Jonatan Wiik | Pontus Boström

Specification and automated verification of dynamic dataflow networks

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Technical Report
No 1170, December 2016

Specification and automated verification of dynamic dataflow networks

Jonatan Wiik
Faculty of Science and Engineering, Åbo Akademi University
jonatan.wiik@abo.fi
Pontus Boström
Faculty of Science and Engineering, Åbo Akademi University
pontus.bostrom@abo.fi

TUCS Technical Report
No 1170, December 2016

Abstract

Dataflow programming has received much recent attention within the signal processing domain as an efficient paradigm for exploiting parallelism. In dataflow programming, systems are modeled as a static network of actors connected through asynchronous order-preserving channels. In this paper we present an approach to contract-based specification and automated verification of dynamic dataflow networks. The verification technique is based on encoding the dataflow networks and contracts in the guarded command language Boogie.

1 Introduction

Modern software systems are increasingly concurrent, as the computing power of modern CPUs is improved mainly by increasing the number of processor cores. At the same time, modern computer platforms are also increasingly distributed and heterogenous, often involving special processing units, such as GPUs or DSPs for performing specific tasks efficiently. Writing software that effectively exploits the capacity of such platforms is hard. The dataflow paradigm has been proposed as a possible solution to this problem and has received a large amount of attention within the signal processing domain. The main advantage of the dataflow paradigm is that it efficiently exploits parallelism in the implemented program.

A dataflow program consists of a static network of actors. Actors are stateful operators communicating exclusively via asynchronous unidirectional order-preserving channels that describe the exchange of data between actors. Each actor can execute concurrently when the required data is available on the incoming channels. As the only communication between actors is performed over channels, computations can easily be mapped to different processing units.

In this paper, we present a hierarchical and modular approach to specification and automated verification of dataflow actors and networks based on assume-guarantee reasoning. The approach is based on giving actors and networks specifications in the form of contracts. We present a novel contract notation for actors and networks. The contracts state functional properties, which the actor or network should adhere to. The goal of the approach is to ensure functional correctness with respect to contracts for actors and networks as well as deadlock freedom for networks. Additionally, we have observed that our contracts could be used to improve the performance of scheduling for dataflow networks. Scheduling methods for dataflow networks [4] are based on finding an execution sequence that returns the communication buffers to the same state. Our contracts explicitly express this buffer state as well as preconditions on input data that also can be utilised in scheduling.

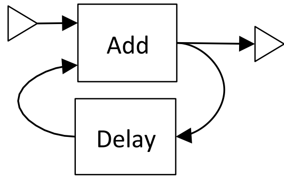
The verification technique is based on encoding the dataflow actors and networks in the guarded command language Boogie [2]. There are several advantages with this approach. The Boogie verifier is tightly integrated with the Z3 [5] SMT solver and provides much of the infrastructure needed to generate efficient verification conditions for this solver. Boogie has also been used as a backend in verifiers for several popular programming languages. This means that our approach could potentially be integrated to support dataflow actors implemented in other host languages for which a translation to Boogie exists.

In this paper we make the following main contributions:

1. A method to specify the behaviour of networks based on the reaction of the network to individual tokens.

2. An encoding of actors, networks and their specifications into a guarded command language.

3. A method to generate invariants needed for verification of a common type of actors.




```
actor Add int x1, int x2 ⟹ int y:
  action x1:[i], x2:[j] ⟹ y:[i+j] end
end

actor Delay(int k) int x ⟹ int y:
  initialize ⟹ y:[k] end
  action x:[i] ⟹ y:[i] end
end
```

Figure 1: Examples of two basic actors and an illustration of a network formed from these actors.

The work presented acts as a generalisation of previous work [3] by the authors on verification of Simulink models. There, Simulink models are translated to synchronous dataflow (SDF) [14, 13] networks for verification. SDF is a subset of the dataflow programs considered here.

The remainder of the paper is structured as follows: We begin by introducing dataflow actors and networks in Section 2 and then informally describe our verification technique in Section 3. In Section 4 we describe the actor and assertion languages. We then describe the encoding of assertions, actors and networks into a guarded command language in Section 5. In Section 6 we describe our invariant generation method and discuss the soundness of our approach in Section 7. We then present results of evaluation of our verification approach on a number of examples in Section 8 before we proceed to related work and conclusions in Section 9 and Section 10.

2 Dataflow actors and networks

The dataflow programs we consider in this paper consist of static hierarchical networks of actors, which are connected via asynchronous, unidirectional, order-preserving channels. The channels are the single mean of communication between the actors.

An actor A is a stateful operator consisting of a set of inports A_{ip}, a set of outports A_{op}, a set of state variables A_{var} and a set of actions A_{act}. An actor performs computations by firing sequences of enabled actions. An action $t \in A_{act}$ is enabled or disabled based on the amount and value of input tokens, as well as the actor state. When an action t fires it consumes a fixed amount of tokens on the inports, produces a static amount of tokens on the outports and updates the actors state. An action hence describes the reaction of the actor to a sequence of input tokens. The amount of tokens consumed or produced on a port is called rate. Actors can be classified as either static or dynamic. An actor is considered static if every action $t \in A_{act}$ consumes and produces the same amount of tokens. Otherwise the actor is considered to be dynamic.

We use a language similar to the CAL actor language [6] to describe our actors and networks. More precisely, our language is a subset of the RVC-CAL [20] lan-

guage, extended with some specification constructs. RVC-CAL is a subset of CAL which is part of the Reconfigurable Video Coding standard.

Some basic examples of actors are given in Figure 1. An actor declaration begins with **actor** followed by a declaration of inports and outports. The actor `Add` has two inports, `x1` and `x2`, and one outport `y`. The datatype of the ports is **int**. The actor has one action with the pattern `x1:[i], x2:[j]` $\Longrightarrow$ `y:[i+j]`, which specifies that the action reads 1 token from each inport, binds them to identifiers `i` and `j`, and outputs the sum, `i + j`, of the read tokens on the outport. The actor `Delay` in Figure 1 delays the data on its input channel with one token. The delay is implemented with a special initialisation action, declared with keyword **initialize**, outputting an initial token on the outport. Initialisation actions are only run once, when the actor is initialised and are not allowed to consume input. In the example, the value of the initial token is given as a parameter to the actor. The actors `Add` and `Delay` are both considered static, as they consume and produce the same amount tokens on their ports each time they fire, disregarding initialisation actions.

Instances of actors are connected to form networks. The graph in Figure 1 illustrates a network consisting of one instance of `Add` and one instance of `Delay`, that calculates the accumulated sum of the input tokens. The goal of this paper is to specify and verify actors and networks like those given in Figure 1 based on contracts. We verify functional correctness with respect to contracts as well as deadlock freedom of actor networks. Note that, for instance, the network in Figure 1 would deadlock without the initialisation action of the `Delay` actor.

3 Verification technique

In this section we informally describe our specifications and our verification technique. The channels of an actor network can be described as streams of data. A channel c is then a stream $\langle c_0, c_1, \dots \rangle$, where each c_i is a data token. Actors can then be considered as stateful operators on streams. The specifications are contracts consisting of preconditions and postconditions for actors and networks. Networks and actors are modularly verified to conform to their contracts.

3.1 Networks

We describe networks using a syntax which resembles the syntax used to describe basic actors. This differs from RVC-CAL, which uses a graphical language to describe networks. The languages are, however, semantically equivalent.

A network declaration begins with the keyword **network**. A wellformed network declaration includes an **entities** block, which declares the actor instances in the network, and a **structure** block, which declares the network topology, i.e. how the actor instances are interconnected. For an example, consider the network `SumNet` given in Figure 2, which is the source code of the network illustrated in Figure 1. The **entities** block defines the actor instances `add` and `del`. In the **structure** block, the channels connecting these actor instances are defined. For instance, that the port `in1` of `add` is connected to the network inport `in`. The channels are

```
network SumNet int in ⟹ int out:
  action in:1 ⟹ out:1
    requires 0 ≤ in[•]
    ensures in[•] ≤ out[•]
    ensures out[0] = in[0]
    ensures 0 < •(out) ⇒ out[•] = out[• − 1] + in[•]
  end
  invariant tokens(b,1)
  chinvariant b[0] = 0
  chinvariant 0 ≤ b[•]
  entities add = Add(); del = Delay(0); end
  structure
    a: in ⟶ add.in1;    b: del.out ⟶ add.in2;
    c: add.out ⟶ out;   d: add.out ⟶ del.in;
  end
end
```

Figure 2: The source code with contract of a network for calculating the accumulated sum of the input tokens.

given labels `a`, `b`, `c` and `d`. These labels can be used to refer to the channels in specifications.

In addition to the **entities** and **structure** blocks, a network also has several specification-oriented constructs. The most central specification construct is network actions, which define the relationship between input and output tokens of the network. As for basic actor actions, network actions describe the reaction of the network to a finite amount of input tokens. However, in contrast to basic actor actions, network actions are merely used for specification and have no impact on the behaviour. A network can be given several network actions, defining different reactions to input tokens.

Definition 1 (Network action). *A network action:*

$$\textbf{action } x\colon n \Longrightarrow y\colon m \textbf{ requires } P \textbf{ ensures } Q \textbf{ end}$$

specifies that, given n input tokens on port x conforming to precondition P, the network outputs m tokens on port y conforming to postcondition Q.

Note that patterns of network actions only specifies the amount of tokes to be produced or consumed an ports. Input tokens are not assigned to identifiers as for basic actors. The reason is that it typically is more intuitive to describe network behaviour in terms of streams and tokens carried in them rather than individual values.

Our verification technique is based on checking that a network behaves according to a network action for a finite window over input and output streams. We consider a window of size n for the input stream x, and size m for the output stream y as specified by the network actions. For an example, consider the network action of our example `SumNet`. It specifies that given 1 input token on port `in`,

it will produce 1 output token on port `out`. The network action of `SumNet` also has a network precondition starting with keyword **`requires`** and 3 postconditions starting with keyword **`ensures`**.

Verifying a network entails checking that the network behaves as one of its network actions for all tokens received. A network action as defined in Definition 1 describes the response of the network to a finite input. To be able to verify the network for the finite window described by the network action, the network should have a periodic behaviour where the period corresponds to one execution of a network action. Additionally, to ensure that the network does not buffer an infinite amount of tokens on any channel, we require that the amount of tokens on channels between periods is fixed. If not explicitly stated otherwise, the channels are required to be empty between network action executions.

It is not always desired or possible to have empty channels between network action executions. The network `SumNet`, for instance, contains a loop and requires that an initial extra token is produced to avoid deadlock. Executing the network action will then always result in an unread token in this loop. This can be specified in network invariants. Network invariants are declared using the keyword **`invariant`** and are required to hold between executions of network actions. In the `SumNet` example, the invariant **`tokens`**`(b,1)` specifies that the network should leave 1 token on the channel `b` between network action executions.

Network invariants are not required to hold during execution of network actions. To track data during execution of network actions we use another type of invariants, which we call channel invariants. The intuition is that a channel invariant needs to hold also during execution of a network action. This means that execution of any sub-actor, i.e. instance of actor or network in the network that is to be verified, should preserve the channel invariant. Channel invariants are declared with the keyword **`chinvariant`**. In `SumNet` example we have a channel invariant $\texttt{b[0]} = \texttt{0}$, stating that the first token produced on channel `b` has the value 0. We use indices to refer to stream tokens in assertions. Hence, $c[i]$ refers to the i:th token produced on channel c.

To write preconditions and postconditions, we want to refer to the tokens produced and consumed during the network action execution. This can be done using the function $\bullet(c)$.

Definition 2 (Bullet)**.** *$\bullet(c)$ is the number of tokens that had been consumed on the channel c when the current network action execution started.*

Based on the definition of $\bullet$, it is possible to refer to the first token produced on a channel c during the current network action execution using $c[\bullet]$ and to the second token produced using $c[\bullet + 1]$. It is also possible to refer to the last token that was produced during the previous execution using $c[\bullet - 1]$. Note that the argument to $\bullet$ is left out in the assertions. The argument is implicit when $\bullet$ is used in the position of an index. Hence, $c[\bullet(c)]$ and $c[\bullet]$ are synonymous.

Consider again the `SumNet` example. The $\bullet$ function is used in the preconditions and postconditions. The network `SumNet` has a precondition, $\texttt{0} \leq \texttt{in}[\bullet]$, which states that the input is required to be non-negative. The postcondition $\texttt{0} < \bullet(\texttt{out}) \Rightarrow \texttt{out}[\bullet] = \texttt{out}[\bullet - \texttt{1}] + \texttt{in}[\bullet]$ states that, for any execution where

$0 < \bullet(\texttt{out})$, the network output should be equal to the previous output plus the current input. The postcondition $\texttt{out[0]} = \texttt{in[0]}$ states that the first token produced by the network should be equal to the first input consumed. Additionally, there is also a postcondition stating that the output is always larger than or equal to the input. It is worth noting that in the invariants above the port names are used to refer to input and output channels. It would be equivalent to refer to the channels using the labels `a` and `c` given to the channels in the **`structure`** block.

We check that a network conforms to its specifications based on an inductive proof. The base step of the inductive proof is checking that the network initialisation establishes the invariants. The inductive step consists in considering an arbitrary execution of a network action and showing that channels will be returned to the same state, i.e. that the network invariants are preserved. Our approach also guarantees deadlock freedom by ensuring that progress is made from the state described by the network invariants when input specified by the network action is received.

It should be noted that networks are in practice not executed atomically for a network action. This means that new input can arrive before the network action has finished executing. In Section 7 we argue that it despite this is sound to verify networks for a finite input, given that actors are continuous. This essentially means that actors should be deterministic.

3.2 Actors

The actors given in Figure 1 are simple static actors without state and their complete behaviour is described by the action patterns. However, actors can have both state as well as dynamic rates. Consider for instance the actor `Sum` in Figure 3. This is a single actor essentially implementing the same functionality as the network in Figure 2. The actor has a state variable `sum` to store the accumulated sum. The action has a body, starting with keyword **`do`**, which updates the state variable. Action bodies are described using a simple imperative programming language.

For specification, the action of the actor `Sum` in Figure 3 has been annotated with a precondition and a postcondition in the same manner as for network actions above. These constructs are not part of the CAL language. The precondition requires the input token to be greater than or equal to 0, while the postcondition requires the output, i.e. the state variable `sum`, to be greater than or equal to the input. To prove that the action satisfies the postcondition, we need to restrict the state variable with an invariant. The invariant $\texttt{0} \leq \texttt{sum}$ allows the action to be verified.

Actor invariants are required to hold between action firings and can, with some restrictions, be used in the place of channel invariants for verification on the network level. The advantage of this approach is that actor invariants can be locally proved and it is then not needed to prove them again on the network level. Instead they can be used directly as assumptions. The requirement on an invariant to be used as on the network level is that it does not refer to state variables, but only to the actors interface, i.e. its inports and outports. We refer to such invariants as input/output invariants, or i/o invariants, as they typically describe the relationship between

```
actor Sum int x ⟹ int y:
  int sum;
  invariant 0 ≤ sum
  invariant tot(y) > 0 ⇒ sum = y[last]
  invariant tot(y) = rd(x)
  invariant ∀ int i · every(y,i,1) ⇒ y[i] = y[i−1]+x[i]
  initialize ⟹ do sum := 0; end
  action x:[i] ⟹ y:[sum]
    requires 0 ≤ i
    ensures i ≤ sum
    do sum := sum+i;
  end
end

actor Split int in ⟹ int pos, int neg:
  invariant rd(in) = tot(pos)+tot(neg)
  action in:[i] ⟹ pos:[i] guard 0 ≤ i end
  action in:[i] ⟹ neg:[i] guard i < 0 end
end
```

Figure 3: The actor `Sum` is an actor with state and `Split` is a data-dependent actor. The functions **`rd`** and **`tot`** used in the invariants refer to the amount of tokens consumed and produced on streams, respectively.

input streams and output streams. The i/o invariants are expressed in a similar way as invariants on streams for networks, but only include the input and output streams of the actor or network it describes.

For some examples of i/o invariants, consider again the actor `Sum` in Figure 3. In this actor, we have defined the i/o invariants `tot(y) = rd(x)` and `∀ int i · every(y,i,1) ⇒ y[i] = y[i−1]+x[i]`. The remaining two invariants refer to the state variable `sum` and are hence not i/o invariants. In the invariants, the function `tot(y)` gives the total number of tokens produced on the stream connected to port `y`. The function `rd(x)` gives the total number of tokens that has been consumed from the stream connected to port `x`. Hence, the first invariant states that the total number of tokens produced on stream `y` is equal to the total number of tokens consumed on stream `x`. In the second invariant, the data tokens of the input and output streams are referred to using indices. In the second invariant, `every(y,i,1)` is a predicate equal to `1 ≤ i < tot(y)`. The invariant hence states that every token produced on stream y with an index i of 1 or larger is equal to the previous output plus the input token consumed during the same firing. The construct `y[last]` used in one of the invariants refers to the last token produced on the channel `y`.

Our verification technique can locally check the i/o invariants and then use them as assumptions in place of channel invariants on the network level. This is possible because of the following reasons: Every channel in a network has a single actor reading and a single actor writing to it. This means that the number of consumed tokens on an input stream x, **rd**(x), cannot be changed by any other

actor. In the same way, the total number of tokens produced on an output stream y, **tot**(y), cannot be changed by any other actor. Hence these values will not change between firings. On the other hand, the number of consumed tokens on output streams and the number of produced tokens on input streams is not known locally and hence cannot be used in assertions that are checked locally. The verifier performs wellformedness checks on assertions to ensure this.

Input/output invariants like those described above can also be provided for networks, either as network invariants or channel invariants. As for basic actors, i/o invariants for networks can only refer to the interface, i.e. the inports and outports of the network, and not to internal channels. Input/output invariants can be used to modularly check a hierarchy of networks one layer at a time. This significantly simplifies the verification task, especially when a large network consists of several instances of the same actor.

The actor `Split` in Figure 3 is a data-dependent actor. We consider an actor to be data-dependent if the number of tokens consumed and/or produced is dependent on the value of the input tokens. The `Split` actor outputs non-negative input tokens on the outport named `pos` and negative input tokens on the outport named `neg`. In cases like this it becomes very hard to write invariants on streams like those in the `Sum` example, because the number of output tokens depend on the value of the input tokens. For instance, the total number of tokens produced on `pos` is equal to the number of non-negative tokens consumed. To express this as an invariant, we would need to express an aggregated sum on the consumed tokens. For actors like this, it is often more feasible to provide the needed properties as channel invariants on the network level instead.

It should be emphasised that invariants on streams like those described above are also needed for the simple actors in Figure 1. However, in these cases the verification tool can automatically infer the invariants, as we describe in Section 6.

Our actor language allows describing actors that are non-deterministic. Consider for instance changing the guard expression of the second action of `Split` to `i` $\leq$ `0`. An input token with value 0 would then enable both actions and the choice of which action to fire would be non-deterministic. The amount of produced tokens on the output channels would then also be non-deterministic. As we discuss in Section 7, our verification technique is not sound for non-deterministic actors. Because of this we perform wellformedness checks on actors to ensure that firing rules are mutually exclusive. This ensures that for any two action t_1 and t_2 of an actor, t_1 cannot be enabled by the same input sequence as t_2 or a prefix of that sequence. This wellformedness requirement applies to both basic actors and networks.

4 Programming and assertion language

In this section we define precisely the language we consider in this paper. The language can be split into two parts: the actor language, which is used to describe actors, networks and their interconnection, and the host language, which is used to implement actor action bodies. The host language we consider is a simple imperative programming language. We also define our assertion language, which is

$$
\begin{array}{lcl}
\mathit{Prog} & ::= & (\mathit{ActorDecl} \mid \mathit{NwDecl})^* \\
\mathit{ActorDecl} & ::= & \texttt{actor}\ \mathit{id}\langle\overline{\mathit{type}\ \mathit{id}}\rangle\ \mathit{PortDecl}\ \mathit{ActorMem}^*\ \texttt{end} \\
\mathit{NwDecl} & ::= & \texttt{network}\ \mathit{id}\langle\overline{\mathit{type}\ \mathit{id}}\rangle\ \mathit{PortDecl}\ \mathit{NwMem}^*\ \texttt{end} \\
\mathit{PortDecl} & ::= & \overline{\mathit{type}\ \mathit{id}} \Longrightarrow \overline{\mathit{type}\ \mathit{id}} \\
\mathit{ActorMem} & ::= & \mathit{ConstDecl} \mid \mathit{VarDecl} \mid \mathit{ActorInv} \mid \mathit{Action} \mid \mathit{Schedule} \mid \mathit{Priority} \\
\mathit{NwMem} & ::= & \mathit{ActorInv} \mid \mathit{ChInv} \mid \mathit{Action} \mid \mathit{Entities} \mid \mathit{Structure} \\
\mathit{Action} & ::= & \mathit{id}\colon (\texttt{action} \mid \texttt{initialize})\ \mathit{Pattern}\ \mathit{ActSpec}\ \mathit{ActBody}\ \texttt{end} \\
\mathit{Entities} & ::= & \texttt{entities}\ (\mathit{id} = \mathit{id}\langle\overline{e}\rangle;)^*\ \texttt{end} \\
\mathit{Structure} & ::= & \texttt{structure}\ (\mathit{id}(.\mathit{id})^? \longrightarrow \mathit{id}(.\mathit{id})^?;)^*\ \texttt{end} \\
\mathit{ActorInv} & ::= & \texttt{invariant}\ A \\
\mathit{ChInv} & ::= & \texttt{chinvariant}\ A \\
\mathit{ActSpec} & ::= & (\texttt{requires}\ A \mid \texttt{ensures}\ A)^* \\
\mathit{ActBody} & ::= & (\texttt{guard}\ e)^?\ (\texttt{do}\ S)^? \\
\mathit{Pattern} & ::= & \overline{\mathit{InPat}} \Longrightarrow \overline{\mathit{OutPat}} \\
\mathit{InPat} & ::= & \mathit{id}\colon [\overline{\mathit{id}}] \\
\mathit{OutPat} & ::= & \mathit{id}\colon [\overline{e}] \\
\mathit{VarDecl} & ::= & \mathit{type}\ \mathit{id}
\end{array}
$$

Figure 4: Grammar of the actor language.

used to express preconditions, postconditions and invariants.

Actor language The grammar of the complete actor language is listed in Figure 4. In the grammar we use $\langle$ and $\rangle$ for concrete parentheses to differentiate them from the meta-parentheses. A program consists of a set of actor and network declarations. In the grammar S is the statement language of the host language, A is an assertion, e is an expression in the host language and id is an identifier.

Host language The host language we consider in this paper is a simple imperative programming language without reference types, which is straight-forward to encode in a verifier. The statement grammar S of the language is given in Figure 5. Essentially, the language is a small subset of the RVC-CAL host language. The language consists of assignments, if statements and while loops. While loops are verified using Hoare logic as in traditional program verification. The expression language grammar e is also given in Figure 5. This language is also a subset of the RVC-CAL expression language, but has been extended with some constructs, such as quantifiers, which are needed to write specifications. The host language could easily be extended or substituted with another language which can be encoded in a verifier.

Assertion language The grammar of the assertion language A is given in Figure 5. The grammar is defined in such a way that the predicate **tokens** is only allowed as an independent assertion. The reason is that the verifier needs to keep track of for which channels **tokens** has been used, because channels which are not mentioned in any **tokens** predicate are required to be empty between network action executions. For the soundness of our approach we also require that **tokens** is not used on network input streams. Requiring that **tokens** is only used as an

S ::=

$id := e$ |	Assignment
if exp **then** S_1 **else** S_2 **end** |	If-Else
while e (**invariant** A)* **do** S **end**	While

$A ::= A_1 \wedge A_2 \mid$ **tokens**$(id, e) \mid e$

e ::=

$e_1\ (+ \mid - \mid * \mid / \mid \%)\ e_2 \mid$	Binary expression
$e_1\ (\wedge \mid \vee \mid \Rightarrow \mid \Longleftrightarrow)\ e_2 \mid$	Binary predicate
$e_1\ (= \mid \neq \mid < \mid > \mid \leq \mid \geq)\ e_2 \mid$	Relational expression
$\neg e \mid$	Negation
$-e \mid$	Unary minus
$(\forall \mid \exists)\ \overline{type\ id}\ \cdot\ e \mid$	Quantifier
if e_1 **then** e_2 **else** e_3 **end** |	Conditional expression
$id\langle\overline{e}\rangle \mid$	Function call
$e[e] \mid$	Accessor
$id \mid$	Identifier
$n \mid$	Numeric literal
true | **false**	Boolean literal

Figure 5: The statement grammar S of the host language, the assertion language grammar A and the expression grammar e.

independent assertion means that it will always be required to hold, as it cannot appear in conditions like, e.g., $e \Rightarrow$ **tokens**$(x, 1)$. In the verifier Chalice [16, 18] they have solved a similar problem by transferring permissions to receive and obligations to send messages as part of assertions. They realise this using a concept of inhaling and exhaling assertions. A similar approach could be used here also, but seems more complex than necessary. We do not have send obligations as the number of tokens to be produced is given statically in actions. Expressions e used in assertions are equivalent to expressions of the host language, but can contain the specification constructs $\bullet(c)$, **rd**(c), **tot**(c) etc., which are not allowed in standard expressions used e.g. in actor bodies.

5 Encoding

In this section we explain how the assertions, as well as the proof rules for networks and actors, can be encoded in a guarded command language similar to Boogie [2]. The Boogie verifier carries out the rest of the proof by computing weakest preconditions of the generated code and proving them using an SMT solver.

5.1 Assertions

Our encoding is based on tracking content of network channels. We do this via a number of global map variables:

$$\mathcal{I}\colon \mathbf{ch} \to \mathbf{int} \qquad \mathcal{R}\colon \mathbf{ch} \to \mathbf{int} \qquad \mathcal{C}\colon \mathbf{ch} \to \mathbf{int} \qquad \mathcal{M}\colon (\mathbf{ch}\langle\beta\rangle, \mathbf{int}) \to \beta$$

$$
\begin{array}{lcllcl}
[\![\bullet(c)]\!] & = & \mathcal{I}[c] & [\![\textbf{prev}(c)]\!] & = & \mathcal{R}[c]-1\\
[\![\textbf{rd}(c)]\!] & = & \mathcal{R}[c] & [\![\textbf{last}(c)]\!] & = & \mathcal{C}[c]-1\\
[\![\textbf{tot}(c)]\!] & = & \mathcal{C}[c] & [\![c[e]]\!] & = & \mathcal{M}[c,[\![e]\!]]\\
[\![\textbf{rd}\bullet(c)]\!] & = & \mathcal{R}[c]-\mathcal{I}[c] & [\![\textbf{tokens}(c,e)]\!] & = & \mathcal{C}[c]-\mathcal{R}[c]=[\![e]\!]\\
[\![\textbf{tot}\bullet(c)]\!] & = & \mathcal{C}[c]-\mathcal{I}[c] & [\![\textbf{history}(c,e)]\!] & = & 0\le[\![e]\!]\wedge[\![e]\!]<\mathcal{I}[c]\\
[\![\textbf{urd}(c)]\!] & = & \mathcal{C}[c]-\mathcal{R}[c] & [\![\textbf{current}(c,e)]\!] & = & \mathcal{I}[c]\le[\![e]\!]\wedge[\![e]\!]<\mathcal{C}[c]\\
[\![\textbf{next}(c)]\!] & = & \mathcal{R}[c] & [\![\textbf{every}(c,e)]\!] & = & 0\le[\![e]\!]\wedge[\![e]\!]<\mathcal{C}[c]
\end{array}
$$

Figure 6: Encoding of assertion constructs.

The type of $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{C}$ is a map from channels to integers. $\mathcal{I}[c]$ tracks the number of tokens read on channel c when the network action started executing. $\mathcal{R}[c]$ tracks the total number of tokens read on channel c. $\mathcal{C}[c]$ tracks the number of tokens that has been produced on c. $\mathcal{M}$ is a two-dimensional map of type $(\mathbf{ch}\langle\beta\rangle, \mathbf{int}) \to \beta$. It is used to track the messages sent on channels. $\mathcal{M}[c, i]$ gives the i:th token produced on channel c. Note that the type of $\mathcal{M}$ is polymorphic and β is the datatype of the messages carried on the channel.

The assertion encoding, denoted with $[\![_]\!]$, of the most significant constructs of our assertion language are given in Figure 6. The functions $\bullet(c)$, $\textbf{rd}(c)$ and $\textbf{tot}(c)$ correspond directly to $\mathcal{I}[c]$, $\mathcal{R}[c]$ and $\mathcal{C}[c]$, respectively. The functions $\textbf{rd}\bullet(c)$, $\textbf{tot}\bullet(c)$ and $\textbf{urd}(c)$ are defined as differences between the values of $\mathcal{I}[c]$, $\mathcal{R}[c]$ and $\mathcal{C}[c]$. The standard logical operators, and other constructs of the expression language e in Figure 5, have direct correspondences in the target language and are not listed here.

5.2 Basic actors

For a basic actor A, we verify that (1) the output of each actor action $T \in A_{act}$ fullfils its postcondition and (2) that each actor action preserves the actor invariants A_{inv}. Additionally, we also verify that the actor initialisation establishes A_{inv}. The complete encodings, Action for verifying normal actor actions and Init for verifying actor initialisation, are listed in Figure 7. It is worth noting that the verification technique used is analogous to how methods are verified in traditional program verification for object-oriented languages.

The encoding Action, to verify an action T is as follows: We assume that A_{inv} hold when the execution of T starts and that the input tokens satisfy the precondition T_{pre}. We also assume that the action guard T_{grd} is satisfied, as T would not fire if T_{grd} is not satisfied.

In the encoding Initialisation we show that A_{inv} is established after executing a possible initialisation action. This verification is similar to normal actions, but we do not initially assume the invariants. Instead we check that the output of the initialisation action I satisfies its postcondition I_{post} and that I establishes A_{inv} if executed from a state where no input tokens have been consumed and no output tokens produced.

```
Init(A, I) =
  assume 0 = R[x] ∧ 0 = C[y];
  ⟦I_body⟧;
  assert ⟦I_post⟧;
  foreach e ∈ e^m {
    M[y, C[y]] := ⟦e⟧;
    C[y] := C[y] + 1;
  }
  assert ⟦A_inv⟧;
```

```
Action(A, T) =
  assume 0 ≤ R[x] ∧ 0 ≤ C[y];
  assume ⟦A_inv⟧;
  foreach k ∈ i^n {
    k := M[x, R[x]];
    R[x] := R[x] + 1;
  }
  assume ⟦T_grd⟧ ∧ ⟦T_pre⟧;
  ⟦T_body⟧;
  assert ⟦T_post⟧;
  foreach e ∈ e^m {
    M[y, C[y]] := ⟦e⟧;
    C[y] := C[y] + 1;
  }
  assert ⟦A_inv⟧;
```

Figure 7: Encoding for verification of basic actors.

5.3 Networks

The goal of the network encoding is to check for each network action that the network behaves according to the network action specification. The verification method is based on checking an arbitrary execution of the network action. The verification relies on tracing data on channels using invariants. Network invariants are required to hold between network action executions. Channel invariants should additionally hold during network action executions, i.e. executing a sub-actor should preserve the channel invariants.

The encoding for verifying networks is split into several parts. These parts are listed in Figure 8 and explained below. In the encoding and below, N is a network, $x \in N_{ip}$ is an inport of N and $y \in N_{op}$ is an outport of N. N_{inst} and N_{ch} are the actor instances of the network and network channels, respectively. Then N_{nwi} and N_{chi} are the network invariants and channel invariants, respectively. It should be emphasised that, in addition to user provided network invariants, N_{nwi} also includes a **tokens** assertion for each channel $c \in N_{ch}$ in the network. If there is no explicit **tokens** assertion for a channel c provided by the user, the verifier adds an implicit **tokens**$(c, 0)$ assertion. N_{subi} is the i/o invariants of the sub-actors, which have been proven locally and can be used as assumptions here. N_{frules} is the firing rules of every sub-actor in the network. We further assume that T is the network action for which we want to verify N and that T has the pattern $x\colon n \Longrightarrow y\colon m$, precondition T_{pre} and postcondition T_{post}.

The network encoding consists of the following parts:

- Initialisation We check that the network initialisation establishes $N_{nwi} \cup N_{chi}$. This is done by assuming that we start from a state with empty channel histories. We then update the buffers according to the initialisation actions of every sub-actor, assume the sub-invariants N_{subi}, and assert $N_{chi} \cup N_{nwi}$.

```
Initialisation(N) =
  foreach c ∈ N_ch
    { assume I[c] = 0 ∧ R[c] = 0 ∧ C[c] = 0; }
  foreach a ∈ N_act { Init(a) }
  assume ⟦N_subi⟧;
  assert ⟦N_chi⟧ ∧ ⟦N_nwi⟧;
  foreach f ∈ N_frules { assert ¬⟦f⟧; }

Input(N, T) =
  assume ⟦N_subi⟧ ∧ ⟦N_chi⟧ ∧ C[x] − I[x] < n;
  C[x] := C[x] + 1;
  assume ⟦T_pre⟧;
  assert ⟦N_chi⟧;

Output(N, T) =
  assume ⟦N_subi⟧ ∧ ⟦N_chi⟧ ∧ C[x] − I[x] = n;
  assume ⟦T_pre⟧;
  foreach f ∈ N_frules { assume ¬⟦f⟧; }
  assert ⟦T_post⟧;
  R[y] := R[y] + m;
  I := R;
  assert ⟦N_chi⟧ ∧ ⟦N_nwi⟧;

Compatibility(N, A, t) =
  assume ⟦N_subi⟧ ∧ ⟦N_chi⟧;
  assume n_t ≤ C[x_t] − R[x_t];
  foreach k ∈ i^{n_t} {
    k := M[x, R[x_t]];
    R[x_t] := R[x_t] + 1;
  }
  assume ⟦t_grd⟧;
  assert ⟦t_pre⟧;
  havoc A_var;
  assume ⟦t_post⟧;
  foreach e ∈ e^{m_t} {
    M[y_t, C[y_t]] := ⟦e⟧;
    C[y_t] := C[y_t] + 1;
  }
  assume ⟦A_inv⟧;
  assert ⟦N_chi⟧;
```

Figure 8: Encoding for verification of networks.

Finally, we also assert that no sub-actor action is enabled from the initial network state, i.e. that each $f \in N_{frules}$ is falsified. This ensures that the state is stable in the sense that N cannot make progress from this state without receiving additional input. The encoding $\mathsf{Init}(a)$ stands for executing the initialisation of the sub-actor a. Essentially, this includes updating the output channels of the network to contain possible tokens specified by a's initialisation action and assuming a's i/o invariants.

- Compatibility We check that the inter-connected sub-actors are compatible, i.e. that the invariants implies the preconditions of each sub-actor action, and that executing any sub-actor preserves N_{chi}. A separate Compatibility proof obligation is generated for each action of every sub-actor. The encoding is as follows for a sub-actor A and action t with pattern $x_t\colon [i^{n_t}] \Longrightarrow y_t\colon [e^{m_t}]$. The invariants $N_{chi} \cup N_{subi}$ are assumed. It is then assumed that there are at least n_t tokens on the input channel x_t, $n_t \leq \mathcal{C}[x_t] - \mathcal{R}[x_t]$. The input tokens are then assigned to identifiers and the action guard t_{grd} is assumed. We then assert the precondition t_{pre}. After this we havoc (non-deterministically assign any type correct value) to the sub-actor state variables. This is done because executing t can change the value of the state variables. We then assume the sub-action postcondition t_{post} and assign the tokens defined in the output pattern to the output channel y. After this we assume the invariants of the sub-actor, A_{inv}, and assert that N_{chi} is preserved. It should be noted that references to inports and outports of the sub-actor, i.e. x_t and y_t, are renamed to refer to the corresponding channels of N.

- Input We check that N_{chi} is preserved when new network input conforming to T_{pre} is received on inport x. This is done by incrementing the value of $\mathcal{C}[x]$. As we consider a finite window, an assumption, $\mathcal{C}[x] - \mathcal{I}[x] < n$, that the amount of input tokens received so far is less than that specified in the action pattern is made.
- Output We check that correct network output has been produced and that the network is returned to a state conforming to N_{nwi} if the following conditions hold: (1) On the input x, n tokens satisfying T_{pre} has been received. (2) No sub-actor can be fired, i.e., each $f \in N_{frules}$ is falsified. (3) All sub-actor actions preserve N_{chi}, as checked in Compatibility. The encoding is as follows. We assert that these assumptions imply that m tokens satisfying T_{post} has been produced on output y. We then mark the output tokens as read by updating $\mathcal{R}[y]$ and assign $\mathcal{I}$ values of $\mathcal{R}$. This models that the network action execution is completed. Note that this is the only update of $\mathcal{I}$ in the encoding and that it ensures that it contains the number of read tokens after the network action execution. Finally, we assert $N_{chi} \cup N_{nwi}$ and check that we have the correct amount of tokens on each channel in the same way as in the Initialisation case.

It is worth noting the similarity between our encoding of networks and how loops are verified using classical Hoare logic. In fact, our network encoding could be expressed as a loop, with the loop body consisting of a nondeterministic branch for each Compatibility check. The channel invariants would then be the loop invariants.

6 Invariant generation

To decrease the number of invariants that must be provided by the user, we aim to automatically generate actor invariants when possible. In this section we present a general method for generating invariants for all stateless static actors. However, this method can also be applied to stateful actors, if the state variables are transformed to feedback loops for the actor.

For an example of the invariants we aim to generate, consider the actor Add in Figure 1. We want to find invariants expressing the relation between input tokens and output tokens for this actor. The following invariants fullfil this:

$$\mathbf{tot}(out) = \mathbf{rd}(in1) \wedge \mathbf{tot}(out) = \mathbf{rd}(in2)$$
$$\forall\ \mathbf{int}\ j \cdot 0 \leq j < \mathbf{tot}(out) \Rightarrow out[j] = in1[j] + in2[j]$$

To see how generation of invariants like above can be automated and generalised to consider any stateless static actor, consider an actor A with the following two actions:

$$\mathbf{initialize} \implies y\colon [d^r]\ \mathbf{end}$$
$$\mathbf{action}\ x\colon [i^n] \implies y\colon [e^m]\ \mathbf{guard}\ g\ \mathbf{end}$$

Here i^n is a sequence of n input variables and e^m and d^r are sequences of m respectively r functions. Hence A has an input rate n and an output rate m.

Additionally it produces r initial tokens. The **initialize** action is only run once when the actor is initialised. We can now relate the number of tokens on the inport to the number of tokens on the outport with the following invariant:

$$(n\ \mathbf{tot}(y)) = (m\ \mathbf{rd}(x)) + r \tag{1}$$

We can also express the relation between values of tokens on the output streams and values of tokens on the input streams using invariants. Such invariants take the following form:

$$\forall\ \mathbf{int}\ j \cdot \mathsf{R} \wedge \mathsf{G} \Rightarrow y[j] = \mathsf{F}(\mathsf{k}) \tag{2}$$

where

$$\begin{array}{lcl} \mathsf{R} & = & r \leq j < \mathbf{tot}(y) \wedge j \,\%\, m = \mathsf{k} \\ \mathsf{F} & = & e_{\mathsf{k}}(i_0 \mapsto x[\mathsf{b}(0)], \ldots, i_n \mapsto x[\mathsf{b}(n)]) \\ \mathsf{G} & = & g(i_0 \mapsto x[\mathsf{b}(0)], \ldots, i_n \mapsto x[\mathsf{b}(n)]) \\ \mathsf{b}(\mathsf{i}) & \hat{=} & (n/m)j + \mathsf{i} - r \end{array}$$

Here the notation $x \mapsto y$ stands for substituting x with y in the expression. A separate invariant with the format in (2) is generated for each $\mathsf{k} \in 0..m-1$. Hence, k is an integer literal and not bound in the quantifier. It should also be noted that separate invariants are needed for each outport. Hence, the total amount of invariants of the format in (2) generated for an actor with p outports and action with m output tokens is $p \times m$. Additionally, the generation is repeated separately for each action.

Invariants generated according to (1) and (2) can be checked locally, by checking that the actor actions preserve the invariants. However, to verify networks it is often also useful to state properties about the initial amount of tokens on channels. For the static actor A we can provide the following invariant:

$$(n \bullet(y)) = (m \bullet(x)) \tag{3}$$

However, the value of function $\bullet$ is not defined on the actor level. Consequently, invariants involving $\bullet$ cannot be proven locally. Instead, invariants of the form in (3) are provided as channel invariants for the network where an instance of A is used.

Using invariants in the form described by (1), (2) and (3), it is possible to verify rate-static actors and networks of rate-static actors such as `Add` and `Delay` in Figure 1 without any user-provided invariants. However, network invariants still have to be provided to describe the state between network action executions.

7 Soundness

In this section we provide an informal argument for the soundness of our approach. We give our actors and actor networks semantics based Kahn Process Networks (KPN) [11]. We then show that it is sound to verify networks for a finite input sequence and that our approach guarantees deadlock freedom for networks.

Our actors and actor networks can be considered to be Dataflow Process Networks (DPN) [15]. Lee and Parks [15] has shown that DPNs can be mapped to

KPNs. In a KPN, computation is performed by a set of independent processes, which communicate through FIFO channels. A KPN process with n inputs and m outputs is defined as a function $F\colon S^n \to S^m$ mapping potentially infinite input sequences to output sequences. Here S^i denotes the set of i-tuples of sequences. KPN processes are required to be continuous in the sense reviewed below. Consider an order relation $x \sqsubseteq x'$, where x and x', meaning that x is a prefix of x'. Now consider a chain w of sequences, where each sequence is comparable using $\sqsubseteq$. Let $\sqcup w$ denote the least upper bound for w. A process F is continuous if for every such chain, $\sqcup F(w)$ exists and $F(\sqcup w) = \sqcup F(w)$ Every continuous process is also monotonic, which means that $x \sqsubseteq x' \Rightarrow F(x) \sqsubseteq F(x')$. Additionally it has also been proven that a network of monotonic processes itself forms a monotonic process [15].

Dataflow process networks are a special case of KPN [15]. A dataflow node can be considered as a pair $\{f, R\}$, where $f\colon S^n \to S^m$ is the firing function and $R \subset S^n$ is the firing rules, expressed as finite sequences. We can then define a Kahn process F based on $\{f, R\}$ as follows, where $s.s'$ denotes the concatenation of the sequences s and s':

$$F(x) = \begin{cases} f(r).F(x') & \text{if there exists an } r \in R \text{ such that } x = r.x' \\ \bot & \text{otherwise} \end{cases}$$

Theorem 1. *Actors are continuous and monotone, given that the firing rules are mutually exclusive.*

Proof. Lee and Parks [15] have shown that sufficient conditions for dataflow process, which is also what our actors are, to be continuous are that each actor is functional and that the firing rules are sequential. Functional here means that the actor does not have side effects and that the output tokens are a function of the input tokens consumed during that firing. Sequential means that the firing rules can be tested in a predefined order using only blocking reads. As we here allow actors with state they do not appear to be functional. However, Lee and Parks [15] note that actor state is just syntactic sugar for having feedback loops in the top-level network. The same conditions hence apply also to actors with state. The condition that firing rules are sequential is ensured by requiring that firing rules are mutually exclusive. □

Our verification approach is based on an inductive argument. Given a network N with a network invariant N_{nwi}, we show that N_{nwi} holds again after an arbitrary execution of the network action T. The inductive base step consists in checking that N_{nwi} is established by the network initialisation. We hence verify the network for a finite input sequence of length n described by the network action. This is sound for a network of monotone actors.

Theorem 2. *Our verification approach, using a finite input sequence, is sound for a network of monotone actors.*

Proof. To be able to verify a network N for a finite input sequence, it is required that the prefix of the output sequence does note change in response to receiving

additional input. More formally, we require that $N(x) \sqsubseteq N(x')$ for two input sequences x and x' where $x \sqsubseteq x'$. This is exactly the definition of monotonicity. $\square$

Our verification technique guarantees that verified networks are deadlock free. A deadlock occurs in a network if no actor in the network is able to fire.

Definition 3 (Deadlock). *A deadlock for a network N is a state where $\forall f \in N_{frules} \cdot \neg f$*

According to this definition, the state described by N_{nwi} can be a deadlock state. In fact, this is desired as N should not make progress from this state without receiving additional input. The deadlock should, however, be resolved at the latest when the amount of input specified in the network action(s) is received. It is possible that a deadlock could be resolved by receiving more than n tokens. Hence, the verification is more strict than necessary.

In our encoding, deadlock freedom is checked in the proof obligation Output by checking the property $N_{chi} \wedge \mathcal{C}[x] - \mathcal{I}[x] = n \wedge T_{pre} \wedge (\forall f \in N_{frules} \cdot \neg f) \Rightarrow N_{nwi}$, assuming that we only have one input port x for simplicity. More intuitively this property states that if N is in a deadlock state and behaves according to its specfication, meaning here that the channel invariants and preconditions hold, we are always in a state satisfying N_{nwi}. A network N is then deadlock free if it makes progress from a state satisfying N_{nwi} when receiving n input tokens. We now provide a proof that this is the case.

Theorem 3. *A network N with input pattern $x\colon n$ is deadlock free if $N_{chi} \wedge \mathcal{C}[x] - \mathcal{I}[x] = n \wedge T_{pre} \wedge (\forall f \in N_{frules} \cdot \neg f) \Rightarrow N_{nwi}$.*

Proof. When a network action execution starts we have $\mathcal{R}[x] - \mathcal{I}[x] = 0$ for the input channel x. We assume $\mathcal{C}[x] - \mathcal{I}[x] = n$, i.e. that all input tokens specified has been received, when the execution is finished. N_{nwi} requires $\mathcal{C}[x] - \mathcal{R}[x] = 0$, i.e. that all tokens have been consumed on x. Consequently, the value of $\mathcal{R}[x]$ must have been increased by n during the execution of N. The only possibility to increase the value of $\mathcal{R}[x]$ is by firing the sub-actor connected to x. To prove N_{nwi}, the channel invariants N_{chi} then have to ensure that progress has been made from the initial state and that no sub-actor deadlocks. $\square$

Note that if a sub-actor would deadlock on the input received we would not be able to prove N_{nwi}, as it contains a property $\mathcal{C}[c] - \mathcal{R}[c] = k_c$ for every channel. However, our verification approach allows a part of a network to be inactive in the sense that no tokens are ever read from some channels. It is also possible to construct a network which would never reach a state satisfying $\forall f \in N_{frules} \cdot \neg f$. Such a network could be seen as being in a livelock state. Proving the absence of livelocks would be analogous to proving termination of loops. We expect livelocks to be rare in practice and have chosen to not handle them in our approach.

Table 1: Summary of evaluation

Name	LOC	Instances	User Invs.	Gen. Invs.	Assertions	Boogie LOC
`SumNet`	41	2	3	15	74	622
`DataDependent`	49	2	7	4	70	604
`Nested`	107	6	16	18	151	1,190
`IIR`	52	6	1	21	118	1,327
`FIR`	80	13	5	27	326	4,609
`LMS`	198	45	9	110	3,789	47,898

8 Evaluation

We have implemented our verification approach in a prototype verification tool[1] to evaluate the usability in practice. We have successfully verified a number of existing networks and actors with both static an dynamic behaviour. The verified networks include, e.g., implementations of digital filters. The largest verified network consists of over 40 actors. Many of the examples consisted mostly of static actors for which our tool automatically generates invariants. In these cases, most of user-provided invariants were used to specify tokens on feedback loops.

The results of evaluation of 6 different networks are summarised in Table 1. The table lists the lines of code, the number of actor instances, the number of user-provided invariants needed, the number of invariants generated by the tool, the number of assertions in the resulting Boogie encoding, and the total number of lines of the Boogie encoding.

In Table 1, the network `SumNet` is the network listed in Figure 2. The network `DataDependent` is a network consisting of the data-dependent actor `Split` listed in Figure 3 and a static actor. Since the verifier cannot generate invariants for `Split` several user-provided invariants are needed to verify the network. The network `Nested` is a network nesting another network. The top-lelvel network nests the network `SumNet` and also contains the actor `Sum` given in Figure 3. The postconditions asserts that the network `SumNet` and the actor `Sum` produces equivalent output streams.

The networks `IIR`, `FIR` and `LMS` describe digital filters and are based on networks available as part of the Orcc[2] compiler infrastructure for RVC-CAL programs. All of these networks are essentially SDF networks, except that some of the actors have state. In the `IIR` and `FIR` examples the postconditions check that the output produced conforms to the difference equations describing the filters. For the `LMS` filter we only checked that the network produces one output token for each input token and that the network is returned to the state described by the network invariant.

In conclusion it can be observed in Table 1 that the number of assertions in the Boogie code is in many cases roughly proportional to the number of actor instances multiplied by the number of invariants. This is expected as channel invariants are asserted for each action of every sub-actor. Proving actor invariants locally instead

[1] `http://users.abo.fi/jonwiik/actortool/`

[2] `http://orcc.sourceforge.net/`

of expressing them as channel invariants on the network level decreases the number of assertions. Decomposing networks into a hierarchies of smaller networks also decreases the number of assertions. All of the evaluated networks, except `LMS`, were verified within 5 seconds on a modern laptop. The `LMS` example took roughly 40 seconds to verify.

9 Related work

Chalice [16, 17] is a programming language and verifier for multi-threaded object-based programs. Chalice also supports channels [18] that can be verified to be deadlock free. There, permissions to receive and obligations to send messages on channels are described in assertions. We have opted to not use this method here since it is more complex than needed in our case. We do not have send obligations as the number of tokens to be produced is given statically in actions. It could, however, be investigated as a method to support more dynamic networks. As our tool, Chalice uses Boogie [2] as a backend for verification.

A classification method for dataflow actors using satisfiability and abstract interpretation has been developed by Wipliez and Raulet [23]. They also use the SMT solver Z3 [5] as a backend. They detect a class of actors, which they call time-dependent. A time-dependent actor can react to the absence of tokens on a channel. Their method is similar to our check of mutual exclusiveness of firing rules. However, the focus of their work is on classifying actors in order to improve compile-time scheduling and not on proving functional properties.

In [4] merging of actors into composite actors is discussed. The actions of the resulting composite actor are similar to our network action, but they try to automatically determine this action, which is not always possible. The purpose of the actor merging is to decrease the number of needed runtime scheduling decisions. They do not consider functional correctness with respect to contracts. We plan to investigate integration of our contracts with this approach as future work.

Some automata-based approaches to static analysis of dataflow networks exist. In [9] a method for modular analysis of Dataflow Process Networks based on Interface Automata is presented. Interface Automata are associated with processes to specify the interface behaviour and environmental assumptions. Based on the automata they deduce properties such as deadlock freedom by checking the consistency of components and interface automaton networks. An extension to Interface Automata, named Counting Interface Automata, is presented in [22] and used for checking CAL actor compatibility. The method can capture temporal and quantitative aspects of an actors interface, as well as the token exchange rate. By composing automata they can prove behavioural type compatibility. However, neither of the approaches [9, 22] consider properties given in contracts.

There is a large amount of work, e.g. [1, 10, 12], on verification of asynchronous object programs. Asynchronous objects are similar to our actors, but there are several differences. We consider static networks of actors, while asynchronous objects can be dynamically created. Restricting ourselves to static networks simplifies reasoning and enables us to prove stronger properties fully automatically.

Formal verification of synchronous languages such as Simulink and Lustre has been studied extensively. An approach [7, 8] to verification of safety properties based on k-induction in Lustre has been developed by Hagen and Tinelli and also implemented in a tool. A contract format for Lustre is presented by Maraninchi and Morel [19]. An approach [21] to verification of Simulink models based on a translation to Boogie has also been developed. However, all of the above mentioned approaches are aimed at synchronous languages and do not support dynamic dataflow actors. Moreover, we allow verification of user-defined actors expressed in an imperative programming language. This is also possible in Simulink through an integration with MATLAB, but the verification approaches above do not handle such blocks. This paper acts as an extension to an approach [3] to contract-based verification of Simulink models. There, Simulink models are first translated to a functionally equivalent SDF representation. Trough this translation, Simulink models could also be handled using the approach presented in this paper.

10 Conclusion

In this paper we have presented an approach to specification and automated verification of dynamic dataflow networks. Our actors and networks are described in programming language similar to the RVC-CAL language, which has been standardised as part of the Reconfigurable Video Coding standard. Our verification approach ensures functional correctness with respect to contracts for actors and networks as well as deadlock freedom for networks. We have also observed that our contracts could be useful in scheduling of dataflow networks. The verification approach is based on checking the network for an arbitrary execution of finite length. Verification is carried out by encoding the dataflow networks and specifications in the guarded command language Boogie. The Boogie verifier then carries out the rest of the proof by computing weakest preconditions and submitting them to an SMT solver. We have also implemented our approach in a prototype verification tool and successfully verified a number of existing networks.

There are several directions for future work. We plan to investigate more extensively the utilisation of our contracts in scheduling of dataflow networks, e.g. by integrating them with the approach in [4]. We also plan to investigate extension of our approach to consider networks where actors can be dynamically created. However, we believe that our approach is a good first step towards fully automatic verification of dataflow actor networks.

Acknowledgements We would like to thank Johan Ersfolk for many fruitful discussions on dataflow programming.

References

[1] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 77(12), 2012.

[2] M. Barnett, B.-Y. E. Chang, R. Deline, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05*, volume 4111 of *LNCS*. Springer, 2006.

[3] P. Boström and J. Wiik. Contract-based verification of discrete-time multi-rate Simulink models. *Software & Systems Modeling*, 15(4), 2016.

[4] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silvén. Actor merging for dataflow process networks. *IEEE Trans. Signal Process.*, 63(10), 2015.

[5] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*. Springer, 2008.

[6] J. Eker. and J. W. Janneck. CAL language report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, 2003.

[7] G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach.* PhD thesis, The University of Iowa, 2008.

[8] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *FMCAD'08*. IEEE, 2008.

[9] Y. Jin, R. Esser, C. Lakos, and J. W. Janneck. Modular analysis of dataflow process networks. In M. Pezzè, editor, *FASE'03*. Springer, 2003.

[10] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. Abs: A core language for abstract behavioral specification. In *FMCO'12*, volume 6957 of *LNCS*. Springer, 2012.

[11] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing '74*, 1974.

[12] I. W. Kurnia and A. Poetzsch-Heffter. Verification of open concurrent object systems. In *FMCO'12*, volume 7866 of *LNCS*. Springer, 2013.

[13] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 100(1), 1987.

[14] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9), 1987.

[15] E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. IEEE*, 83(5), 1995.

[16] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP'09*, volume 5502 of *LNCS*. Springer, 2009.

[17] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *FSAD V*, volume 5705 of *LNCS*. Springer, 2009.

[18] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP'10*, volume 6012 of *LNCS*. Springer, 2010.

[19] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *EUROMICRO'04*. IEEE, 2004.

[20] M. Mattavelli, I. Amer, and M. Raulet. The reconfigurable video coding standard. *Signal Processing Magazine, IEEE*, 27(3), 2010.

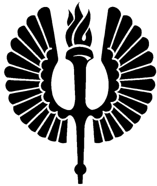
[21] R. Reicherdt and S. Glesner. Formal verification of discrete-time MATLAB/Simulink models using Boogie. In *SEFM'14*, volume 8702 of *LNCS*. Springer, 2014.

[22] E. Wandeler, J. W. Janneck, E. A. Lee, and L. Thiele. Counting interface automata and their application in static analysis of actor models. In *SEFM'05*. IEEE, 2005.

[23] M. Wipliez and M. Raulet. Classification of dataflow actors with satisfiability and abstract interpretation. *IJERTCS*, 3(1):49–69, 2012.

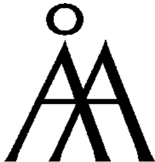
Turku Centre for Computer Science

Vesilinnantie 3, 20520 TURKU, Finland | www.tucs.fi



University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Sciences



Åbo Akademi University
- Computer Science
- Computer Engineering

ISBN 978-952-12-3486-6
ISSN 1239-1891