Reachability and Design of Communication Fabrics

A Master Thesis

Sanne Wouda

Supervisors:
dr. Julien Schmaltz
prof.dr.ir. Jan Friso Groote
prof.dr. Johan J. Lukkien

version 0.9.3 (August 17, 2015)

Eindhoven
We propose a technique for reachability analysis in micro-architectural models of communication fabrics. We group channels in what we call transfer islands. Combining transfer islands in different ways results in synchronous, asynchronous and interleaving semantics. We give semantics to the primitive models using process algebra. To justify our abstraction, we prove equivalence between the process algebra and the transfer islands semantics. I/O automata are added to enable integrated verification of protocols and fabrics. We encode the micro-architectural model together with a given state reachability property in the input format of nuXmv. Reachability is solved either using BDDs or IC3. Combined with inductive invariant generation techniques, the approach shows promising results. For the design of larger communication fabrics, we also introduce a domain-specific language prototype.
## Contents

1. **Introduction**  
   1.1. Context  
      1.1.1. xMAS  
      1.1.2. Process Algebra  
   1.2. Motivation  
   1.3. Contribution  

2. **Process Algebra-based Semantics**  

3. **Transfer Islands**  
   3.1. Defining Islands  
   3.2. Synchrony and Asynchrony  
   3.3. Equivalence to Process Algebra-based Semantics  
   3.4. Island Extraction  
   3.5. State Machine Extension  
      3.5.1. I/O Automata  
      3.5.2. In Process Algebra  
      3.5.3. In Transfer Islands  
      3.5.4. Transition System  

4. **Coding in xMAS: CodexMAS**  

5. **Experimental Work**  
   5.1. Small Examples  
   5.2. 2-agents Example  
   5.3. Execution Times  

6. **Conclusion**  
   6.1. Limitations  
   6.2. Future Work  

A. **nuXmv Translation**  

B. **CodexMAS Syntax**  

Bibliography
List of Figures

1.1. The eight xMAS primitives. ................................. 2
1.2. A simple example of an xMAS network. ................. 3

3.1. A Venn diagram of state spaces. ......................... 15
3.2. An I/O automaton modeling a merge primitive. ......... 22

5.1. An unreachable state in asynchronous semantics. ...... 34
5.2. A false deadlock with fractions of messages. ........... 34
5.3. The red and blue example. ................................. 34
5.4. 2-agents. .................................................. 35

List of Tables

5.1. Results for the synchronous semantics. ................. 36
5.2. Results for the asynchronous semantics. ................. 36

List of Algorithms

1. Computation of transfer islands. ............................ 21
1. Introduction

In modern computing systems, performance improvements come from increases in parallelism [1]. Systems-on-Chips (SoCs) as found in e.g. smartphones are increasingly parallel by including more processors on one chip. Processors themselves are increasingly parallel through an increase in the number of cores per processor. Bus architectures suffer performance degradation when the number of communicants increases. Networks-on-Chips (NoCs) [2, 3] are a better but more complex alternative. Techniques to aid the design of communication fabrics, the interconnect infrastructure of NoCs, has been studied recently. Intel [4, 5] has proposed a number of primitives—xMAS for eXecutable MicroArchitectural Specifications—to describe communication fabrics. A tool called WickedXmas [6] developed around xMAS is capable of analyzing complex safety and liveness properties such as deadlock freedom.

WickedXmas uses invariants to over-approximate the state space [7, 8] in order to be efficient. As a result it is incomplete and can report false deadlock scenarios, that is, deadlocks that are not reachable from an empty network. Additionally, the tool only allows the specification of networks by graphical means which is impractical for large or complicated fabrics.

To solve the reachability question, we propose a technique to translate xMAS networks to a model suitable for model checking. To this end, we introduce the concept of transfer islands and mathematically prove its correctness against a novel semantics for xMAS-like networks. Finally, we propose a textual language for specifying xMAS networks.

This introduction continues with a discussion of related work.

1.1. Context

Our discussion of related work starts with a brief overview of the xMAS language and work related to xMAS. Then we briefly detail a process algebra used in later chapters.

1.1.1. xMAS

An xMAS model is a network of instantiations of primitives, called components, connected via typed, one-directional channels. We write a network $\mathcal{N} = (C, X)$ as a set of components and a set of channels. A channel is connected to an initiator and a target component. A channel is composed of three signals. Channel signal $x.irdy$
indicates whether the initiator is ready to write to channel $x$. Channel signal $x.\text{trdy}$ indicates whether the target is ready to read from channel $x$. Channel signal $x.\text{data}$ contains data that is transferred from the initiator output to the target input if and only if both signals $x.\text{irdy}$ and $x.\text{trdy}$ are set to true.

Figure 1.1 shows the eight primitives of the xMAS language. A function primitive manipulates data. Its parameter is a function (written $c.f$ for a function component $c$) that produces an outgoing packet\(^1\) from an incoming packet. Typically, functions are used to convert packet types and represent message dependencies inside the fabric or in the model of the environment. A fork duplicates an incoming packet to its two outputs. Such a transfer takes place if and only if the input is ready to send and the two outputs are both ready to read. A join is the dual of a fork. The function parameter determines how the two incoming packets are merged. A transfer takes place if and only if the two inputs are ready to send and the output is ready to read. A switch uses its function parameter to determine to which output an incoming packet must be routed. A merge is an arbiter. It grants its output to one of its inputs. The arbitration policy is a parameter of the merge. A queue stores data. Messages are non-deterministically produced and consumed at sources and sinks. A source or sink may process multiple packet types. We refer to ports by their names as in Figure 1.1. For example, $q.i$ is the input channel of queue $q$.

Intuitively, the execution semantics of an xMAS network consists of a combinatorial and a sequential part. The combinatorial part updates the values of channel signals. The sequential part is the synchronous update of all queues according to the values of the channel signals. A simulation cycle consists of a combinatorial and a sequential update. A sequential update only concerns queues, sinks and sources. We denote these primitives as sequential primitives. Other primitives are denoted as combinatorial.

For each output port $o$, signal $o.\text{irdy}$ is set to true if the primitive can transmit a packet towards channel $o$, i.e., port $o$ is ready to transmit to its target. For each input port $i$, signal $i.\text{trdy}$ is set to true if the primitive can accept a packet from input channel $i$, i.e., the target of channel $i$ is ready to receive. In a sequential primitive, the values of output signals depend on the values of the input signals and an internal

\(^1\)We use the terms packet and message interchangeably.
state. Queues accept packets only when they are empty. A source and a sink produce or consume a packet according to an internal oracle which models non-determinism.

**Example 1.** Figure 1.2 shows an example of an xMAS network. The top source injects requests or responses which are duplicated in queues $q_0$ and $q_1$. The other source only injects requests in $q_1$. At the output of $q_1$ requests are routed to a sink via a merge. Responses are routed back to a join synchronizing with $q_1$ before traveling further to the sink. We will use this simple network as a running example throughout this thesis.

A deadlock is characterized by a message that is stuck at the head of a queue, unable to make progress [6]. A channel $x$ cannot make progress, i.e. it is blocked, if $x$ is ready to send ($irdy$) but is unable to ever do so (globally not $trdy$). Deadlocks can be hunted using Boolean equations [9]. Verbeek and Schmaltz [7] improve on this technique by reducing the state space with generated invariants, but this is an over-approximation.

**Example 2.** Figure 1.2, with $q_0$ containing only requests and $q_1$ empty, is in a deadlock configuration. We have $q_0.oidy$, since there is at least one message in $q_0$. However, $q_0.oidy$ will never hold, since $q_1$ contains no responses. It will never do so because $c_0$ cannot insert a response because $q_0$ is full.

An extension of xMAS with an I/O automata primitive was proposed by Verbeek et al. [10]. The extension improves the specification of protocols in xMAS, while maintaining the ability to generate invariants over protocol state and network state together.

Overall we assume xMAS networks that are syntactically correct and without combinatorial cycles. Procedures exist to check for these properties [11].

**1.1.2. Process Algebra**

Here we give a brief description of a process algebra, based on principles by Baeten, Basten and Reniers [12]. We abbreviate or combine some definitions for brevity where possible.

Given a set of channels $X$ and data $D$, we describe communicating processes with the following operations:

- When data $d \in D$ is read from channel $x \in X$, this is denoted as a read $x?d$. Similarly, a write of $d$ on $x$ is denoted as $x!d$. Alternatively, a step that neither reads nor writes is a non-communicating step and is written without $?$ or $!$. 

We can perform a set of reads, writes or steps $u$ simultaneously and continue with process $P$. This is denoted as $u . P$, so $u . P$ is a process if $P$ is a process and $u$ is a set of reads, writes and steps. Instead of using a set notation for $u$, we will omit the brackets $\{}$, separate the elements of $u$ by $|$ and call it an action as is usual in process algebra. For example, we write $x_0 ? d_0 | x_1 ! d_1 | a . P$ for $u . P$ when $u = \{ x_0 ? d_0, x_1 ! d_1, a \}$. Note that $u . P = P$ iff $u = \emptyset$.

The set of actions is called $A$, as in process algebra. The set of non-communicating steps is called $A^\perp$.

- Perform either process $P$, or process $Q$: $P + Q$.
- Perform the processes $P$ and $Q$ in parallel: $P \parallel Q$.
- Choose only the maximal transitions of $P$: $\theta(P)$.

Apart from the processes described above, we allow recursive definitions of processes. The intended process is the least fixed-point of the recursive definition.

We abbreviate a simultaneous read $x ? d$ and write $x ! d$ as a transfer $x / d$. A transfer $x / d$ always occurs within an action rather than as a separate element, as $\{ x ? d \}$ is a shorthand for $\{ x ! d, x ? d \}$, or $x ! d | x ? d$ in the process algebra convention. We write $x ? d \in u$ for $\{x ? d, x ! d\} \subseteq u$. For actions $u, v \in A$ we write $u | v$ for $u \cup v$, as is usual in process algebra.

We introduce $\approx$ as an equivalence between processes. In this equivalence, the operations $+$ and $\parallel$ are associative and commutative. Operator $\parallel$ distributes over $+$, or formally:

\begin{align*}
P + Q & \approx Q + P \quad (1.1) \\
P + (Q + R) & \approx (P + Q) + R \quad (1.2) \\
P \parallel Q & \approx Q \parallel P \quad (1.3) \\
P \parallel (Q \parallel R) & \approx (P \parallel Q) \parallel R \quad (1.4) \\
P \parallel (Q + R) & \approx (P \parallel Q) + (P \parallel R) \quad (1.5)
\end{align*}

Processes describe automata in the sense that a process can perform a transition. We denote a transition as $P \xrightarrow{u} Q$, where $u$ is a set of communications performed by the process $P$ and $Q$ is the resulting process. The communications arise when two processes simultaneously write and read from the same channel. Readers who are familiar with process algebra may recognize that by only allowing communications as actions, we make the encapsulation operation implicit in our definition.

We express the transitions for each process based on its structure as follows:

- A communication between parallel processes. We write $\parallel_i P_i$ for $P_0 \parallel \cdots \parallel P_n$. Let $a_i$ be sets of steps and let $u_i$ be sets of reads and writes such that the
union $\bigcup_i u_i$ contains a read $x?d$ for every write $x!d$ and vice versa:
\[
\| (u_i|a_i, P_i) \|_{\bigcup_i (u_i|a_i)} \| P_i
\]
if $x?d \in \bigcup_i u_i \iff x!d \in \bigcup_i u_i$
(1.6)

- A choice:

\[
\text{if } P \xrightarrow{u} R \text{ then } P + Q \xrightarrow{u} R
\]
(1.7)

- A step allowed by an equivalent process.

\[
\text{if } P \approx Q, R \approx S \text{ and } P \xrightarrow{u} R, \text{ then } Q \xrightarrow{u} S
\]
(1.8)

- Given a process $P$ and some sets of possible transitions, we write $\theta(P)$ to limit the transitions to those that are maximal, but only with respect to read and writes on channels:

\[
\theta(P) \xrightarrow{u} \theta(Q)
\]
if $P \xrightarrow{u} Q$ and there is no $v \supset u$ s.t. $P \xrightarrow{v} Q$, with $v \cap A^- = u \cap A^-$
(1.9)

Readers with a background in process algebra will see a slight deviation: Equation 1.8 is usually a notion of equivalence that follows from similarities in the transition relation. We have included it in the transition relation to write Equations 1.6 to 1.8 more succinctly. Similarly, Equation 1.9 explicitly contains an ordering for the familiar priority operator $\theta$.

### 1.2. Motivation

From the survey of related work, we see a gap in the knowledge as to the elimination of false positives in existing deadlock detection tools. Furthermore, the existing description of semantics to xMAS primitives is unsuitable for proving correctness of theorems relating to the behavior of networks. During the specification of the networks required by this thesis, we have found a lack in tooling support for designing even small networks.

### 1.3. Contribution

In this thesis we contribute a novel semantics of an xMAS-like language, using process algebra. We propose a technique, called transfer islands, to translate networks to a model suitable for model-checking reachability. We provide an algorithm to compute transfer islands. Experimental results compare the performance of symbolic model-checking (BDDs) with IC3 [13]. Performance is further improved using inductive
invariants automatically generated from the model using existing techniques [14, 15] and their implementations in a design tool for xMAS [6]. These contributions will be presented at MEMOCODE 2015 [16].

New in this thesis is a discussion of semantics by combining different sets of transfer islands. We extend our process algebra description and transfer islands with I/O automata [10]. We introduce a domain-specific language for specifying XMAS-like networks, called CodexMAS. A domain-specific language, as opposed to a generic programming language, is specifically designed to solve problems in a particular domain [17]. Advantages of a domain-specific language include the ability to express solutions at the right abstraction level, to increase productivity and to allow verification at the domain level. To describe messages and message functions in our domain-specific language, we use a language inspired by Van Gastel [11].

The process algebra-based semantics are presented in Chapter 2. Transfer islands are introduced and discussed in Chapter 3. CodexMAS is introduced by way of a tutorial in Chapter 4. Chapter 5 presents the experimental work. The thesis is concluded in Chapter 6.
2. Process Algebra-based Semantics

We define a process algebra-based semantics for arbitrary networks. We show that xMAS primitives can be described using the process algebra from the previous chapter. These definitions coincide with those of the original xMAS language, with exception of the merge primitive. The latter is abstracted away and our semantics assume a non-deterministic choice at arbitration points. For reachability, any sequence of transfers allowed by our process algebra semantics can be mimicked with an appropriate arbitration policy in the xMAS merge. Conversely, any arbitration policy can be mimicked by appropriate choices at arbitration points. We therefore conclude that this abstraction is sound for reachability analysis.

When creating communication fabrics using process algebra, we use a network of channels $X$ and components $C$. Each component $c$ is specified as a process on a subset of all channels. The mapping between components and channels is such that each channel has exactly one component that can perform reads and one that can perform writes. The final process that is described, is the parallel composition of the processes of all components.

Used this way, the process algebra allows us to give descriptions of xMAS components as processes, without being explicit about the indy and trdy wires.

Queue  The xMAS queue is defined as a “synchronous FIFO with a standard interface.” The size, or capacity, of the queue $k > 0$ is a parameter of the queue process. Process $Q^k_\sigma$, with $\sigma \in D^*$, models this behavior.

There are three cases. When the queue is empty, only write to the queue is possible. Process $Q^k_\epsilon$ receives a message $d \in D$ through an $i?d$ read. Data $d$ is stored as the new contents of the queue resulting in process $Q^k_d$.

$$Q^k_\epsilon = \sum_{d \in D} i?d . Q^k_d$$

When a queue is full ($|\sigma d| = k$) only a send is allowed. Process $Q^k_{\sigma d}$ sends a message through the $o!d$ write and $d$ is removed from the head of $\sigma d$ resulting in process $Q^k_\sigma$.

$$Q^k_{\sigma d} = o!d . Q^k_\sigma$$

When a queue is neither empty nor full ($|\sigma d| < k$), both a send and receive can execute separately or simultaneously.

$$Q^k_{\sigma d} = \sum_{e \in D} \left( i?e | o!d . Q^k_{e \sigma} + i?e . Q^k_{e \sigma d} \right) + o!d . Q^k_\sigma$$
Function  A function simultaneously receives a message \( d \) via read \( i?d \), modifies it according to \( f \) and sends it onwards through write \( o!f(d) \). Note that the send and receive happen simultaneously.

\[
\text{Function}_f = \sum_{d \in D} i?d|o!f(d) \cdot \text{Function}_f
\]

Source  A source sends a message \( d \) into the network via the write \( o!d \). Action \( c \) is a non-communicating step. The operator \( \theta \) does not consider this \( c \) step in defining a priority between two actions.

\[
\text{Src} = \sum_{d \in D} c|o!d \cdot \text{Src}
\]

Sink  The sink is similar to the source, receiving a message through read \( i?d \). Again non-communicating step \( c \) ensures correct behavior through the priority operator.

\[
\text{Sink} = \sum_{d \in D} c|i?d \cdot \text{Sink}
\]

Fork  A fork reads message \( d \) on its input channel via \( i?d \) and writes two identical messages \( d \) to its two output channels: \( a!d \) and \( b!d \). As with a function (and all combinatorial primitives), all messages are sent and received simultaneously.

\[
\text{Fork} = \sum_{d \in D} i?d|a!d|b!d \cdot \text{Fork}
\]

Join  The join is quite similar to the fork. Instead though, two messages are received simultaneously which are combined using function parameter \( h \). The duality between fork and join is especially visible in the process definition.

\[
\text{Join}_h = \sum_{d,e \in D} a?d|b?e|o!h(d,e) \cdot \text{Join}_h
\]

We call this the unrestricted join, as opposed to the control join. An unrestricted join takes any arbitrary function parameter, while the control join only allows a function passing either the message from \( a \), or the message from \( b \).

Merge  xMAS specifies the arbitrator with a policy. As mentioned before, we abstract from this policy leaving only a non-deterministic choice between reading on port \( a \) or reading on port \( b \).

\[
\text{Merge} = \sum_{d \in D} (a?d|o!d \cdot \text{Merge} + b?d|o!d \cdot \text{Merge}) \quad (2.1)
\]
Switch  Finally, the switch routes messages based on its parameter function $f$. The switch performs read $i?d$. If $f(d)$ is satisfied, then the switch writes $d$ to port $a$ with $a!d$. If $f(d)$ is not satisfied, then the switch writes to port $b$ instead.

$$Sw_f = \sum_{d \in D \land f(d)} i?d!a!d \cdot Sw_f + \sum_{d \in D \land \neg f(d)} i?d!b!d \cdot Sw_f$$

Recall that an xMAS network is a composition of components through channel connections. To model the behavior of an xMAS network as a whole, we use a suitable composition of our primitive processes, such that the behavior of the process sufficiently matches the behavior of an xMAS network. Using standard parallel composition to compose the primitive processes, process $P_N$ gives semantics to xMAS network $N = (C, X)$ and is defined as follows:

$$P_N = \theta \left( \parallel_{c \in C} P_c \right).$$

In the process definitions, all primitives except queues are stateless. The aggregate state of the network is hence the aggregate state of the queues in the network. We call this network state.

**Definition 1** (network state). For a network $N = (C, X)$, a state $s$ is a mapping from queues to a string of messages $D^*$. The state space $S_N$ is defined as all such mappings. The initial state of the network maps each queue to the empty string $\epsilon$.

We refer to the state of a stateful component using the following notation.

**Notation 2** (queue state). The state of a queue $c$ in network state $s$ is written $s_c$.

Henceforth we simple write state when we are referring to network state. We write $P^s_c$ for the process of a component $c$ in component state $s$ and $P_c$ for the process of a component $c$ in the initial state.

**Definition 3** (process at a state). The process in a state $s$ is written as $P^s = \theta \left( \parallel_{c \in C} P^s_c \right)$.

The priority operator $\theta$ only considers actions that have the same non-communicating steps. The source and sink processes always perform a non-communicating step in parallel with a read or write. It is therefore not the case that a maximal number of sources and sinks must transfer. For each source or sink the choice can be made non-deterministically.

The process $P_N$ defines a synchronous execution of the network. We also define process $\bar{P}^s$, which gives an asynchronous execution to the network. Process $\bar{P}^s$ removes the priority operator, allowing any number of channels to transfer at once as long as the reads and writes match.

$$\bar{P}^s = \parallel_{c \in C} P^s_c$$
3. Transfer Islands

The previous chapter introduced a process algebra-based semantics for communication fabrics. In order to answer the reachability question, we propose a technique to describe the transition relation for a complete network, which encapsulates the combinatorial part of the network.

The key notion is that a transition of the network can be characterized by which channels transfer at the same time. There are some channels that cannot transfer at the same time, e.g. the two inputs of a merge. On the other hand, some channels must transfer in a synchronized fashion, e.g. the input and output of a function. These dependencies are captured by the concept of combinatorial-closed sets of channels. The smallest such sets are especially nice to reason about, so we call these transfer islands.

Ultimately, we pre-compute which islands can transfer based solely on the state of the queues in the network. The end result is that we abstract away from any intermediary irdy, trdy and data signals, which simplifies the creation of an efficient model. This also eliminates the need for explicit parallelism in the model.

Example 3. Consider the running example in Figure 1.2. All channels between source $c_0$ and the input channels of the queues ($x_0, x_1, x_2$ and $x_4$) always transfer together. This is due to the fork that creates this synchronization of channels. The set composed of all these synchronized channels forms a transfer island. This network has four islands in total, which are shown in Example 4.

In the next section, we will formally define combinatorial-closedness and transfer islands. For transfer islands—and combinations thereof—we define when they can transfer and what state the network will have afterwards. Section 3.2 discusses semantics by changing the allowed sets of transfer islands. In Section 3.3, transfer island semantics is proven equivalent to the process algebra-based semantics. Section 3.4 will give an algorithm to compute the full set of transfer islands for any network. The chapter concludes with an extension with I/O automata in Section 3.5.

3.1. Defining Islands

As a warning to the reader, the following sections are fairly technical and presume a strong knowledge of process algebra and mathematical logic.

Combinatorial-closed sets of channels are the key concept for defining transfer islands. This notion is used to identify synchronized channels. The idea is that
switches and merges create an exclusive dependency—and hence distinct islands—while the other components combine inputs and outputs into one island. Sequential primitives like queues impose no dependencies between inputs and outputs.

**Definition 4** (combinatorial-closed). Given a network $\mathcal{N} = (C, X)$, a set of channels $I \subseteq X$ is combinatorial-closed iff:

- for any merge $c$ connected to $I$ we have $c.i \in I$ and $(c.a \in I \oplus c.b \in I)$;
- for any switch $c$ connected to $I$ we have $c.o \in I$ and $(c.a \in I \oplus c.b \in I)$;
- for any other combinatorial component $c$, connected to $I$, all inputs and outputs are in $I$,

where $\oplus$ denotes the exclusive or.

A transfer island is formally defined as a non-empty set of combinatorial-closed channels. We take the smallest among such sets.

**Definition 5** (transfer island). Given a network $\mathcal{N} = (C, X)$, a set of channels $I$ with $\emptyset \neq I \subseteq X$ is a transfer island iff

- $I$ is combinatorial-closed and
- if there exists a combinatorial-closed set of channels $I'$ with $\emptyset \neq I' \subseteq I$ then $I' = I$.

**Example 4.** Applying the definition to the example in Figure 1.2, we obtain the following transfer islands: $I_0 = \{x_0, x_1, x_2, x_4\}$, $I_1 = \{x_3, x_4\}$, $I_2 = \{x_5, x_6, x_7, x_9, x_{10}\}$, and $I_3 = \{x_5, x_8, x_{10}\}$.

**Transition System**

To give semantics to transfer islands, we define a finite transition system. The transition system for a network $\mathcal{N} = (C, X)$ consists of the state space $S_N$ (see Definition 1) and a set of transitions $T_N : S_N \times S_N$.

The set of transitions is defined in stages. First, we define when a transfer island is enabled in a state. Then, we define when a set of transfer islands is enabled. Finally, we define what the next state is based on which channels transfer.

It follows that there is a transition $(s, s') \in T_N$, with $s, s' \in S_N$, if and only if there exists a set of transfer islands that are enabled in $s$ and have $s'$ as next state.

**Enabledness**

We define predicate $\rho_I(s)$ indicating that the transfers of $I$ are enabled in state $s$. We first define that a channel is ready to transfer some data $d$.

**Definition 6** ($d$-transfer-ready). A channel $x$ is ready to transfer message $d \in D$ in state $s$ iff $P^s \xrightarrow{\nabla} P^{s'}$ and $x!d \in u$, for some next state $s'$.
In other words, when the process $P^*$ can do an action which communicates data $d$ over channel $x$, then it is ready to transfer $d$ in $s$.

Usually, there is no need to refer to the data that is to be transferred.

**Definition 7** (transfer-ready). A channel $x$ is transfer-ready iff $x$ is $d$-transfer-ready for some $d \in D$.

A transfer island is enabled if and only if all its channels are ready to transfer.

**Definition 8** (island-enabled). Given a network $\mathcal{N} = (C, X)$, a transfer island $I \subseteq X$ is island-enabled, usually just ‘enabled’, in a state $s \in S_\mathcal{N}$ iff all channels $x \in I$ are transfer-ready in $s$. We write $\rho_I(s)$ if $I$ is enabled in $s$.

For a set of transfer islands to be enabled, not only should each transfer island be enabled, but they must not share any channels. Note that transfer islands in general can share a channel. This follows from the closed-conditions for merges and switches. For example, the islands containing input $a$ of a merge will also contain output $o$. The same holds for islands containing input $b$. These sets of islands are distinct, but they all contain output $o$.

**Definition 9** (set-enabled). A set of transfer islands $\mathcal{T}$ is set-enabled in state $s \in S_\mathcal{N}$, written $\rho_\mathcal{T}(s)$, if and only if

- for each $I \in \mathcal{T}$, we have that $I$ is island-enabled in $s$ and
- for all $I, I' \in \mathcal{T}$, if $I \neq I'$ then $I \cap I' = \emptyset$.

A synchronous network executes all enabled transfer islands, as long as they do not conflict. In case of conflict, we need to make a choice. In any case only the maximal set of transfer islands is enabled.

**Definition 10** (maximally-enabled). A set of transfer islands $\mathcal{T}$ is maximally-enabled in state $s$, iff

- $\mathcal{T}$ is set-enabled in $s$ and
- if there exists a set transfer islands $\mathcal{T}'$ enabled in $s$ and $\mathcal{T} \subseteq \mathcal{T}'$ with $\mathcal{T}$ and $\mathcal{T}'$ identical in inputs and outputs of sources and sinks, then $\mathcal{T} = \mathcal{T}'$.

Next, we consider what the next state could be after a set of channels has transferred.

**Next state**

A next state depends on the previous state and the channels that transfer. We define predicate $\tau_Y(s, s')$ indicating that $s'$ is a next state after the transfer of $Y$, for any of transfer-ready channels $Y \subseteq X$. We say that $s'$ is a $Y$-successor of $s$. 

13
Definition 11 (Y-successor). Given a network $\mathcal{N} = (C, X)$, states $s, s' \in S_\mathcal{N}$ and a set of channels $Y \subseteq X$ with for each $x \in Y$, $x$ is $d_x$-transfer-ready in $s$. We have $\tau_Y(s, s')$, $s'$ is a Y-successor of $s$, iff the following conditions hold for all queues $c$:

- if $c.i, c.o \in Y$, then $s'_c d_{c.o} = d_{c.i} s_c$
- otherwise if $c.i \in Y$, then $s'_c = d_{c.i} s_c$
- otherwise if $c.o \in Y$, then $s'_c d_{c.o} = s_c$
- otherwise $s_c = s'_c$

Example 5. Consider transfer island $I_0$ from Example 4. The input channels of $q_0$ and $q_1$—namely $x_1$ and $x_4$—are members of $I_0$. A next state is found by inserting data from $x_1$ and $x_4$ into $q_0$ and $q_1$.

Intermediate data

With our transfer island technique, we can recursively define the value of data for each channel. We call this data propagation.

The data in the network depends on the input packets from the sources. We generalize this to an input/output function $M : C \rightarrow D$ specifying the inputs and outputs of the network. We say that an action $u$ obeys $M$ iff 1) for every source $c$, we have $c.o.d \in u$ iff $M(c) = d$ and 2) for every sink $c$ we have $c.i.d \in u$ iff $M(c) = d$.

Definition 12 (data propagation). Given a set of transfer-ready and combinatorial-closed channels $Y \subseteq X$, state $s$, input/output function $M$ and channel $x \in Y$. We define data propagation $\pi_x(s, M, Y)$.

Let $c$ be the initiator of $x$. We define $\pi_x(s, M, Y)$ to equal the following

- $M(c)$, if $c$ is a source;
- $d$, if $c$ is a queue and $s_c = \sigma d$;
- $c.f(\pi_{c.i}(s, M, Y))$, if $c$ is a function;
- $\pi_{c.i}(s, M, Y)$, if $c$ is a fork or switch;
- $c.h(\pi_{c.a}(s, M, Y), \pi_{c.b}(s, M, Y))$, if $c$ is a join;
- $\pi_{c.a}(s, M, Y)$, if $c$ is a merge and $c.a \in Y$;
- $\pi_{c.b}(s, M, Y)$, if $c$ is a merge and $c.b \in Y$.

This definition is well-founded because networks have no combinatorial cycles and $Y$ is combinatorially-closed.

Corollary 1. Given a set of transfer islands $\mathcal{T}$, state $s$ and input function $M$. Let $X'$ be the union of the islands in $\mathcal{T}$. It holds that $\mathcal{T}$ is maximally-enabled in $s$ if and only if for each channel $x \in X'$ we have that $x$ is $\pi_x(s, M, X')$-transfer-ready.

Proof. Follows from the definitions of primitive processes.
Conditionals

For the translation to a model, we use conditions independent of the process algebra-based semantics.

Definition 13. The set of combinatorial-closed channels $Y$ is conditionally enabled in $s$, with input/output function $M$, iff for each $x \in Y$:

- if the initiator of $x$, say $c$, is a queue, then $s_c \neq \epsilon$,
- if the target of $x$, say $c$, is a queue with capacity $k$, then $|s_c| \neq k$,
- if $x = c.a$ and $c$ is a switch, then $c.f(\pi_x(s,M,Y))$,
- if $x = c.b$ and $c$ is a switch, then $\neg c.f(\pi_x(s,M,Y))$,
- if the initiator of $x$, say $c$, is a source, then $M(c)$ is defined and
- if the target of $x$, say $c$, is a sink, then $M(c)$ is defined.

Corollary 2. A set of combinatorial-closed channels $Y$ is conditionally enabled iff all channels in $Y$ are transfer-ready.

Proof. Follows from the definitions of primitive processes. \qed
3.2. Synchrony and Asynchrony

In this section we discuss what effects the choice of transfer island sets has on semantics. We discuss four variants, resulting in different types of semantics. The resulting state spaces are visualized in a Venn diagram in Figure 3.1.

The simplest variant allows any set of transfer islands as long as it is enabled by set-enabledness (Definition 9). Recall that transfer islands are set-enabled when the individual islands are enabled and have no channels in common. Any combination of independent transfer islands can transfer during the sequential update. We call this the *asynchronous* variant. We will write \( S_{\text{async}} \) for the reachable states from the empty network using this variant. In Section 3.3 we will show that the asynchronous variant is equivalent to the process algebra semantics without priority operator.

The second variant allows only one transfer island to execute at a time. This variant is useful for model checking, since a transfer island corresponds one-to-one with a transition. We call this the *interleaving* variant and write \( S_{\text{leaf}} \) for the states reachable from the empty network.

The relation between interleaving and asynchronous semantics is interesting. Clearly, interleaving semantics can be simulated by the asynchronous variant since the latter is less restrictive. However, interleaving semantics can also simulate the asynchronous semantics. Any asynchronous transition can be mimicked by transferring the islands one by one. We conclude that \( S_{\text{async}} = S_{\text{leaf}} \).

The third variant allows only a maximal set of non-intersecting transfer islands. Intuitively, all enabled transfer islands that do not share a channel with another enabled transfer island will transfer in this case. If there are shared channels between enabled transfer islands, then a choice has to be made. We call this the *synchronous* variant and it neatly maps to a hardware implementation where all enabled channels transfer during a sequential step. Note that in a hardware or model checking context any choice must be implemented explicitly.

Finally, the original semantics of xMAS as given by Intel’s reasearchers is slightly different from the synchronous variant. The original xMAS semantics is more restrictive as an explicit arbitration policy is given, while transfer islands and process algebra semantics abstract away from it. Persistency is also an issue. This is discussed in Section 6.1.

In Chapter 5 we give model checking results for interleaving and synchronous semantics.

3.3. Equivalence to Process Algebra-based Semantics

To justify the correctness of transfer islands, we prove semantic equivalence between transfer island-based and process algebra-based semantics. We prove that a transfer occurs on a channel of a transfer island if and only if a transfer occurs on this channel in the process algebra semantics.

The proof is divided into two parts. We first prove equivalence for asynchronous
semantics. Using the asynchronous proof, we can ultimately prove that transfer island and process algebra semantics are equivalent in a synchronous semantics.

**Theorem 1** (asynchronous equivalence). Given a network $N = (C, X)$, states $s, s' \in S_N$ and input/output function $M$. There exists an action $u$, with $u$ obeying $M$, such that $P^s \xrightarrow{u} P^{s'}$ iff there exists a set of transfer islands $T$ such that

- $T$ is set-enabled in $s$ and
- $s'$ is an $X'$-successor of $s$,

where $X'$ is the union of all islands in $T$.

**Proof.** ($\Rightarrow$) Take an action $u$ such that $\bar{P}^s \xrightarrow{u} \bar{P}^{s'}$. Let $X_u \subseteq X$ be all channels that transfer in $u$, that is, $X_u = \{x \mid \exists d \in D \exists x\!\vdash d \in u\}$. Let $T$ be a partition of $X_u$ into transfer islands which exists per Lemma 1 and Lemma 2 (appearing at the end of the section).

For all islands $I \in T$, we have that $I$ is enabled in $s$ (see Definition 8), since we have $I \subseteq X_u$ and for each $x \in I$ there is some $x\!\vdash d \in u$. Furthermore, all islands are non-intersecting and non-empty, since $T$ is a partition.

We show that $s'$ is an $X'$-successor of $s$, where $X' = \bigcup_{I \in T} I$. Since $T$ is a partition of $X_u$ it follows that $X_u = X'$. Consider the primitive process for the queue as defined in Chapter 2. We have that $\bar{P}^s \xrightarrow{u} \bar{P}^{s'}$ implies for any queue $c$ and messages $d, e \in D$ that

- if $c.i\!\vdash d, c.o\!\vdash e \in u$ then $s'_c e = ds_c$
- otherwise, if $c.i\!\vdash d \in u$ then $s'_c = ds_c$
- otherwise, if $c.o\!\vdash e \in u$ then $s'_c e = s_c$
- otherwise, $s_c = s'_c$.

By Definition 11 it follows that $s'$ is an $X'$-successor of $s$.

($\Leftarrow$) Assume $T \subseteq 2^X$ is a set of transfer islands such that $T$ is enabled in $s$ and $s'$ is an $X'$-successor of $s$, with $X' = \bigcup_{I \in T} I$.

Let action $u$ be defined as follows:

$$u = \{x\!\vdash d \mid x \in X' \land d = \pi_x(s, M, X')\}$$
$$\cup \{c \mid \text{source or sink } c \text{ with } c.o \text{ or } c.i \text{ in } X'\}$$

By this definition, $u$ obeys $M$.

We show that $\bar{P}^s \xrightarrow{u} \bar{P}^{s'}$, that is

$$\bigparallel_{c \in C} P^s_c \xrightarrow{u} \bigparallel_{c \in C} P^{s'}_c$$

We define $u_c \subseteq u$, for each $c \in C$, as the smallest sets satisfying: if $x\!\vdash d \in u$, then
• \( x!d \in u_c \) if \( c \) is the initiator of \( x \),

• \( x?d \in u_c \) if \( c \) is the target of \( x \) and

If \( c \in u \) then \( c \in u_c \).

It follows that

1. we have \( P^s_c \xrightarrow{u_c} P'^s_c \), for each \( c \in C \), by the definitions of \( u, u_c \), data propagation (Definition 12) and the primitive processes and

2. by the definition of \( u_c \) there exists a read action for every write action, and vice versa. Therefore, the parallel composition \( \parallel_{c \in C} P^s_c \) can perform exactly action \( u \) and the resulting process is \( \parallel_{c \in C} P'^s_c \).

And hence \( \bar{P}^s \xrightarrow{u} \bar{P}'^s \).

\[ \text{Theorem 2 (synchronous equivalence).} \]

Given a network \( \mathcal{N} = (C, X) \), states \( s, s' \in S_N \) and input/output function \( M \). There exists an action \( u \), with \( u \) obeying \( M \), such that \( P^s \xrightarrow{u} P'^s \) iff there exists a set of transfer islands \( T \) such that

• \( T \) is maximally-enabled in \( s \) and

• \( s' \) is an \( X' \)-successor of \( s \),

where \( X' \) is the union of all islands in \( T \).

\[ \text{Proof.} \ (\Rightarrow) \ \text{Take an action } u \text{ such that } P^s \xrightarrow{u} P'^s. \text{ By the structure of } P^s, \text{ we also have } \bar{P}^s \xrightarrow{u} \bar{P}'^s. \text{ Let } T \text{ be the set of transfer islands as in Theorem 1. We already have that } T \text{ is set-enabled and that } s' \text{ is an } X'\text{-successor of } s. \]

Suppose a set of transfer islands \( T' \supseteq T \), with \( T' \) and \( T \) identical with respect to channels attached to sources and sinks and \( T' \) is maximally-enabled. By Theorem 1 there exists an action \( u' \) such that \( P^s \xrightarrow{u'} P'^s \), for some state \( s' \). Since \( T' \) and \( T \) are identical with respect to channels attached to sources and sinks, it follows that \( u \) and \( u' \) are identical with respect to non-communicating steps and \( u' \supseteq u \). By the priority operator, there is no \( u' \supseteq u \), so \( u' = u \). Hence we have \( T' = T \) and \( T \) is maximally-enabled in \( s \).

\[ (\Leftarrow) \ \text{Take a set of transfer islands } T \text{ such that } T \text{ is maximally-enabled in } s \text{ and } s' \text{ is an } X'\text{-successor of } s. \text{ Let } u \text{ be defined as in the proof of Theorem 1. As before, } u \text{ obeys } M \text{ and we have } \bar{P}^s \xrightarrow{u} \bar{P}'^s. \]

Suppose an action \( u' \supseteq u \), with \( u' \) and \( u \) differing only in communication actions and \( P^s \xrightarrow{u'} P'^s \). By Theorem 1 there exists a set of transfer islands \( T' \) such that \( T' \) set-enabled in \( s \). Since \( u \) and \( u' \) are identical with respect to non-communicating steps, it follows that \( T' \) and \( T \) are identical with respect to channels attached to sources and sinks and \( T' \supseteq T \). By the definition of maximally-enabled, there is no \( T' \supset T \), so \( T' = T \). Hence we have \( u' = u \) and \( P^s \xrightarrow{u} P'^s \). \]
The following two Lemmas are used in the proof of Theorem 1.

**Lemma 1** (Enabled implies closed). Given network $\mathcal{N} = (C, X)$, state $s \in S_{\mathcal{N}}$ and a set of channels $Y \subseteq X$. If there exists an action $u$, such that $P^s$ can do a $u$ and for each $x \in X$ there exists an $x?d \in u$ iff $x \in Y$, then $Y$ is combinatorial-closed.

**Proof.** Take an action $u$ such that for each $x \in Y$ there exists an $x?d \in u$. Let $c$ be any component. If $c$ is connected to a channel $x$ in $Y$, then there is an action $x?d \in u$. By the structure of $P^s$, we have that $P^s_c$ does an $x!d$ action or an $x?d$ action. By the primitive process definition for $c$ it follows that for each channel $x'$ required by the combinatorial-closed conditions we have that $P^s_c$ also performs a read or write action on $x'$. By the structure of $P^s$, there must be a transfer $x'?d' \in u$, for some $d' \in D$. This implies that $x' \in Y$ and the combinatorial-closed condition holds. Hence, the closed conditions hold for all $c \in C$.

**Lemma 2** (Island partition). Given a set of channels $Y \subseteq X$ and $Y \neq \emptyset$. If $Y$ is combinatorial-closed, then there exists a partition into transfer islands.

**Proof.** Let $(Y, E)$ be an undirected graph and let $E$ be defined as follows: for each $x, x' \in Y$ we have $\{x, x'\} \in E$ iff there is a component $c$ such that $x$ and $x'$ are connected to $c$ and $c$ is not a queue. Take the set of connected components of $(Y, E)$. This is a partition of $Y$. A connected component, say $I$, is combinatorial-closed since $Y$ is combinatorial-closed and for each component $c$, those channels in $Y$ connected to $c$ are in the same connected component. Furthermore, $I$ is the smallest such set. Hence, each connected component is a transfer island and the set of connected components of $(Y, E)$ is a partition into transfer islands.

### 3.4. Island Extraction

Consider a given network $\mathcal{N} = (C, X)$. Algorithm 1 processes all components in $C$ in an order such that, for each channel $x \in X$, the initiator of $x$ is processed before the target of $x$. One such order is as follows: process all sources and queues, then repeatedly process components of which all input channel initiators were already processed. We call this transfer order.

**Example 6.** For the network in Figure 1.2, transfer order is:

$$c_0, c_1, q_0, q_1, c_2, c_3, c_4, q_0, q_1, c_5, c_6, c_7$$

Let $B_x$ be the set of transfer islands containing channel $x$. Algorithm 1 computes a set of transfer islands for network $\mathcal{N} = (C, X)$. The algorithm directly follows the definition of a transfer island, that is, a set of minimally combinatorial-closed channels (Definition 4). We illustrate the algorithm using the simple network in Figure 1.2.

For each source or queue, a new island is created containing the output channel of the source. When the component is a fork, the two output channels are added to all islands that contain the input of the fork component.
Example. Running these steps on the example in Figure 1.2, we obtain the following islands:

\{x_0, x_1, x_2\}, \{x_3\}, \{x_6\}, \{x_5\}

When the component is a merge, the output channel is added to all islands that contain at least one input of the merge.

Example. Next, we process component \(c_3\), a merge. The set of transfer islands becomes:

\{x_0, x_1, x_2, x_4\}, \{x_3, x_4\}, \{x_6\}, \{x_5\}

When the component is a switch, we duplicate each island containing the input channel. To the first set of islands we add the first output channel and to the new set of islands we add the second output.

Example. Next, we arrive at component \(c_4\), a switch. The set of transfer islands becomes:

\{x_0, x_1, x_2, x_4\}, \{x_3, x_4\}, \{x_6\}, \{x_5, x_7\}, \{x_5, x_8\}

When the component is a queue that we have processed before, nothing happens and the transfer island set does not change.

Example. We process queues \(q_0\) and \(q_1\) for the second time and the set of transfer islands does not change.

When the component is a join, we merge any pair of islands \(I, I'\), where \(I\) contains the first input channel and \(I'\) the second input channel, into one island \(I \cup I'\) together with the output of the join.

Example. We arrive at component \(c_5\), a join. The set of transfer islands becomes:

\{x_0, x_1, x_2, x_4\}, \{x_3, x_4\}, \{x_5, x_6, x_7, x_9\}, \{x_5, x_8\}

Example. After processing merge \(c_6\) and sink \(c_7\), the final set of transfer islands is:

\{x_0, x_1, x_2, x_4\}, \{x_3, x_4\}, \{x_5, x_6, x_7, x_9, x_{10}\}, \{x_5, x_8, x_{10}\}

The correctness follows from the definition of combinatorial-closed. The transfer islands are minimal, since we only add channels when required by combinatorial-closedness.

The running time of the algorithm is linear in the number of components and quadratic in the number of resulting transfer islands. The number of transfer islands is exponential in the number of switches. For each switch that we encounter, the total number of transfer islands potentially doubles.
for $c \in C$ in transfer order do
  if $c$ is a source then
    Create transfer island $\{c.o\}$.  
  end
  else if $c$ is a fork then
    for $I \in B_{c.i}$ do
      Add $c.a$ and $c.b$ to $I$.  
    end
  end
  else if $c$ is a merge then
    for $I \in B_{c.a} \cup B_{c.b}$ do
      Add $c.o$ to $I$.  
    end
  end
  else if $c$ is a queue and visited for the first time then
    Create transfer island $\{c.o\}$.  
  end
  else if $c$ is switch then
    for $I \in B_{c.i}$ do
      Let $I'$ be a copy of $I$.  
      Add $c.a$ to $I$ and $c.b$ to $I'$.  
    end
  end
  else if $c$ is a join then
    Let $X = B_{c.a}$ and $X' = B_{c.b}$.  
    for $I \in X$ do
      for $I' \in X'$ do
        Let new transfer island $I'' = I \cup I' \cup \{c.o\}$.  
      end
    end
    Remove all islands in $X$ and $X'$.  
  end
  else
    Skip.  
end

Algorithm 1: Computation of transfer islands.
3.5. State Machine Extension

Verbeek et al. [10] extend xMAS with a primitive for I/O automata to better compute invariants. Finite state machines can be simulated in pure xMAS, but this requires an unrestricted join which invariant generation techniques cannot handle.

In this section, we similarly extend our definitions from earlier sections. The next section provides a formalization of I/O automata, which is used in Section 3.5.2 and Section 3.5.3 to extend process algebra and transfer island semantics respectively.

3.5.1. I/O Automata

A deterministic\(^1\) I/O automata is described by the tuple \((I, O, S, s_0, \delta, \lambda)\), where

- \(I\) is the input alphabet;
- \(O\) is the output alphabet;
- \(S\) is a finite set of states;
- \(s_0 \in S\) is the initial state;
- \(\delta : 2^I \times S \rightarrow S\) is a transition function;
- \(\lambda : 2^I \times S \rightarrow 2^O\) is an output function.

We require that \(\lambda(u, s)\) is defined iff \(\delta(u, s)\) is defined. We write \(\epsilon(u, s)\) when \(\lambda(u, s)\) and \(\delta(u, s)\) are defined. If \(\epsilon(u, s)\) is false then no transition from state \(s\) with input \(u\) is possible.

**Example 7.** Consider the I/O automaton in Figure 3.2, which implements the behavior of an xMAS merge without an arbitration policy. Formally, this automaton

\(^1\) An extension to non-deterministic I/O automata remains as future work.
is described by the tuple \((I, O, S, s_0, \delta, \lambda)\), where

\[
I = \{a?d \mid d \in D\} \cup \{b?d \mid d \in D\}
\]
\[
O = \{old \mid d \in D\}
\]
\[
S = \{s\}
\]
\[
s_0 = s
\]
\[
\delta = \{(a?d, s) \mapsto s \mid d \in D\} \cup \{(b?d, s) \mapsto s \mid d \in D\}
\]
\[
\lambda = \{(a?d, s) \mapsto old \mid d \in D\} \cup \{(b?d, s) \mapsto old \mid d \in D\}.
\]

We use process algebra notation to write sets of reads and writes, omitting \(
\{
\}).

**Notation 14.** We write \(X_{IO}\) for the channels that are connected to the I/O automaton \((I, O, S, s_0, \delta, \lambda)\). That is,

\[
X_{IO} = \{x \mid \exists d \in D \ x?d \in I \lor x!d \in O\}.
\]

Similarly, we write \(X_I\) and \(X_O\) for the input respectively output channels.

### 3.5.2. In Process Algebra

An I/O automaton as defined in the previous section can be transformed into a process in our process algebra. Given automaton \((I, O, S, s_0, \delta, \lambda)\) we get the following process.

For each state \(s \in S\), we introduce an equation \([s] = T_s\) where \([s]\) is a unique name.

The process algebra term describing the behavior of \([s]\) is defined as

\[
T_s = \sum_{u \subseteq I \land \epsilon(u, s)} u[\lambda(u, s) \cdot [\delta(u, s)]],
\]

and \([s_0]\) is the initial term.

**Example 8.** We translate the I/O automaton from Example 7 to a process. Let \(P = [s]\), then we get the term

\[
P = \sum_{u \subseteq I \land \epsilon(u, s)} u[\lambda(u, s) \cdot [\delta(u, s)]]
\]
\[
= \sum_{d \in D} a?d[old \cdot [s]] + \sum_{d \in D} b?d[old \cdot [s]],
\]

which simplifies to

\[
P = \sum_{d \in D} (a?d[old \cdot P] + b?d[old \cdot P]).
\]

Note the similarity with Equation 2.1.
3.5.3. In Transfer Islands

We extend transfer islands to account for the I/O automaton primitives. As in Section 3.1, we define transfer islands by providing combinatorial-closed conditions that a set of channels has to satisfy to be a transfer island.

We introduce the concept of IO-combined actions, which aggregate input actions with the corresponding output actions.

**Definition 15** (IO-combined actions). Given an automaton \((I, O, S, s_0, \delta, \lambda)\), let \(Z_{IO}\) be the set of IO-combined actions. We define \(Z_{IO}\) as the smallest set satisfying for each \(s \in S\) and \(u \subseteq I\) — for which we have \(\epsilon(u, s)\) — that \(u|\lambda(u, s) \in Z_{IO}\).

**Example 9.** Consider the automaton from Example 7. We have

\[ Z_{IO} = \{a?d|old \mid d \in D\} \cup \{b?d|old \mid d \in D\} \].

We introduce some notation for the channels that transfer in an action.

**Notation 16.** Given an action \(u \in A\), let \(X_u\) be the channels that transfer in \(u\). That is

\[ X_u = \{x \mid \exists d \in D x?d \in u \lor x!d \in u\} \].

We introduce the concept of weakly-connected channels. A set of channels is weakly-connected if there is an IO-combined action which fires exactly those channels.

**Definition 17** (weakly-connected channels). Given an automaton \((I, O, S, s_0, \delta, \lambda)\). A set of channels \(Y \subseteq X_{IO}\) is weakly-connected iff there exists an IO-combined action \(u \in Z_{IO}\) with \(X_u = Y\).

**Example 10.** Consider the automaton from Example 7. We have the sets of weakly-connected channels \({a, o}\) and \({b, o}\).

Using the concept of weakly-connected channels, we can define the combinatorial-closed condition for automata.

**Definition 18** (combinatorial-closed condition for automata). Given a set of channels \(Y\) and a component \(c\) instantiated from automaton \((I, O, S, s_0, \delta, \lambda)\).

The combinatorial-closed condition for \(P\) holds iff \(Y\) is not connected to \(c\) or there is a set of weakly-connected channels \(X'\) such that

\[ Y \cap X_{IO} = X' \].

In other words, for each I/O automaton connected to \(Y\), exactly one weakly-connected set of channels must be in \(Y\).

**Example 11.** Consider the automaton from Example 7. Note that \(X_{IO} = \{a, b, o\}\). A set of channels \(Y\) is combinatorial-closed only if \(Y \cap X_{IO} = \emptyset\), \(Y \cap X_{IO} = \{a, o\}\) or \(Y \cap X_{IO} = \{b, o\}\). Note that this is equivalent to: if \(Y\) is connected to the automaton, then \(o \in Y\) and \(a \in Y\) xor \(b \in Y\). It is no coincidence that this is the closed-condition for the xMAS merge from Definition 4.
Finally, we adjust the definition of transfer islands from Chapter 3.

**Definition 19** (transfer island with automata). *Given a network* \((C, X)\). A set of channels \(Y\) is a transfer island iff the channels in \(Y\) satisfy the combinatorial-closed conditions for all \(c \in C\) (Definition 4 and 18).

### 3.5.4. Transition System

To define the semantics of the transfer islands, we adapt the definitions of a finite transition system from Section 3.1.

Recall that the state space is a mapping from components to their individual states. Queues were the only component contributing to the network state space. We now add I/O automata. The individual state space of an I/O automaton \((I, O, S, s_0, \delta, \lambda)\) is, obviously, \(S\). The network state is still a function from components to their individual states.

In Section 3.1, we defined enabledness and next-states based on process algebra. With the translation of I/O automata to process algebra from the previous section Definition 8 and 11 still capture the concepts.

As before, it is needed to derive enabledness and next-states separate from the Process Algebra. The two ancillary definitions of data propagation (Definition 12) and conditionals (Definition 13) are extended to account for I/O automata.

**Definition 20.** Given an I/O automaton \((I, O, S, s_0, \delta, \lambda)\), a network state \(s\) and a set of channels \(Y\), we define data propagation \(\pi\).

Let \(P^s \in S\) denote the state of the automaton in \(s\). Let \(u \subseteq I\) be the input action enabled in \(s\). That is,

\[
    u = \{x?\pi_x(s, M, Y) \mid x \in X_I \land \pi_x(s, M, Y) \text{ is defined}\}
\]

We have for all \(x \in X_O\), if \(x!d \in \lambda(u, P^s)\) for some \(d \in D\), then \(\pi_x(s, M, Y) = d\).

**Definition 21.** Given an I/O automaton \((I, O, S, s_0, \delta, \lambda)\), a network state \(s\) and a set of channels \(Y\), we define conditional-enabledness of \(Y\).

Let \(P^s\) and \(u\) as in Definition 20. Then \(Y\) is enabled in \(s\) iff we have \(u \subseteq I\) and \(\epsilon(u, P^s)\) holds.

Using these definitions, the proof from Section 3.3 can be extended to include networks containing I/O automata. Similarly, Algorithm 1 can be extended to account for automata using the combinatorial-closed conditions of Definition 18.
4. Coding in xMAS: CodexMAS

Before CodexMAS, there were two ways to define xMAS networks. One could either write C++ code, or use a graphical tool. To quickly build tests for our transfer island and nuXmv algorithms, neither option worked very well. The C++ code was time-consuming to write and prone to errors. The graphical tool makes sense for non-programmers; it is very easy to see what one is doing. However, a coder is used to working in text and is very productive in doing so. A coder has a fast and trusty text editor, as well as a smart compiler which immediately notifies of any silly mistakes. CodexMAS is the coder’s version of xMAS.

In this chapter CodexMAS is introduced by way of a tutorial. A grammar is included in Appendix B, while the development of formal semantics and a full implementation remains as future work.

A taste of CodexMAS

The basic idea of the language is to define the structure of a network by defining xMAS channels and how they connect to components. The following statement connects a queue with a capacity of 2 to channels a and b. Channel a is the input of the queue, while b is the output. The channels must be declared (see line 1) before they can be used.

1 chan a, b
2 let b := Queue(2, a)

Next, we’d like to inject a message into channel a. We use the source primitive, with a specification of what kind of messages can be injected through this source.

3 let a := Source(type in {req} && x in [0..10])

This defines a source feeding into a. The messages have two fields, namely type and x. type is an enumeration field with one possible value, namely req. x is an interval field which can take any value between zero (inclusive) and ten (exclusive).

Finally, we want to close the loop by feeding the output of the queue, namely channel b, into a sink.

4 Sink(b)
Note that the sink has no output channel, so we do not have to assign it to a channel. In fact, it would be illegal to do so.

This admittedly simple network is now finished. Note that each channel is assigned and consumed exactly once. This corresponds to the fact that each channel in xMAS has exactly one initiator and one target. A channel lacking an initiator or target would result in an invalid network, so this is forbidden (and enforced) by the language tools.

In the next example, we introduce the remaining xMAS primitives. This network corresponds almost entirely with that in Figure 1.2. The channels are named accordingly.

1. `chan x0 := Source(type in {req, rsp})`

First of all note that we can declare a channel and assign to it in one statement. This statement declares channel `x0` and connects it to a source. The source makes a non-deterministic choice between injecting either a `req`-typed message, or a `rsp`-typed message, or nothing at all, just like an xMAS source.

2. `chan x1, x2 := Fork(x0)`
3. `chan x6 := Queue(2, x1)`

Channels `x1` and `x2` are declared and assigned as the two outputs of a fork. The input of the fork is the previous `x0`. In line 3, `x1` is connected to a queue.

4. `chan x3 := Source(type in {req})`
5. `chan x5 := Queue(2, Merge(x2, x3))`

Now we define channel `x3` to a source injecting `req`-typed messages. Channel `x5` becomes the output of a queue, whose input is an unnamed channel. This channel is implicitly created between the queue and its argument, a merge taking channels `x2` and `x3` as input. This construct guarantees that this unnamed channel has exactly one initiator and target. This channel is therefore always valid.

6. `chan x7, x8 := Switch(type in {rsp}, x5)`

Here we use a switch to send all `rsp`-typed messages from `x5` to `x7`, while all other messages go to `x8`.

7. `chan x8prime := Function(type := type with {_, rsp}, x8)`

We modify the network in Figure 1.2 slightly to demonstrate the function primitive. The function primitive takes a message transformation function and an input channel as arguments. This transformation turns all messages into `rsp`-typed messages and sends them on to channel `x8'`.

8. `chan x9 := CtrlJoin(x7, x6)`
9. `chan x10 := Merge(x8prime, x9)`
To wrap the network up neatly, a control join and a merge feed messages into the sink attached to channel $x_{10}$.

The next step is to combine standard xMAS components into larger, user-defined components. As a simple example, we define a component that simulates a delay in the network.

```plaintext
component Delay(chan i) => chan o {
    let o := CtrlJoin(i, Source(type in {token}))
}
```

The keyword `component` signals that we will define a new component. `Delay` is the name of the component, which has one input channel (as a parameter) named `i` and an output channel (as a return parameter) named `o`. In the body of the component (enclosed in curly brackets) these channels are treated as if they were already declared. To make any component valid, all input channels must be used exactly once, while all output channels must be assigned once. This is precisely what we do inside the body, using a control join to forward input from channel `i` if and only if the source of token-type produces a message.

User-defined components can have any number of input or output channels, separated by commas. It is even possible to take the capacity of a queue as an argument, using `int` as the type.

Sources and switches take conditions as arguments. The signature for these two components is as follows:

```plaintext
component Source(cond m(msg)) => chan o
component Switch(cond s(msg), chan i) => chan a, chan b
```

For the source this means that any message for which `m` evaluates to `true` can be injected by that source. The switch will send a message to channel `a` if `s` evaluates to `true` and to channel `b` if it evaluates to `false`.

A function takes a message function, as well as an input channel.

```plaintext
component Function(msg f(msg), chan i) => chan o
```

A join primitive takes a message function over two input messages.

```plaintext
component Join(msg h(msg, msg), chan a, chan b) => chan o
```

CODEXMAS has two more features to make designing larger systems easier. The first feature is to bundle a number of channels into a bus. For example, given two channels `a` and `b`, we define a bus `B` as follows:

```plaintext
bus<2> B := (a, b)
chan c, d
```
let (c, d) := B

Enclosing channels in parentheses casts them into a bus, allowing assignments. A second method of creating and using buses is through indexing. The following example has the same effect as the previous one:

bus<2> B
let B[0] := a
let B[1] := b
chan c := B[0]
chan d := B[1]

Buses are useful as parameters to components, bundling channels with a similar purpose.

To define networks with a repetitive structure, we introduce recursive components. Recursion is tied closely to buses: it is only allowed on the structure of a bus. The following example defines a component that merges an arbitrary number of inputs into a single output channel.

component MultiMerge([chan head]) => chan o {
  let o := head
}
component MultiMerge([chan head, tail]) => chan o {
  let o := Merge(head, MultiMerge(tail))
}
bus<4> B
chan o := MultiMerge(B)

When MultiMerge is called with a channel or bus with size 1, then the first definition is used which simply passes the input through as output. If the argument matches as a head channel and tail bus, then the second definition is used. This combines the head channel with the merged tail using a regular Merge primitive.

Messages in CodexMAS have a simple structure. A message consists of a number of named fields. Each field can either be an enumeration value or an integer in a certain interval. Channels are not explicitly typed, but their types are inferred.

Source primitives are the only source of new messages (ignoring i/O automata for now). The messages that can be injected by a source are defined by a condition.

\[
\text{cond } m(\text{msg}) = \text{type in \{req, rsp\} and source in [0..10] and destination } \rightarrow = 9
\]

The message condition \( m \) allows messages with exactly three fields: a type, a source and a destination.

We can also combine message conditions into more complex conditions.
cond n(msg) = type in {req}
cond n_or_m(msg a) = m(a) or n(a)

These conditions are used for both sources and switches in regular xMAS.

Primitives like the function and join allow the computation of new messages based on other messages. We use message functions for this.

msg f(msg a) = x := a.x + 1
msg h(msg a, msg b) = y := a.x; type := b.type with {req : rsp, _ : err}

Function $f$ results in a message with field $x$ with the value of $x$ from message $a$ incremented by one. Function $h$ places the value of $x$ from message $a$ in field $y$. It also takes the type of message $b$, changes a request to a response and any other value into an error.

The extension with I/O automata allows for a holistic view of the network and the protocols running on top of the network. Just like components, processes are connected to the network through input and output channels.

process P(chan i) => chan o {
  state S
      trans (type in {req}) i? | o!(type in {rsp}) -> T
  state T
      trans i? -> S
      trans o!(type in {rsp}) -> S
}

Process $P$ encodes an I/O automata with one input channel and one output channel. It has two states, namely $S$ and $T$, of which the first, $S$, is the initial state. State $S$ has a transition to state $T$, receiving a message on $i$ and simultaneously sending a message on $o$. The output message has a single field type, which can only have the value $rsp$. The condition in parentheses restricts the input messages to requests. State $T$ has two transitions, both back to $S$. The first accepts a message from $i$. The second transition (re)sends the response to $o$.

Sometimes two or more transitions are possible at the same instant, based on the channel readiness. In this case, a non-deterministic choice is made.\(^1\)

An interesting feature of processes is the ability to parameterize states. To define a state that captures (parts of) messages we write:

state S(msg m)

and in the definitions of transitions from $S$ we can use $m$ like an input message.

state S(msg m)

\(^1\)The syntax allows for non-deterministic automata, an extension mentioned as future work in Section 3.5
\texttt{trans \rightarrow S(x := m.x + 1)}

This example shows how we use a message function to change the value stored in state $S$. Additionally, this shows how to define an internal transition; a transition that does not send or receive any messages. A condition in parentheses is also allowed here.

We do not allow processes to be defined recursively. It is, however, possible to combine multiple processes using (recursive) components.

The full implementation, as well as a treatment of the formal semantics, are future work. The examples as found in the next chapter can, and are, specified in the CodexMAS language.
5. Experimental Work

We implemented Algorithm 1 and the translation to NuXmv in C++. This allows us to use the internal data structures used in the WickedXmas tool set [6]. For the convenience of the reader, we put the resulting NuXmv models on-line\(^1\). The method to transform a collection of transfer islands into a NuXmv model is illustrated by example in Appendix A. Before computing reachability, we infer all channel types using the technique proposed by Gastel et al. [11]. Our experiments were performed on an iMac Intel Core i3 540 3.07 GHz running Fedora 22, Linux kernel 4.0.4, with 4MB cache and 4GB main memory. We used NuXmv version 1.0.1. To evaluate the performance of our reachability analysis, we used several small examples and a larger one to show some scalability. We first describe all examples, then give tables with execution times of NuXmv.

5.1. Small Examples

Our examples are taken from Verbeek’s thesis [18]. These networks are special cases where his deadlock hunting tool finds false deadlocks, that is, deadlock states that are not reachable from the initial state. We show that our technique can make his analysis complete by automatically proving these states unreachable.

**Asynchronous deadlock** Consider the example in Figure 5.1. The source injects exactly one message into the network. The sink is dead, that is, it never accepts messages. The shown configuration is a deadlock. The message in \(q_4\) is blocked by the dead sink. The switches are priority switches: they will always forward a message to an enabled channel. The message injected by the source is split over \(q_0\) and \(q_1\). It then moves in lock-step to \(q_2\) and \(q_3\). For a message to end up in \(q_4\), both \(q_1\) and \(q_2\) need to contain a message, but this is never the case. With synchronous semantics, our reachability checker shows that this configuration is indeed not reachable. In asynchronous semantics, the deadlock is reachable and our tool finds it. Even if the sink is dead, this network has no deadlock.

**Rationals** Figure 5.2 shows a network where exactly one message is inserted. Assuming a fair arbiter, this message has a 50\% chance of ending up in either queue, after which it will never progress. To create a deadlock, assume the sink to be dead.

\(^1\)http://www.win.tue.nl/~jschmalt/publications/memo15/memo15.html
Verbeek’s tool finds a deadlock with a message blocked in $q_1$. The two half messages are combined into a half message in $q_0$, two half messages in the next two queues and finally (combining the probabilities) one message in $q_1$. Our tool correctly shows that this state is unreachable.

**Red and blue** Figure 5.3 shows a network where the source injects either blue or red messages. The fork ensures that colours of messages are always in the same order in both queues. The invariants generated on this example are not strong enough to rule out this configuration, as they do not state anything about the order of messages. They only state that the number of blue or red messages is equal in both queues. Our tool shows indeed that this state is unreachable.

### 5.2. 2-agents Example

Figure 5.4 shows a network consisting of two agents communicating through a fabric using virtual channels. Credit counters make sure that no more messages are in transit than the opposing agent can buffer. When the credit counters are too large, the network can deadlock. The deadlock configuration is shown in the figure: all
buffers are full, as well as the main transfer buffer in the fabric. We explore a number of versions of this network. The queues in the fabric are always size 2; queues that buffer messages (including buffers for the credit counters) are of size $N$; the credit counters’ queues are of size $M$. A version of the 2-agents network is then denoted as “twoagents $N/M$”.

### 5.3. Execution Times

Table 5.1 shows the runtimes in seconds for running reachability questions using our synchronous semantics of transfer islands. Designs with reachable deadlocks are marked with an asterix (*). The first two columns compare the BDD engine with IC3. In the second column, inductive invariants are added to the BDD or IC3 engines. Without invariants, the BDD engine is clearly faster than IC3. Invariants improve performance for both engines. IC3 benefits so much from the invariants, that it becomes overall faster than BDDs.

Note that the generation of the NuXmv models, including the runtime of Algorithm 1, takes less than 0.01 seconds on the experimenting machine.

To further experiment, we defined an *asynchronous* version of our transfer islands. We use the interleaving semantics to model check this version. Table 5.2 shows the runtimes in seconds for the asynchronous semantics. As expected, the running times are larger because of the necessary interleaving of the possible choices between enabled transfer islands. In contrast to the synchronous case, without invariant IC3 is already overall faster than BDDs. The difference is even more visible when invariants are added.
### Table 5.1.: Results for the synchronous semantics.

<table>
<thead>
<tr>
<th>design</th>
<th>BDD</th>
<th>IC3</th>
<th>BDD + invariants</th>
<th>IC3 + invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>asynchronous</td>
<td>0.06</td>
<td>0.07</td>
<td>0.05</td>
<td>0.13</td>
</tr>
<tr>
<td>rationals</td>
<td>0.02</td>
<td>0.06</td>
<td>0.03</td>
<td>0.06</td>
</tr>
<tr>
<td>blue red</td>
<td>0.03</td>
<td>0.04</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>twoagents 2/2</td>
<td>0.21</td>
<td>10.34</td>
<td>0.20</td>
<td>0.45</td>
</tr>
<tr>
<td>twoagents 2/3*</td>
<td>0.85</td>
<td>12.85</td>
<td>0.92</td>
<td>1.31</td>
</tr>
<tr>
<td>twoagents 4/4</td>
<td>4.11</td>
<td>253.55</td>
<td>3.93</td>
<td>0.87</td>
</tr>
<tr>
<td>twoagents 4/5*</td>
<td>41.13</td>
<td>80.98</td>
<td>20.30</td>
<td>3.21</td>
</tr>
</tbody>
</table>

### Table 5.2.: Results for the asynchronous semantics.

<table>
<thead>
<tr>
<th>design</th>
<th>BDD</th>
<th>IC3</th>
<th>BDD + invariants</th>
<th>IC3 + invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>asynchronous *</td>
<td>0.03</td>
<td>0.28</td>
<td>0.32</td>
<td>0.22</td>
</tr>
<tr>
<td>rationals</td>
<td>0.02</td>
<td>0.05</td>
<td>0.02</td>
<td>0.07</td>
</tr>
<tr>
<td>blue red</td>
<td>0.02</td>
<td>0.05</td>
<td>0.02</td>
<td>0.06</td>
</tr>
<tr>
<td>twoagents 2/2</td>
<td>5.58</td>
<td>7.27</td>
<td>5.15</td>
<td>1.40</td>
</tr>
<tr>
<td>twoagents 2/3*</td>
<td>53.49</td>
<td>119.55</td>
<td>68.03</td>
<td>6.48</td>
</tr>
<tr>
<td>twoagents 4/4</td>
<td>413.70</td>
<td>50.67</td>
<td>754.31</td>
<td>21.43</td>
</tr>
<tr>
<td>twoagents 4/5*</td>
<td>1387.4</td>
<td>108.52</td>
<td>2765.40</td>
<td>46.56</td>
</tr>
</tbody>
</table>
This thesis developed a model checking solution to the problem of state reachability in communication fabrics. Chapter 2 introduced process algebra semantics for communication fabrics. Transfer islands were presented in Chapter 3 together with a proof of equivalence to the process algebra semantics. We discussed differences in semantics in Section 3.2 and extended our method with I/O automata in Section 3.5. Chapter 5 showed experimental work solving reachability in small and larger examples, comparing the BDD and IC3 engines in nuXmv. Based on the results we can conclude that IC3 in combination with pre-generated invariants performs best. It seems to be the case that these techniques are feasible even in quite large networks. Chapter 4 introduced a domain-specific language for designing complex communication fabrics.

To end the conclusion we discuss some limitations of our method followed by a discussion of future work.

6.1. Limitations

The process algebra semantics in Chapter 2 does not exactly model xMAS. In this section we will explore exactly how, and why, it differs.

The acronym xMAS stands for executable micro-architectural specification. Executable in this instance means that an xMAS network can easily be translated to a hardware implementation. An important property of this implementation is that all (physical) wires must have determinate values at the moment of inspection: the clock tick. One consequence is that we must disallow combinatorial cycles, that is, circular dependencies of wires. Another, more subtle, consequence is the requirement of persistency. Once an irdy wire goes up, both irdy and its associated data bus must not change until a transfer has occurred. Recall that a transfer occurs when both irdy and trdy are high at the clock tick. Similarly, a trdy wire must not go down until after a transfer has occurred.

The solution of xMAS to the problem of persistency is to make merges and sources stateful components. In xMAS, suppose a merge sees that a message $m$ is ready on input $a$: the irdy wire is high. It will signal on its output channel, by setting irdy high, that it is ready to send message $m$. The merge is now in an intermediate state where it cannot change the data output, even if a message arrives on $b$.

The merge in our process algebra model is not persistent. While an xMAS merge persistently chooses an input channel to pass to the output, our model could change to a different input channel later. This results in an over-approximation of the state
space using our model. Hence, we may identify a state as reachable when it is not reachable in the XMAS network. Therefore, our approach still suffers from false deadlocks albeit in a subset of cases compared to earlier methods.

6.2. Future Work

We propose a number of areas for future work based on this thesis.

First, the limitations related to XMAS semantics from the previous section could be solved by extending the process algebra. This undoubtedly will affect the transfer island abstraction. With a semantics that is both sound and complete with respect to XMAS, it will be possible to check temporal properties beyond reachability using model checkers such as nuXmv.

Second, the extension with I/O automata could be further enhanced to handle non-deterministic automata. If the issues related to persistency can be solved in the context automata, then it should be possible to simulate XMAS exactly using automata.

Third, the experimental work in this thesis does not fully investigate the practical limits of reachability checking. It would be very interesting to apply the techniques developed here to large and complex communication fabrics and protocols from industry. A comparative study between the nuXmv translation from this thesis and industrial tools for the verification of hardware descriptions such as Verilog would better showcase any improvements over existing techniques. Preliminary experiments in this direction show promising results.

Fourth, the development of CodexMAS is still in its infancy. Discussions and feedback from experts in the field will be required to develop the language in the right direction. A number of features described in Chapter 4 have not yet been implemented.
A. nuXmv Translation

Here we present a translation from transfer islands to nuXmv, specifically the synchronous variant. A translation of the interleaving variant exists, but it is omitted here.

Queues are modeled by two variables: an array modeling the contents of the queue and a counter representing the number of messages stored in the queue. The counter is needed as arrays in nuXmv are of fixed length. Unused queue slots are indicated using a special ‘none’ value. All queues are initialized with a counter set at 0 and all their positions filled with a ‘none’ value.

**Example 12.** Considering the example in Figure 1.2, queue $q_0$ is declared as follows:

```nuXmv
VAR
  Queue_0: array 0..1 of {rsp, req, none};
  Queue_0_n: 0..2;

and it is initialised with a counter at 0 and ‘none’ values

init(Queue_0_n) := 0;
init(Queue_0[0]) := none;
init(Queue_0[1]) := none;
```

We use input variables to model messages produce by sources. In the case a source can produce several messages, a choice is made non-deterministically. If there is no choice, nuXmv automatically reduces the input variable to a constant. Sources, sinks and merges require an oracle to decide whether to send or receive a message, or to determine which input channel should be served. We use Boolean input variables to represent these non-deterministic choices.

**Example 13.** The following nuXmv declaration model the input variables and oracles of the example in Figure 1.2.

```nuXmv
IVAR Source_0: {req};
IVAR Source_1: {rsp, req};
IVAR Source_0_oracle: boolean;
IVAR Source_1_oracle: boolean;
IVAR Sink_0_oracle: boolean;
IVAR Sink_1_oracle: boolean;
IVAR Merge_0_oracle: boolean;
```

State updates are modeled using nuXmv ‘next’ statements. These statements allow us to represent the conditions—given by Corollary 2—under which counters
and queues are updated. A transition consists of the execution of all next statements for which the condition holds. The key part of our translation is to generate these conditions for counters and queue places.

Regarding queue counters, we consider three cases: 1) the queue is empty, so we can only add a message to the queue; 2) the queue is full, so we can only remove a message from the queue; 3) the queue is neither full nor empty, hence we can add or remove a message at the same time. The counter is then increased, decreased, or left unchanged. Decisions to increase or decrease are represented using a function converting a transfer condition to 1 if the condition is true, or to 0 otherwise. We then add incoming transfers and subtract outgoing transfers. For each counter, we generate a ‘next’ statement with these three cases. The transfer conditions are extracted from the transfer islands.

**Example 14.** Here is the next statement for the counter of queue $q_0$ in Figure 1.2. Due to the information captured in transfer islands we can state a next statement of queue $q_0$ even if it depends on the state of the other queue $q_1$.

$$\text{next}(\text{Queue}_0_\text{n}) :=$$

$$\begin{cases} & \text{Queue}_0_\text{n} = 0 \& \text{Queue}_1_\text{n} \neq 2 \& \text{oracle}_\text{Source}_1 \& \text{oracle}_\text{Merge}_0: 1; \\
& \text{Queue}_0_\text{n} = 2 \& \text{Queue}_1_\text{n} \neq 0 \& \text{Queue}_0_\text{n} \neq 2 \& \text{Queue}_1[1] = \text{rsp} \& \\
& \text{oracle}_\text{Sink}_1: 1; \\
& \text{TRUE: Queue}_0_\text{n} + \\
& \quad \text{toint}(\text{Queue}_0_\text{n} \neq 2 \& \text{Queue}_1_\text{n} \neq 0 \& \text{oracle}_\text{Source}_1 \& \text{oracle}_\text{Merge}_0) - \\
& \quad \text{toint}(\text{Queue}_1_\text{n} \neq 0 \& \text{Queue}_0_\text{n} \neq 0 \& \text{Queue}_1[1] = \text{rsp} \& \text{oracle}_\text{Sink}_1); \\
\end{cases}$$

Messages are always taken from the end of the queue (Queue$_0[1]$) and pushed to the tail (Queue$_0[0]$) if there are no messages in the queue yet, Queue$_0[0]$ otherwise. If a message is popped, all other messages shift one position to the end of the queue.

For computing the next value of a queue element, we consider four cases: 1) nothing happens; 2) a message is pushed; 3) a message is popped; or 4) messages are both pushed and popped from the queue. The first case takes the previous value of the queue element. For case 2), we use a similar technique as for the queue counter: we compute a predicate encoding when a certain message will be inserted. Then, depending on the value of the queue counter, we either take the new message, or keep the old one in place. For case 3) we compute a predicate describing when a message will be popped, in which case we take the value of the previous queue element. Case 4) is a combination of 2) and 3), pushing and popping at the same time. For each queue, we generate a next statement containing these four cases. The values of messages are given by Corollary 1.

**Example 15.** The code extract below gives the next statements for the two positions of queue $q_0$ in Figure 1.2. The first conditions correspond to case 4 above, where a push and a pop happen. Because the queue only has two places, the first place
(Queue_0[0]) still contains “none”. The second place (Queue_0[1]) contains the value given by the source. The second and third conditions respectively correspond to a push only or a pop only.

next(Queue_0[0]) :=
case
  Queue_0_n = 1 & Queue_1_n = 1 & oracle_Source_1 & oracle_Merge_0 &
  Queue_1[1] = rsp & oracle_Sink_1: none;
  Queue_0_n != 2 & Queue_1_n != 2 & oracle_Source_1 & oracle_Merge_0:
    (Queue_0_n = 1 ? Source_1 : Queue_0[0]);
  Queue_1_n != 0 & Queue_0_n != 0 & Queue_1[1] = rsp & oracle_Sink_1: none;
  TRUE: Queue_0[0];
esac;

next(Queue_0[1]) := case
  Queue_0_n = 1 & Queue_1_n = 1 & oracle_Source_1 & oracle_Merge_0 &
  Queue_1[1] = rsp & oracle_Sink_1: Source_1;
  Queue_0_n != 2 & Queue_1_n != 2 & oracle_Source_1 & oracle_Merge_0:
    (Queue_0_n = 0 ? Source_1 : Queue_0[1]);
  Queue_1_n != 0 & Queue_0_n != 0 & Queue_1[1] = rsp & oracle_Sink_1:
    Queue_0[0];
  TRUE: Queue_0[1];
esac;

Our translation makes one extra optimization. If a queue has a type that consists of just one message, it is represented by a counter only.
Here we present the EBNF grammar for CodexMAS.

```plaintext
modifier = "param"
simple_type = "chan" | "bus" "<" add-exp ">" | "msg" | "cond" | "int"
decl = [modifier] simple_type [ID {"," ID}] ["(" decl_list ")"]
rec_decl = 
[decl | rec_decl | decl]
decl_list = (rec_decl | decl) ["," decl_list]
definition = declaration [":=" expr | ":=" expr]
assignment = "let" assign-exp
state = "state" ID ["(" decl_list ")"]
transition = "trans" ["(" expr ")"] pa-exp
primary-exp = 
["[" expr "].." expr "]" | constant
| ID ["(" expr ")"] | 
["[" expr "]" | 
| "(" expr ")"]
| "{" expr "}" // enum value
unary-exp = primary-exp | "!" unary-exp
mult-exp = unary-exp [ ("*"|"/"|"mod"|"%"|"mul"|"div") mult-exp ]
add-exp = mult-exp [ ("-"|"+"|"-"|"+") add-exp ]
rel-exp = add-exp [ ("<"|">"|"<="|">="|"=="|"!="|"in"|"!="|"notin") add-exp ]
log-exp = rel-exp [ ("and"|"&&"|"or"|"||") log-exp ]
with-exp = log-exp [ "with" 
{" ID ":=" ID { "," ID ":=" ID } "}
assign-exp = log-exp [ ":=" with-exp ]
assign-list-exp = assign-exp [";" assign-list-exp ]
cond-exp = assign-list-exp [ "?" assign-list-exp ":=" assign-list-exp ]
rw-exp = primary-exp [ "!" primary-exp | ":?" ]
action-exp = rw-exp [ ":" action-exp ]
next-exp = action-exp ":->" primary-exp | ":->" primary-exp
pa-exp = next-exp
expr = cond-exp [";"," expr]
component = ("component"|"process") ID [ "{" param_decl_list "}"]
network-stat = definition | assignment | component | expr
statement = definition | assignment | transition | state | expr
network = {network-stat}
```
Bibliography


