Analysis of Static Data Flow Structures

Danil Sokolov*, Ivan Poliakov, Alex Yakovlev
School of Electrical, Electronic and Computer Engineering
Newcastle University, Newcastle upon Tyne, UK
danil.sokolov@ncl.ac.uk; Alex.Yakovlev@newcastle.ac.uk

Abstract. A token-based model for asynchronous data paths, called static data flow structures (SDFS), is formally defined. Three token game semantics are introduced for this model, namely atomic token, spread token and counterflow. The SDFS semantics are analysed using a simple benchmark example; their advantages and drawbacks are highlighted. A combination of spread token and counterflow models, which employs the advantages of both, is presented. A technique is described for mapping the high-level SDFS token game semantics into the low level of underlying Petri nets (PNs). The PNs are employed as a back-end for verification of SDFS models. For analysis and comparison of SDFS semantics a software tool has been developed, which integrates all SDFS semantics into a consistent framework, implements their conversion into PNs and provides an interface to existing model checking tools.

1. Introduction

Design of an asynchronous circuit starts at the high level of behavioural specification, where control and data paths are described together using a conventional HDL (e.g., Verilog, VHDL, System-C). This high-level representation is subsequently decomposed into a set of communicating processes. The interaction between the processes is captured using a formal modelling language from which the specifications of control and data paths are extracted. The control and data paths are optimised and synthesised separately to improve the design features of each path independently. For example, the control path is often optimised for low latency and size, while the data path is optimised for power consumption and security. Such separation of control and data paths is employed in Pipefitter [3], TAST [11], VeriSyn [21], etc.

The synthesis of asynchronous control path is well developed and supported by tools (PETRIFY [10], Minimalist [12], 3D [8], OptiMist [22]). A designer can choose among the methods aiming at speed,
size and power optimisation. Decomposition techniques [28, 30] can be applied at this level to reduce the complexity of the controller specification.

On the other hand, the synthesis of asynchronous data path is not sufficiently studied. Usually, a designer makes the whole data path either bundled-data (to achieve smaller size of the circuit) or dual-rail (to get an average-case performance and hazard-free logic). In both cases the conventional EDA synthesis tools (e.g., Ambit, Synopsis) are used to obtain the data path combinational logic. The calculation of the matching delay for bundled-data [4] and conversion of the single-rail logic into NCL-X [15, 23] or other dual-rail implementations [31] have been automated. However, the synchronous nature of the underlying synthesis tools does not allow to exploit the full potential of asynchronous data path. For instance, the early evaluation cannot be controlled and influenced at the early synthesis stage.

The conventional EDA tools force data encoding to single-rail, which is justified for synchronous designs, but may result in redundancy when the obtained circuits are converted to dual-rail (e.g., the output of a three-way comparator can be 1-of-3 rather than 3 dual-rail lines). While alternative synthesis methods exist which allow the use of arbitrary delay-insensitive codes [27, 2], the choice of the particular encoding is not automated and is left to a designer intuition.

Note that combinational logic for asynchronous data path can also be obtained by the tools, which traditionally are used for synthesis of asynchronous controllers, for example PETRIFY [10]. These tools are based on the state-space exploration and require a circuit specification at low level of signal transitions. Therefore, the data path, first, has to be decomposed into relatively small blocks whose state space would not exceed the possibilities of modern computers. Then, each block needs to be refined to the level of signal transitions. Finally, these blocks are synthesised separately and connected together to form the data path circuit.

The choice of suitable synthesis methods for different parts of data path is also relevant. It may be the case that implementing some branch of a data path as “expensive” dual-rail does not give any speed advantage because there is a concurrent branch with is very slow and never exhibits early evaluation.

In order to decompose, optimise and efficiently synthesise the asynchronous data path it must be analysed at the level of formal technology independent model. Currently there is no formal model which adequately represents the asynchronous data path. The formal models, such as Petri nets (PNs) [18] and finite state machines (FSMs), are too abstract and do not mimic the behaviour of asynchronous data path. The models which naturally capture the asynchronous data path, such as SDFS [25], are not formally defined and require further research. In particular, modelling preemption, early evaluation and speculation in the asynchronous data path by SDFS is of great interest. Preemption is a technique which allows to destroy data items in a computation pipeline if the result of computation is no longer needed, thus reducing the power consumption. Early evaluation and speculation techniques are based on the preemption idea. Early evaluation allows a circuit to compute the output using a subset of its inputs and preempting the inputs which are not needed. In speculation, all conflicting branches of computation run concurrently without waiting for the selecting condition; once the selecting condition is computed the incorrect branches are preempted.

There has recently been an increase in research on design of self-timed data path logic and pipeline structures with much more sophistication in dynamic behaviour than simple Muller pipelines. However, the modelling, analysis and synthesis support is very limited. As a result there are examples of circuit level solutions [6, 1] that are insufficiently well analysed and the published circuits behave with certain undesirable effects [24]. Part of our goal is to provide a good formal model and tool support to enable designers to analyse such structures early on in their design process.
In this paper we formally define the SDFS model and introduce three token game semantics on this model: atomic token, spread token and counterflow. These semantics are compared and the advantages of each of them are studied. Atomic token model is intended as a formalisation of the original SDFS. Spread token SDFS model addresses the drawbacks of the atomic token model and introduces a rudimentary early evaluation support. The counterflow semantics is capable of modelling preemption, early evaluation and speculation in asynchronous data path. In order to support simulation, analysis and verification of these models a software tool has been developed which integrates arbitrary token-based models into a consistent framework.

The rest of the paper is organised as follows. Section 2 gives a basic background on PNs and STG models, which are used for verification of SDFS models. The SDFS model is formally defined in Section 3. Three token game semantics, namely, atomic token, spread token and counterflow, are introduced in Sections 4, 5 and 6 respectively. An interesting combination of spread token and counterflow models is presented in Section 7. A technique for converting the high-level SDFS models into underlying low-level Petri nets is described in Section 8. In Section 9 a software tool is presented which has been developed for analysis, comparison and verification of the SDFS semantics. Section 10 summarises the advantages and drawbacks of the SDFS semantics introduced in this paper.

2. Petri nets and signal transition graphs

This section presents the basic concept of PNs model and its special interpretation as STG. The definitions of this section are used in Section 8 for mapping the high-level SDFS models into the low-level PNs. A PNs model, first defined in [18], is a graphical and mathematical representations of discrete distributed systems. PNs extend the FSMs with a notion of concurrency, which makes them especially convenient for the specification and verification of asynchronous circuits. An STG is a special kind of PNs whose transitions represent signal events. This model was independently introduced in [9] and [20] and can be considered as a formalisation of the widely used timing diagrams.

Formally, a PN is defined as a tuple $PN = (P, T, F, M_0)$ comprising finite disjoint sets of places $P$ and transitions $T$, arcs denoting the flow relation $F \subseteq (P \times T) \cup (T \times P)$ and initial marking $M_0$. There is an arc between $x \in P \cup T$ and $y \in P \cup T$ iff $(x, y) \in F$. An arc from a place to a transition is called consuming arc, and from a transition to a place - producing arc. The preset of a node $x \in P \cup T$ is defined as $\bullet x = \{ y \mid (y, x) \in F \}$, and the postset as $x \bullet = \{ y \mid (x, y) \in F \}$.

The dynamic behaviour of a PN is defined as a token game, changing marking according to the enabling and firing rules. A marking is a mapping $M : P \rightarrow \mathbb{N}$ denoting the number of tokens in each place, $\mathbb{N} = \{0, 1\}$ for 1-safe PNs. A transition $t$ is enabled iff $\forall p, p \in \bullet t \Rightarrow M(p) > 0$. The evolution of a PN is possible by firing the enabled transitions. Firing of a transition $t$ results in a new marking $M'$(p) = \[
\begin{cases} 
M(p) - 1 & \text{if } p \in \bullet t \setminus \bullet t, \\
M(p) + 1 & \text{if } p \in t \bullet \setminus \bullet t, \\
M(p) & \text{otherwise}
\end{cases}
\] for all $p \in P$.

Graphically, places of a PN are represented as circles (○), transitions as boxes (□), consuming and producing arcs are shown by arrows (→), and tokens of the PN marking are depicted by dots in the corresponding places (●).

An extension of a PN model is a contextual net [17]. It uses additional elements such as non-consuming arcs, which only control the enabling of transitions and do not influence their firing.
A PN extended with a special type of non-consuming arcs, namely read-arcs, is defined as $PN = (P, T, F, R, M_0)$. A set of read-arcs $R$ is defined as $R \subseteq (P \times T)$, there is a read-arc between $p$ and $t$ if $(p, t) \in R$. The read-preset of a transition $t \in T$ is defined as $*t = \{ p | (p, t) \in R \}$, and the read-postset of a place $p \in P$ as $p* = \{ t | (p, t) \in R \}$. A transition $t$ is enabled iff $\forall p, p \in *t \cup t \Rightarrow M(p) \neq 0$. The rules for firing of the transitions are preserved. A read-arc is depicted as a line without arrows.

Formally, an STG is a 1-safe PN whose transitions are labelled by signal events, i.e. $STG = (P, T, F, M_0, \lambda, Z, v_0)$, where $\lambda$ is a labelling function, $Z$ is a set of signals and $v_0 \in \{0, 1\}^{|Z|}$ is a vector of initial signal values.

The labelling function $\lambda : T \rightarrow Z\pm$ maps transitions into signal events $Z\pm = Z \times \{+,-\}$. The signal events labelled $z+$ and $z-$ denote the transitions of signals $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. The labelling function does not have to be 1-to-1, i.e. transitions with the same label may occur several times in the net. In order to distinguish between transitions with the same label and refer to them from the text an index $i \in \mathbb{N}$ is attached to their labels as follows: $\lambda(t)/i$, where $i$ differs for different transitions with the same label.

STGs inherit the operational semantics of their underlying PNs, including the notions of transition enabling and firing. In this paper a set of read-arcs is also included into the STG model, which is an enhancement w.r.t. [20].

Important properties of STG are consistency and persistency. These properties are necessary, e.g., to derive circuits from STGs by logic synthesis [10]. An STG is called consistent if for each signal $z \in Z$ transitions labelled $z-$ and $z+$ alternate in any firing sequence starting from $M_0$. An STG is called persistent if no enabled transition can be disabled by firing another enabled transition.

### 3. Static data flow structures

The SDFS is a high-level model for asynchronous data path that can be viewed as an equivalent to register transfer level (RTL) in synchronous design. The SDFS has been informally introduced in [25] concentrating on the structural and syntactical aspects of the model. However, the token game semantics (enabling and firing rules) is only defined by examples and is ambiguous in some cases.

This section focuses on the structure and the syntax of the SDFS model. Token game semantics is an independent issue as it is closely related to the architecture of the asynchronous data path. The most interesting token game semantics are studied separately in the following sections.

**Static Data Flow Structure (SDFS)**

An SDFS is a directed graph $G = (V, E, D, M_0)$, 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 vertices $x \in V$ and $y \in V$ iff $(x,y) \in E$. There are two types of vertices with different semantics: register nodes (or simply registers) $R$ and combinational logic nodes (or simply logic) $L$. $R \cup L = V$. The registers can contain tokens, thus defining the marking $M$ of SDFS. The tokens can be associated with data values from the semantic domain $D$. The marking of SDFS may evolve by enabling and subsequent firing of register nodes. The rules of enabling and firing are defined by the token game semantics and are discussed separately for each semantics.
Presets and postsets
The \textit{preset} of a vertex \( x \in V \) is defined as \( \bullet x = \{ y \mid (y, x) \in E \} \) and the \textit{postset} as \( x \bullet = \{ y \mid (x, y) \in E \} \).

Source and sink
Only registers can have empty presets and postsets. A register with empty preset is called a \textit{source}, and with empty postset is called a \textit{sink}. Note that source and sink nodes can represent inputs and outputs of a data path respectively, thus modelling a device-environment interface.

Path and cycle
A sequence of vertices \((z_0, z_1, \ldots, z_n)\) such that \((z_{i-1}, z_i) \in E, i = 1 \ldots n\) is called a \textit{path} from \( z_0 \in V \) (called a \textit{start vertex}) to \( z_n \in V \) (called an \textit{end vertex}) and is denoted as \( \delta (z_0, z_n) \). Note that there can be several paths from one vertex to another or no path at all. A \textit{cycle} is a path whose start vertex is the same as end vertex. A path with no repeated vertices is called a \textit{simple path}, and cycle with no repeated vertices aside from the start/end vertex is a \textit{simple cycle}.

Deadlock and liveness
An SDFS reaches a \textit{deadlock} state if no firing can happen. If a deadlock state is not reachable the SDFS is called \textit{deadlock-free}.

An SDFS is called \textit{live} if all its registers can fire infinitely many times. In order to be live it is necessary for SDFS to have at least one token in every cycle. This leads to an important structural property of the SDFS model that any simple cycle must contain at least one register. Note that this condition may be not sufficient as liveness property also depends on token game semantics. For example, applying a token game semantics to SDFS model may further limit this requirement to at least three registers per simple cycle (similar to direct mapping from Petri nets [14]).

Projection
\textit{Projection} of a path \( \delta \) onto a set of vertices \( X \) is defined as \( \delta \downarrow X = \text{Set}(\delta) \cap X \), where \( \text{Set}(\delta) \) is the set of vertices in sequence \( \delta \).

R-preset and register R-postset
The \textit{R-preset} of a vertex \( x \in V \) is defined as \( \star x = \{ r \in R \mid \exists \delta (r, x) : \delta (r, x) \downarrow R = \{ r, x \} \cap R \} \), i.e. a register \( r \) is in R-preset of a node \( x \) iff there exists a path \( \delta (r, x) \) with no other registers except \( r \) and \( x \) (if \( x \) is a register). Similarly, the \textit{R-postset} is defined as \( x \star = \{ r \in R \mid \exists \delta (x, r) : \delta (x, r) \downarrow R = \{ x, r \} \cap R \} \), i.e. a register \( r \) is in R-postset of a node \( x \) iff there exists a path \( \delta (x, r) \) with no other registers except \( x \) (if \( x \) is a register) and \( r \).

Graphical representation
Graphically, the combinational logic nodes are represented as boxes (□), the registers as boxes with two vertical lines (□), and the edges are depicted by arrows (→). The tokens are usually drawn as filled cycles (●), however, this representation varies for different token game semantics (see Sections 3, 4 and 6).

For example, the SDFS fragment shown in Figure 1 consists of 11 nodes: 5 combinational logic nodes \((L1, L2, L3, L4 \text{ and } L5)\) and 6 registers \((R1, R2, R3, R4, R5 \text{ and } R6)\). Note that \( R1 \) and \( R2 \)
are sources while $R5$ and $R6$ are sinks. The preset of node $L3$ is $\{L2, R3\}$ and it postset is $\{L4\}$; the R-preset of node $L3$ is $\{R2, R3\}$ and its R-postset is $\{R4, R5\}$.

In this section we have formally defined the structure and syntax of SDFS model using [25] as a guideline. The following sections introduce different token game semantics for the SDFS model.

4. Atomic token semantics

The atomic token semantic of the SDFS model, or simply atomic token model, is a formalisation of the intuitive token game which is presented in [25] on a set of simple examples.

Marking semantics
The marking in the atomic token model is defined as a mapping $M : R \rightarrow \{0, 1\}$, i.e. a register can contain maximum one token. The marking in this model represents data validity. The presence of a token in a register means it stores a valid data. The absence of a token in a register represents invalid data or spacer.

A current marking in atomic token model can be viewed as a front of computation phase in a circuit, followed by a reset phase. Subsequent computation phases must not overlap, therefore for a marked register $r \in R$ all registers in its preset and postset must be unmarked, i.e.:

$$\forall r \in R, q \in \star r \cup r\star : M(r) \Rightarrow \overline{M(q)}.$$

Evaluation and reset of combinational logic nodes
The evaluation state of the atomic token model is a mapping $\Xi : L \rightarrow \{0, 1\}$ which defines if a combinational logic node $l \in L$ has computed its output ($\Xi(l) = 1$) or has not computed it yet ($\Xi(l) = 0$). A node $l \in L$ is said to be in reset state if $\Xi(l) = 0$; it is said to be in evaluated state if $\Xi(l) = 1$. The switching of a combinational logic node form reset to evaluated state is called evaluate transition; its change from evaluated to reset state is called reset transition. Note, that words “state” and “transition” are often omitted in the text if it is clear from the context what is referred: the state of a node or its transition from one state to another.

Initially all combinational logic nodes are in reset states. A reset combinational logic node may evaluate iff all the combinational logic nodes in its preset are in evaluated states and all the registers in its preset are marked. This is the evaluation condition. Similarly, an evaluated combinational logic node may reset iff all the combinational logic nodes in its preset are in reset states and all the registers in its preset are unmarked. This is the resetting condition. For a combinational logic node $l \in L$ the evaluation
condition $\xi_+ (l)$ and resetting condition $\xi_- (l)$ can be formally expressed as:

$$
\xi_+ (l) = \bigcap_{k \in l \cap L} \Xi(k) \land \bigcap_{q \in l \cap R} M(q)
$$

$$
\xi_- (l) = \bigcap_{k \in l \cap L} \Xi(k) \land \bigcap_{q \in l \cap R} \overline{M(q)}
$$

In other words, a combinational logic node $l \in L$ may evaluate when $\xi_+ (l) = 1$ and may reset when $\xi_- (l) = 1$. The evaluation and resetting conditions of atomic token SDFS are similar to the firing conditions of phased logic [16], where a gate is enabled when all its input phases are opposite to the gate output phase; when an enabled gate fires, its outputs toggle to the opposite phase.

### Enabling and disabling of registers

The *enabling state* of the atomic token model is a mapping $\Sigma : R \rightarrow \{0, 1\}$ which defines if a register $r \in R$ is disabled ($\Sigma (r) = 0$) or enabled ($\Sigma (r) = 1$).

Initially all unmarked registers are disabled and all marked registers are enabled. A disabled and unmarked register becomes enabled iff all the combinational logic nodes in its preset are evaluated and all the registers in its preset are marked. This is a register *enabling condition*. Similarly, an enabled and marked register becomes disabled iff all the combinational logic nodes in its preset are reset and all the registers in its preset are unmarked. This is a register *disabling condition*. The enabling condition $\sigma_+ (r)$ and disabling condition $\sigma_- (r)$ of a register $r \in R$ can be formally represented as follows:

$$
\sigma_+ (r) = \overline{M(r)} \land \bigcap_{k \in r \cap L} \Xi(k) \land \bigcap_{q \in r \cap R} M(q)
$$

$$
\sigma_- (r) = M(r) \land \bigcap_{k \in r \cap L} \Xi(k) \land \bigcap_{q \in r \cap R} \overline{M(q)}
$$

A register $r \in R$ becomes enabled when $\sigma_+ (r) = 1$ and it becomes disabled when $\sigma_- (r) = 1$.

### Propagation of tokens

In order to be marked with a token a register must be enabled first; and to be unmarked a register needs to get disabled. Therefore a register cycles through the following four phases: enabling, marking, disabling and unmarking, as shown by a register state graph in Figure 2. Each state of the graph is encoded by a vector $(M(r), \Sigma(r))$. The excited variables (the ones which may change in the current state) of this vector are denoted by "*" symbol on top right. In the initial state $00^*$, which outlined by a box, a register...
is disabled and unmarked. This register may get enabled, which is denoted by the ‘*’ symbol next to the \( \Sigma (r) \) component of the vector. After being enabled it may be marked with a token, then get disabled and finally be unmarked, thus coming to the initial state.

To prevent overlapping of tokens from subsequent phases of computation, when propagating a token, the following two conditions have to be satisfied: i) a token can leave a disabled register iff all the registers in its R-postset are unmarked; ii) a token can move into an enabled register iff all the registers in its R-preset are marked. Following these conditions, in a current marking, a set of registers \( R_- \) from which tokens can be potentially removed and a set of registers \( R_+ \) which can potentially receive tokens are defined as:

\[
R_- = \{ r \in R \mid M (r) \land \overline{\Sigma (r)} \}, \quad R_+ = \{ q \in R \mid \overline{M (q)} \land \Sigma (q) \}
\]

A token propagation takes place when i) each register in \( R_- \) also belongs to R-preset of some register in \( R_+ \), i.e.: \( \forall r \in R_- \land q \in r^* \Rightarrow q \in R_+ \); and ii) each register in \( R_+ \) belongs to R-postset of some register in \( R_- \), i.e. \( \forall r \in R_+ \land q \in \overline{*r} \Rightarrow q \in R_- \). When these two conditions hold, the registers in \( R_- \) may fire in a single action, removing tokens from all registers of \( R_- \) and producing tokens in each register of \( R_+ \).

The atomic nature of token propagation in this model is similar to firing in PNs, where places correspond to registers and transitions correspond to (possibly empty) combinational logic ‘clouds’ between the registers.

This token game semantics works for simple examples but can be problematic for more complex SDFS. For instance, consider the SDFS in Figure 3. At Step 1 only register \( R1 \) is enabled and has a token. It enables register \( R2 \) at Step 2 and the token propagates from \( R1 \) to \( R2 \) at Step 3. Now the token in register \( R2 \) allows combinational logic node \( L1 \) to evaluate and enable the register \( R3 \).

Note that \( R2 \) still cannot fire and produce a token into \( R3 \), because there is register \( R5 \) in its R-postset which is still disabled. This results in a concurrency reduction, where the whole branch \{\( R3, L2, R4 \)\} waits for evaluation of the concurrent branch \{\( L3 \)\}.
Another problem arises at Step 4, when the evaluation of the combinational logic node $L_3$ leads to a deadlock. Indeed, after Step 4 the combinational logic node $L_4$ cannot evaluate until a token propagates to the register $R_3$ and then to $R_4$. At the same time the register $R_3$ can only receive a token when the combinational logic node $L_4$ evaluates and enables register $R_5$.

These concurrency reduction and deadlock problems can be avoided in two different ways. The easiest would be to introduce a set of constraints for well-formed SDFS. For example, a necessary constraint would be: if one of the concurrent branches contain a register, then all the other branches concurrent to it also must have a register. However, this approach would significantly restrict the class of circuits the model can capture. A more practical approach is to define the token game rules which would naturally capture the pipeline-style behaviour of the asynchronous data path. For example, the firing can be split in two atomic actions: i) propagation of the tokens into the next-stage registers (can be associated with a request signal in a pipeline), and ii) removing the tokens from the previous stage registers (models an acknowledgement signal). Thus, a token can stretch over a chain of registers before being removed from the beginning of the chain. This token game semantics is called spread token and is formally defined in Section 5.

5. Spread token semantics

The spread token semantics of the SDFS model, or simply spread token model, is an extension of the atomic token semantics. It models asynchronous circuits of Muller pipeline architecture. The spread token semantics does not capture preemption or token borrowing though. However, it can be extended to model token borrowing as in low-latency structures with slack [7].

Marking, evaluation and enabling

The marking semantics, evaluation and reset of combinational logic nodes, and also enabling and disabling of registers in this model are exactly the same as in atomic token. The only difference is in the way tokens propagate from a register to a register. Also a concept of early evaluation will be introduced in this model, which has not been discussed in [25]. Therefore, in this section we concentrate on modelling the early evaluation and the new rules of token propagation. The rest of the terminology is adopted from Section 4.

Early evaluation

It is often sufficient to have only a subset of the inputs ready to evaluate a combinational logic node. This is called early evaluation and can be modelled by modifying the evaluation condition $\xi_0$ of the node. For example, a combinational logic node $l \in L$ which evaluates as soon as any of its inputs is ready, has the following evaluation condition:

$$\xi_+(l) = \bigcup_{k \in \bullet \cap L} \Xi(k) \lor \bigcup_{q \in \bullet \cap R} M(q).$$

Modification of the evaluation and resetting conditions is not limited to early evaluation. In fact, any reasonable expressions can be assigned to conditions $\xi_+(l)$ and $\xi_-(l)$ of a combinational logic node $l \in L$. For example, it is reasonable to assume that $\xi_+(l) \land \xi_-(l) \equiv 0$, i.e. evaluation and resetting conditions of a node are mutually exclusive to prevent a node from enabling and resetting at the same time.
time. Also it is reasonable to assume that evaluation and resetting conditions depend on the marking and evaluation state of SDFS, i.e. they are not constant 1 or constant 0.

The concept of early evaluation can also be applied to enabling and disabling of a register. However, the same result can be achieved by splitting such a register into an early evaluation combinational logic node and a register itself. Therefore, the notion of early evaluation is restricted to combinational logic nodes.

**Propagation of tokens**

A token can be put into an enabled register iff all the registers in its R-presets are marked and all the registers in its R-postsets are unmarked. This is a *marking condition*. Similarly, a token can be removed from a disabled register iff all the registers in its R-presets are unmarked and all the registers in its R-postsets are marked. This is *unmarking condition*. Formally this can be represented by assigning each register \( r \in R \) a marking condition \( m^+ (r) \) and unmarking condition \( m^- (r) \):

\[
m^+ (r) = \sum (r) \land \bigcap_{q \in \delta r} M (q) \land \bigcap_{s \in \delta^* r} \overline{M (s)}
\]

\[
m^- (r) = \sum (r) \land \bigcap_{q \in \delta r} \overline{M (q)} \land \bigcap_{s \in \delta^* r} M (s)
\]

A register \( r \in R \) can be marked with a token when \( m^+ (r) = 1 \) and can be unmarked when \( m^- (r) = 1 \).

Note that an unmarked source is always enabled because its R-presets are empty. A new token can be put into an enabled source as soon as its R-postset is unmarked (its R-presets are empty). Similarly, a token can be removed from a disabled sink as soon as its R-presets are unmarked (its R-postsets are empty). These features of the source and sink registers are useful to model the communication with the environment which produces new tokens and consumes old ones.

Consider the spread token model on a simple example of Figure 4. Enabled registers and evaluated combinational logic nodes are highlighted. Note that combinational logic node \( L4 \) is labelled with *EE* tag. This tag means the node exhibits early evaluation, i.e. for \( L4 \) to evaluate it is sufficient to have \( R4 \) marked or \( L3 \) evaluated: \( \xi^+ (L4) = M (R4) \lor \Xi (L3) \). Therefore, on Step 2, when token propagates to \( R2 \), combinational logic nodes \( L3 \) and \( L4 \) evaluate and register \( R5 \) becomes enabled. However, \( R5 \) cannot be marked with a token until all registers in its R-presets are marked. For this to happen two more steps are needed: at Step 3 the register \( R3 \) is marked and at Step 4 token propagates to \( R4 \). At Step 5 register \( R5 \) is finally marked. Similarly, the token cannot be removed from \( R2 \) until Step 6 when all registers in its R-postset are marked. Because of these restrictions the token spreads over four registers (at least four, as a token could still stay in \( R1 \) and \( R2 \)) at Step 5. Finally, the tokens are removed one by one from the tail of the spread token, as shown at Steps 6-8.

The spread token model solves the concurrency reduction and deadlock problems of the atomic token semantics. It also has some rudimentary means to model early evaluation. However, in this model a register can only accept a token when all the registers in its R-presets are marked. This limits the early evaluation to one pipeline stage only and makes the model useless for capturing preemption and speculation.
It would be natural to allow further propagation of a token into an enabled register without waiting for all the tokens in its R-preset. There is a risk of mixing tokens from different computation cycles though. In order to avoid this mixture, when a token propagates into an enabled register, all unmarked registers in its R-preset should be marked with a negative token. The next data token to arrive into a register with negative marking must be ignored as it carries old data. Therefore the data token and the negative token cancel each other. The described technique is called token borrowing. Different types of token borrowing and one of SDFS models implementing this technique are discussed in Section 6.

6. Counterflow semantics

The token borrowing techniques can be partitioned into two classes: passive borrowing and active borrowing. In the passive borrowing a special join block is responsible for counting the number of tokens borrowed from each of its inputs. The borrowing does not propagate further in the direction opposite to the token flow. The passive borrowing is introduced as a feature of the change diagrams model and is also modelled by unsafe (places can be marked with more than one token) Petri nets [29]. A model and
an implementation of a join element capable of unbounded borrowing are presented in [7]. The main
disadvantage of the passive borrowing is the lack of preemption mechanism in the unwanted branches,
which may result in a higher power consumption and longer computation time.

The active borrowing is characterised by negative tokens which are able to propagate in the direction
reverse to the data token flow. When collide, a data token and a negative token eliminate each other. The
major drawback of this technique is caused by the resolution of the conflicts when a data token and a
negative token want to occupy the same register simultaneously. Usually, such conflicts result in arbitra-
tion which cause significant implementation overheads (increase in circuit size, power consumption and
latency). On the positive side, preemption is captured naturally by active borrowing.

Both, passive and active borrowing can be defined as token game semantics for the SDFS model.
However, in this paper we concentrate on active borrowing only, as it is somewhat superior to passive
borrowing and is advantageous for implementing the preemption mechanism. There are two SDFS model
of active borrowing mechanism, namely antitoken model and counterflow model.

The antitoken semantics of the SDFS model, or simply antitoken model, is based on the idea of the
two pipelines of opposite directions, one for data tokens and the other for negative tokens. Data tokens
and negative tokens eliminate each other when collide. Similar idea is employed in counterflow pipeline
processor (CFPP) [26] which allows instructions to move one way along a processing pipeline while
results flow freely in the opposite direction; when collide instructions are executed on the correspon-
ding data.

The main disadvantage of the antitoken model is that in order not to miss each other, data tokens
and negative tokens must synchronise within each pipeline stage. This requires arbitration which is
associated with metastability problems at the level of circuit implementation. The arbitration problem
is avoided in counterflow semantics of SDFS model which is the main focus of this section. For more
details on antitoken SDFS semantics the reader is referred to [24].

The counterflow semantics of SDFS model, or simply counterflow model, is based on the idea of OR-
causality [29], which allows to avoid arbitration inherent in antitoken model. Data tokens and negative
tokens are not distinguished in this model at the level of individual stages: the first to arrive propagates
in both directions (as a data token forward and as a negative token backward), the second one is ignored.
The idea of antitokens without arbitration is introduced in [6, 5] and is revisited with minor modifications
in [1].

Marking semantics
In the counterflow SDFS model there are two types of tokens: OR-tokens and AND-tokens. The marking
in the counterflow model is defined as $M = M^O \times M^A$, where mapping $M^O : R \to \{0, 1\}$ is the
OR-marking and $M^A : R \to \{0, 1\}$ is the AND-marking. The presence of an OR-token in a register
means either that data has been received from its R-preset or that data is not needed anymore by its R-
postset (e.g. due to early evaluation form another branch). An AND-token in a register means that data
has been received from its R-preset and has been consumed (or ignored, in case of early evaluation) by
its R-postset. Graphically, an OR-token is depicted as a filled triangle (▲) while an AND-token as a filled
box (■).

Evaluation and reset of combinational logic nodes
Forward evaluation state of SDFS is a mapping $\Xi^F : L \to \{0, 1\}$ which defines if a combinational logic
node $l \in L$ has computed its output ($\Xi^F(l) = 1$) or has not computed it yet ($\Xi^F(l) = 0$). A node
A combinational logic node \( l \in L \) is said to be forward evaluated if \( \Xi^F(l) = 1 \) and forward reset if \( \Xi^F(l) = 0 \). Initially all combinational logic nodes are forward reset. A forward reset combinational logic node may forward evaluate iff all the combinational logic nodes in its preset are forward evaluated and all the registers in its preset have OR-tokens. Similarly, a forward evaluated combinational logic node may forward reset iff all the combinational logic nodes in its preset are forward reset and all the registers in its preset do not have OR-tokens. These are forward evaluation condition and forward resetting condition respectively.

Backward evaluation state of SDFS is a mapping \( \Xi^B : L \to \{0, 1\} \) which defines if the output of a combinational logic node \( l \in L \) has been consumed and is not needed any longer (\( \Xi^B(l) = 1 \)) or the output has not been received yet and is still awaited (\( \Xi^B(l) = 0 \)). A combinational logic node may backward evaluate iff all the combinational logic nodes in its postset are backward evaluated and all the registers in its postset have OR-tokens. Similarly, a backward evaluated combinational logic node may backward reset iff all the combinational logic nodes in its postset are backward reset and all registers in its postset do not have OR-tokens. These are backward evaluation condition and backward resetting condition.

Formally, the forward evaluation condition \( \xi^F(l) \) and the forward resetting condition \( \xi^F_E(l) \) of a combinational logic node \( l \in L \) can be expressed as:

\[
\xi^F(l) = \bigcap_{k \in \bullet \cap L} \Xi^F(k) \land \bigcap_{q \in \bullet \cap R} M^{OR}(q)
\]

\[
\xi^F_E(l) = \bigcap_{k \in \bullet \cap L} \Xi^F(k) \land \bigcap_{q \in \bullet \cap R} \overline{M^{OR}(q)}
\]

Similarly, the backward evaluation condition \( \xi^B(l) \) and the backward resetting condition \( \xi^B_E(l) \) are:

\[
\xi^B(l) = \bigcap_{k \in \bullet \cap L} \Xi^B(k) \land \bigcap_{q \in \bullet \cap R} M^{OR}(q)
\]

\[
\xi^B_E(l) = \bigcap_{k \in \bullet \cap L} \Xi^B(k) \land \bigcap_{q \in \bullet \cap R} \overline{M^{OR}(q)}
\]

A combinational logic node \( l \in L \) may forward evaluate when \( \xi^F(l) = 1 \) and it may forward reset when \( \xi^F_E(l) = 1 \). Similarly, a combinational logic node \( l \in L \) may backward evaluate when \( \xi^B(l) = 1 \) and it may backward reset when \( \xi^B_E(l) = 1 \).

The above conditions do not allow early forward (backward) evaluation because the change on all the node inputs (outputs) is required to change its forward (backward) state. By analogy with spread token model, the effect of early evaluation in counterflow semantics can be modelled by modifying the evaluation and resetting conditions of a combinational logic node, so, that a subset of node inputs (outputs) is sufficient to trigger its forward (backward) state.

Enabling and disabling of registers
Forward enabling state of SDFS is a mapping \( \Sigma^F : R \to \{0, 1\} \) which defines if a register \( r \in R \) is forward enabled (\( \Sigma^F(r) = 1 \)) or forward disabled (\( \Sigma^F(r) = 0 \)). Similarly, backward enabling state of SDFS is a mapping \( \Sigma^B : R \to \{0, 1\} \) which defines if a register \( r \in R \) is backward enabled (\( \Sigma^B(r) = 1 \)) or backward disabled (\( \Sigma^B(r) = 0 \)).
Initially all registers without AND-tokens are both forward disabled and backward disabled. All
registers which are marked with AND-tokens are both forward enabled and backward enabled.

A register without an AND-token becomes forward enabled iff all the combinational logic nodes
in its preset are forward evaluated and all the registers in its preset have OR-tokens. A register with
an AND-token becomes forward disabled iff all the combinational logic nodes in its preset are forward
reset and all the registers in its preset do not have OR-tokens. These are forward enabling condition
and forward disabling condition. Note that a source without AND-token becomes forward enabled and a
source with AND-token becomes forward disabled (because its preset is empty).

A register without an AND-token becomes backward enabled iff all the combinational logic nodes in
its postset are backward evaluated and all the registers in its postset have OR-tokens. A register with an
AND-token becomes backward disabled iff all the combinational logic nodes in its postset are backward
reset and all the registers in its postset do not have OR-tokens. These are backward enabling condition
and backward disabling condition respectively. Note that a sink without AND-token becomes backward
enabled and a sink with AND-token becomes backward disabled (because its postset is empty).

Formally, the forward enabling condition \( \sigma^F_+ (r) \) and the forward disabling condition \( \sigma^F_- (r) \) of a
register \( r \in R \) is defined as:

\[
\sigma^F_+ (r) = \overline{MA (r)} \land \bigcap_{k \in r \land L} \Xi^F (k) \land \bigcap_{q \in \bullet \land R} MO (q)
\]

\[
\sigma^F_- (r) = MA (r) \land \bigcap_{k \in r \land L} \Xi^F (k) \land \bigcap_{q \in \bullet \land R} MO (q)
\]

The backward enabling condition \( \sigma^B_+ (r) \) and the backward disabling condition \( \sigma^B_- (r) \) formally are:

\[
\sigma^B_+ (r) = \overline{MA (r)} \land \bigcap_{k \in r \land L} \Xi^B (k) \land \bigcap_{q \in \bullet \land R} MO (q)
\]

\[
\sigma^B_- (r) = MA (r) \land \bigcap_{k \in r \land L} \Xi^B (k) \land \bigcap_{q \in \bullet \land R} MO (q)
\]

A register \( r \in R \) becomes forward enabled when \( \sigma^F_+ (r) = 1 \) and it becomes forward disabled when
\( \sigma^F_- (r) = 1 \). Similarly, a register \( r \in R \) becomes backward enabled when \( \sigma^B_+ (r) = 1 \) and it becomes
backward disabled when \( \sigma^B_- (r) = 1 \).

**Propagation of tokens**

A register can be marked with an OR-token iff it does not have an AND-token, it is either forward
enabled or backward enabled and neither its R-preset nor its R-postset is marked with AND-token. An
OR-token can be removed from a register iff it is marked with AND-token, it is either forward disabled
or backward disabled and its R-preset and R-postset are both marked with AND-tokens. These are OR-
marking condition \( m^O_+ (r) \) and OR-unmarking condition \( m^O_- (r) \) of a register \( r \in R \), which are formally
defined as:

\[
m^O_+ (r) = \overline{MA (r)} \land (\Sigma^F (r) \lor \Sigma^B (r)) \land \bigcap_{q \in \bullet \cup r \cup r^*} \overline{MA (q)}
\]

\[
m^O_- (r) = MA (r) \land (\Sigma^F (r) \lor \Sigma^B (r)) \land \bigcap_{q \in \bullet \cup r \cup r^*} MA (q)
\]
A register can be marked with an AND-token iff it has an OR-token and it is both forward enabled and backward enabled and its R-preset and R-postset are marked with OR-tokens. An AND-token can be removed from a register iff it does not have an OR-token and it is both forward disabled and backward disabled and its R-preset and R-postset are not marked with OR-tokens. These are OR-marking and OR-unmarking conditions. These AND-marking condition \( m_A^+ (r) \) and AND-unmarking condition \( m_A^- (r) \) are assigned to each register \( r \in R \) as follows:

\[
m_A^+ (r) = M^O (r) \land \Sigma^F (r) \land \Sigma^B (r) \land \bigcap_{q \in \star \cup \star \ast} M^O (q)
\]

\[
m_A^- (r) = M^O (r) \land \Sigma^F (r) \land \Sigma^B (r) \land \bigcap_{q \in \star \cup \star \ast} M^O (q)
\]

A register \( r \in R \) can be marked with an OR-token when \( m_O^+ (r) = 1 \) and the OR-token can be removed when \( m_O^- (r) = 1 \). Similarly, a register \( r \in R \) can be marked with an AND-token when \( m_A^+ (r) = 1 \) and the AND-token can be removed when \( m_A^- (r) = 1 \).

A counterflow register operation is represented by the state graph in Figure 5. Each state of the graph is encoded by a vector \( \langle M^A (r), M^O (r), \Sigma^B (r), \Sigma^F (r) \rangle \). In the initial state \( 000^*0^* \), which outlined by a box, a register is both forward and backward disabled and does not have tokens. This register may be forward and/or backward enabled, which is denoted by \( ^* \) symbol next to \( \Sigma^B (r) \) and \( \Sigma^F (r) \) variables. Changing any of the excited variables leads to the next state, where the variable \( M^O (r) \) becomes excited, i.e. the register may be marked by an OR-token, and so on.

Note the states where two variables are excited, e.g. the state \( 00^*0^*1 \). Changing one of the excited variables does not remove the excitation from the other one. Eventually both excited variables have to switch leading to the same state \( 0^*111 \). It is also possible for both excited variables to change simultaneously, which is depicted by dotted arcs.

There are two distinctive phases in the operation of a counterflow register: marking phase and cleaning phase. At the marking phase a register gets enabled (forward and/or backward), then marked with OR-token and finally marked with AND-token. At the cleaning phase it is, firstly, disabled (forward and/or backward), then OR-token leaves the register and finally AND-token is removed.

Figure 6 illustrates the counterflow SDFS semantics on a simple example. Forward (backward) enabled registers and forward (backward) evaluated combinational logic nodes are highlighted on top
Figure 6. Counterflow SDFS example

(bottom). The combinational logic node \( L_4 \) labelled with \( EE \) tag exhibits early forward evaluation: 
\[
\xi^F (L4) = M^O (R4) \lor \xi^F (L3), \quad \xi^E (L4) = M^O (R4) \land \xi^E (L3).
\]

At Step 1 only register \( R1 \) has an OR-token, which forward enables register \( R2 \) (this models a request signal in the circuit). The OR-token propagates to \( R2 \) at Step 2 and backward enables register \( R1 \) (this models an acknowledgement signal). Also the combinational logic nodes \( L_1, L_3 \) and \( L_4 \) evaluate at this step (note that \( L_4 \) exhibits early evaluation). This allows forward enabling of registers \( R3 \) and \( R5 \). At Step 3 an AND-token is produced in register \( R1 \) because it is both forward enabled and backward enabled; as AND-token appears in \( R1 \) and it is a source, it becomes forward disabled. Also the OR-tokens propagate to forward enabled registers \( R3 \) and \( R5 \). Now register \( R4 \) becomes both forward
enabled and backward enabled. As it does not have a token yet, first, an OR-token is generated in $R4$ at Step 4. After that, at Step 5, an AND-tokens appear registers $R3$ and $R4$ as they are both forward enabled and backward enabled. At Step 6 OR-token disappears from the forward disabled register $R2$, which leads to forward disabling of $R3$. OR-token leaves the forward disabled register $R3$ and register $R4$ becomes forward disabled at Step 7, therefore OR-token is removed from $R4$ which forward disables $R5$ at Step 8. Now, at Step 9, OR-token leaves $R5$ and registers $R2$ and $R4$ become both forward disabled and backward disabled, therefore AND-tokens can be removed from them, as show at Step 10.

Note that at Step 4 it does not matter which register, $R3$ or $R5$, initiates the OR-token in $R4$ - the resultant marking is the same. Thus, the merge of the data tokens (moving in forward direction) and the negative tokens (moving in backward direction) is modelled by OR-causality instead of arbitration. This is the main advantage of the counterflow model over antitoken model.

The major drawback of the counterflow model is the complex behaviour of its registers. It is difficult to design a fully indicating and hazard-free controller for counterflow registers. Interesting implementations of such controller were proposed in [5, 1]. Due to the complexity of the counterflow protocol these implementations are several times larger than a Muller pipeline stage. This is particularly disadvantageous when no token borrowing is actually possible. For example, consider a long linear pipeline with a small section $S$ having parallel branches, e.g. for speculative computation. The token borrowing is only possible within section $S$, but in order to satisfy the counterflow protocol the whole pipeline has to be implemented using large counterflow controllers.

A combination of counterflow pipeline (for the sections which require preemption) and Muller pipeline (for the rest of the circuit) is a promising way to build asynchronous data path. Such data path has all advantages of counterflow pipelines (no arbitration, preemption, early evaluation, speculation) for the price of moderate area increase compared to Muller pipeline. The hybrid data path can be modelled by combining spread token and counterflow semantics of SDFS model as is described in Section 7.

### 7. Hybrid semantics

The idea of combining a counterflow pipeline with a Muller pipeline originates from [24], where PN models and gate-level implementations for converters between different pipeline types were proposed. The subject of this section is to capture the behaviour of such hybrid pipeline in special SDFS model, which is a combination of spread token and counterflow models. The main idea for this model is that only those parts of data path which may exhibit preemption should be modelled by the counterflow semantics while the rest of the data path should have the spread token semantics. Such a syndication of the token game semantics is called hybrid SDFS model. One of the ways to achieve this hybrid functionality is to introduce a pair of converters between the spread token SDFS nodes and the counterflow SDFS nodes. For this the set of SDFS registers $R$ needs to be extended with a special kind of registers $C \subseteq R$, which have spread token type of interface on one side and counterflow interface on the other side.

A spread token to counterflow (ST2CF) converter behaves as a spread token register to its preset and as a counterflow register to its postset. Only nodes with spread token semantics are allowed in the preset of an ST2CF converter and only nodes with counterflow semantics are allowed in its postset. The set of ST2CF converters is denoted as $C^{ST2CF} \subseteq C$.

A counterflow to spread token (CF2ST) converter appears as a counterflow register to its preset and as a spread token register to its postset. The preset of a CF2ST converter can only contain nodes with
counterflow semantics, while its postset only allows nodes with spread token semantics. The set of
ST2CF converters is denoted as $C^{ST2CF} \subseteq C$.

The behaviour of ST2CF and CF2ST converters is somewhat symmetrical. They are used in pairs
forming structures of fork-join type. An ST2CF converter is used as fork interface from a part of the data
path without early propagation to the part with several concurrent branches where preemption mechanism
is employed. These concurrent branches are subsequently joined into a CF2SF converter which limits
the early propagation and preemption to the fork-join part of the data path.

Marking semantics
The ST2CF and CF2ST converters should be able to accept three types of tokens: ordinary tokens (used
in spread token model), OR-tokens and AND-tokens (used in counterflow model). Therefore the marking
of the converters is defined as $M_C = M_C^S \times M_O \times M_A$, where mapping $M : C \rightarrow \{0, 1\}$ is spread token
marking, $M_O : C \rightarrow \{0, 1\}$ is OR-marking and $M_A : C \rightarrow \{0, 1\}$ is AND-marking. The semantics of
these markings are the same as in spread token and counterflow models, respectively.

Enabling state
The hybrid enabling state for SDFS converters comprises of three components. The first component
is enabling state $\Sigma : C \rightarrow \{0, 1\}$ for the spread token part of all converters. The other two are
$\Sigma^F : C \rightarrow \{0, 1\}$ and $\Sigma^B : C \rightarrow \{0, 1\}$ which are forward enabling and backward enabling states of the
counterflow parts. The semantics of these enabling states are the same as for the registers of spread token
model and counterflow model.

Operation of ST2CF converter
The enabling and disabling conditions for the spread token part of an ST2CF converter $c \in C^{ST2CF}$ are
the same as for a spread token register:

$$\sigma_+ (c) = \overline{M(c)} \land \bigcap_{k \in c \cap L} \Xi(k) \land \bigcap_{q \in c \cap R} M(q)$$

$$\sigma_- (c) = M(c) \land \bigcap_{k \in c \cap L} \Xi(k) \land \bigcap_{q \in c \cap R} M(q)$$

The spread token part of an ST2CF converter $c \in C^{ST2CF}$ becomes enabled when $\sigma_+ (c) = 1$ and it
becomes disabled when $\sigma_- (c) = 1$.

The forward enabling and forward disabling conditions for the counterflow part of an ST2CF converter
$c \in C^{ST2CF}$ are similar to those of a counterflow register. The major simplification is because an
ST2CF converter does not have any counterflow nodes in its preset and the marking of its spread token
part is taken into account instead:

$$\sigma^F_+ (c) = \overline{M^A(c)} \land M(c) : \sigma^F_- (c) = M^A(c) \land \overline{M(c)}$$

The backward enabling and backward disabling conditions are the same as for a counterflow register:

$$\sigma^B_+ (c) = \overline{M^A(c)} \land \bigcap_{k \in c \cap L} \Xi^B(k) \land \bigcap_{s \in c \cap R} M^O(s)$$
\[ \sigma_B^-(c) = MA(c) \land \bigcap_{k \in c \cap L} B(k) \land \bigcap_{s \in c \cap R} MO(s) \]

The counterflow part of an ST2CF converter \( c \in C^{ST2CF} \) becomes forward enabled when \( \sigma_F^+(c) = 1 \) and it becomes forward disabled when \( \sigma_F^-(c) = 1 \). Similarly, it becomes backward enabled when \( \sigma_B^+(c) = 1 \) and it becomes backward disabled when \( \sigma_B^-(c) = 1 \).

Once the spread token part of an ST2CF converter is enabled, it may accept a spread token, providing all the spread token registers in its R-preset are marked and its counterflow part does not have an OR-token. When the spread token part becomes disabled it may lose the token. Formally, these marking conditions are:

\[ m_+ (c) = MO(c) \land \Sigma(c) \land \bigcap_{q \in c^*} M(q) \]

\[ m_- (c) = MO(c) \land \Sigma(c) \land \bigcap_{q \in c^*} M(q) \]

The spread token part of a converter \( c \in C^{ST2CF} \) can be marked with a token when \( m_+ (c) = 1 \) and can be unmarked when \( m_- (c) = 1 \).

The marking and unmarking conditions for the counterflow part of an ST2CF converter are identical to those of a counterflow register. The only simplification is that there is no counterflow nodes in the preset of a ST2CF converter:

\[ m_+^O (c) = MA(c) \land \Sigma^F(c) \lor \Sigma^B(c) \land \bigcap_{q \in c^*} MO(q) \]

\[ m_-^O (c) = MA(c) \land \Sigma^F(c) \lor \Sigma^B(c) \land \bigcap_{q \in c^*} MO(q) \]

\[ m_+^A (c) = MO(c) \land \Sigma^F(c) \land \Sigma^B(c) \land \bigcap_{q \in c^*} MO(q) \]

\[ m_-^A (c) = MO(c) \land \Sigma^F(c) \land \Sigma^B(c) \land \bigcap_{q \in c^*} MO(q) \]

The counterflow part of a converter \( c \in C^{ST2CF} \) can be marked with an OR-token when \( m_+^O (c) = 1 \) and the OR-token can be removed when \( m_-^O (c) = 1 \). Similarly, it can be marked with an AND-token when \( m_+^A (c) = 1 \) and the AND-token can be removed when \( m_-^A (c) = 1 \).

**Operation of CF2ST converter**

The forward enabling and forward disabling conditions for the counterflow part of a CF2ST converter \( c \in C^{CF2ST} \) are identical to those of a counterflow register:

\[ \sigma_F^+(c) = MA(c) \land \bigcap_{k \in c \cap L} E^F (k) \land \bigcap_{q \in c \cap R} MO(q) \]

\[ \sigma_F^-(c) = MA(c) \land \bigcap_{k \in c \cap L} E^F (k) \land \bigcap_{q \in c \cap R} MO(q) \]
For the backward enabling and backward disabling conditions there is a significant simplification compared to the counterflow registers. This is due to the fact that there is no counterflow nodes in the postset of a CF2ST converter and the marking of its spread token part is taken into account instead:

\[ \sigma^B_+ (c) = M^A (c) \land M (c) ; \quad \sigma^B_- (c) = M^A (c) \land \overline{M (c)} \]

The counterflow part of a CF2ST converter becomes forward enabled when \( \sigma^F_+ (c) = 1 \) and it becomes forward disabled when \( \sigma^F_- (c) = 1 \). Similarly, it becomes backward enabled when \( \sigma^B_+ (c) = 1 \) and it becomes backward disabled when \( \sigma^B_- (c) = 1 \).

The spread token part of a CF2ST converter \( c \in C^{CF2ST} \) becomes enabled when there is an OR-token in its counterflow part; it becomes disabled when OR-token leaves the converter. These enabling and disabling conditions can be formalised as:

\[ \sigma^O_+ (c) = M^O (c) \land M^O (c) \land \bigcup_{q \in \ast c} M^O (q) \]

Marking and unmarking conditions of the counterflow part of a CF2ST converter are similar to those of a counterflow register. Formally, for a CF2ST converter \( c \in C^{CF2ST} \) the OR-making/unmarking and AND-marking/unmarking conditions are:

\[ m^O_+ (c) = \overline{M^A (c)} \land (\overline{M^F (c)} \lor \overline{M^B (c)}) \land \bigcap_{q \in \ast c} M^A (q) \]

\[ m^O_- (c) = M^A (c) \land (\overline{M^F (c)} \lor \overline{M^B (c)}) \land \bigcap_{q \in \ast c} M^A (q) \]

\[ m^A_+ (c) = M^O (c) \land M^F (c) \land M^B (c) \land \bigcap_{q \in \ast c} M^O (q) \]

\[ m^A_- (c) = \overline{M^O (c)} \land M^F (c) \land M^B (c) \land \bigcap_{q \in \ast c} M^O (q) \]

The counterflow part of a converter \( c \in C^{CF2ST} \) can be marked with an OR-token when \( m^O_+ (c) = 1 \) and the OR-token can be removed when \( m^O_- (c) = 1 \). Similarly, it can be marked with an AND-token when \( m^A_+ (c) = 1 \) and the AND-token can be removed when \( m^A_- (c) = 1 \).

Finally, the marking and unmarking of the spread token part of a CF2ST converter \( c \in C^{CF2ST} \) are determined by the following conditions:

\[ m_+ (c) = \overline{M (c)} \land \bigcap_{s \in \ast c} M (s) ; \quad m_- (c) = M (c) \land \bigcap_{s \in \ast c} M (s) \]

These conditions are derived from the marking and unmarking conditions for the spread token register, assuming there is no spread token register in the R-preset of a CF2ST controller. The spread token part of a converter \( c \in C^{CF2ST} \) can be marked with a token when \( m_+ (c) = 1 \) and can be unmarked when \( m_- (c) = 1 \).

Consider the operation of the hybrid SDFS model on a simple example shown in Figure 7. At Step 1 only ST2CF converter \( R2 \) is enabled and a token propagates into it as Step 2. This forward enables the counterflow part of the controller and it gets an OR-token at Step 3; the counterflow register \( R3 \)
and the CF2ST converter R5 are forward enabled now. Also the tail of spread token is removed from disabled register R1 at this step. At Step 4 both forward enabled register R3 and forward enabled CF2ST converter R5 get marked with OR-tokens and backward enable the counterflow part of ST2CF converter R2. The OR-token in register R3 also forward enables register R4 and the OR-token in CF2ST converter R5 enables its spread token part. At Step 5 the counterflow part of ST2CF converter R2 is marked with AND-token because it has an OR-token and is both forward enabled and backward enabled. Also a token propagates to the spread token part of the CF2ST converter R5. At Step 6 a token is removed
from the disabled spread token part of the ST2CF converter $R_2$; also a token propagates from the CF2ST converter $R_5$ to the register $R_6$. The forward disabled counterflow part of the ST2CF converter $R_2$ is freed of OR-token at Step 7, which forward disables the register $R_3$. At Steps 8 and 9 OR-tokens first leave the register $R_3$ and then the register $R_4$, which forward disables the counterflow part of the CF2ST converter $R_5$. Now OR-token disappears from the forward disabled CF2ST converter $R_5$, thus disabling its spread token part. Also the register $R_4$ and the ST2CF converter $R_2$ become backward disabled, see Step 10. Finally, the rest of the registers return to the initial state at Steps 11 and 12.

8. Verification of SDFS models

Direct verification of the SDFS models is a difficult task as there is no formal methods and no software tools to do this. It is tempting to reuse the variety of verification methods and model checking tools developed for PNs. In order to do this a conversion technique is required, which maps SDFS models into equivalent PNs.

An SDFS model with its token game semantics is a high level paradigm. At the low level this model can be viewed as a PN, or more precisely an STG, in which each state variable of the SDFS model is represented by an elementary cycle.

An elementary cycle models a state of a binary variable $x \in \{0, 1\}$ by two places $x = 0$ and $x = 1$, which represent the value associated to variable $x$. There is at least one transition $x+$ and one transition $x-$ between places $x = 0$ and $x = 1$, such that $x+ \in (x = 0) \bullet$, $x+ \in \bullet (x = 1)$, $x- \in (x = 1) \bullet$, $x- \in \bullet (x = 0)$. Transition $x+$ determines the change of variable state from 0 to 1, while $x-$ represents the change of the state from 1 to 0. Transitions $x+$ and $x-$ may also connected to read-arcs which enable the transitions only when a certain condition is held.

Consider the mapping of spread token model into elementary cycles of PN. In this model a combinational logic node $l \in L$ is associated with a single evaluation state variable $\Xi (l)$ and a pair of evaluation condition $\xi_+ (l)$ and resetting condition $\xi_- (l)$ (see Section 5 for details). At the PN level this is modelled as an elementary cycle $\Xi (l)$ shown in Figure 8(a). The read-arc connected to $\Xi (l) +$ allows this transitions to fire only when enabling condition $\xi_+ (l) = 1$ is held. Similarly, transition $\Xi (l) -$ becomes enabled only if its enabling condition $\xi_- (l) = 1$ is held. Note that for readability of the figure the variable name $\Xi (l)$ is only shown in the middle of the elementary cycle; places and transitions associated with this variable are labelled in a shorthand notation. In particular, places $\Xi (l) = 0$ and $\Xi (l) = 1$ are labelled '0' and '1' while transitions $\Xi (l) +$ and $\Xi (l) -$ are labelled '1' and '0' respectively.
Mapping of a spread token register into a PN is illustrated in Figure 8(b). There are two state variables associated with a register \( r \in R \): enabling state \( \Sigma (r) \) and marking \( M(r) \). Therefore two elementary cycles are required to capture the register behaviour by a PN. Conditions \( \sigma_+ (r) = 1 \) and \( \sigma_- (r) = 1 \) control transitions \( \Sigma (r) + \) and \( \Sigma (r) - \) respectively. The former denotes when the register is enabled and the later when it is disabled. Likewise, the change of register marking is defined by conditions \( m_+ (r) = 1 \) and \( m_- (r) = 1 \), which enable transitions \( M(r) + \) and \( M(r) - \) respectively.

Usually the enabling conditions on the read-arcs are more complex than a single variable. Such conditions should be represented into a disjunctive normal form (DNF). Then each DNF clause is mapped into a separate transition of the elementary cycle and each variable of the clause is read by its own read-arc.

In order to illustrate how the enabling conditions are represented by means of read-arcs consider a simple spread token example shown in Figure 9(a). Note that the combinational logic node \( l2 \) is tagged with \( EE \) label, which means it can evaluate as soon as one of its input is ready. Let us concentrate on mapping of this node into an elementary cycle \( \Xi (l2) \). The evaluation condition associated with this node is \( \xi_+ (l2) = \Xi (l1) \lor M(r2) \) while the resetting condition is \( \xi_- (l2) = \Xi (l1) \land M(r2) \).

For the evaluation phase \( \xi_+ (l2) = 1 \) implies \( (\Xi (l1) = 1) \lor (M(r2) = 1) \). This expression has two DNF clauses, therefore transition \( \Xi (l2) + \), which is controlled by the condition \( \xi_- (l2) \), is split into a pair of transitions \( \Xi (l2) + 1 \) and \( \Xi (l2) + 2 \). Transition \( \Xi (l2) + 1 \) is enabled when place \( \Xi (l1) = 1 \) is marked and transition \( \Xi (l2) + 2 \) is enabled by a token in place \( M(r2) = 1 \), as shown in Figure 9(b). Firing either of these transitions changes the evaluation state of node \( l2 \in L \), which models the early evaluation.

At the reset phase, \( \xi_- (l1) = 1 \) implies \( (\Xi (l1) = 0) \land (M(r1) = 0) \). This expression has a single DNF clause and therefore both read-arcs, one from place \( \Xi (l1) = 0 \) and the other from place \( M(r1) = 0 \), are connected to the same transition \( \Xi (l2) - \). This means that both places must be marked to allow the reset of node \( l2 \in L \), i.e. no early reset is possible.

Elementary cycles for the rest of the nodes are built the same way. Note that the resultant STG is consistent by construction because the positive and negative transitions of each signal (or variable) alternate in each the elementary cycle.

Consider the conversion of SDFS models into PNs on a more realistic benchmark, e.g. ARISC processor whose SDFS model is shown in Figure 10(a). This is a relatively small example which consists of 17 combinational logic nodes and 14 registers. However, its underlying PN is quite big even for a basic spread token semantics without early propagation, see Figure 10(b). The PN consists of 45 elementary cycles: 17 elementary cycles for combinational logic nodes and 28 elementary cycles to represent 14 registers. The names of places and transitions are hidden as they are not readable at this scale. It is still possible to see the correspondence of the elementary cycles to the original SDFS nodes - their relative layout is preserved.

Due to high concurrency this PN has more than \( 10^7 \) states and therefore cannot be verified by analysing the whole state space in reasonable time. For example, it took PETRIFY three hours before it ran out of memory. Instead, verification tools based on analysis of unfolding prefixes should be employed. The unfolding prefix for this PN has only 164 events and is built by PUNF [13] in 18ms. Analysis of the resultant unfolding by MPSAT confirms that the model of ARISC processor does not have deadlocks.

In this section a method for mapping of high-level spread token SDFS model into low-level PN has been presented. The same technique can be used to build underlying PNs for other SDFS models. The only difference is in the number of elementary cycles representing the state of SDFS nodes. For example,
in counterflow model each combinational logic node \( l \in L \) is associated with two state variables, the forward evaluation state \( \Xi^F (l) \) and the backward evaluation state \( \Xi^B (l) \) which are mapped into a pair of elementary cycles. A counterflow register \( r \in R \) has four state variables: forward enabling \( \Sigma^F (r) \), backward enabling \( \Sigma^B (r) \), OR-marking \( M^O (r) \) and AND-marking \( M^A (r) \). Each of these variables is represented by its own elementary cycle. The transparent correspondence between SDFS and PNs allows to reuse model checking tools developed for PNs to verify SDFS specifications.
Figure 10. ARISC processor
Table 1. Comparison of SDFS token game semantics

<table>
<thead>
<tr>
<th>Token game semantics</th>
<th>Model complexity</th>
<th>Model power</th>
<th>Early evaluation</th>
<th>Preemption mechanism</th>
<th>Conflict resolution</th>
<th>Control complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic token</td>
<td>simple</td>
<td>limited</td>
<td>no</td>
<td>no</td>
<td>n/a</td>
<td>simple</td>
</tr>
<tr>
<td>Spread token</td>
<td>simple</td>
<td>good</td>
<td>partially</td>
<td>no</td>
<td>n/a</td>
<td>simple</td>
</tr>
<tr>
<td>Antitoken</td>
<td>complex</td>
<td>excellent</td>
<td>yes</td>
<td>yes</td>
<td>arbitration</td>
<td>complex</td>
</tr>
<tr>
<td>Counterflow</td>
<td>moderate</td>
<td>excellent</td>
<td>yes</td>
<td>yes</td>
<td>OR-causality</td>
<td>moderate</td>
</tr>
<tr>
<td>Hybrid</td>
<td>simple/moderate</td>
<td>excellent</td>
<td>yes</td>
<td>yes</td>
<td>OR-causality</td>
<td>simple/moderate</td>
</tr>
</tbody>
</table>

9. WORKCRAFT framework

Simulation of SDFS models, their conversion into PNs and verification by existing model checking tools - all these are automated in a consistent framework called WORKCRAFT. The framework has a plug-in driven architecture and supports run-time scripting, which makes it a flexible and expendable environment. Its underlying J AVA technology provides robust cross-platform operation. WORKCRAFT uses OPENGL hardware acceleration for real-time visualisation, which allows interactive animated simulation of large specifications.

All SDFS models presented in this paper were studied and analysed using the WORKCRAFT framework. For this, each SDFS model has been implemented as a WORKCRAFT model plug-in. A model plug-in is just a set of rules which define the structure and the token game semantics of the model. A general purpose conversion algorithm has been implemented as a WORKCRAFT tool. It takes an SDFS graph and a set of rules defining its token game semantics, and converts this model into a low-level PN as has been described in Section 8. The framework also provides a transparent interface to model checking tools, such as PUNF and MPSAT, for seamless verification of the high-level SDFS models using their low-level PNs representation.

The capabilities of WORKCRAFT are not limited to SDFS models. There are plug-ins to support generic graphs, PNs, unfoldings, gate level models, etc. More information about this framework can be found in [19].

10. Comparison of SDFS token game semantics

All the token game semantics presented in this paper have their advantages and drawbacks. In this section the models are informally compared in few aspects, which are summarised in Table 1. In particular, the model complexity, model power, control complexity, support for early evaluation and preemption are compared.

The SDFS token game semantics can be classified as basic and advanced models. The former models only capture basic features of the asynchronous data path, while the later are able to capture more advanced concepts, such as preemption and speculation. Clearly, the atomic token and the spread token semantics belong to the class of basic models, while the antitoken and the counterflow are advanced models.
In the basic model category, both the atomic token and the spread token models have similar complexity. However, the atomic token semantics can only be applied to some class of well-formed SDFS, which limits its model power. The spread token semantics represents a much wider class of asynchronous data path circuits and has a rudimentary support for early evaluation (within one pipeline stage). Therefore, the spread token semantics a better choice for basic SDFS modelling.

In the category of advanced models the difference is mostly in the complexity of the semantics and the implementation of control logic. Both, antitoken and counterflow semantics capture early evaluation and preemption. However, the counterflow semantics has simpler token game rules. Also, the use of OR-causality (as opposed to arbitration in antitoken semantics) for the resolution of conflicts between tokens results in a simpler implementation for control logic. These advantages make counterflow semantics a better choice for modelling SDFS with early evaluation and preemption.

The hybrid token game semantics has the advantages of both, basic and advanced models. In this model the relatively complex counterflow semantics is only used in those parts of SDFS where preemption can be exploited to speed up the data path. In the rest of the SDFS simple spread token semantics is employed. At the level of implementation this results in significant area decrease compared because the ordinary Muller pipeline stages are much smaller than counterflow pipeline controllers.

Verification of SDFS models is based on their conversion into schematic PNs, as has been described in Section 8. The verification tools which use the explicit state space representation of the underlying PN fail even on relatively small SDFS examples. The reason for this is a high level of concurrency in SDFS models, which leads to the state space explosion. The high level of concurrency does not cause a problem for unfolding-based verification tools because unfolding prefixes capture the concurrency in a very compact form, comparable to the size of original PNs. Choice becomes a problem for unfolding though, because each choice branch needs to be unfolded and stored explicitly. However, there is not much choice in the SDFS models. The only source of choice is early evaluation, which is usually limited to few nodes where concurrent branches synchronise. In our experiments, if no early evaluation was allowed, the unfolding time did not exceed few seconds even on relatively large SDFS examples containing few hundred nodes. If early evaluation was enabled, then benchmarks of up to a hundred counterflow SDFS nodes could be verified using unfolding-based tools. The benchmark results based on PUNF unfolder and MPSAT model checker [13] are presented in Table 2.

All the benchmarks in Table 2 have combinational logic nodes with early evaluation. In the small benchmark, which has 27 nodes only, the presence of early evaluation is not critical for the unfolder - it handles both spread token and counterflow semantics within a second. For the average benchmark, which has 70 nodes, the counterflow semantics becomes a problem - the unfolding prefix grows much larger than the PN and it takes nearly two minutes to build. The hybrid SDFS model becomes useful in this case. If the counterflow semantics is only applied to those 12 nodes which can exhibit preemption, then the unfolding size is much smaller and the computation time is just 4 seconds. The large benchmark, which consists of 524 nodes, is verified in 2 seconds under spread token semantics. However, if the counterflow semantics is applied, the computation time exceeds 38 minutes; if the hybrid semantics is used with 96 nodes exhibiting preemption, then the computation time is reduced to 8 minutes. Therefore, few hundred nodes is a practical limit for the size of SDFS models which can be verified by our method within acceptable time.

Let us study the influence of early evaluation on the size of unfolding prefix and computation time using benchmarks ee2, ee3 and ee4. These benchmarks are essentially the same SDFS, but with different number of early evaluating fork-join blocks - two, three and four early evaluating blocks, respectively.
The early evaluating block in these benchmarks is such that any of its three inputs is sufficient to produce the output. For the spread token semantics the number of early evaluation blocks does not change the unfolding time or size much because there is no preemption in this model and the early evaluation is limited to a single pipeline stage. Contrary, for the counterflow semantics both the size of unfolding prefix and its computation time grow exponentially with the number of early evaluation blocks. This is due to the choices introduced by early evaluating combinational logic nodes.

In the last benchmark, called deadlock, the evaluating and resetting conditions of combinational logic nodes were modified to force a deadlock in the model. Verification of the model reviled a trace leading to a deadlock state.

### 11. Conclusions

In this work we have formally defined a token-based SDFS model, which captures the asynchronous data path. Three token game semantics are introduced on this model: atomic token, spread token and...
counterflow. A hybrid SDFS model, which combines the advantages of spread token and counterflow semantics, has been presented.

A method has been developed for converting high-level SDFS models into low-level Petri nets for subsequent verification and model checking by existing tools. To study different token game semantics of SDFS model a software tool called WORKCRAFT has been developed. This tool is capable of integrating arbitrary token-based models into a consistent framework. The models presented in this paper have been implemented as plug-ins to WORKCRAFT which were used to analyse the advantages and drawbacks of SDFS models on a set of benchmarks.

Acknowledgements. We are grateful to Victor Khomenko, Charles Brej and Will Toms for numerous insightful discussions and to the anonymous referees for very useful comments. This work is supported by EPSRC grant EP/D053064/1 (SEDATE).

References


