Process algebra semantics & reachability analysis for micro-architectural models of communication fabrics

Sanne Wouda, Sebastiaan J. C. Joosten, and Julien Schmaltz
Department of Mathematics and Computer Science
Eindhoven University of Technology
s.wouda@student.tue.nl, {s.j.c.joosten,j.schmaltz}@tue.nl

***Abstract*—We propose an algorithm for reachability analysis in micro-architectural models of communication fabrics. The main idea of our solution is to group transfers in what we call transfer islands. In an island, all transfers fire at the same time. To justify our abstraction, we give semantics of the initial models using a process algebra. We then prove that a transfer occurs in the transfer islands model if and only if the same transfer occurs in the process algebra semantics. We encode the abstract 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, our approach shows promising results.**

This is a preprint to https://doi.org/10.1109/MEMCOD.2015.7340487

I. INTRODUCTION

Modern Systems-on-Chips (SoC's) are multi-processors, and modern processors are multi-core. The increase in the number of interconnected components integrated in a single system creates new challenges. The design and validation of interconnection structures used to connect cores in processors or components in SoC's is a bottleneck. Analysing such networks is challenging for formal methods because the large number of queues induces a very large state space and distributed control makes localisation difficult to apply [1].

Recently, Intel [2], [3] introduced a dedicated language to model and verify communication fabrics. This language is called XMAS (for: eXecutable Micro-Architectural Specification). Intel demonstrated that using such models, it was possible to automatically infer inductive invariants that ease model checking of safety [4], [5] and deadlock properties [6] on hardware generated from the models. Verbeek and Schmaltz [7], [8] show that deadlock can be hunted even more efficiently directly at the XMAS level. All these methods actually aim at proving complex liveness properties, like request response. To be efficient, they all use invariants to over-approximate the state-space. As a result, they are incomplete and can report false deadlock scenarios, that is, deadlocks that are not reachable from the initial state.

Our contribution is an algorithm for reachability analysis of micro-architectural models of communication fabrics described in XMAS and similar languages. The basic element of this language is a communication *channel* connecting an initiator primitive to a target primitive. A transfer occurs when the channel is such that both the initiator is ready to send data and the target is ready to receive data. Our algorithm is based on the extraction of sets of channels that fire simultaneously, that is, a transfer occurs in one channel in the set if and only if a transfer occurs in all other channels of the set. We call such a set a *transfer island*. A transfer island actually captures the symbolic expressions about transfer decisions. Islands also ensure locality of these conditions. Transfer decisions in one island are independent of the rest of the network. We provide an algorithm to automatically compute all transfer islands. The symbolic expressions captured in the transfer islands are encoded together with a reachability property in the language of the NUXMV tool set [9]. Experimental results compare the performance of symbolic model-checking (BDDs) with IC3 [10]. Performance is further improved using inductive invariants automatically generated from the model using existing techniques [4], [5] and their implementations in a design tool for XMAS [11].

The next section introduces the XMAS language and the necessary pre-requisites about process algebra. Section III gives process algebra semantics to XMAS. Section IV defines transfer islands and proves semantic equivalence between executions for transfer islands and the process algebra. Section V gives our algorithm for extracting transfer islands and encoding the result in NUXMV for reachability analysis. Finally, Section VI presents experimental results and Section VII concludes the paper.

II. PRELIMINARIES: XMAS AND PROCESS ALGEBRA

In this section, we present the XMAS language used to represent micro-architectural models of communication fabrics. We also recall the basic definitions of the process algebra used to give semantics to XMAS.

A. The XMAS language

An XMAS model is a network of primitives connected via typed *channels*. A channel is connected to an *initiator* and

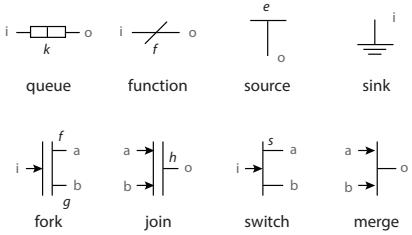

Fig. 1: The eight XMAS primitives.

This research is supported by the Netherlands Organisation for Scientific Research (NWO) under the project Effective Layered Verification of Networks on Chip (ELVeN) under grant no. 612.001.108.

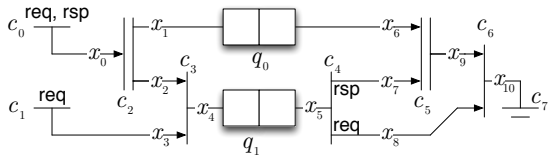

Fig. 2: A simple example of an xMAS network.

a *target* primitive. A channel is composed of three signals. Channel signal $x.irdy$ indicates whether the initiator is ready to write to channel x. Channel signal $x.trdy$ indicates whether the target is ready to read from channel x. Channel signal $x.data$ contains data that are transferred from the initiator output to the target input if and only if both signals $x.irdy$ and $x.trdy$ are set to true. Figure 1 shows the eight primitives of the xMAS language. A *function* primitive manipulates data. Its parameter is a function that produces an outgoing packet 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.

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 $o.irdy$ 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 $i.trdy$ 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 state. Queues accept packets only when they are not full. A source and a sink produces or consumes a packet according to an internal oracle modelling non-determinism.

Example 1. *Figure 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 the join synchronising with q_1 before travelling further to the sink. We will use this simple network as a running example throughout the paper.*

We assume xMAS networks that are syntactically correct and without combinatorial cycles. Procedures exist to check for these properties [12].

B. Process Algebra

In this section, we give a description of Process Algebra, as described by Baeten, Basten, and Reniers [13]. We repeat the parts needed for this work. 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. The semantics of these operations are defined later.

- 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 $x!d$. We can perform a set of reads and writes 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 and writes. Instead of using a set notation for u, we will omit the brackets $\{\}$, and separate the elements of u by $|$, and call it an *action*, as is usual in Process Algebra. For example, we write $x_0?d_0|x_1!d_1.P$ for $u.P$ when $u = \{x_0?d_0, x_1!d_1\}$.
- Similarly for sources and sinks c and $d \in D$: read $c?d$ and write $c!d$.
- Perform either process P, or process Q: $P + Q$.
- Perform the processes P and Q in parallel: $P \| Q$.
- 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 $\{x!d, x?d\} \subseteq u$.

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

$$P + Q \approx Q + P \tag{1}$$
$$P + (Q + R) \approx (P + Q) + R \tag{2}$$
$$P \| Q \approx Q \| P \tag{3}$$
$$P \| (Q \| R) \approx (P \| Q) \| R \tag{4}$$
$$P \| (Q + R) \approx (P \| Q) + (P \| R) \tag{5}$$

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, which arise when two processes simultaneously write and read from the same channel. Readers who are familiar with process algebra may recognise 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 $\|_i P_i$ for $P_0 \| \cdots \| P_n$. Let u_i be sets of reads

and writes such that their union $\bigcup_i u_i$ contains a read $x?d$ for every write $x!d$ and vice versa:

$$\begin{array}{l} \Big\|_i (u_i.P_i) \xrightarrow{\bigcup_i u_i} \Big\|_i P_i \\ \text{if } x?d \in \bigcup_i u_i \Leftrightarrow x!d \in \bigcup_i u_i \end{array} \tag{6}$$

- A choice:

$$\text{if } P \xrightarrow{u} R \text{ then } P + Q \xrightarrow{u} R \tag{7}$$

- A parallel step in one of the branches:

$$\text{if } P \xrightarrow{u} R \text{ then } P \| Q \xrightarrow{u} R \| Q \tag{8}$$

- A step allowed by an equivalent process.

$$\text{if } P \approx Q,\ R \approx S \text{ and } P \xrightarrow{u} R, \text{ then } Q \xrightarrow{u} S \tag{9}$$

- Given a process P, and some sets of possible transitions, we write $\theta(P)$ to limit the transitions to those that are maximal:

$$\begin{array}{l} \theta(P) \xrightarrow{u} \theta(Q) \\ \text{if } P \xrightarrow{u} Q \text{ and there is no } v \supset u \text{ s.t. } P \xrightarrow{v} Q \end{array} \tag{10}$$

Readers with a background in process algebra will see a slight deviation: Equation 9 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 6 to 9 more succinctly. Similarly, the familiar priority operator θ already contains the ordering implicitly in Equation 10.

III. Process Algebra Semantics

We show that all xMAS primitives can be described in Process Algebra. 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 oracle in the xMAS merge, and vice versa. 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 primitives C. Each primitive c is specified as a process on a subset of all channels. The mapping between primitives and channels is such that each channel has exactly one primitive that can perform reads, and one that can perform writes. A channel on which a primitive can read is referred to as an input port, a channel on which it can write will be called an output port. The final process that is described, is the parallel composition of the processes of all primitives.

Used this way, process algebra allows us to give descriptions of xMAS primitives as processes, without having to be explicit about the $irdy$ and $trdy$ wires.

a) Queue: The xMAS queue is defined as a "synchronous FIFO with a standard interface," namely a read and write port. Additionally, the queue size $k > 0$ is a parameter of the queue. Process Q^k_σ, with $\sigma \in D^*$, models this behaviour.

When the queue is empty, only a receive is possible. Process Q^k_ϵ receives a message $d \in D$ through an $i?d$ action, d is the new σ resulting in process Q^k_d.

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

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

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

When a queue is neither empty nor full, both a send and receive can happen simultaneously.

$$Q^k_{\sigma d} = \sum_{e \in D} \left(i?e | o!d.Q^k_{e\sigma} \ + \ i?e.Q^k_{e\sigma d} \right) \ + \ o!d.Q^k_\sigma$$

b) Function: A function can simultaneously receive a message d via $i?d$, modify it according to f and send it onwards via $o!f(d)$. Note that the send and receive happen simultaneously, or not at all.

$$\mathit{Function}_f = \sum_{d \in D} i?d | o!f(d).\mathit{Function}_f$$

c) Source: A source sends a message d into the network via an $o!d$ action. The $o!d$ action models the sending of the message on the channel of the source. The action $c?d$ is externally visible and models our external behaviour, namely the insertion of a message d at source c.

$$\mathit{Src} = \sum_{d \in D} c?d | o!d.\mathit{Src}$$

d) Sink: Similar to the source. Again, the external behaviour is modelled by the $c!d$ action.

$$\mathit{Sink} = \sum_{d \in D} c!d | i?d.\mathit{Sink}$$

e) Fork: A fork can receive a message d on its input channel via $i?d$ and send 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, or not at all. This corresponds exactly to the $irdy$ and $trdy$ definitions of the wire semantics.

$$\mathit{Fork} = \sum_{d \in D} i?d | a!d | b!d.\mathit{Fork}$$

f) Join: Similar to `fork`. Instead, two messages are received simultaneously which are combined using function parameter h. The duality between fork and join is especially visible in the process definitions.

$$\mathit{Join}_h = \sum_{d,e \in D} a?d | b?e | o!h(d,e).\mathit{Join}_h$$

g) Merge: The specification of arbitration in xMAS is rather complex using wire semantics. However, the principle is easy. Of two possible inputs, at most one can progress. So, either a message d is received on the first channel ($a?d$) and sent to the output ($o!d$), or the second channel is relieved from its heavy duty ($b?d$).

$$\mathit{Merge} = \sum_{d \in D} (a?d|o!d.\mathit{Merge} + b?d|o!d.\mathit{Merge})$$

h) Switch: Finally, the switch routes messages based on its parameter function s. The switch can either send message d from the input channel ($i?d$) to output a or to output b, depending on the value of $s(d)$. The multi-actions are similar to those of the merge.

$$\mathit{Sw}_s = \sum_{d \in D \wedge s(d)} i?d|a!d.\mathit{Sw}_s \; + \sum_{d \in D \wedge \neg s(d)} i?d|b!d.\mathit{Sw}_s$$

An xMAS network is a composition of primitives through channel connections. To model the behaviour of an xMAS network as a whole, we must find a suitable composition of our primitive processes, such that the behaviour of the process matches the intended behaviour of the xMAS network. Using standard parallel composition to compose the primitive processes, process $P_{\mathcal{N}}$ giving semantics to xMAS network $\mathcal{N} = (C, X)$ is defined as follows:

$$P_{\mathcal{N}} = \theta \left(\parallel_{c \in C} P_c \right)$$

In our 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. When we refer to a state s, we are referring to the network state.

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

We refer to the state of a particular primitive using the following notation.

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

Finally, we write P_c^s for the process of a primitive c in state s and P_c for the process of a primitive c in the initial state.

Definition 3 (process at a state)**.** *The process in a state s is written as $P_{\mathcal{N}}^s = \theta\left(\|_{c \in C} P_c^{s_c}\right)$.*

In process $P_{\mathcal{N}}$ all sources and sinks are dead. That is, the network will never send a message through a source or sink. In xMAS networks, sinks and sources can send and receive non-deterministically. Internally, such a primitive c determines whether to execute a transfer using a boolean oracle, denoted $c.\mathit{oracle}$. To simulate this non-determinism, process $P_{\mathcal{N}}$ executes in parallel with an environment process P_E. This process satisfies the following properties:

- for source c, $c.\mathit{oracle}$ iff P_E can do an action containing $c!d$ for some $d \in D$
- for sink c, $c.\mathit{oracle}$ iff P_E can do an action containing $c?d$ for some $d \in D$

We write a state s combined with an environment process P_E as s/E.

Definition 4 (process in environment)**.** *The process modelling network $\mathcal{N}$ in a state s within environment E is defined as*

$$P_{\mathcal{N}}^{s/E} = \theta \left(\parallel_{c \in C \cup \{E\}} P_c^s \right)$$

IV. Transfer Islands

A. Transfer Islands definitions

To aid in the model checking process, we abstract from irdy and trdy signals in the combinatorial part of the network, as well as intermediate values of data. The idea is to group channels that always transfer together in *transfer islands*.

Example 2. *Consider the running example in Figure 2. All channels between the top source and the input channels of the queues (x_0, x_1, x_2 and x_4) always transfer together. A transfer occurs in one of those channels if and only if a transfer occurs in all the other ones. This is due to the fork that creates this synchronisation of channels. The set composed of all these synchronised channels forms a transfer island. This network has four islands in total (see Example 3)*

A key notion needed to define transfer islands is "combinatorial-closed". This notion is used to identify synchronised channels. The idea is that switches and merges create distinct islands while the other primitives are combined in one island.

Definition 5 (combinatorial-closed)**.** *Given a network $\mathcal{N} = (C, X)$, a set of channels $I \subseteq X$ is combinatorial-closed iff*

- *for all merges c connected to I we have $c.i \in I$ and $(c.a \in I \oplus c.b \in I)$, and*
- *for all switches c connected to I we have $c.o \in I$ and $(c.a \in I \oplus c.b \in I)$, and*
- *for all other combinatorial primitives c connected to I all ports of c are connected to 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 6 (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.*

Example 3. *Applying this definition to the example in Figure 2, we obtain the following transfer islands: $I_0 = \{x_0, x_1, x_2, x_4\}$, $I_1 = \{x_3, x_4\}$, $I_2 = \{x_6, x_5, x_7, x_9, x_{10}\}$, and $I_3 = \{x_5, x_8, x_{10}\}$.*

To give semantics at the level of transfer islands, we need to know when all transfers are enabled. We define predicate

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

Definition 7 (d-transfer-ready)**.** *A channel x is ready to transfer message $d \in D$ in state s within environment E iff $P_{\mathcal{N}}^{s/E} \xrightarrow{u}$, and $x!d \in u$.*

Definition 8 (transfer-ready)**.** *A channel x is transfer-ready iff x is d-transfer-ready for some $d \in D$.*

Finally, a transfer island is enabled if and only if all its channels are ready to transfer.

Definition 9 (I-enabledness)**.** *Given a network $\mathcal{N} = (C, X)$, a transfer island $I \subseteq X$, a state $s \in S_{\mathcal{N}}$ and environment E, we have $\rho_I(s/E)$ iff all channels in I are transfer-ready in s/E.*

Definition 10 ($\mathcal{T}$-enabledness)**.** *A set of transfer islands $\mathcal{T}$ is enabled in state $s \in S_{\mathcal{N}}$ within environment E, written $\rho_{\mathcal{T}}(s/E)$, if and only if*

- *for each $I \in \mathcal{T}$, $\rho_I(s/E)$*
- *for all $I, I' \in \mathcal{T}$, if $I \neq I'$ then $I \cap I' = \emptyset$*
- *for all sets of transfer islands $\mathcal{T}'$ with $\mathcal{T} \subset \mathcal{T}'$, there is an $I \in \mathcal{T}'$ with $\neg\rho_I(s/E)$.*

Now we have defined when a transfer island is enabled. We consider the computation of the state reached by executing an enabled transfer island. We also define predicate $\tau_Y(s/E, s'/E')$ indicating that s'/E' is the Y-successor (the next state after the transfers of Y) of s/E.

Definition 11 (Y-successor)**.** *Given a network $\mathcal{N} = (C, X)$, states $s, s' \in S_{\mathcal{N}}$, environments E, E', and a set of channels $Y \subseteq X$ with for each $x \in Y$, x is d_x-transfer-ready in s/E. We have $\tau_Y(s/E, s'/E')$, s'/E' is a Y-successor of s/E, 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 $s_c = s'_c$,*

and $P_E \xrightarrow{u} P_{E'}$, where u is the smallest set satisfying, for all sources or sinks c:

- *if $c.o \in Y$ (c is a source), then $c!d_{c.o} \in u$*
- *if $c.i \in Y$ (c is a sink), then $c?d_{c.i} \in u$.*

Example 4. *Consider transfer island I_0 of the previous example. For q_0 and q_1, the input channels – namely, x_1 and x_4 – are members of I_0. The next state is according to this definition defined by inserting data into these two queues.*

We now define the successor state obtained after executing a set of transfer islands.

Definition 12 ($\mathcal{T}$-successor)**.** *Given a network $\mathcal{N} = (C, X)$, a set of transfer islands $\mathcal{T} \subseteq 2^X$, states $s, s' \in S_{\mathcal{N}}$, environments E, E' and $\mathcal{T}$ enabled in s/E. Let $X_{\mathcal{T}} = \bigcup_{I \in \mathcal{T}} I$ be the set of channels covered by transfer islands $\mathcal{T}$. We say s'/E' is a $\mathcal{T}$-successor of s'/E', or $\tau_{\mathcal{T}}(s/E, s'/E')$ iff $\tau_{X_{\mathcal{T}}}(s/E, s'/E')$.*

We now define *data propagation*, used later when proving the soundness of our transfer island definitions, and as the basis for our translation to a NUXMV model. Given a state s, an environment E and a set of channels Y , we can determine per channel x what message is sent through it when all channels in Y perform a transfer.

Definition 13 (data propagation)**.** *For any transfer-ready channels $Y \subseteq X$, channel $x \in Y$, state s, and environment E, we define function $\pi_x(s/E, Y)$. Let c be the initiator of x.*

- *If c is a source, then $P_E \xrightarrow{u}$, with $c.o!d \in u$ and $\pi_x(s/E, Y) = d$,*
- *if c is a queue, then $s_c = \sigma d$ and $\pi_x(s/E, Y) = d$,*
- *if c is a function with f, then $\pi_x(s/E, Y) = f(\pi_{c.i}(s/E, Y))$.*
- *if c is a fork or switch, then $\pi_x(s/E, Y) = \pi_{c.i}(s/E, Y)$.*
- *if c is a join with h, then $\pi_x(s/E, Y) = h(\pi_{c.a}(s/E, Y), \pi_{c.b}(s/E, Y))$.*
- *if c is a merge and $c.a \in Y$, then $\pi_x(s/E, Y) = \pi_{c.a}(s/E, Y)$*
- *if c is a merge and $c.b \in Y$, then $\pi_x(s/E, Y) = \pi_{c.b}(s/E, Y)$*

This definition is well-founded because networks have no combinatorial cycles.

Corollary 1. *Given a set of transfer islands $\mathcal{T}$, let $X_{\mathcal{T}}$ be the union of the transfer islands. If $\mathcal{T}$ is enabled in s/E, then for each channel $x \in X_{\mathcal{T}}$ we have that x is $\pi_x(s/E, X_{\mathcal{T}})$-transfer-ready.*

For our translation into a NUXMV model, we also need a way to determine if a channel is enabled, independent of the Process Algebra semantics.

Corollary 2. *The set of channels Y is enabled in s/E iff for each $x \in Y$*

- *if the initiator of x, say c, is a queue, then $s_c \neq \epsilon$, and*
- *if the target of x, say c, is a queue with capacity k, then $|s_c| \neq k$, and*
- *if the initiator of x, say c, is a source, then $P_E \xrightarrow{u}$ and $c!d \in u$, and*
- *if the target of x, say c, is a sink, then $P_E \xrightarrow{u}$ and $c?d \in u$, and*
- *if $x = c.a$ and c is a switch with f, then $f(\pi_x(s/E, Y))$, and*
- *if $x = c.b$ and c is a switch with f, then $\neg f(\pi_x(s/E, Y))$.*

B. Justification for transfer islands

To justify the correctness of the transfer islands, we prove semantic equivalence between the transfer islands and the process algebra 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. We first prove our main theorem referring to lemma's which are proven after the theorem.

Theorem 1 (Semantic equivalence)**.** *Given a network $\mathcal{N} = (C, X)$, states $s, s' \in S_{\mathcal{N}}$, and environments E, E'. There exists an action Z such that $P_{\mathcal{N}}^{s/E} \xrightarrow{Z} P_{\mathcal{N}}^{s'/E'}$ iff there exists a set of transfer islands $\mathcal{T}$ such that $\mathcal{T}$ is enabled in s/E, and s'/E' is a $\mathcal{T}$-successor of s/E.*

Proof: ($\Rightarrow$) Take an action Z such that $P_{\mathcal{N}}^{s/E} \xrightarrow{Z} P_{\mathcal{N}}^{s'/E'}$. Take all channels that transfer in Z, that is $X_Z = \{x \mid x \in X \wedge d \in D \wedge x!?d \in Z\}$. Let $\mathcal{T}$ be a partition of X_Z into transfer islands (which exists per Lemma 1 and Lemma 2).

We have that $\mathcal{T}$ is enabled (see Definition 10), since

- for each transfer island $I \in \mathcal{T}$, since $I \subseteq X_Z$ and for all $x \in X_Z$ we have some $x!?d \in Z$, it follows that $\rho_I(s/E)$, and
- all islands are non-intersecting and non-empty, since $\mathcal{T}$ is a partition, and
- $\mathcal{T}$ is a maximal enabled set. Suppose for a contradiction that there exists a strict superset $\mathcal{T}'$ with for all $I \in \mathcal{T}'$ we have I is enabled in s/E. Let I be a transfer island in $\mathcal{T}'$ but not in $\mathcal{T}$. Then there exists a $Z' \supset Z$ such that $P_{\mathcal{N}}^{s/E} \xrightarrow{Z'} P_{\mathcal{N}}^{s''/E''}$ for some s'' and E''. However, by the definition of priority θ, Z would not be enabled in s/E. This is a contradiction.

We show that s'/E' is an X_Z-successor of s/E. Since $\mathcal{T}$ is a partition of X_Z it follows that $X_Z = \bigcup_{I \in \mathcal{T}} I$.

By the processes of the queue, source and sink primitives defined in Section II-B, we have that $P_{\mathcal{N}}^{s} \xrightarrow{Z} P_{\mathcal{N}}^{s'}$ implies that for all queues c and messages $d, e \in D$ that

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

and $P_E \xrightarrow{Z_E} P_{E'}$ where we define Z_E such that, for all sources or sinks c

- if $c.o!d \in Z$ then $c.o?d \in Z_E$
- if $c.i?d \in Z$ then $c.i!d \in Z_E$

By Definition 11 it follows that s'/E' is a X_Z-successor of s/E. By Definition 12 it follows that s'/E' is a $\mathcal{T}$-successor of s/E.

($\Leftarrow$) Assume $\mathcal{T} \subseteq 2^X$ is a set of transfer islands such that $\mathcal{T}$ is enabled in s/E and s'/E' is a $\mathcal{T}$-successor of s/E. Let $X_{\mathcal{T}}$ be the set of all channels covered by $\mathcal{T}$, that is $X_{\mathcal{T}} = \bigcup_{I \in \mathcal{T}} I$. Let action Z be defined as follows:

$$\begin{aligned} Z = &\{x!?d \mid x \in X_{\mathcal{T}} \wedge d = \pi_x(s/E, X_{\mathcal{T}})\} \\ &\cup \{c!?d \mid \text{source } c \wedge c.o \in X_{\mathcal{T}} \wedge d = \pi_{c.o}(s/E, X_{\mathcal{T}})\} \\ &\cup \{c!?d \mid \text{sink } c \wedge c.i \in X_{\mathcal{T}} \wedge d = \pi_{c.i}(s/E, X_{\mathcal{T}})\} \end{aligned}$$

We show that $P_{\mathcal{N}}^{s/E} \xrightarrow{Z} P_{\mathcal{N}}^{s'/E'}$. By definition this amounts to

$$\theta \left(\Vert_{c \in C \cup \{E\}} P_c^{s_c} \right) \xrightarrow{Z} \theta \left(\Vert_{c \in C \cup \{E'\}} P_c^{s'_c} \right).$$

We define $Z_c \subseteq Z$, for each $c \in C \cup \{E\}$, as the smallest sets satisfying

- if $c!?d \in Z$ and c is a source, then $c?d \in Z_c$ and $c!d \in Z_E$, and
- if $c!?d \in Z$ and c is a sink, then $c!d \in Z_c$ and $c?d \in Z_E$, and
- if $x!?d \in Z$, then $x!d \in Z_c$ and $x?d \in Z_{c'}$, where c and c' are the initiator and target of x respectively.

We have

1) $P_E \xrightarrow{Z_E} P_{E'}$, since there is a $\mathcal{T}$-transfer from s/E to s'/E' which implies that the sources and sinks transfer data exactly in accordance with Z_E;
2) $P_c^{s_c} \xrightarrow{Z_c} P_c^{s'_c}$, for each $c \in C$, by the definitions of Z, Z_c, data propagation (Definition 13) and the primitive processes;
3) by the definition of Z_c there exists a read action for every write action, and vice versa. Therefore, the parallel composition $\Vert_{c \in C \cup \{E\}} P_c$ can perform exactly action Z.
4) there is no enabled $Z' \supset Z$ since there is no transfer-enabled set of channels $X'_{\mathcal{T}} \supset X_{\mathcal{T}}$, because if there was such an $X'_{\mathcal{T}}$, there would exist a transfer island $I \notin \mathcal{T}$ which is enabled in s/E, which is a contradiction with the assumption that $\mathcal{T}$ is enabled in s/E;

Therefore $P_{\mathcal{N}}^{s/E} \xrightarrow{Z} P_{\mathcal{N}}^{s'/E'}$. ■

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 Z, $P_{\mathcal{N}}^{s}$ can do a Z, and for each $x \in X$ there exists a $x!?d \in Z$ iff $x \in Y$, then Y is combinatorial-closed.*

Proof: Take an action Z such that for each $x \in Y$ there exists an $x!?d \in Z$. Let c be any primitive. If c is connected to a channel x in Y, then there is an action $x!?d \in Z$. By the structure of $P_{\mathcal{N}}^{s}$, we have that $P_c^{s_c}$ 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 $P_c^{s_c}$ also performs a read or write action on x'. By the structure of $P_{\mathcal{N}}^{s}$, there must be a transfer $x'!?d' \in Z$ (for some 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 primitive 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 was combinatorial-closed, and for each primitive c, those channels in Y connected to c are in the same connected component. Furthermore, I is the smallest such set, since only those channels in Y that are required by the combinatorial-closed conditions are in Y. Hence, each connected component is a transfer island, and the set of connected components of (Y, E) is a partition into transfer islands. ∎

V. Reachability Analysis

A. Transfer islands extraction

Consider a given network $\mathcal{N} = (C, X)$. Algorithm 1 processes all primitives 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 any primitive of which the initiators of all input channels are already processed. We call this *transfer order*.

Example 5. *For the network in Figure 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 the set of transfer islands $\mathcal{I}$ 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 5).

For each source or queue, a new island is created containing the output channel of the source. When the primitive is a fork, the two outputs channels are added to all islands that contain the input of the fork primitive.

Example. *Running these steps on the example in Figure 2, we obtain the following islands:*

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

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

Example. *Next, we process primitive 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 primitive is a switch, then 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 primitive 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 primitive is a queue that we have processed before, then nothing happens and the transfer island set does not change.

Example. *We process queues q_0 and q_1 for the second time, so the set of transfer islands does not change.*

When the primitive is a join, then 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 primitive c_5, a join. The set of transfer islands becomes:*

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

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

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

The correctness follows from the definition of combinatorial-closed. The transfer-islands are minimal, since we only add channels when required by the combinatorial-closed conditions. The running time of the algorithm is exponential in the number of primitives.

Algorithm 1: compute transfer islands

```
for c ∈ C in transfer order do
    if c is a source then
        Create transfer island I = {c.o}.
    else if c is a fork then
        for I ∈ B_{c.i} do
            Add c.a and c.b to I.
    else if c is a merge then
        for I ∈ B_{c.a} ∪ B_{c.b} do
            Add c.o to I.
    else if c is a queue and first time then
        Create transfer island I = {c.o}.
    else if c is switch then
        for I ∈ B_{c.i} do
            Let I' be a copy of I.
            Add c.a to I and c.b to I'.
    else if c is a join then
        Let X = B_{c.a} and X' = B_{c.b}.
        for I ∈ X do
            for I' ∈ X' do
                Let new transfer island
                I'' = I ∪ I' ∪ {c.o}.
        Remove all islands in X and X'.
    else
        Skip.
```

B. Translation to NuXmv

Queues are modelled by two variables: an array modelling 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 initialised with a counter set at 0 and all their positions filled with a 'none' value.

Example 6. *Considering the example in Figure 2, the top queue is declared as follows:*

```
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 7. *The following* NUXMV *declaration model the input variables and oracles of the example in Figure 2.*

```
  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 modelled 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 there 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 8. *Here is the next statement for the counter of queue q_0 in Figure 2. Thanks to the information captured in transfer islands we can easily state a next statement of queue q_0 that depends on the state of the other queue q_1.*

```
next(Queue_0_n) :=
  case
    Queue_0_n = 0 & Queue_1_n != 2 &
    oracle_Source_1 & oracle_Merge_0: 1;

    Queue_0_n = 2 & Queue_1_n != 0 &
    Queue_0_n != 0 & Queue_1[1] = rsp
    & oracle_Sink_1: 1;

    TRUE: Queue_0_n +
    toint(Queue_0_n != 2 & Queue_1_n != 2
    & oracle_Source_1 & oracle_Merge_0) -
    toint(Queue_1_n != 0 &
    Queue_0_n != 0 & Queue_1[1] = rsp &
    oracle_Sink_1);
  esac;
```

Messages are always taken from the end of the queue (Queue_0[1]) and inserted at the beginning (Queue_0[1] if there are no messages in the queue yet, Queue_0[0] otherwise). If a message is removed, 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 inserted; 3) a message is removed; and 4) messages are both inserted and removed from the queue. The first case is easy, just take 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 removed, in which case we take the value of the previous queue element. Case 4) is a combination of 2) and 3), shifting and taking on a message at the same time. For each queue, we generate a next statement containing these four cases. Values of messages are given by Corollary 1.

Example 9. *The code extracts below give the next statements for the two positions of queue q_0 in Figure 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 optimisation. If a queue has a type that consists of just one message, it is represented by a counter only.

VI. Experimental results

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 [11]. For the convenience of the reader, we put the resulting nuXmv models on-line[1]. Before computing reachability, we infer all channel types using the technique proposed by Gastel et al. [12]. 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, 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 scalability. We describe all examples and give tables with execution times of nuXmv.

A. Small examples

Our examples are taken from Verbeek's thesis [14]. These networks are special cases where his deadlock hunting tool finds false deadlocks, that is, deadlock states that are actually 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 3. The source injects exactly one message into the network. The sink is dead, that is, it never accepts messages. The shown configuration is supposedly 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, and 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. Our reachability checker shows that this configuration is indeed not reachable. Even if the sink is dead, this network has no deadlock.

Rationals: Figure 4 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. Verbeek's tool still 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.

[1] http://www.win.tue.nl/~jschmalt/publications/memo15/memo15.html

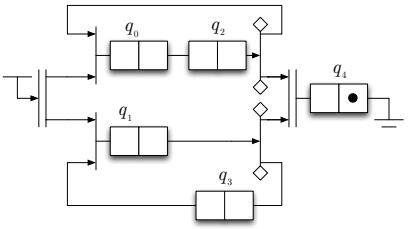

Fig. 3: An unreachable state in asynchronous semantics

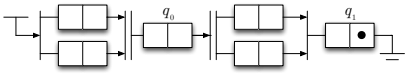

Fig. 4: A false deadlock with fractions of messages.

Red and blue: Figure 5 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.

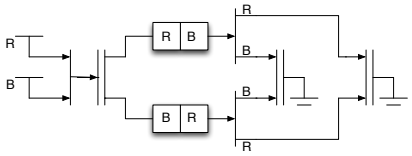

Fig. 5: The red and blue example.

B. 2-agents example

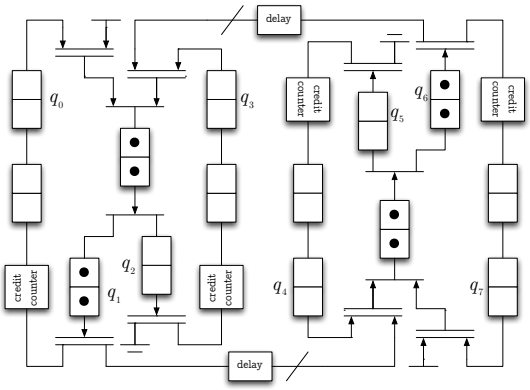

Fig. 6: 2-agents.

Figure 6 shows a network consisting of 2-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".

C. Execution times

Table I 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, automatically generated, 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.

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

TABLE I: Results for the synchronous semantics.

To further experiment, we defined an *asynchronous* version of our transfer islands. The idea is that instead of executing all transfers of all enabled transfer islands, only one enabled transfer island is executed at a time. There is therefore a need to compute all possible orderings of the execution of the transfer islands. Table II 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.

design	BDD	IC3	BDD + invariants	IC3 + invariants
asynchronous *	0.03	0.28	0.32	0.22
rationals	0.02	0.05	0.02	0.07
blue red	0.02	0.05	0.02	0.06
twoagents 2/2	5.58	7.27	5.15	1.40
twoagents 2/3 *	53.49	119.55	68.03	6.48
twoagents 4/4	413.70	50.67	754.31	21.43
twoagents 4/5 *	1387.4	108.52	2765.40	46.56

TABLE II: Results for the asynchronous semantics.

VII. CONCLUSION

We presented a method for checking reachability of states in micro-architectural models of communication fabrics. Our analysis first captures the symbolic expressions of transfers in what we called *transfer islands*. These symbolic expressions are then translated to NUXMV together with a reachability property. We then compared the BDD and IC3 engines on several small and larger examples. We also evaluated the impact on inductive invariants automatically generated from the micro-architectural models. The conclusion is that IC3 with invariants provides the overall best performance. It seems from these results that reachability of states is feasible even in quite large networks.

Future experiments should explore this further. In particular, even if some of the examples used in this paper are extracted from realistic designs, they are still rather small. Note that in the examples the number of different message types is limited to two. Having more message types will clearly impact performance. More experiments are needed to evaluate this impact. Our method answers the basic question of state reachability. Future work includes extending our approach to larger classes of properties, which requires explicit fairness constraints on oracles and arbitration policies.

REFERENCES

[1] S. Ray and R. K. Brayton, "Scalable progress verification in credit-based flow-control systems," in *DATE*, W. Rosenstiel and L. Thiele, Eds. IEEE, 2012, pp. 905–910.

[2] S. Chatterjee, M. Kishinevsky, and Ü. Y. Ogras, "Quick formal modeling of communication fabrics to enable verification," in *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'10)*, 2010, pp. 42–49.

[3] ——, "xMAS: Quick formal modeling of communication fabrics to enable verification," *IEEE Design & Test of Computers*, vol. 29, no. 3, pp. 80–88, 2012.

[4] S. Chatterjee and M. Kishinevsky, "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics," in *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, July 2010.

[5] ——, "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics," *Formal Methods in System Design*, vol. 40, no. 2, pp. 147–169, Apr. 2012.

[6] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, "Verifying deadlock-freedom of communication fabrics," in *Verification, Model Checking, and Abstract Interpretation (VMCAI '11)*, vol. 6538, 2011, pp. 214–231.

[7] F. Verbeek and J. Schmaltz, "Hunting deadlocks efficiently in microarchitectural models of communication fabrics," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11, Austin, TX, 2011, pp. 223–231.

[8] ——, "Automatic generation of deadlock detection algorithms for a family of micro architectural description languages," *IEEE International High Level Design Validation and Test Workshop (HLDVT'12)*, November 2012.

[9] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Computer Aided Verification*. Springer, 2014, pp. 334–342.

[10] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2011, pp. 70–87.

[11] S. Joosten, F. Verbeek, and J. Schmaltz, "Wickedxmas: Designing and verifying on-chip communication fabrics," in *Proceedings of the 3rd International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS'14)*, 2014, pp. 1–8.

[12] B. van Gastel, F. Verbeek, and J. Schmaltz, "Inference of channel types in micro-architectural models of on-chip communication networks," in *Proceedings of the 22nd IFIP/IEEE International Conference on Very Large Scale Integration*, 2014.

[13] J. C. Baeten, T. Basten, and M. Reniers, *Process algebra: equational theories of communicating processes*. Cambridge university press, 2010, vol. 50.

[14] F. Verbeek, "Formal verification of on-chip communication fabrics," Ph.D. dissertation, Radboud University Nijmegen, 2013.