Static Data Flow Structures with Dynamic Elements

Ivan Poliakov†, Charles Brej‡, Danil Sokolov†, Alex Yakovlev†
ivan.poliakov@ncl.ac.uk, cb@cs.man.ac.uk, danil.sokolov@ncl.ac.uk, alex.yakovlev@ncl.ac.uk
†Newcastle University, ‡University of Manchester

***Abstract*— The paper proposes an extension of the Static Data Flow Structure model (SDFS) that simulates the influence of the control path elements on the data path. One of the shortcomings of the SDFS is the lack of non-deterministic data path execution, which makes it difficult to model real-life examples of hardware that often rely on choice. Several additional elements are introduced in this paper to resolve this issue, and the method for verification of the extended SDFS models is discussed.**

I. Introduction

Traditionally, the asynchronous circuits described using a high-level hardware description language such as Verilog or VHDL are decomposed into two parts for the purpose of analysis and optimisaton: the data path and the control path. Such separation is required because the goals for these paths are often different: the control path is optimised for low latency and size, while the data path is optimised for power consumption and security. The synthesis and analysis of the control path is well supported by tools such as Petrify [1], but the theory and tools for the data path are lacking.

Representing the complex system of handshake elements that constitute an asynchronous circuit datapath in a way not only intelligible to the designer, but also amenable to automated analysis and verification is a daunting task. The Static Data Flow Structure (SDFS) model proposed in [9] is a flexible and elegant way of achieving both the visual comprehensibility and robust verification mechanism. This paper builds on the work presented in [9] in an attempt to expand the class of asynchronous circuits that can be described using SDFS.

II. Static Data Flow Structures

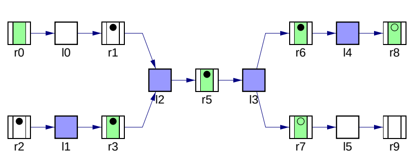


Fig. 1. A simple SDFS

The static data flow structures (SDFS), introduced informally in [10] and refined in [9], are a high-level model for asynchronous data path that is as an equivalent to register transfer level (RTL) in synchronous design. The marking of the SDFS and the token game semantics are closely related to the architecture of the asynchronous data path, e.g. the propagation of tokens through the SDFS can be associated with the propagation of "request" signals, while removal of the tokens can be associated with the reception of "acknowledge" signals.

Definition 1: An SDFS [9] is a directed graph $G = \langle V, E, D, M_0 \rangle$, where V is a set of vertices (or nodes), $E \subseteq V \times V$ is a set of edges denoting the flow relation, D is a semantic domain of data values and M_0 is an initial marking of the graph. There is an edge between nodes $x \subseteq V$ and $y \subseteq V$ iff $(x, y) \in E$. There are two types of vertices with different semantics: registers R and combinational logic nodes L, $R \cup L = V$. The registers can contain tokens, thus defining the marking M of an SDFS. The tokens can be associated with data values from the semantic domain D. The marking of an SDFS may evolve by firing, which is a process of marking the registers with tokens or removing the tokens from the registers according to the firing rules. Several sets of firing rules are defined in [9] that assign different semantics to the token propagation process. A graphical representation of a simple SDFS is shown in Figure 1. In the figure, the combinational logic nodes are shown as boxes, either filled or empty according to the evaluation state, and the register nodes are shown as boxes with two vertical lines. A token is shown as a filled circle.

A. Problematic case

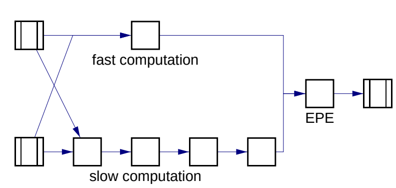


Fig. 2. Deterministic behaviour limits the flexibility of SDFS

Consider an example shown in Figure 2. The data that comes into this section of the data path may need to be processed via two alternative computation paths, one significantly slower than the other. If this situation is modelled using plain SDFS, both paths will start executing the computation simultaneously. Although the faster result can be output immediately by making the join element early propagative (EPE), in order to start the next wave of the computation the execution of the slower path still needs to be completed. If a more complex token game, such as counterflow, were used in the pipeline, then the execution of the slower path could be interrupted. However, it is very likely that in the modelled system only one path is enabled at a time, and the computation in the other branch should not start at all. Hence, it is impossible to produce the expected behaviour using the SDFS model.

III. Dynamic elements

To resolve the limitation explained in Section II-A, it is necessary to introduce elements that would model the influence of the control path on the underlying data path. These elements are called *dynamic,* because they modify the otherwise static, or deterministic, execution flow of the model. To model the activity of the control signals, it is necessary to introduce a new class of tokens, called *control tokens,* that would represent the propagation of the control signals in a way similar to the propagation of data. As opposed to the data tokens, that represent abstract data items in the SDFS model, the control tokens need to be associated with the actual data values. In the scope of this paper, only 2 values are used, depicted as ⓪-token and ①-token.

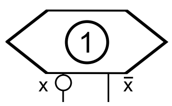


Fig. 3. Graphical representation of the Control node

A. Control

The Control node acts similarly to the spread token SDFS register, with the exception that it propagates control tokens preserving their values. Note that the Control is allowed to be connected only to the Push/Pop nodes or to another Control node.

A Control node is initially in a disabled state. It can be *enabled* iff all nodes in its preset are marked with a token. An enabled node can be *marked* with a ⓪-token if it is enabled, not yet marked and all nodes in its preset are marked with a ⓪-token, and, similarly, it can be marked with a ①-token iff all nodes in its preset are marked with a ①-token, thus achieving the propagation of the tokens while preserving their values. A marked Control node can become *disabled* iff any of its preset nodes become unmarked, and the token can be removed from a disabled node iff none of its preset nodes hold a token.

If a Control node has an empty preset, it is called an *external control* node. An external control node is always enabled and can be marked either with a ⓪-token or a ①-token in a free choice. A Control node is not allowed to have an empty postset.

B. Push

Push is an element that, depending on the choice made by the control, either forwards the data token or destroys (acknowledges) it. Paired with pop, it can be used to select one of several possible paths of the data flow. The Push element is comprised of the three blocks: the *outer interface (OI)*, the *inner interface (II),* and the *control interface (CI)* (Figure *4*). The outer and inner interfaces act as a pair of regular SDFS registers for the other SDFS nodes, i.e. they can be enabled,

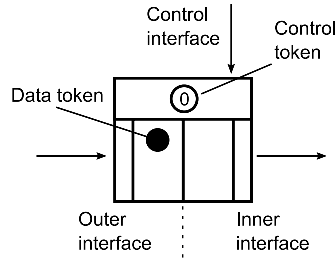


Fig. 4. Graphical representation of the Push and Pop nodes

disabled, marked and unmarked; however the marking visible to its postset and preset nodes is different. If an SDFS node is in the Push node's preset, it reads the marking of the *outer interface.* If an SDFS node is in the Push node's postset, it reads the marking of the *inner interface.*

The transfer of tokens between the outer and inner interfaces is governed by a special set of rules, which are as follows. Note that *preset, postset, r-preset, r-postset* are defined for the Push and Pop elements in the same way they are defined for regular SDFS elements [9].

The OI, II and CI are initially disabled and unmarked. The OI can become enabled iff all registers in the Push's preset are marked and all logic nodes in the Push's preset are evaluated. The OI can become marked with a token iff it is enabled, the II and CI are unmarked, the r-preset of the Push is marked. OI can become disabled iff any of the registers in the Push's preset becomes unmarked or any of the logic nodes in the Push's preset becomes reset. The disabled OI can be unmarked iff the r-preset of the Push is unmarked and the II is marked.

The II can become enabled iff the OI holds a token and the CI holds a ①-token. The enabled II can be marked iff the r-postset of the Push is unmarked.

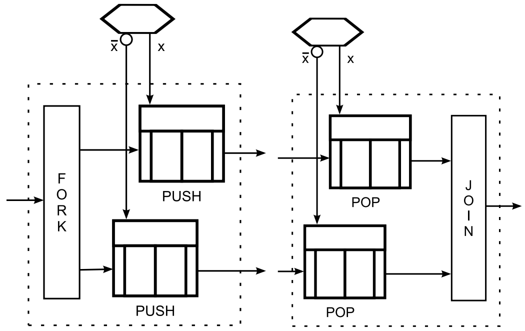
The CI behaves according to the similar set of rules as a Control node, with the exception that it can only accept a token when the OI is marked, and can be unmarked when the OI is unmarked.

To summarise, the Push element synchronises a data token on the outer interface with a control token. If the control token is a ①-token, it forwards the data token by transferring it into its inner interface, and then allows the token to be removed from the outer interface. If the data token is a ⓪-token, it allows the token to be removed from the outer interface without transferring it into the inner interface.

C. Pop

Pop is an element that, depending on the choice made by the control, either forwards the data token or produces a dummy token. Paired with push, it can be used to select one of several possible execution paths. Its structure is similar to the Push, but the marking rules are different and are as follows.

The OI, II and CI are initially disabled and unmarked. The OI can become enabled iff all registers in the Pop's preset are marked and all logic nodes in the Pop's preset are evaluated. The OI can become marked with a token iff it is enabled, the II is unmarked, the CI is holds a ①-token and the r-preset of the Pop is marked. OI can become disabled iff any of the



(a) Demux (b) Mux

Fig. 5. Implementation of the multiplexer and demultiplexer using dynamic components

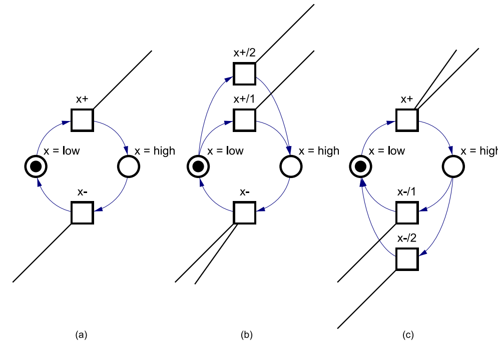


Fig. 6. Elementary cycle examples

registers in the Pop's preset becomes unmarked or any of the logic nodes in the Pop's preset becomes reset. The disabled OI can be unmarked iff the r-preset of the Pop is unmarked and the II is marked.

The II can become enabled if the OI holds a token and the CI holds a ①-token, or if the OI does not hold a token and CI holds a ⓪-token. The enabled II can be marked iff the r-postset of the Pop is unmarked.

The CI behaves according to the similar set of rules as a Control node, with the exception that in can only be marked when the II is unmarked, and can be unmarked when the II is marked.

To summarise, the Pop element first receives a control token. If it is a ①-token, it then synchronises it with a data token on the outer interface and transfers it into the inner interface. If the token is a ⓪-token, it immediately produces a dummy data token on the inner interface.

D. Mux and Demux

The multiplexer and demultiplexer are good examples of how the basic dynamic elements can be used. In Figure 5 (a), the demux is an element that, depending on the choice made by the control, forwards a data token from its input to one of its outputs. This is implemented using Push elements. Depending on the value of the control token, one of the Push elements receives a ①-token and forwards the input token received via the fork element, and the other one receives a ⓪-token and blocks the token from entering its corresponding data path.

The mux (Figure 5 (b)) is an element that, depending on the choice made by the control, forwards a data token from one of its inputs to its output. Mux is implemented using two Pop elements. Depending on the value of the control token, one of the Pop elements receives a ①-token and forwards the input token to the join element, while the other one receives a ⓪-token and generates a dummy token that is also sent to the join element, where it is OR-ed with the actual data token resulting in the propagation of the data from the selected channel.

The combination of these two elements allows us to address the limitation of the original SDFS model as explained in Section II-A.

IV. Verification of the Dynamic Elements

A. Generalised Petri net mapping technique

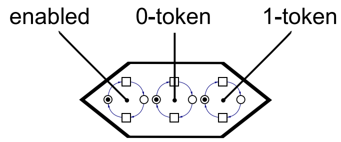
Definition 2: A *signal* is a 4-tuple S = $\langle n, s, f_{set}, f_{reset}\rangle$, where *n* is the symbolic signal name, *s* is the signal state, f_{set} and f_{reset} are Boolean functions; *s* can initially be either 0 (low) or 1 (high), and can be changed from 0 to 1 at any time when f_{set} evaluates to true, and from 1 to 0 at any time when f_{reset} evaluates to true.

A set of signals associated with a single high-level object *obj* is called an *expansion* E_{obj}. A high-level object that has an associated expansion is called a *source node N*. For example, a source node can be a register R the state of which is described using two signals: $E_R = \{S_{marked}, S_{enabled}\}$.

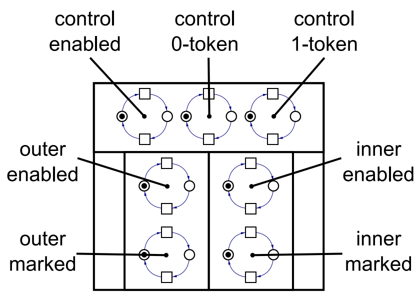
A *source system* Sys_{src} is a set of source nodes. A Petri net into which the source system is mapped is called a *destination system* Sys_{dst}.

1) Circuit Petri Nets: Circuit, or level-based Petri nets [3], [11], is a class of Petri Nets [6] that best describes systems comprised of a set of interdependent binary signals. A signal is represented by an *elementary cycle* (Figure 6) that consists of two complementary places: one place has a token when the signal is high, and the other one when the signal is low. The places cannot hold tokens simultaneously, which is ensured by the fact that they are only connected via a number of "positive" transitions that transfer the token from the negative place to the positive place, and "negative" transitions that do the opposite. The transitions are controlled by *read arcs* [5] that non-destructively test the presence of tokens in the places of other cycles, essentially reading the state of other signals.

2) Mapping algorithm: The mapping of the source system Sys_{src} into a Petri net Sys_{dst}, is obtained in two steps. First, for each signal of Sys_{src} an elementary cycle is added into Sys_{dst} and its places are labeled in such a way that it is possible to derive the source node name and signal name from the label and vice versa. Second, for each elementary cycle the positive and negative transitions are constructed from f_{set} and f_{reset} by first converting them into disjunctive normal form (a disjunction of conjunctive clauses), then for each resulting DNF clause adding correspondingly a positive or a negative transition between places, and then adding read arcs going



(a) Control element



(b) Push/Pop element

Fig. 7. Petri net mapping of the dynamic elements

from the transition to places that correspond to the signals present in DNF clause as literals. For the detailed description of the algorithm, please see [7].

3) Verification based on the result Petri net mapping: As soon as the Petri net mapping is constructed, it is passed to the external tools for verification. When the verification result is received, it is treated as either successful or failed. If the verification has failed, the trace of events that lead to the problematic state, on the Petri net level, is retrieved from the tool's output. Since it is possible to determine the high-level object corresponding to each place, a high-level trace can be built [7]. All these steps can be carried out transparently using the Workcraft framework [8]: the Petri net mapping and verification can be done behind-the-scenes, so the user can focus on the high-level model. Currently the main tool used by the Workcraft framework for the verification of Petri nets is MPSAT, which is based on unfoldings [4].

B. Mapping of the dynamic SDFS elements

To apply the verification method explained in Section IV-A to the newly proposed elements, it is necessary to define the set of signals that are to be mapped into elementary Circuit Petri Net cycles (Figure 6).

For the Control node (Figure 7 (a)), 3 signals are necessary: the enabling state of the Control node, and the presence of the control token. Because the control token can carry data, it has to be represented with more than one signal to encode the "value" of the token. In the scope of this paper, the control tokens are allowed to only have 2 different values: 0 and 1, and thus two signals are enough to encode the value. The token presence signal can also be encoded using the same signals, similar to dual-rail encoding: the 00 value means "no token", 10 means "0-token present", 01 means "1-token present" and the value of 11 is not allowed. To build the firing rules that need to test only for the presence of a token (and do not care about its value) in the form of Boolean equations an OR construct is used. This approach also allows to extend the data domain if need arises simply by adding additional cycles.

For the Push and Pop nodes(Figure 7 (b)), the number of required signals is higher because they act as a 3-way node: they accept control tokens, data tokens and can generate (dummy) data tokens themselves. The signals for the outer and inner interfaces are the same as for the usual SDFS register: enabling and marking, and the signals for the control interface are the same as for the Control node: enabling and 2 signals for the control token value.

Once the elementary Circuit Petri Net cycles are constructed for each of the signals, they are ready to be interconnected with read arcs to impose the firing rules. The rules are explained informally in Section III, and for the more formal explanation of the read arcs construction procedure please see [7].

V. Conclusions and future work

Several additional elements are introduced in the paper that extend the SDFS model. The new elements assist in simulation of the data path behaviour that is non-deterministic and is dependent on the signals produced by the control path. The token game rules for the new elements are defined in such a way that they are compatible with the previously defined SDFS elements, which allows the new elements to be easily integrated into the existing SDFS models. A method for verification of the extended model is given. The proposed extensions improve the flexibility and usability of the SDFS, making it possible to translate a high-level hardware description, written in language such as Balsa [2], directly into the SDFS model for verification and analysis. A method for such translation is the focus of the future work.

Acknowledgement: this work is supported by EPSRC, GR/D053064/1 and GR/D052238/1.

References

[1] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.

[2] Doug Edwards and Andrew Bardsley. Balsa: an asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[3] J. Grabowski. On the analysis of switching circuits by means of Petri nets. *Elektronische Informations-verarbeitung und Kybernetik*, (14):611–617, 1978.

[4] Victor Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, University of Newcastle upon Tyne, School of Computing Science, 2003.

[5] U. Montanari and F. Rossi. Contextual nets. *Acta Informacia*, 32(6):545–596, 1995.

[6] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[7] Ivan Poliakov, Andrey Mokhov, Danil Sokolov, and Alex Yakovlev. High-level model verification within workcraft framework. In *19th UK Asynchronous Forum*, 2007.

[8] Ivan Poliakov, Danil Sokolov, and Andrey Mokhov. Workcraft: A static data flow structure editing, visualisation and analysis tool. In *Petri Nets and Other Models of Concurrency - ICATPN 2007*, 2007.

[9] Danil Sokolov, Ivan Poliakov, and Alex Yakovlev. Asynchronous data path models. In *International Conference Application of Concurrency to System Design*, July 2007.

[10] Jens Sparsø and Steve Furber. *Principles of asynchronous circuit design: a system perspective*. Kluwer Academic Publishers, 2001.

[11] V. Varshavsky, M. Kishinevsky, V. Marakhovsky, V. Peschansky, L. Rosenblum, A. Taubin, and B. Tzirlin. *Self-Timed Control of Concurrent Processes*. Kluwer Academic Publisher, Dordrecht, The Netherlands, 1990.