Synthesis from Waveform Transition Graphs

Alberto Moreno*, Danil Sokolov*, Jordi Cortadella†
*School of Engineering, Newcastle University, Newcastle upon Tyne, United Kingdom, NE1 7RU
†Department of Computer Science, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain

***Abstract*—Asynchronous circuits are a promising class of digital circuits that has numerous advantages over their synchronous counterparts. Nonetheless, adoption has not been widespread, which in part is attributed to the difficulty of entry into complex models like Signal Transition Graphs (STGs) by electronic designers. This work formally introduces Waveform Transition Graphs (WTGs) which resembles the timing diagrams, that are very familiar to circuit designers, and defines their behaviour semantics. This formalization enables translation of the WTGs into equivalent STGs in order to reuse the existing body of research and tools for verification and logic synthesis of speed-independent circuits. The development of WTGs has been automated in the WORKCRAFT toolkit, allowing their conversion into STGs, verification and synthesis.**

I. INTRODUCTION

Asynchronous circuits forfeit the use of a global clock signal in favor of local synchronization between components [1]. This gives important advantages in terms of power consumption and electromagnetic emissions, as well as avoiding clock-related problems, such as clock skew. Moreover, for newer technology nodes, asynchronous circuits are fundamentally more tolerant to voltage, temperature and process variability [2]. New application domains, such as *analogue and mixed-signal* (AMS) systems, also benefit from asynchronous controllers and demonstrate clear advantages over synchronous solutions beyond the classical examples. For instance, asynchronously controlled power converters significantly improve the response time to power demand – from the order of several clock cycles to just few gate delays. This helps to reduce the voltage ripple and peak current, which can be efficiently traded off for smaller inductor size [3].

Yet, considering all the benefits and technical advantages, asynchronous controllers have a relatively modest use in commercially produced systems. Arguably, one of the main reasons for this lack of success lays in the challenging adoption path by the electronic engineers. While there is a wide availability of powerful synthesis and verification tools, such as PETRIFY [4], ATACS [5] and MPSAT [6], they are based on *Signal Transition Graphs* (STGs) or other rather powerful, yet complex, formal models. Indeed, STGs is a very expressive model, however, this expressiveness is doubled edged. In essence, the underlying model for STGs is Petri nets, for which most electronic engineers are not familiar with. The use of STGs necessitates a deep understanding of such modelling aspects as the token game, encoding conflicts, and output persistency, which are better suited for scientists than engineers.

In an attempt to improve this situation, a new model for asynchronous design, the *Waveform Transition Graph* (WTG), was proposed [7]. WTGs model is designed from its inception to be simple and familiar to electronic engineers, yet expressive enough to specify a wide class of asynchronous controllers. In particular, the concurrency and choice are explicitly differentiated in WTGs. The concurrency is localized within *waveforms*, which are notationally identical to *Timing Diagrams* (TDs) and specify partial order of signals in each distinctive mode of a circuit operation. The choice between the modes of operation is restricted to the *nodal states*, which are strongly related to the notion of states in *Finite State Machines* (FSMs). In essence, a WTG is an FSM whose arcs are associated with TDs. As both TDs and FSMs are common tools for electronic engineers the adoption of WTGs becomes very intuitive to them.

Development of WTG model was influenced by several previous works, e.g the *Waves* model and the synthesis method implemented in JANUS tool [8]. The model makes use of waveform diagrams to describe behavior of signals and, similar to WTG, leverage the familiarity of TDs with designers. However, it lacked a path towards generating a complete synthesizable model of the control circuit behavior. *Waves* was targeted at synchronous implementation through the syntax-direct translation, and hence lacked efficiency compared to what is possible via explicit logic synthesis.

Comparisons can also be made with *Burst Mode* and *eXtended Burst Mode* (XBM) automata, supported by the synthesis tools MINIMALIST and 3D [9]. These models have a limited expressiveness when compared to WTGs. A WTG can be seen as an XBM automaton in which each arc represents a waveform rather than an input/output burst. While the input/output bursts are necessarily sequential in XBM, with an output burst always following an input burst, the waveforms in WTG allow concurrency between inputs and outputs, thus increasing the expressive power of WTGs.

Another relevant model, that inspired the explicit support of *undefined* signal behavior and *guarded* selection of waveforms in WTGs, is the *Generalized STGs* (GSTGs) [10]. The primary goal of GSTGs was to enable specification of mixed mode asynchronous/synchronous systems and arbitration. This was achieved by extending conventional STGs with several new modelling primitives, such as level transitions and Boolean guard functions. These new features, however, negatively impact on accessibility of already rather complicated STGs model. Synthesis of GSTGs is supported by PETRIFY, but often relies on an experienced designer to provide the timing

assumptions that make the specification implementable.

Some similarity to WTGs can also be found in binary expansion of Symbolic STGs (SSTGs) [11]. Here the change of symbolic variables between two high-level states is represented by a partial order STG fragment (resembles WTG waveforms) which is inserted between a pair of Petri net places that correspond to the high-level states (similar to nodal states). There is, however, no explicit separation of choice and concurrency and inherent complexity of SSTGs makes them hardly accessible by the circuit designers, which is the primary goal of WTGs.

The above issues are addressed in the design flow for development and synthesis of WTGs. WTGs can only express a subset of behaviors that can be specified with STGs. This restriction of expressiveness allows to significantly reduce the chance for design errors when specifying the circuit behaviour and also streamline the circuit synthesis. Note that WTGs can be mechanically translated into STGs, which enables the reuse of all the infrastructure for synthesis and verification that is already built around STGs.

The main contributions of the paper are as follows:

- Formal definition of WTGs model and its behavioural semantics (see Section II).
- Introduction of implementability constraints for WTGs (see Section III).
- Translation of WTGs into STGs for reuse of verification and synthesis methods (see Section IV).
- Design automation strategies for synthesis of WTGs (see Section V).

II. WTG DEFINITION AND SEMANTICS

A Waveform Transition Graph (WTG) is a bipartite connected and directed graph in which vertices are *nodal states* and *waveforms*. Arcs connect waveforms with nodal states and vice versa. Before going into details, it is important to first define what a waveform is.

A. Waveform

In essence, a Waveform (WF) is a formal model for Timing Diagrams (TDs) represented as an acyclic graph in which vertices are *transitions*. Transitions represent events for the signals, while arcs define the causality among events.

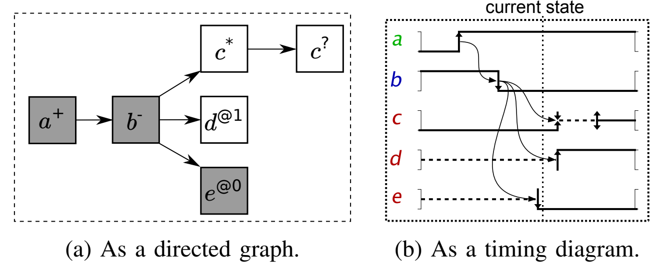
Formally, a WF is the tuple $w = \{T, F, \sigma, \lambda\}$ where:

- T is a finite set of transitions.
- $F \subseteq T \times T$ is the flow relation that defines a strict partial order (irreflexive, transitive and antisymmetric).
- $\sigma = \{In \cup Out \cup Int\}$ is the set of input, output and internal signals (pairwise disjoint).
- $\lambda : T \rightarrow \sigma \times \{+, -\} \cup In \times \{*, ?, @0, @1\}$ is the labelling function.

The minimal elements of F (transitions without predecessors) are the *initial transitions* of the WF, whereas the maximal elements (transitions without successors) are the *final transitions*. When a WF is *activated*, transitions can *fire* in any order that honors the precedence defined by F. We say that a WF has been *executed* when all its transitions have fired. When

	0	1	*	?
0		+	*	
1	–		*	
*	@0	@1		?
?			*	

TABLE I: Valid signal transitions from one state (left column) to another state (top row). The empty cells represent invalid transitions.



(a) As a directed graph. (b) As a timing diagram.

Fig. 1: Visual representation of a Waveform.

a WF has been executed, it becomes *inactive*. An active WF cannot be re-activated until it has been executed. At any given moment, the *state* of a WF is defined by the set of transitions that have fired.

The set of signals σ is the union of *input* (*In*), *output* (*Out*) and *internal* (*Int*) signals. The labeling function λ assigns a signal event to every transition. Every signal can be associated to a $+/-$ event, while only inputs can be labelled with $*, ?, @0$ or $@1$. The firing of transitions in a WF implicitly induces a state for every signal, called *signal state*. The signal state induced by firing a transition depends on the specific transition label:

- *Rise (Fall)*: A rise (fall) event is denoted by $+$ ($-$) and indicates that a signal changes its state from 0 to 1 (1 to 0).
- *Destabilize*: A *destabilize* event ($*$), indicates that a signal becomes *unstable*.
- *Stabilize*: A *stabilize* event (?) indicates that a signal becomes *stable* at an unknown value.
- *Stabilize-at-0 (1)*: A *stabilize-at-0 (1)* event is denoted by @0 (@1) and indicates that a unstable signal becomes 0 (1).

Table I summarizes all the valid signal transitions induced by the events.

Intuitively, the *destabilize* transition $*$ indicates that a signal may arbitrarily change its value. Alternatively, it is a way of saying that the state of a signal can be ignored, hence why this transition is sometimes referred to as *don't care*. Conversely, the *stabilize* transition ? indicates that a signal will not change value. Yet, the value of a signal in *stable* state remains irrelevant.

Fig. 1a shows the representation of a WF as a graph. Shadowed transitions indicate that they have already fired, defining the state as $\{a^+, b^-, e^{@0}\}$. In this WF, only a^+ belongs to the initial transitions, while the set of final transitions is composed by $\{c^?, d^{@1}, e^{@0}\}$.

Note that the representation of a WF in Fig. 1a, while strictly matching the definition, is not very intuitive. A more user-friendly way to visualize WFs is by means of conventional TDs, e.g. as shown in Fig. 1b generated in WORKCRAFT [12]. Here every transition is represented as a vertical arrow, while connections between transitions appear as curved arrows. To further match the appearance of a TD, horizontal lines are used to represent the state of a signal after a transition. The state of a signal can be identified by an upper/lower horizontal line (high/low) and a slashed/straight middle line (unstable/stable). The current state of the WF is indicated by the dotted vertical line across the waveform – all the events to its left have fired. For convenience the type of signals is color-codded: input are red, outputs are blue and internal signals are green.

The waveform from Fig. 1b includes every possible type of transition. In particular, *rise* and *fall* transitions are shown as upward and downward arrows (signals a and b). *Destabilize* and *stabilize* transitions (signal c) are represented by two arrows pointing inwards and outwards respectively. Finally, *stabilize at* 0 and 1 can be seen as upward and downward arrows in signals d and e respectively.

B. Waveform Transition Graph

In a simplistic way, a WTG can be seen as a Finite-State Machine in which the arcs are waveforms. Formally, a WTG is defined as the tuple $G = \{S, W, P_r, P_o, s_0, \Sigma, \gamma\}$ where:

- S is a finite set of nodal states.
- W is a finite set of waveforms.
- $P_r : W \to S$ is the waveform pre-set function.
- $P_o : W \to S$ is the waveform post-set function.
- s_0 is the initial nodal state.
- $\Sigma = \{\cup_{\sigma_i}\}$ is the set of signals of the WTG, composed by the union of signals of the waveforms $w_i \in W$, with $w_i = \{T_i, F_i, \sigma_i, \lambda_i\ \}$.
- $\gamma : W \times \mathit{In} \to \{0, 1, *\}$ is the waveform guard function.

The behavior of a WTG is reminiscent to that of a Petri net in the sense that a *token* decides the activation rules for waveforms. Yet, for the WTG case, only one token can exist at any time. Initially, this token is placed in the initial nodal state s_0.

The waveform *pre-set* function P_r defines which WFs can be *activated* if the token is in a specific nodal state. The *post-set* function P_o indicates which nodal state will hold the token after a WF has been executed. We refer to the *post-set* and *pre-set* of a WF as the set of nodal states obtained from functions of P_o and P_r, respectively. Similarly, the *post-set* and *pre-set* of a nodal state is inferred from P_r and P_o, respectively. A nodal state s with more than one WF in its post-set is a *choice*. From now on, we consider that the graph induced by P_r and P_o is always connected. The state of a WTG is defined by the position of the token (i.e. the nodal state) and a set of transitions fired in a WF in its post-set.

A complete depiction of a WTG can be seen in Fig. 2, which specifies a simple buck controller. In this example, the initial state is also a choice between two waveforms: *late_or_not_zc* and *early_zc*. After the execution of one of the two waveforms, a final waveform, *oc_handling*, is executed before looping back to the initial state. The nodal state *choice* is in the pre-set of the WFs *late_or_not_zc* and *early_zc*, and in the post-set of *oc_handling*. Similarly, the post-set of *choice* is *late_or_not_zc* and *early_zc*, with the pre-set containing only *oc_handling*.

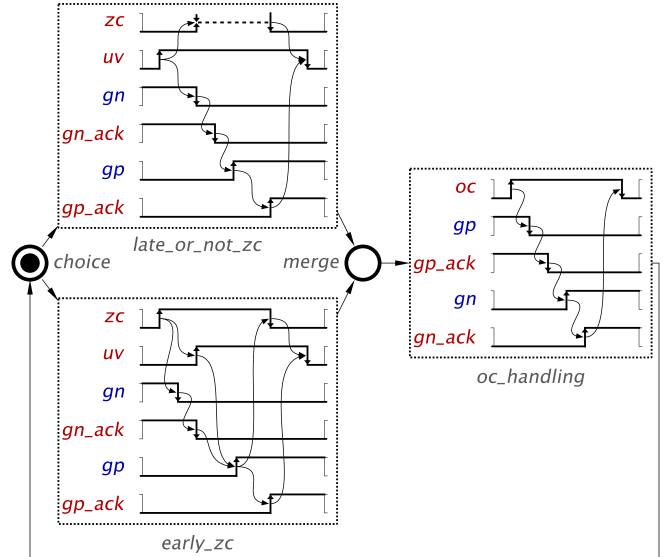


Fig. 2: WTG for a simple buck controller.

C. Boolean guards

Every WF can have a Boolean guard defined by a product of literals of the input signals. For example, the guard $a \cdot \overline{b}$ indicates that the WF can only be activated when $a = 1$ and $b = 0$. If $a = *$ in a guard, it indicates that the value of a is irrelevant. An empty guard is one in which all input signals have been assigned to $*$ (i.e. no literals are present in the guard). A WF is said to be *unguarded* if it is associated to an empty guard, otherwise it is said to be *guarded*.

Section III will discuss different structural properties of WTGs. One of them enforces the signals comprising a guard to be stable. Typically, guards are used to select WFs based on the value of stable input signals for which the actual value is still unknown. Additionally, every choice must have a consistent definition of guards in its post-set, i.e., either all WFs are guarded or all WFs are unguarded. In this sense, we distinguish between *guarded choices* and *unguarded choices* depending on whether the WFs in the post-set of a choice are all guarded or unguarded, respectively. Section III will also discuss the Boolean properties of the guarded choices.

An example on the use of guards can be seen in Fig. 3 with the specification of a small decoder from 2-bit binary to one-hot positional code. In this case, a pair of input signals $in1$ and $in0$ is destabilized in the initialization, only becoming stable before a request is signaled by req. A guarded choice then selects which waveform is executed: *code0*, *code1*, *code2* or *code3*. This depends on what values the signals $in1$ and $in0$ stabilize at, which is indicated by the guard expression on top

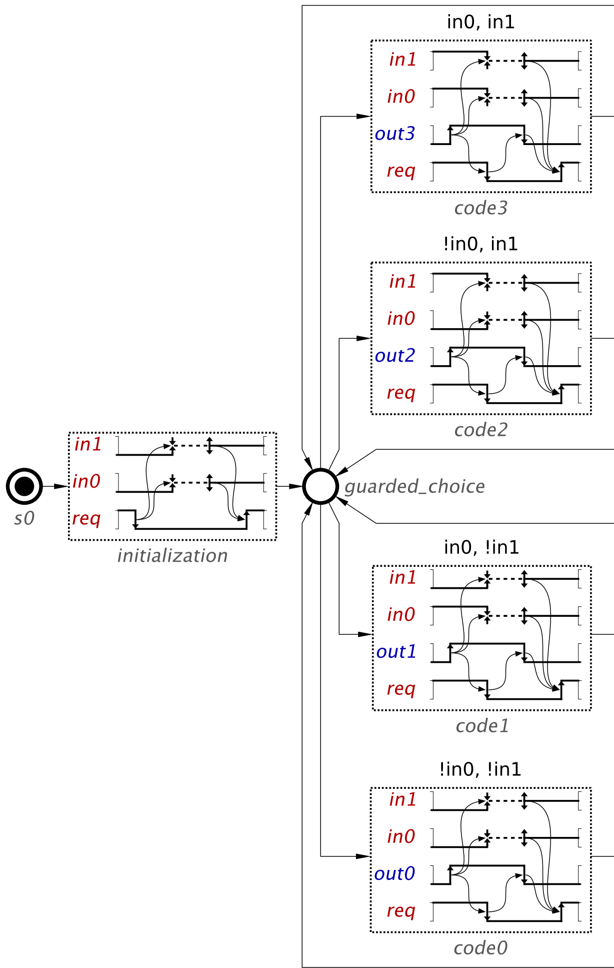


Fig. 3: Decoder from 2-bit binary to one-hot positional code.

of every waveform. The expression is a coma-separates list of expected values of input signals, e.g. a guard expression $in0, !in1$ indicates that the waveform should be executed when $in0$ is high and $in1$ is low. The waveform whose guard evaluates to true is executed: it first sets the corresponding output in the positional code high and then resets it when req goes low.

III. Implementability constraints

WTGs allow a wide range of specifications, some of which cannot be converted into hazard-free asynchronous circuits. Furthermore, the inclusion of new transitions and states, such as the *don't care* states, introduces some complexities that might be confusing to non-experts. There is, in particular, two properties that can alleviate these problems and should be intrinsic to the design of WTGs: *output-persistency* and *output-determinacy*.

Output-persistency violations can arise when transitions are *disabled*. A transition becomes disabled if it was enabled at some state, but the firing of a different transition leads to a state in which it is no longer enabled. If either transition, the disabling or the disabled, belongs to a non-input signal, this is considered a violation of output-persistency. These violations are important because they cause *hazards* in the circuit obtained by the synthesis procedure and should be avoided.

For synthesis, it is necessary for the specification to be output-determinate [13]. In essence, output-determinacy guarantees that the model is not self-contradictory, i.e. two identical traces of events must produce the same output. A model with *destabilize* transitions may easily fall into output-determinacy violations.

Here we propose a set of easy-to-check *structural properties* which guarantee that the complying WTG models are both output-persistent and output-determinate:

1) Transitions of non-input signals can only be triggered by *rise* or *fall* transitions.
2) Transitions of output signals can belong to the initial transition set of a WF only if the final transition set of every WF in the pre-set of the nodal state in the pre-set are composed exclusively by *rise/fall* transitions.
3) The initial transitions of the waveforms in the post-set of an *unguarded choice* must:
   3a) Belong to input signals.
   3b) Be disjoint between the waveforms in the post-set of the choice.
   3c) Be either *rise* or *fall*.
4) The guard conditions for the waveforms in the post-set of a *guarded choice* must:
   4a) Not be empty (i.e. all the waveforms in a guarded choice have a guard).
   4b) Be pair-wise independent.
   4c) The literals belong to signals in *stable* state.
   4d) Be complete, i.e. the disjunction of Boolean expressions of all guards is a tautology.
5) The state of all signals in the initial nodal state must be either *high* or *low*.

Note that conditions for *unguarded choices* do not apply for *guarded choices* and vice versa.

Disabling a signal requires some form of choice. In WTGs this can only happen in *choice* nodal states or as a side-effect of transitions different than *rise/fall*. Condition 3a) prevents disabling output transitions in choices. Transitions different than *rise/fall* cannot be directly observed by the system and can potentially disable an output. These issues are avoided by condition 1) and 2). Condition 2) is an extension for 1) to prevent triggering an output transition between different WFs. These conditions are then sufficient to avoid violations on output-persistency.

Most of the remaining conditions are in place to guarantee output-determinacy. Conditions 3b) and 3c) aim to determinize choices in a WF, preventing output-determinacy violations. Without condition 3b), two WFs after a choice might produce indistinguishable sequences of input transitions that lead to different output transitions. Condition 3c) prevents a similar case, in which a choice is decided by a transition that cannot be observed by the system.

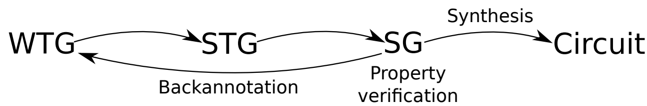


Fig. 4: Synthesis and verification flow.

Conditions 4a), 4b) and 4c) similarly aim to avoid scenarios in which output-determinacy is not preserved. In the case of guarded choices, the execution of a WF is decided according to signal levels, which are determined by past transitions. For that reason, output-determinacy can be guaranteed with conditions over the guards, rather than the initial transitions. In essence, there can be no intersection of what Boolean conditions are true. This explains conditions 4a) and 4b). Condition 4c) ensures that the signal levels for the signals in the guard remain stable, enabling unique identification of the WF which is being executed. Condition 4d) aims to avoid involuntary *dead-locks* in the specification by requesting at least one of the guards for the waveforms in conflict evaluates to true.

Finally, condition 5) exists to prevent cases in which signals that are, initially, in the *destabilize/stable* state do not transition before reaching a guard. This may result in a violation of output-determinacy, since potentially two identical traces could enable two mutually exclusive WFs with different output transitions.

IV. Synthesis and verification flow

One of the cornerstones on the design of WTG is to be able to use existing infrastructure and tools for asynchronous circuits. This is possible because WTG corresponds to a subset of STG and an easy transformation from WTG to STG is available. This is predominantly exploited in the synthesis and verification flow of WTG.

Fig. 4 shows an overview of the synthesis flow. First, WTG is transformed into STG while making use of *back-annotation* into every place and transition. In essence, this back-annotation allows tracking the correspondence between every element of the STG into the WTG. A similar process is then performed from STG to SG, along with additional back-annotation that enables the same kind of tracking between STG and SG. Once in SG, the process of synthesis and verification can take place using preexisting tools, such as Petrify [4]. It is also possible to work directly with the STG with tools that support it, as is the case with MPSat [6]. If problems arise during verification or even synthesis, the back-annotation allows traces with errors to be propagated all the way to WTG.

Thanks to this, all the process related to synthesis and verification can easily take place without further tool implementations other than the transformation into STG.

A. Conversion of WTG into STG

The conversion of WTG into STG is very straightforward. In WTG, the meaning of nodal states is strongly related to the notion of places in STG, while transitions have identical semantics in both models. In fact, this conversion would be almost trivial were it not for the support of the extended transitions and guards.

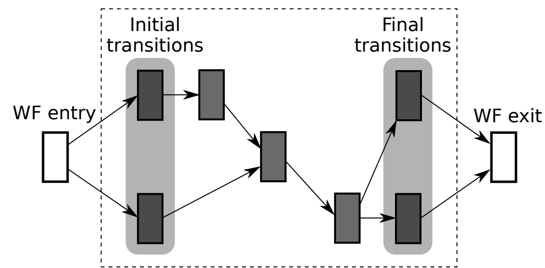


Fig. 5: STG representation of a WTG waveform.

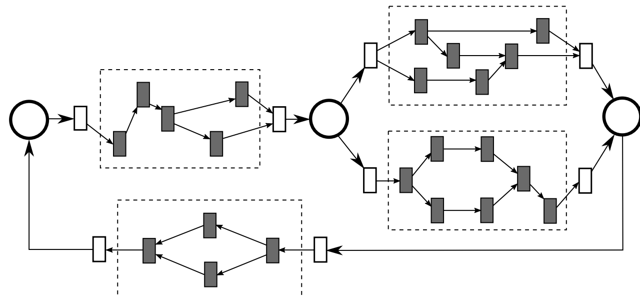


Fig. 6: STG representation of a WTG.

For simplicity, let us provisionally assume that there are no guards and only *rise/fall* transitions are supported. Fig. 5 depicts the STG obtained from a waveform. In it, every WF transition is converted into a STG transition and every arc defined by F corresponds to an arc between transitions in the STG. The figure also highlights two groups of transitions: the *initial* and *final* transitions. These transitions were defined earlier as the minimal and maximal elements in a WF, respectively. The conversion into STG requires two new dummy transitions, the *WF entry* and *WF exit* transitions, which are represented as empty boxes in Fig. 5. These dummy transitions must be connected to the *initial* and *final* transitions.

The nodal states are directly converted into places in the STG. For every connection between a nodal state and a WF, the STG will contain an arc between the place representing the nodal state and the WF entry/exit transition of the WF, depending on the direction of the arc. Finally, a token is added to the STG place that represents the initial nodal state in the WTG. Fig. 6 shows an example of an STG converted from a WTG.

Since the standard STG does not recognize transitions different than *rise/fall* (along with dummy, which do not explicitly exist in WTG), the modeling of these require the addition of a structure that *simulates* their behavior. In WTG, at any moment, a signal is in two of the following four states:

- High or low.
- Stable or unstable.

This can be simulated in STG by adding, for every input signal (non-input transitions can only be *rise/fall*), four places representing each of the four states. In this structure, a token

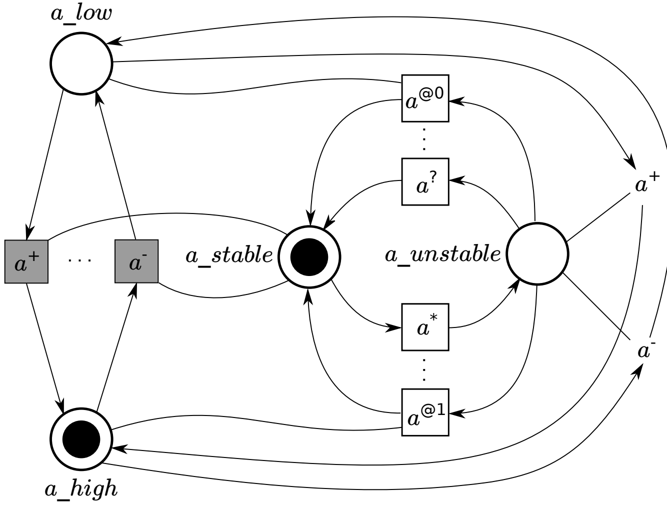


Fig. 7: STG structure simulating WTG transitions for a.

in a given place determines the state of a signal.

The inclusion of these transitions requires the following steps:

- Every transition different than *rise/fall*, is converted into a *dummy* transition in the STG.
- For every input signal, a new structure comprised of four places and two transitions is added to the STG.
- Every transition of every input signal is connected to the simulating structure in a specific way.

The new structure and its connections is represented by Fig. 7 for a signal a. In this figure, transitions enclosed in a box represent transitions that belong to a specific WF, while the two transitions without a box (a^+ and a^-, on the right side of the figure) belong to the structure itself. *Rise* and *fall* transitions appear shadowed to indicate that they are not dummy. Every other transition is converted into a dummy transition in the STG. The places representing the four states are identified by the name of the signal, following the state: *a_high*, *a_low*, *a_stable* and *a_unstable*. At any moment, there must be a token in either *a_high* or *a_low* and a token in either *a_stable* or *a_unstable*.

As it can be seen, a token in *a_unstable* allows the STG to arbitrarily transition between *high* and *low*. This is accomplished by connecting, with a *read-arc* (i.e. a bidirectional arc), the place *a_unstable* with transitions a^+ and a^-. Conversely, a token in the *a_stable* place disallows the firing of the transitions a^+ and a^- that belong to the simulating structure. The connection between *a_high* and *a_low* to those same transitions enables the transfer of the token from one to the other as the signal fires *rise* and *fall* transitions in a WF.

Additionally, every transition in a WF for every input signal must be connected to the simulating structure in a specific way, as represented by Fig. 7. For the example signal a, these connections are:

- *Rise/Fall*: These transitions are only allowed if the signal is *stable* and change the state between high and low. This is enforced by connecting these transitions with a read-arc to *a_stable* and the appropriate connections to the places representing *high* and *low* states. As an example, for *rise* this is ensured by a connection from *a_low* to a^+ and from a^+ to *a_high* (i.e. moving the token from *low* to *high*).
- *Destabilize/Stabilize*: These transitions must move the token between the *stable/unstable* states. As such, a $a^?$ transition has an arc from *a_unstable* and an arc towards *a_stable*. For a a^*, the arcs go in the opposite way, from *a_stable* to a^* and from a^* to *a_unstable*.
- *Stabilize at 0/1*: These transitions are similar to $a^?$ and so share the same arcs: from *a_unstable* and towards *a_stable*. Additionally, since they must also ensure a *high/low* state, they also include a read-arc with *a_high* or *a_low*, for $a^{@1}$ and $a^{@0}$ respectively.

The last elements that remain to be translated are guards. These are now very easy to convert by making use of the same simulating structure. In particular, a guard for a waveform must ensure that a signal has a specific state. This can be enforced by adding read-arcs between the WF entry transition and the places in the STG that represent the required state for every signal. Fig. 8 shows an example of the conversion of a guarded choice into STG. For simplicity, the WFs are represented as a graph and guard conditions are expressed in the arcs. In the example, the guard for the function $a \cdot \overline{b}$ is simulated by adding a read-arc between the dummy transition representing the WF entry and *a_high* and *b_low*. Since both signals must be stable before reaching the guard, additional read-arcs are added towards *a_stable* and *b_stable*. The other guards are similarly simulated by their own corresponding arcs.

V. Design automation

The design automation of WTG is currently implemented in the Workcraft [12] toolkit. It offers a a comprehensive implementation of WTG that allows the design, verification and synthesis of specifications. Besides tools, it is worth considering that electronic designers might not be used to the classical design flow of asynchronous circuits. For that reason, we additionally propose a set of strategies that address the design flow of WTG at different degrees of simplicity: the use of design guidelines to guarantee synthesis, exploiting back-annotation to report issues, and making necessary timing assumptions when no other option is available. These strategies, which can be used individually or in combination, are described in the following subsections.

A. Guideline-driven development

The guideline-driven development is the only strategy currently implemented in Workcraft. The idea behind this strategy is to abstract the designer from the issues and quirks of asynchronous circuit design that prevent synthesis, such as irreducible encoding conflicts. To accomplish that, the

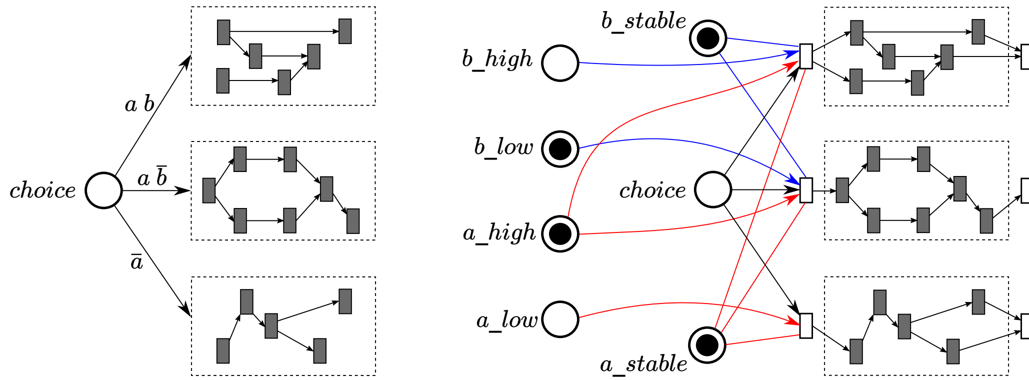


Fig. 8: Representation of a guarded choice in WTG on the left. Its conversion into STG is shown on the right. Red and blue lines represent arcs from the simulating structure of a and b respectively.

toolkit must enforce the structural properties for circuit implementability and a set of additional constrains (the guidelines) that give a strong guarantee of synthesis.

In order to be implementable as a circuit the specification must have *complete state coding* (CSC). Two states are in *CSC conflict* if they exhibit the same value of all the signals (encoding), but enable different non-input signals. There is no way to distinguish such states, and yet different outputs should be produced. The tools are usually capable of resolving CSC conflicts automatically by inserting new internal signals in such a way that differentiate the encoding of the conflicting states. There are, however, *irreducible CSC conflicts* that cannot be handled automatically.

A CSC conflict becomes irreducible when the only way to distinguish conflicting states is by inserting a signal before an input event. This, however, cannot be done automatically as delaying the input would requite the change of communication protocol between the circuit and its environment. There are two situations leading to irreducible CSC conflicts: i) conflicting states are separated by a sequence of input events; ii) there is a choice state from which the conflicting states are reachable via the input-only traces. In both cases, in order to resolve the conflict, a signal would need to be inserted before an input event, which cannot be done automatically as it requires designer to explicitly change the specification.

In the context of WTG, two additional constrains are sufficient (but not necessary) to prevent irreducible CSC conflicts:

- Transitions of the same input signal must be separated by an output signal transition.
- Waveforms in the post-set of an unguarded choice cannot contain sequences of input transitions that correspond to different permutations of the same events before an output signal fires.

It is important to note that the application of these guidelines has an impact on the expressiveness of the model. Yet, for non-expert designers, it is useful to have a strong guarantee that the design can be synthesized without having to understand some of the complexities inherent to asynchronous design.

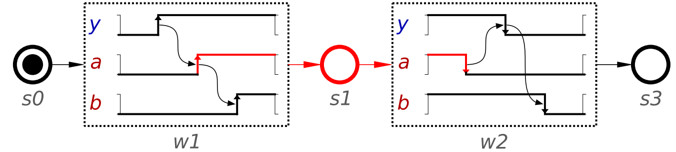


Fig. 9: Report of an irreducible conflict in WTG. The alternating transitions of a, which produce the conflict, are highlighted in red.

B. Back-annotation of synthesis issues

As previously mentioned, the proposed synthesis flow includes the transformation of WTG into STG and SG, along with back-annotations that allow tracking elements of the specification between the models. This can be exploited in the design stage in order to report errors, warnings and different issues that might arise during synthesis. It is possible, for example, to detect an irreducible CSC conflict in the SG and report a trace in WTG. This can be further enhanced by the inherent ability of WTG to display graphical information to the user.

Fig. 9 shows an example of such a report. The WTG presents an irreducible conflict that prevents synthesis and requires an action on the part of the designer. This can be displayed to the user by using, in this case, a red highlight that clearly indicates the location of the problem. It is then up to the designer to chose the best path of action to resolve the issue.

This synthesis strategy gives more control and flexibility to the user and is reminiscent to the classic design flow of other models for asynchronous design. The drawback is that a designer needs to understand the issues and their possible solutions, so it might not be appropriate for newcomers.

C. Timing assumptions

The addition of *destabilize/stabilize* transitions and guards makes WTG particularly well suited to specifications that include synchronous signals. For these cases, it is often impossible to synthesize a circuit that belongs to the speed-independent family. Fig. 10 shows a WTG specifying the

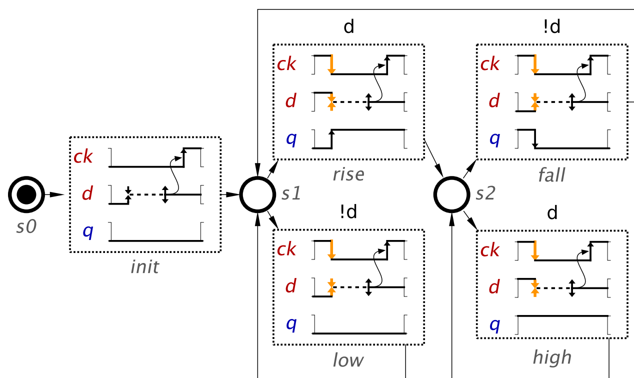


Fig. 10: WTG specification of a D Flip-Flop.

behaviour of a D Flip-Flop. The use of guards makes this specification simple to read. Yet, this specification contains irreducible conflicts that cannot be solved.

A way to go around this is the introduction of *timing assumptions*. In this case, synthesis is possible as long as we assume that the environment (i.e. ck and d) is slower than the circuit. This corresponds, for the case of a D Flip-Flop, to a *hold* constrain. In the example, the *fall* transitions for ck and the *destabilize* transitions for d (shown in the figure as thick amber lines) should only be fired after the circuit has had time to update its state. This necessary condition can be identified by the synthesis tool and reported to the user by highlighting the transitions that require timing assumptions.

The use of timing assumptions is not new. PETRIFY implements the option to perform synthesis with assumptions of a slow environment, as discussed in [14]. This can be extended to WTG by detecting when this is required for synthesis. The user can then have the option to give up speed-independence for those cases in which no other alternative is available.

VI. Conclusions

This work addresses one of the main issues preventing asynchronous circuits from proliferating in commercial systems. Despite the many advantages for this type of circuits, the steep learning curve and unfamiliarity with current models for electronic designers has proved to be too high an entry barrier.

The WTG model presented in this paper is a serious attempt to overcome this challenge. Designed to be as familiar as possible to the designer, WTG remains very expressive in comparison with existing models. Furthermore, its easy conversion into STG allows the re-use of all the previous work and tools build by the asynchronous community.

In order to further increase the ease of access, we describe a set of strategies for the design flow. These range from ensuring synthesizability by following a set of guidelines to go around synthesis issues by allowing timing assumptions. Yet more work still needs to be done in this area, as these strategies currently lack proper implementation and support for WTG.

Acknowledgements

This work was supported in part by funds from the Spanish Ministry for Economy and Competitiveness and the European Union (FEDER funds) under *Grant TIN2017-86727-C2-1-R* and in part by the Generalitat de Catalunya under *Grant 2017 SGR 786* and *Grant FI-DGR 2015*. WORKCRAFT design automation for WTGs was sponsored by EPSRC Impact Acceleration Account under grant *Waveform-based design flow for A4A circuits*.

References

[1] J. Spars and S. Furber, *Principles asynchronous circuit design*. Springer, 2002.

[2] S. Nowick and M. Singh, “Asynchronous design – part 1: Overview and recent advances,” *IEEE Design & Test*, vol. 32, no. 3, pp. 5–18, 2015.

[3] D. Sokolov, V. Dubikhin, V. Khomenko, D. Lloyd, A. Mokhov, and A. Yakovlev, “Benefits of asynchronous control for analog electronics: Multiphase buck case study,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017, pp. 1751–1756.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, “A region-based theory for state assignment in speed-independent circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, pp. 793–812, 1997.

[5] W. Belluomini, C. Myers, and U. Hofstee, “Verification of delayedreset domino circuits using Atacs,” in *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 1999, pp. 3–12.

[6] V. Khomenko, M. Koutny, and A. Yakovlev, “Logic synthesis for asynchronous circuits based on STG unfoldings and incremental SAT,” *Fundamenta Informaticae*, vol. 70, no. 1–2, pp. 49–73, 2006.

[7] J. Cortadella, A. Moreno, D. Sokolov, A. Yakovlev, and D. Lloyd, “Waveform transition graphs: A designer-friendly formalism for asynchronous behaviours,” in *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2017, pp. 73–74.

[8] G. Borriello, “A new interface specification methodology and its application to transducer synthesis,” University of California, Berkeley, Tech. Rep., 1988.

[9] K. Yun, D. Dill, and S. Nowick, “Synthesis of 3D asynchronous state machines,” in *International Conference on Computer Design (ICCD)*, 1992, pp. 346–350.

[10] P. Vanbekbergen, C. Ykman-Couvreur, B. Lin, and H. De Man, “A generalized signal transition graph model for specification of complex interfaces,” in *European Design and Test Conference (EDAC)*, 1994, pp. 378–384.

[11] A. Yakovlev, A. Petrov, and L. Rosenblum, “Synthesis of asynchronous control circuits from symbolic signal transition graphs,” in *Asynchronous Design Methodologies*, ser. IFIP Transactions, S. Furber and M. Edwards, Eds., vol. A-28. Elsevier Science Publishers, 1993, pp. 71–85.

[12] 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*, A. Mokhov, Ed. Newcastle University, 2016, available online http://async.org.uk/ay-festschrift/paper25-Alex-Festschrift-2ed.pdf.

[13] V. Khomenko, M. Schaefer, and W. Vogler, “Output-determinacy and asynchronous circuit synthesis,” *Fundamenta Informaticae*, vol. 88, no. 4, pp. 541–579, 2008.

[14] J. Cortadella, M. Kishinevsky, S. M. Burns, A. Kondratyev, L. Lavagno, K. S. Stevens, A. Taubin, and A. Yakovlev, “Lazy transition systems and asynchronous circuit synthesis with relative timing assumptions,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 2, pp. 109–130, 2002.