the output transition is inverted:

```
inverter a b = rise a ~> fall b
             <> fall a ~> rise b
```

A *C-element* is a gate with two inputs, in this example $a$ and $b$, and one output $c$, which synchronizes both rising and falling input transitions via AND-causality:

```
cElement a b c = rise a ~> rise c
               <> rise b ~> rise c
               <> fall a ~> fall c
               <> fall b ~> fall c
```

An alternative way to express the same concept is to reuse the buffer concept:

```
cElement a b c = buffer a c <> buffer b c
```

A C-element combines the constraints imposed on the output transitions by two 'virtual' buffers. Behaviour of other gates can be similarly defined in this way. An expanded example of a C-element and other examples can be found in [4].

*3) Protocol-level concepts:* In addition to gate-level concepts described above it is often important to specify *protocols* of interaction between multiple gates, components or signals. Here we demonstrate how one can use concepts to specify asynchronous handshakes.

Given two signals $a$ and $b$, a *handshake* between them is the following composition of causality concepts:

```
handshake a b = rise a ~> rise b
              <> rise b ~> fall a
              <> fall a ~> fall b
              <> fall b ~> rise a
```

Intuitively, we have a two-way asynchronous communication channel, where one party sends transitions $a^+$ and $a^-$ and the other party responds by corresponding $b^+$ and $b^-$ transitions. Note that the four causality concepts match those found in the buffer and inverter concepts, which leads to an alternative way to express a handshake between $a$ and $b$:

```
handshake a b = buffer a b
              <> inverter b a
```

This conceptual understanding of a handshake as being composed from a buffer and an inverter is often used by circuit designers as a convenient way of reasoning.

### B. Concepts required for translation

The concepts we have discussed so far are aimed at specifying the behaviour of signals in a circuit. When translating these to STGs however, these behaviours do not result in a complete specification, meaning they will not be verifiable by standard tools, and therefore not synthesizable.

This is due to various parts of an STG that we have not yet discussed, which are important for specification, namely *interface* and *initial states*.

*1) Interface concepts:* An important part of a specification is how these signals interact with the outside world, which could be another scenario or another circuit, for example. These signals can be inputs from the outside world, outputs or an internal signal, which is used only within this circuit.

To specify the type of a signal (*input*, *output* or *internal*) we introduce the `interface` concept. Signal types are composed according to the following rules:

| <> | Input | Output | Internal |
|---|---|---|---|
| Input | Input | Output | Internal |
| Output | Output | Output | Internal |
| Internal | Internal | Internal | Internal |

The intuition is as follows:

- If a signal is an input in one component of the system, but is an output in another component, then in the composition it will be an output.
- An internal signal is similar to an output signal in the sense that it is driven by the circuit (not the environment), but it is hidden, i.e. not accessible via the circuit interface. Once a signal is hidden and declared internal it cannot be revealed.

Specifying signal types is important when designing asynchronous circuits, as it helps to quickly identify errors, for example, when an input transition is caused by a hidden internal transition, as internal transitions are used only by the circuit and cannot be accessed from the environment, and as such, an internal transition cannot cause an input transition.

We can then reuse existing tools for circuit simulation, verification and synthesis. Signal type information is also used in the algorithm for automated translation of concepts to STGs (Section III).

Concepts `inputs`, `outputs`, `internals` are defined for specifying the type of sets of signals, and can be included inline with other concepts. For example, to specify that signals $a$ and $b$ are inputs, $c$ is an output, and $t$ is internal:

```
inputs [a, b] <> outputs [c]
      <> internals [t]
```

*2) Initial state concepts:* Specifying the initial state is important, as it determines what the first transitions of a scenario will be. Without these, no transition can occur.

Each signal must have its initial state declared before translation can occur. In order to specify the initial state of a handshake between signals $a$ and $b$, we use the `initialise` concept. The possible initial states are *low* or *high*, referred to as *0* or *1* respectively:

```
initialise a 0 <> initialise x 1
```

A signal can only be declared as initially high or low. If the initial state of a signal is not defined, an error will occur, and the translation will not continue. In the event a signal has its initial state declared as both high and low, and is thus inconsistent, then the translation will also fail.

For ease-of-use, initial states can also be declared in lists:

```
initialise0[a, b, c] <> initialise1[x,y]
```

## III. Concepts to STG translation algorithm

In this section, we will display an example of how the translation algorithm operates, and present a pseudocode form in Algorithm 1.

**Algorithm 1** Algorithm for concepts to STG translation

1: **for each** effect $e$ **do**
2: $allCauses \leftarrow$ **Concatenate** lists of causes for $e$
3: $transitionList \leftarrow$ **CartesianProduct** $allCauses$
4: **addAll** $transitionList$ **to** $transitions$
5: **end for**
6: **for each** signal $s$ **in** system **do**
7: **define interface** of $s$ as *Input/Output/Internal*
8: **add** place $s$**.Name**0
9: **add** place $s$**.Name**1
10: **for each** transition $t$ **of** $s$ **do**
11: **if** $t$ **is** $rise$ **then**
12: **connect** (place $t$.signalName0, $t$)
13: **connect** ($t$, place $t$.signalName1)
14: **else if** $t$ **is** $fall$ **then**
15: **connect** (place $t$.signalName1, $t$)
16: **connect** ($t$, place $t$.signalName0)
17: **end if**
18: **end for**
19: **end for**
20: **for each** initial state $state$ **do**
21: **if** $state$ **is** low **then**
22: **add-token**(**signalName.place**0)
23: **else if** $state$ **is** high **then**
24: **add-token**(**signalName.place**1)
25: **end if**
26: **end for**
27: **for each** transition $t$ **in** $transitions$ **do**
28: **if** $t$ **is** $rise$ **then**
29: **for each** effect transition $e$ **of** $t$ **do**
30: **read-arc** (place $t$.signalName1, $e$)
31: **end for**
32: **else if** $t$ **is** $fall$ **then**
33: **for each** effect transition $e$ **of** $t$ **do**
34: **read-arc** (place $t$.signalName0, $e$)
35: **end for**
36: **end if**
37: **end for**

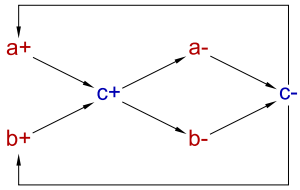


Figure 1: Example STG

We will use a C-element with environment example, which features two inputs to a C-element, $a$ and $b$, and the output of this, $c$, connected to the input of an inverter, the output of which is used to drive the inputs, $a$ and $b$. For reference, a simplified STG of this is featured in Figure 1.

Working from this STG, one could describe the low-level operations by inputs and outputs:

```
circuit a b c = protocol <> ports
              <> init
 where
  protocol = outRise <> outFall
           <> inRise <> inFall
  outRise  = [rise a,rise b] ~&~> rise c
  outFall  = [fall a,fall b] ~&~> fall c
  inRise   = fall c ~> rise a
           <> fall c ~> rise b
  inFall   = rise c ~> fall a
           <> rise c ~> fall b
  ports    = inputs [a, b]
           <> outputs [c]
  init     = initialise0 [a, b, c]
```

This provides four separate behavioural concepts, `outRise` and `outFall` which describe how the outputs transition, and `inRise` and `inFall`, describing how the inputs transition. These are good to describe the low-level behaviour of each signal in relation to the other signals, but this concept specification is fairly long.

We have already defined C-element and inverter concepts in Section II-A2. Therefore, we can simplify this specification, to use gate-level concepts instead:

```
circuit a b c = protocol <> ports
              <> init
 where
  protocol = cElement a b c
           <> inverter c a
           <> inverter c b
  ports    = inputs [a, b]
           <> outputs [c]
  init     = initialise0 [a, b, c]
```

The signal-level concepts between the signal pairs of $a$ and $c$, and $b$ and $c$ also match those of a handshake. So we can also specify the operation as:

```
handshake a c <> handshake b c
```

There can be multiple ways of representing a specification using concepts, however, all levels of abstraction available to the designer are built out of signal-level concepts. Given a specification, we can therefore break down all gate- and protocol-level constructs into 'atoms', which significantly simplifies the translation task. For now, we will ignore initial states and interface concepts, and focus on the arcs only.

Table I: Lists of cause transitions by effect transitions

| Causes | Effect |
|---|---|
| $a^+$, $b^+$ | $c^+$ |
| $a^-$, $b^-$ | $c^-$ |
| $c^-$ | $a^+$ |
| $c^+$ | $a^-$ |
| $c^-$ | $b^+$ |
| $c^+$ | $b^-$ |

To start with we list all causes for each effect. This is performed on Line 2 of the algorithm, for every effect transition. These lists can be found in Table I. All of the included causalities are AND-causalities, and hence, there is at least one single cause transitions for each effect transition.

If OR-causality was included, the list of possible causes would be included with these AND-causalities. For example: $a^+$, $b^+$, $[x^+, y^+]$ causing $c^+$. $x$ and $y$ are in a list of their own, and this shows OR-causality; either $x$ or $y$ must transition high for $c^+$ to occur. However, this does not account for the other AND-causalities.

In this case, we perform the *Cartesian product* (Algorithm 1, Line 3), which combines all AND- and OR-causalities. For the example above, this would produce:

$$a^+,\ b^+,\ x^+ \text{ causes } c/1^+$$
$$a^+,\ b^+,\ y^+ \text{ causes } c/2^+$$

This includes one of the OR-causality transitions with all of the AND-causality transitions. It also leaves us with two separate $c^+$ transitions. These are both needed in order to show the two possible combinations of transitions necessary for $c^+$, and would be included as two separate transition objects in an STG.

However, for the C-element with environment example, there is no OR-causality. We still perform the step of applying the Cartesian product to each of these lists of transitions for each effect transition, but the result will be the same as the original lists, seen in Table I.

This concludes the part of the algorithm which defines the arcs and transitions. Next, the algorithm begins to build the STGs. This is where we start to use the initial state and interface concepts.

First of all, the interface is defined, by listing all signals as *inputs*, *outputs* or *internals*, based on the specification. This is performed by Line 7 of the algorithm, with Line 6 ensuring every signal has its interface declared.

Now we need to add places for each signal. These are used to show whether a signal has transitioned high or low at any point. For example a token in a 0 place shows that this signal has transitioned low. The algorithm performs this on Lines 10-11.

Now we include all of the transitions for each signal, high and low, and connect these to the places to form consistency loops This ensures that all signals in this system will not repeat a same-direction transition, i.e. a signal will not transition high, and then high again sometime later.

This is done by taking each transition, and if it is a $+$ transition, connecting the 0 place for this signal to the transition, then connecting this transition to the 1 place for this signal. If it is a $-$ transition, we connect the 1 place for this signal to the transition, and connect this transition to the 0 place.

Algorithm 1, Lines 12-13 perform this operation for a $+$ transition, and Lines 15-16 do the same for a $-$ transition. Line 10 ensures this is performed for each and every transition object for a signal.

Next, the algorithm introduces the initial states. All of the signals are specified to be initially 0 in this example, meaning that the first transition will be the $+$. In the consistency loops,

we need to place a token in each 0 place for each signal. Lines 20-26 of the algorithm create these tokens, with Lines 21-22 creating a token for an initially 0 signal, and Lines 23-24 creating a token for an initially 1 signal.

Finally, we can add the connections to the transitions, and cause the expected interaction between signals. This is done using read-arcs, which are arcs which require, but do not consume tokens for transitions. We connect a read-arc from the place after a cause transition, to the effect transition object. For example, for `rise a ~> rise b` we connect the place after $a^+$, namely $a0$, to the transition $b^+$. In this case, $b^+$ will not occur until $a0$ contains a token, and when it does occur, the token will remain in $a0$ following the $b^+$ transition.

The read-arc connections are performed by Line 30 for a $rise$ transition, connecting the cause signals 1 to the effect transition object, and Line 34 connects the 0 place of a falling transition to the effect transition object.

Following this, the specification is fully translated, and the resulting STG will be the same as shown in Figure 2. This can now be used with tools, to verify and synthesize. It can also be *resynthesized* using these tools, to produce a more compact STG, such as that seen in Figure 1

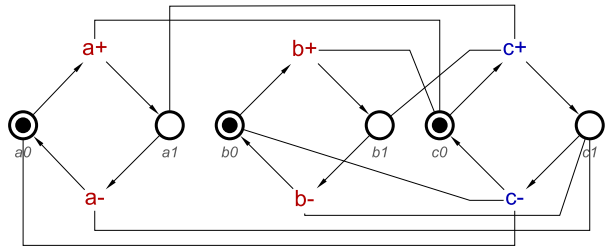


Figure 2: Fully translated C-element with environment STG

## IV. Usage of the Tool

We will now discuss the design flow, including the preparation of a concepts file, the translation methods from within WORKCRAFT, and the uses of the translated STGs.

### A. Concepts file layout

```
module CElement where
import Tuura.Concept.STG

circuit a b c = protocol <> ports
              <> init
 where
  protocol = cElement a b c <> env
  env      = inverter c a
           <> inverter c b
  ports    = inputs[a, b] <> outputs[c]
  init     = initialise0 [a, b, c]
```

Figure 3: A concepts file

The concepts file we will discuss is found in Figure 3. Concepts files are an extension of Haskell files, and hence, use the ".hs" file extension. The following describes important information about specific lines.

Line 1 This line must be included in all concept files, with the module name chosen by the user.

Line 2 is included so that the built-in 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 for the concept to be translated. A user can choose what names they wish to represent their signals.

Line 4 is simply "`where`". This is used to separate the main concept definition from other user-defined concepts.

The lines discussed above are the basics of writing concepts. With this information, a user can write concept files. We have used the C-element with environment example here. We have defined the interface, initial state and the operation of this separately, by defining these concepts following the "**`where`**". The full circuit specification is then composed of all of these defined concepts.

### B. Using PLATO from within WORKCRAFT

PLATO, can be used on its own using a command-line interface, and has been integrated into WORKCRAFT, an open-source EDA tool. This can visualise many graphical modeling methods, including STGs, and also features several other tools integrated, which can perform verification and synthesis of these models automatically.

STGs are a featured modeling method within WORKCRAFT, which can be automatically imported from PLATO, either from concept specifications written by a user from within WORKCRAFT, or by importing a previously written concept file. This section will discuss how to use PLATO from within WORKCRAFT.

*1) Translating and authoring concepts:* To start specifying and translating concepts, open the concepts dialog. This is done from the menu bar, by selecting the "*Conversion*" menu, and then the "*Translate concepts...*" option. It will look similar to the dialog on the left in Figure 4.

From within this dialog, one can write their own concepts, from the default template provided. One can 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. Translated concepts will produce an STG in a form similar to that on the right in Figure 4.

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. Any further changes to this STG, based on the results of these operations can be made to this STG or to the concepts file.

*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.

Any concept files imported will be automatically translated to an STG.

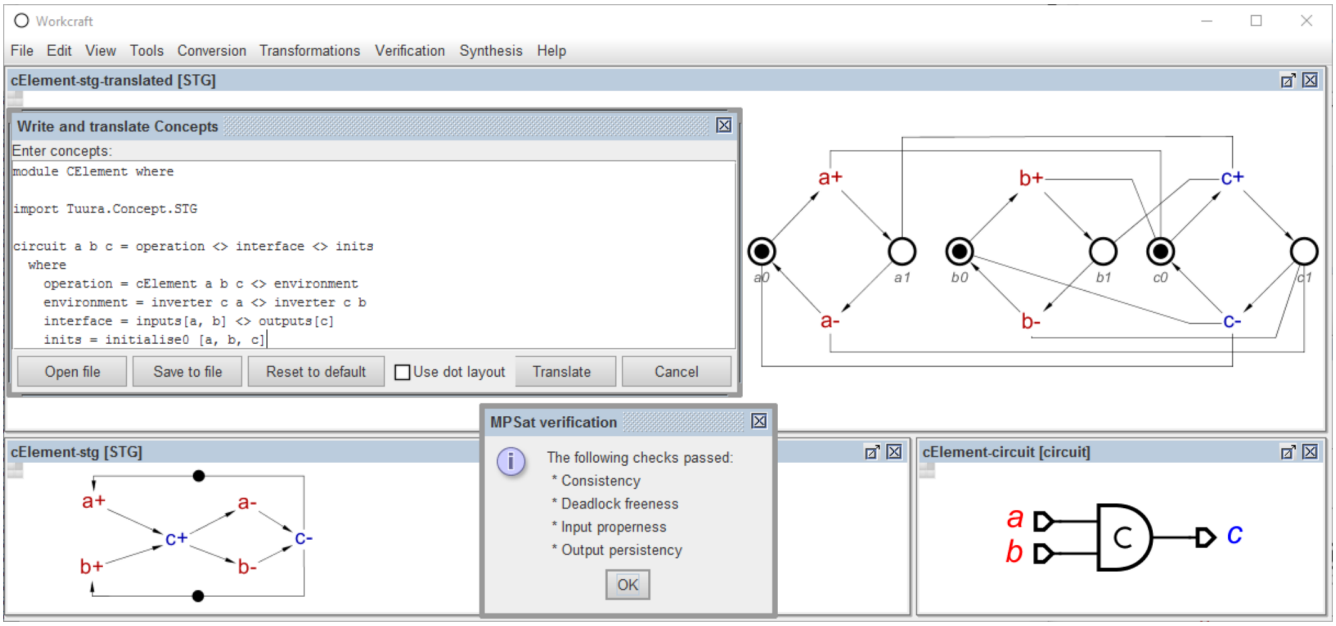


Figure 4: WORKCRAFT and PLATO usage.

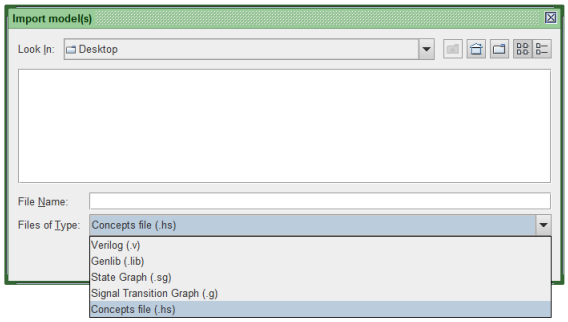


Figure 5: The import menu and the option of *.hs* files.

## V. CONCLUSIONS AND FUTURE WORK

In this work, we have displayed the open-source tool, PLATO [13]. This tool implements a domain-specific language for behavioural concepts, featuring some built-in concepts at varying levels, allowing users to specify behaviours in a preferred way.

This tool also features a method of converting concepts into Signal Transition Graphs, output in a format usable by other existing design, verification and synthesis tools.

PLATO, can be used on its own using a command-line interface, and is integrated into WORKCRAFT, which can visualise many graphical modeling methods, including STGs, which can be automatically imported from PLATO either from concept specifications written by a user from within WORKCRAFT, or by importing a previously written concept file. WORKCRAFT also has several verification and synthesis tools integrated, which can automatically use STGs translated from concepts.

Using concepts, a user can reduce the time of designing an asynchronous control circuit from the ground up, as well as reuse components by importing them from previously written user concept files which can reduce design-time of future projects. Composition of concepts can help reduce errors and save time in comparison to performing these manually. This method can help to make asynchronous circuits more appealing to industrial designers.

The PLATO tool we have discussed is available from [13], and is included with WORKCRAFT [11]. A manual is included with the tool, which contains descriptions of the features.

## REFERENCES


[1] M.B. Josephs and J.T. Udding. An overview of d-i algebra. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume i, pages 329–338 vol.1, Jan 1993.

[2] A. Mokhov, D. Sokolov, and A. Yakovlev. Adapting asynchronous circuits to operating conditions by logic parametrisation. In *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, pages 17–24, May 2012.

[3] A. Mokhov and V. Khomenko. Algebra of Parameterised Graphs. *ACM Transactions on Embedded Computing*, 13(4s), 2014.

[4] J. Beaumont A. Mokhov D. Sokolov A. Yakovlev. Compositional design of asynchronous circuits from behavioural concepts. In *ACM-IEEE International Conference on Formal Methods and Models for System Design MEMOCODE15*, June 2015.

[5] T.-A. Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[6] A. Yakovlev L. Rosenblum. Signal graphs: from self-timed to timed ones. *International Workshop on Timed Petri Nets*, pages 199–206.

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

[8] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state encoding conflicts in stg unfoldings using sat. *Fundamenta Informaticae*, 62(2):221–241, 2004.

[9] O. R. i Mansill. *Formal Verification and Testing of Asynchronous Circuits*. PhD thesis, Citeseer, 1997.

[10] D. Sokolov, V. Khomenko, and A. Mokhov. Workcraft: Ten years later. In *This asynchronous world. Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday*. Newcastle University, 2016. Available online http://async.org.uk/ay-festschrift/paper25-Alex-Festschrift.pdf.

[11] Workcraft. www.workcraft.org.

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

[13] Plato repository. https://github.com/tuura/plato, 2016.