A Contract-Based Approach to Scheduling and Verification of Dynamic Dataflow Networks

Jonatan Wiik  
Faculty of Science and Engineering  
Åbo Akademi University  
E-mail: jonatan.wiik@abo.fi

Johan Ersfolk  
Faculty of Science and Engineering  
Åbo Akademi University  
E-mail: johan.ersfolk@abo.fi

Marina Waldén  
Faculty of Science and Engineering  
Åbo Akademi University  
E-mail: marina.walden@abo.fi

Abstract—Restricted dataflow models of computation have gained widespread adoption in the safety-critical and real-time domains. As more complex functionality is being incorporated in embedded systems, there is a need for more expressive languages while maintaining high analysability. We present a contract-based approach to specification, scheduling and verification of dynamic dataflow networks. The approach is based on finding static schedules based on contracts and utilising this information in the verification process to reduce the number of invariant annotations needed. Moreover, we show that contracts can be used to make compile-time scheduling decisions, hence, improving runtime performance.

I. INTRODUCTION

The dataflow programming paradigm has become popular in the development of embedded systems. In this paradigm, a program is a static network of concurrent actors, which communicate asynchronously through order-preserving channels. Restricted models of computation (MoC), such as Synchronous Dataflow (SDF) [1], have also gained widespread use in the safety-critical and real-time domains because of their high analysability, providing guarantees such as deadlock freedom and boundedness of channels. However, for more complex functionality, restricted MoCs like SDF are often not expressive enough. In the signal processing domain, dynamic dataflow has, however, not gained significant uptake in the safety-critical and real-time domains. As more complex functionality is being incorporated in languages such as the CAL Actor Language [2], Dynamic dataflow has, however, not gained significant uptake in the safety-critical or real-time domains because scheduling decisions have to be made at runtime. Quasi-static scheduling, e.g. [3], has been proposed as a mean of decreasing the runtime scheduling overhead, by making as many scheduling decisions as possible at compile-time.

In this paper, we present a contract-based approach to scheduling and verification of dynamic dataflow networks. Our approach is based on finding static schedules based on contracts and utilising this information in the verification process. We verify actors and networks to be functionally correct with respect to contracts as well as deadlock free. We build on previous work [4] and show that utilising schedules in the verification process significantly reduces the number of invariant annotations needed. Additionally, we show that the use of contracts can improve runtime performance by allowing scheduling decisions to be made at compile-time.

In a dataflow program, all communication is made explicit through channels, giving actors strong encapsulation with clearly defined interfaces. This also makes contracts a very suitable concept for describing actors and their interaction. In contracts, the developer can state functional properties which the actor or network should adhere to, and make implicit assumptions explicit. Our work is based on two observations: (1) The set of contracts often naturally describe the cyclic behaviour of a dataflow actor or network and can thereby be used to obtain schedules. This means that a contract-based design enables generating more efficient code, as contracts aid the search for schedules. (2) Schedules can make the verification with respect to contracts easier by significantly reducing the number of invariant annotations needed. This is because we can obtain sequential programs from the schedules and verify networks and actors using well-established verification techniques for sequential programs. It has been shown that, given some restrictions on actors, all valid schedules for a dataflow network are functionally equivalent [5]. Hence, we can verify a network for one single schedule, and conclude that the contracts also hold for any valid schedule.

Our scheduling approach is based on state space analysis using the SPIN model checker to find schedules that are optimal with respect to a cost function. As in [4], we use an encoding into the guarded command language Boogie [6] for verification. However, the encoding used here differs from [4] in that it takes into account obtained schedules. Our approach is scalable, as both the scheduling and verification are done hierarchically in a bottom-up manner. It should be noted that it is not always possible to hierarchically schedule networks containing feedback loops. Consequently, it is not possible to schedule every network that would be schedulable non-hierarchically using our approach. However, this can usually be circumvented by flattening one level in the hierarchy. The language we use to describe actors and networks is similar to the RVC-CAL [7] subset of CAL [2]. RVC-CAL has been standardised as part of an MPEG standard.

The main contributions of this paper are the following: (1)
We show that contracts can aid in the scheduling of dynamic dataflow networks and enables scheduling of actors that do not conform to any well-known statically schedulable model.

(2) We present a scalable hierarchical scheduling method based on implementation independent contracts, which can be used to obtain schedules that are optimal with respect to a cost function. (3) We present a method to verify that actors and networks are deadlock free and functionally correct with respect to their contracts by utilising schedules.

In the next section we introduce a running example to motivate our work and to concretise the concepts explained throughout the paper. We then introduce the contract language in Section III and discuss the scheduling and verification based on contracts in Section IV and Section V. In Section VI we briefly argue for the soundness of our approach. In Section VII, we present results from the experimental evaluation of our approach before we discuss related work and conclude in Section VIII and Section IX.

II. RUNNING EXAMPLE

As motivation for our work we introduce an example. Consider the network Texture illustrated in Figure 1. This network is based on a part of a fully functional MPEG-4 decoder available as part of the Orcc compiler infrastructure for RVC-CAL programs. The network consists of five actors and a subnetwork named DCReconstruction. All of the actors are complex actors, in most of the cases consisting of several actions and state variables. The coding of an MPEG-4 frame (i.e. a single picture) is based on the discrete cosine transform of 8x8 blocks. The blocks may further be predicted from surrounding blocks (intra prediction) or from previous frames (inter prediction). Which actions are executed by the different actors depends on the type of MPEG block received. The MPEG block description is received on the BTYPE port. After handling a block, the actors will always return to an initial state and, hence, have a cyclic behaviour.

For all of the actors in Texture, a specific sequence of actions is always fired for a specific MPEG block type. Hence, it is possible to obtain static schedules for each block type for each actor and also, by extension, for the entire network. However, it is often difficult to deduce that this is possible completely automatically, because of assumptions made by the developer, which are not made explicit in the source code. For instance in the Texture example, it is not stated explicitly that the bit denoting an intra block and the bit denoting inter block in a token received on BTYPE cannot be set at the same time.

Using our contracts, we can describe the behaviour at a higher and more intuitive level, as a contract acts as a form of composite action, describing a sequence of normal action firings.

III. CONTRACTS FOR ACTORS AND NETWORKS

The contract language we use is based on the language of an existing approach [4] to contract-based verification of dynamic dataflow networks. However, we extend on this work in that we use the same contract annotation, used there only for networks, also for basic actors. Below we will use the term component to mean either an actor or a network.

We consider channels in a dataflow network as streams. A channel c is then a stream \((c_0,c_1,...)\), where each \(c_i\) is the \(i\)th data token produced on that channel. An actor or network can then be considered as a function \(F\) mapping a set of input streams \(x\) to a set of output streams \(y\), i.e. \(y = F(x)\). However, in practice the output stream is computed in steps by firing actor actions.

A contract describes the response of an actor or network to a finite sequence of input tokens, i.e. how a finite window over the input stream \(x\) corresponds to a finite window over the output stream \(y\).

Definition 1 (Contract). A contract:

\[
C: \text{contract } x: n \implies y: m \\
\text{guard } G \\
\text{requires } P \\
\text{ensures } Q \\
\end{eqnarray}
\]

defines a contract with label \(C\), which specifies that, given \(n\) input tokens on port \(x\) conforming to \(G \land P\), the component outputs \(m\) tokens on port \(y\) that conforms to \(Q\).

In Def. 1, the difference between the guard \(G\) and the precondition \(P\) is that the guard \(G\) describes on which input the contract is enabled. This means that a component will not fire on an input sequence that does not satisfy \(G\). Hence, guards can be used to describe that different input tokens trigger different behaviour, or modes, in a component. Different modes may have different contract window sizes. The precondition \(P\) is required to hold whenever the contract is enabled.

Our verification technique is based on checking that the output stream of a network consists of finite windows, where each window matches a contract. The size of the window is given by the contract. For a contract \(C\) we consider a window of size \(n\) for the input stream and size \(m\) for the output stream as defined in the contract.

To write contract conditions, it is convenient to be able to refer to the tokens that are consumed on the input and produced on the output during the considered contract window. The operator \(\bullet\) is used for this purpose.

Definition 2 (Bullet). \(\bullet(c)\) is the total number of tokens that have been consumed on the channel or port \(c\) before the current contract window.

\(^1\)http://orcc.sourceforge.net
If \( \bullet \) is used in the position of an index, the argument is implicit. Hence, \( c[\bullet] \) refers to the first unread token on \( c \) at the start of the considered contract window.

As an example, consider the 4 contracts in Figure 2 for the network Texture of Figure 1. The contracts are labelled Start, Intra, InterAC, and Other, respectively. Each of the contracts specifies the behaviour of the network for different types of MPEG blocks. Start specifies the behaviour for NewVop blocks which indicate the start of a new frame, Intra specifies the behaviour for intra-coded blocks, InterAC and Other specifies the behaviour for inter-coded blocks with and without texture data, respectively. Here the size of the contract window is equal to that of an MPEG block. Based on bits set on the token received on the input BTYPE it is decided which contract is enabled. The different contracts have different sizes of contract windows, as is illustrated in Figure 2. Here the input channels contain contract windows for one Start block followed by two Intra blocks and one Other block. The dashed lines mark the boundaries of the contract windows. In the contract guards in Figure 2, BTYPE[\bullet] is used to refer to the first token consumed on BTYPE during the considered contract window. The operator & denotes bitwise conjunction, while 0x000 denotes a 12-bit integer of value 0. Note that if no window size is given for a input or output in a contract it means that the window on this port has size 0.

The subcomponents of Texture are also annotated with contracts and verified hierarchically. Consider the source code of the sub-actor DCSplit in Figure 3. It takes as inputs the coefficients of an MPEG block. An MPEG 8x8 block consists of one DC coefficient and 63 AC coefficients, of which the first one in the stream is the DC coefficient. The functionality of the actor is to output the DC coefficient on the outputport DC and the AC coefficients on the port AC. The actor consists of two actions, labelled ac and dc, respectively, and a state variable count. The coefficients are recognised by counting the number of received coefficients using the count variable.
Our approach is based on scheduling actors and networks based on contracts. A static schedule corresponding to a contract window is obtained for each contract of a component. A valid schedule is a sequence of firings, i.e., executions of actions, that returns the component to a state satisfying the contract invariants.

A schedule $S$ for a contract $C$ of a component $D$ is a sequence $S = [v_0, \ldots, v_n]$, where each $v_i$ is a tuple of an actor instance $e$ and an action $t$. We use $e.t$ to denote such a tuple. Since our approach is hierarchical, $t$ can be either an action or a contract of $e$. If $D$ is a network, $e$ is one of the subcomponents of $D$. If $D$ is an actor, $e$ always refers to an arbitrary instance of $D$ itself.

We have opted to use the SPIN$^2$ model checker for scheduling. The SPIN input language, Promela, is aimed at describing concurrent processes and includes support for e.g., communication channels. This makes the translation from our dataflow networks straightforward. CAL actor networks has been translated into Promela for the purpose of scheduling before [8]. While our translation is heavily based on this translation, a major difference is that we use contracts both to generate a valid input sequence to the network and to abstract the behaviour of components at a lower level in the network hierarchy. More precisely, we use the SMT solver Z3 [9] to generate arbitrary input and output sequences that satisfy the contracts. These sequences are then included in the Promela code to model the subcomponent.

Using state space analysis, we search for a state where all the input tokens defined by $C$ have been consumed and the component has been returned to a state satisfying the contract invariants. A schedule can then be obtained as an execution trace of a counter-example from the model checker. Typically, there is a very large number of valid schedules for a network, and a model checker may return any of these valid schedules. However, there is no control over which properties a schedule obtained in this way has. To obtain schedules with certain properties, we use the Branch-and-Bound algorithm [10] to search for schedules which are optimal with respect to a cost function. We can then find schedules that are optimal with respect to e.g., communication buffer sizes or code locality.

IV. SCHEDULING

One of the actions is enabled based on the value of $count$. The actor has here been annotated with a contract $Block$ describing the behaviour for an entire MPEG block. It states that the actor takes as input 64 tokens on port $IN$ and outputs 1 token on port $DC$ and 63 tokens on port $AC$. Note also that the actions $ac$ and $dc$ have been annotated with postconditions. These conditions describe how the state variable $count$ is updated. Because the state variable $count$ appears in the action guards, these conditions are needed to verify that an obtained schedule is correct, while abstracting the action bodies. In the actor code, $int(8)$ denotes an 8-bit integer type, while $int(x,s)$ denotes an integer of size $s$ with value $x$. The construct $prev(x)$ denotes the previous value of a state variable.

Verifying a component entails checking for each contract that executing the component with an arbitrary input sequence satisfying $G \land P$ gives an output sequence satisfying the postcondition $Q$. In addition, it is checked that invariants are preserved. There are two types of invariants: contract invariants and action invariants. A contract invariant is required to hold between contract windows. An action invariant additionally has to hold between each action firing. In the case of a network, this means firing of any sub-actor. Consider the contract invariant of $DCSplit$ in Figure 3. It requires that $count$ is equal to 0 between contract windows. For networks, there are implicit contract invariants requiring that channels are empty between contract windows if nothing else is explicitly stated. This prevents networks from buffering an infinite amount of tokens on any channel.

Our verification technique is based on an inductive proof. We consider an arbitrary contract window and show that the contract invariants are maintained when the component is executed for this window.

\begin{verbatim}
actor DCSplit int(13) IN \implies int(13) DC, int(13) AC:
  Block: contract IN:64 \implies DC:1, AC:63
    ensures DC[\(\bullet\)] = IN[\(\bullet\)]
    ensures \(\forall \ i \cdot \ 0 \leq i \land i < 63 \implies\)
    AC[i+1] = IN[i+1+1]
  end
  contract invariant count = int(0,8)
  int(8) count := int(0,8)
  dc: action IN:[ x ] \implies DC:[ x ]
    guard count = int(0,8)
    do count := int(1,8);
  end
  ac: action IN:[ x ] \implies AC:[ x ]
    guard count \neq int(0,8)
    ensures count = (prev(count) + int(1,8)) \& int(64,8)
    do count := (count + int(1,8)) & int(63,8);
  end
\end{verbatim}

Fig. 3. The source code of the actor $DCSplit$.

http://spinroot.com
an actor which outputs the absolute value of its input, implemented with two separate actions for negative and non-negative input respectively. Both actions consume one token and output one token. Clearly, this actor satisfies the contract

\[ \text{contract } x:1 \implies y:1 \text{ end, } \] but a valid static schedule for every input allowed by this contract cannot be obtained. The reason is that negative input requires a different schedule than non-negative input. If we, on the other hand, provide separate contracts for negative and non-negative input, valid schedules can be obtained using our approach. However, as we do model checking with one instance of input satisfying the contract, we will obtain a schedule for one of the cases also with the contract without guards. Consequently, we also have to check that the obtained schedule is valid for each input allowed by the contract. To do this, we verify the obtained schedule as described in the next section. This means that our scheduling approach essentially consists of two steps: We first use model checking to find a schedule which returns the component to a state satisfying the contract invariants. We then verify that the obtained schedule is valid for each input allowed by the contract.

As an example, consider again the running example in Figure 1. A static schedule is obtained for each of the contracts in Figure 2. As our scheduling technique works hierarchically in a bottom-up manner, the sub-actors and the subnetwork DCReconstruction are first scheduled based on their respective contracts. The schedules for the whole network then consist of contract firings of the subcomponents.

V. Verification

To ensure that actors and networks are correct with respect to their contracts and deadlock free, we need perform verification. We use the finite state model checker SPIN only to search for schedules for one instance of valid input, as described in the previous section. To verify correctness with respect to contracts and that the schedule is valid for all allowed inputs, we encode networks, actors and the specification constructs in the guarded command language Boogie [6]. Boogie is a so called intermediate verification language, as it acts as an intermediate step between the programming language and input to an SMT solver. The Boogie verifier computes weakest preconditions for the input program and proves them using the SMT solver [9]. This section describes our encoding into Boogie.

Our encoding has two main goals: (1) To verify that an obtained schedule is valid for any input sequence allowed by the contract, and (2) to utilise the obtained schedule to verify that a component is functionally correct with respect to its contract. A schedule \( S \) for a component \( D \) with a contract \( C \) is valid if each firing \( e.t \in S \) is fireable with any component input allowed by \( C \). We can then verify that the \( S \) fulfils \( C \) by asserting the postconditions and contract invariants after executing \( S \). Lee and Parks [5] have shown that, for a deterministic dataflow network, all valid schedules are functionally equivalent. Hence, if we ensure determinism, we can also conclude that \( D \) satisfies its contract \( C \) with any valid interleaving of firings. By verifying these properties, we also guarantee deadlock freedom, as no firing in the schedule can deadlock.

The Boogie encoding is based on the encoding proposed in [4]. However, the encoding presented here differs in that we check and utilise schedules in the verification process. The encoding is based on tracking the content of channels using a number of global map variables:

\[
\begin{align*}
I & : \text{ch} \rightarrow \text{int} & R & : \text{ch} \rightarrow \text{int} \\
C & : \text{ch} \rightarrow \text{int} & \mathcal{M} & : (\text{ch}(\beta), \text{int}) \rightarrow \beta
\end{align*}
\]

Here, \( I, R \) and \( C \) are maps from channels to integers. \( I[c] \) gives the number of tokens read on channel \( c \) when the schedule started. \( R[c] \) gives the total number of tokens read on channel \( c \). \( C[c] \) gives the total amount of tokens that has been produced on \( c \). \( \mathcal{M} \) is a two-dimensional map of type \( (\text{ch}(\beta), \text{int}) \rightarrow \beta \). It is used to track the messages sent on channels. \( \mathcal{M}[c, i] \) gives the \( i \)th message produced on channel \( c \). Note that the type of \( \mathcal{M} \) is polymorphic and \( \beta \) is the datatype of the messages carried on the channel. Based on the definitions of \( I, R, C \) and \( \mathcal{M} \), it is possible to define the encoding of the specification constructs in Boogie. For instance, \( \bullet(c) \) is encoded as \( I[c] \) in Boogie. Similarly, \( c \bullet \) is encoded as \( \mathcal{M}[c, I[c]] \). A more complete description of the encoding is given in [4].
The Boogie encoding for the most central proof obligations is described in Figure 4 in the form of sequential Boogie procedures. The proof obligation Schedule is used to check that the schedule is valid, i.e. that no step in the schedule can deadlock, that the contract postcondition is fulfilled, and that contract invariants are preserved. The proof obligation Action is used to check that actor actions fulfil their postconditions and preserve the action invariants.

In the proof obligation Schedule in Figure 4, $D$ is a component with the import $x$ and the output $y$. $C$ is then a contract

\[
C: \text{contract } x : C_x \implies y : C_y \\
\quad \text{guard } C_{\text{grd}} \\
\quad \text{requires } C_{\text{pre}} \\
\quad \text{ensures } C_{\text{post}}
\]

of $D$ and $S$ is a schedule obtained by scheduling $D$ for $C$. $D_{\text{inv}}$ is the set of invariants of $D$. The encoding is as follows: $D_{\text{inv}}$ and the locally proven invariants $e_{\text{inv}}$ of every sub-actor $e$ are assumed. To model receiving new input, $C[x]$ is increased with $C_x$ and $C_{\text{grd}} \land C_{\text{pre}}$ is assumed. After this, the following is repeated for each firing $e.t \in S$, where $v$ denotes the channel that the import of $e$ is connected to and $w$ denotes the channel that the output of $e$ is connected to: It is asserted that the firing rule and precondition of $t$ holds, $R[v]$ is then updated to model consuming $t_v$ tokens. The set of state variables $t_{\text{var}}$ is then havoced, i.e. non-deterministically assigned any type-correct value. Note that $t_{\text{var}}$ only contains the variables appearing in $t_{\text{grd}}$, i.e. the variables that determines if an actor is fireable or not. After this, $C[w]$ is updated to model producing $t_w$ tokens on $w$ and the postcondition $t_{\text{post}}$ and the invariants $e_{\text{inv}}$ of $e$ are assumed. After the steps of $S$, it is asserted that $C_y$ tokens has been produced on $y$ and that $C_{\text{post}}$ holds. Then, $I$ is updated to the value of $R$, modelling that the contract window is complete. Finally, the set invariants $D_{\text{inv}}$ is asserted.

The proof obligation Action in Figure 4 checks that action bodies fulfil their postconditions and maintain the action invariants. The encoding is analogous to how methods are verified in traditional program verification. In this proof obligation, $A$ is an actor and $T$ is an action

\[
T: \text{action } x : T_x \implies y : T_y \\
\quad \text{guard } T_{\text{grd}} \\
\quad \text{requires } T_{\text{pre}} \\
\quad \text{ensures } T_{\text{post}} \\
\quad \text{do } T_{\text{body}}
\]

of $A$. In the encoding it is assumed that there are at least $T_x$ tokens on $x$ and that $T_{\text{grd}} \land T_{\text{pre}}$ hold. $R[x]$ is then increased by $T_x$ to model consuming $T_x$ tokens, after which the action body $T_{\text{body}}$ is executed. Finally $C[y]$ is increased by $T_y$ to model producing $T_y$ tokens, after which the postcondition $T_{\text{post}}$ and the action invariants $A_{\text{inv}}$ are asserted. In addition to Schedule and Action, there are proof obligations to check that component initialisation establishes the invariants and to check mutual exclusiveness of contracts and actions.

VI. Soundness

In this section we provide an informal argument for the soundness of our approach. Our approach is based on previous results [5] showing that, for a network of continuous actors, all schedules are functionally equivalent. Intuitively, an actor is continuous if the behaviour does not change as a response to receiving additional input tokens and it does not wait for an infinite number of tokens to produce output. In essence, this means that for two actor actions $a$ and $b$, the firing rule of $a$ cannot be a prefix of the firing rule of $b$, i.e. $a$ cannot fire on a subsequence of what $b$ can fire on. To discuss this formally, we give our actors and networks semantics based Kahn Process Networks (KPN) [11].

Our actors and networks can be seen as Dataflow Process Networks (DPN) [5]. Lee and Parks [5] have 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 $F: S^n \rightarrow S^m$ 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 $x \subseteq x'$, where $x$ and $x'$ are sequences, meaning that $x$ is a prefix of $x'$. Now consider a chain $w$ of sequences, where each sequence is comparable using $\subseteq$. 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 $F(\sqcup w) = \sqcup F(w)$.

DPN is a special case of KPN [5]. An actor can be considered as a pair $\{f,R\}$, where $f: S^n \rightarrow S^m$ is the firing function and $R \subset S^n$ is the set of firing rules, expressed as finite sequences. We can then define a Kahn process $F$ based on $\{f,R\}$ as follows, where $\bot$ denotes the empty sequence and $s.s'$ 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.** An actor with mutually exclusive firing rules is continuous.

**Proof.** Lee and Parks [5] have shown that sufficient conditions for a DPN to be continuous are that each actor is functional and has sequential firing rules. 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 [5] 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 asserting that firing rules are mutually exclusive.

**Theorem 2.** For a network of continuous actors, all valid schedules are functionally equivalent.
Proof. This follows from Theorem 1, as the behaviour of a continuous actor does not change in response to receiving additional tokens. Hence, the order in which actors are executed will also not change behaviour.

We can then verify a component for one valid schedule and conclude that the network is correct for any execution of the network given that we verify that the firing rules are mutually exclusive, i.e. that the actors are continuous.

VII. Evaluation

In this section, we evaluate our approach on a number of example programs to assess the practical usability. Our approach has been implemented in a prototype open-source verification tool named Actris\(^3\). We compare our approach to the original verification approach by Wiik and Boström [4]. Additionally, we also measure the runtime speedup we gain by statically scheduling the networks based on contracts compared to making scheduling decisions at runtime.

A. Verification approach

The examples used in the evaluation consist of several realistic actors and networks, including digital filters, a ZigBee transmitter and an MPEG-4 decoder. Most of the examples are based on networks available as part of the Orcc compiler infrastructure, and are available on the Actris webpage. In the larger examples we only verify properties needed for scheduling, while the smaller examples also have more detailed postconditions describing correctness conditions on the output values.

The comparison to the approach in [4] is summarised in Table I. Here Unscheduled refers to the approach in [4], while Scheduled refers to the approach presented in this paper. The networks SumNet and DataDependent are simple examples including feedback loops and data-dependent firing rules. In the SumNet case, the same invariants are needed with both approaches. The invariants express the presence of a delay token on a network channel and properties about this token, which are required to prove the postconditions. In the example DataDependent, no invariants are needed as the network is stateless and behaviour is described using contracts for which static schedules can be obtained.

The networks FIR, IIR and LMS describe a finite impulse response filter, a infinite impulse response filter and a least mean squares filter, respectively. In these cases, the actors are mostly stateless, with invariants mainly describing the network state between contract windows. In these cases, the verification times with the two approaches are comparable, modulo extra overhead introduced by scheduling.

The ZigBee network describes a ZigBee transmitter. It consists 4 actors of which 2 are complex dynamic actors with several actions where both the number of produced and consumed tokens depend on previous input values. Our contract language is not expressive enough to completely describe this behaviour as the number of produced and consumed tokens specified in a contract has to be static. Hence, we could only verify the network for a number of different input lengths, but not for an arbitrary input length. Complete support for networks of this type, where the input rates and output rates are functions on input values, is future work. The unscheduled approach here requires many complex action invariants involving complicated relationships between state variables and channels to describe the dynamic behaviour. Using scheduling we can completely eliminate the need of action invariants. We then only need one contract invariant for each of the complex actors, describing the value of state variables between contract windows. On the other hand, one of the actors result in a very long schedule and the verification time hence increases significantly with the scheduled approach. However, the reduced number of invariants can significantly improve usability, as finding good invariants typically is a difficult task.

The remaining networks in Table I are the actors of our running example. Here Texture is the top-level network illustrated in Figure 1, DCReconstruction is a subnetwork and the rest (Addressing, DCRInvpred, DCSplit, Algo_IS, Algo_IAP, Dequant and Algo_IDCT2D) are actors. For all the components, the unscheduled approach requires a significant number of action invariants to describe the dynamic behaviour. Using the scheduled approach, no action invariants are needed. We then only need contract invariants describing the state between contract windows. For the networks, no user-provided invariants are needed, since all the channels are empty between contract windows. For some of the actors the verification time even decreases using the scheduled approach, because the schedules are short and just a few invariants need to be checked. For networks, the verification time includes scheduling each subcomponent in a bottom-up manner. Hence,

\(\text{https://jwiik.github.io/actris-verifier}\)

<table>
<thead>
<tr>
<th>Program</th>
<th>Unscheduled Invariants</th>
<th>Unscheduled Time (s)</th>
<th>Scheduled Invariants</th>
<th>Scheduled Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SumNet</td>
<td>3</td>
<td>1.7</td>
<td>3</td>
<td>5.1</td>
</tr>
<tr>
<td>DataDependent</td>
<td>6</td>
<td>1.9</td>
<td>0</td>
<td>7.4</td>
</tr>
<tr>
<td>IIR</td>
<td>2</td>
<td>1.9</td>
<td>2</td>
<td>5.3</td>
</tr>
<tr>
<td>FIR</td>
<td>4</td>
<td>2.4</td>
<td>4</td>
<td>5.8</td>
</tr>
<tr>
<td>LMS</td>
<td>9</td>
<td>14.6</td>
<td>9</td>
<td>12.1</td>
</tr>
<tr>
<td>ZigBee</td>
<td>28</td>
<td>6.8</td>
<td>2</td>
<td>250.7</td>
</tr>
<tr>
<td>Addressing</td>
<td>11</td>
<td>3.4</td>
<td>1</td>
<td>12.3</td>
</tr>
<tr>
<td>DCRInvpred</td>
<td>25</td>
<td>38.2</td>
<td>1</td>
<td>18.2</td>
</tr>
<tr>
<td>DCSplit</td>
<td>7</td>
<td>11.8</td>
<td>1</td>
<td>8.6</td>
</tr>
<tr>
<td>Algo_IS</td>
<td>8</td>
<td>3.4</td>
<td>2</td>
<td>72.0</td>
</tr>
<tr>
<td>Algo_IAP</td>
<td>5</td>
<td>2.7</td>
<td>1</td>
<td>22.3</td>
</tr>
<tr>
<td>Dequant</td>
<td>3</td>
<td>2.2</td>
<td>1</td>
<td>11.1</td>
</tr>
<tr>
<td>Algo_IDCT2D</td>
<td>2</td>
<td>3.6</td>
<td>0</td>
<td>13.4</td>
</tr>
<tr>
<td>DCReconstruction</td>
<td>13</td>
<td>4.6</td>
<td>0</td>
<td>40.6</td>
</tr>
<tr>
<td>Texture</td>
<td>29</td>
<td>5.9</td>
<td>0</td>
<td>103.9</td>
</tr>
</tbody>
</table>
most of the verification time in these cases is spent on scheduling. In Figure 5 the invariants needed to verify the actor DCSplit, given in Figure 3, using the unscheduled approach. In the invariants, \( \text{rd}_i(x) \) and \( \text{tote}_i(x) \) denote the number of tokens consumed and produced on a channel during the considered contract window, respectively. The function \( \text{id}_x \) converts an actor state variable to a channel index. This conversion is needed because we have chosen to represent integer variables as bitvectors and map indices using mathematical integers in the Boogie encoding. This provides convenient support for bitwise operations commonly used in RVC-CAL while at the same time efficiently utilising the built-in map datatype of Boogie to track channel contents.

**B. Runtime performance**

In addition to assessing the practical usability of our approach in terms of invariant annotations and verification time, we also evaluated the runtime performance gains obtained by scheduling actors and networks according to contracts. To do this, our tool generated a composite actor based on the obtained schedules. The actions of the composite actor correspond to the contracts of the component. We then used the Orcc C code generator to generate executable code based on the actor. We conducted our experiments by statically scheduling units of different sizes and measuring the performance gains on two different platforms: a modern laptop with an Intel i5 processor and an Odroid platform with an ARMv7 processor. The performance gains are due to the fact that dataflow programs are often described using many small actors. While this exposes parallelism, executing several concurrent actors on the same processor incurs a runtime overhead [8]. By merging actors into composite actors, it is possible to eliminate overhead while also enabling compilers to perform optimisations on larger code blocks.

We performed our evaluation on a MPEG-4 decoder. Our findings are summarised in Table II. Here Original refers to the case where all scheduling decisions are done at runtime. The performance is given in decoded frames per second (fps). The other columns show the relative speedups (%) compared to the original case. Each case in Table II has results for two different videos. Classifier refers optimisations done by the Orcc code generator similar to our static scheduling but only applied to static actors. To understand the remaining cases, it should be noted that the decoder consists of the two main networks Texture and Motion. The case Tex/Mot buf Sz refers to the case where these networks are separately scheduled by optimising for minimal buffer sizes. Tex/Mot loc. means that schedules are optimised for code locality (i.e. run as many actions as possible of the same actor in a row). Decoder loc. means that Texture and Motion have been further scheduled into a single actor. In DecMacro loc. the sequences of some contracts are extended to describe several MPEG blocks. This is based on properties specified in the MPEG standard stating that some types of blocks are always followed by a fixed number of blocks of the same type. This is an optimisation that would not be possible to perform automatically without stating these assumptions in contracts. To summarise the results, the Intel i5 processor with a very large cache performed better the larger unit scheduled. With the simpler ARM processor the differences between the evaluated scheduling strategies were small, but the optimal one was Tex/Mot loc. Overall, we got speedups ranging from 74.1 % to 96.4 % compared to doing all scheduling decisions at runtime.

It should be noted that our scheduling approach does not require that the entire network is statically scheduled or even schedulable. It allows scheduling units of suitable size, or up to a certain level in the network hierarchy, and then mapping these composite components to processing units. Hence, it is possible to achieve a level of parallelism suitable for the platform in question. It is also possible to leave some components unscheduled and perform the scheduling decisions for those components dynamically at runtime. Our verification approach can also fall back to the unscheduled approach of [4] for subcomponents for which a static schedule cannot be found.

**VIII. RELATED WORK**

Our work extends on an approach [4] to contract-based specification and verification of dynamic dataflow networks. We reuse the contract language presented there and our Boogie encoding is also based on this work. In [4] it is verified that a network is correct with respect to the contract for any valid schedule. In this work, we compute a concrete schedule for a contract and verify and use this schedule in the verification process. Encoding with schedules may result in larger Boogie procedures, but the the properties to be proven are generally simpler. Using schedules also decreases the number of invariant annotations needed significantly resulting in improved practical usability.

Automata-based static analysis of Dataflow Process Networks has been studied. In [12] DPNs are modularly analysed based on Interface Automata. Processes are associated with automata specifying the interface and environmental assumptions. They then deduce properties such as deadlock freedom by checking consistency of components and interface automaton networks. Counting Interface Automata, able to capture temporal and quantitative aspects of actor interfaces as well as token rates is presented in [13]. However, neither of the approaches [12], [13] consider properties given in contracts.

Formal verification of synchronous languages such as Lustre has been studied extensively. One recent contract-based ap-
proach is CoCoSpec [14]. They present a mode-aware contract language. This is similar to annotating a network with several contracts in our approach. However, CoCoSpec, as well as other work aimed at synchronous languages, do not consider asynchronous, dynamic actors.

An extensive amount of research has been done on scheduling of (dynamic) dataflow networks. An approach close to ours in that they merge actors has been presented by Boutellier et al. [8]. They use model checking to find schedules for parts of dataflow networks. Essentially, their approach tries to automatically deduce information that we make explicit in contracts. This is not always possible. Model checking has also been used to optimise schedules for e.g. SDF networks [15], [16]. Our approach enables this for a broader class of dataflow programs.

Wippliez and Raulet [17] presented classification of dataflow actors into more restricted but easier to schedule MoCs. Abstract interpretation and satisfiability is used to check whether a DPN actor can be described using a more restrictive MoC such as SDF. Classification is essentially about checking that requirements imposed by a specific MoC, holds for the actor. Actors which can be classified as static can be specified with a single contract in our approach.

Other approaches are based on describing the dynamic behavior more explicitly. Siyoum et al. [18] present a scenario-based approach where dynamic dataflow programs are implemented according to a Disciplined Dataflow Network MoC. Dynamic behavior is captured by special actors. These actors can be seen as forming contracts with the difference that these actors are actually part of the program and not independent specifications.

Actor composition does not guarantee rate consistency and deadlock freedom [19]. Tripakis et al. [19] proposed DSSF (Deterministic SDF with Shared FIFOs) profile-based methodology to handle actor composition in hierarchical dataflow networks. A similar problem is solved by Falk et al. [20] by a rule-based quasi-static scheduling approach, where static actors are clustered together to form composite actors such that the quasi-static schedule of the composite actors guarantee deadlock-freedom globally. In our approach deadlock freedom is ensured by verification based on contracts.

IX. CONCLUSION

We have presented a contract-based approach to specification, verification and scheduling of dynamic dataflow networks. In our approach, actors and networks are statically scheduled based on contracts. Using contracts, implicit assumptions can be made explicit, allowing more scheduling decisions to be made at compile-time. Moreover, schedules are used in the verification process to reduce the number of invariant annotations needed. We verify that networks and actors are deadlock free and functionally correct with respect to their contracts.

It is not always possible to schedule networks containing feedback loops hierarchically for a complete repeating period. To solve this, contract level state in combination with finer grained scheduling would be needed. We plan to investigate this as future work. In conclusion, however, our results suggest that contract-based design is useful in dataflow programming at several stages in the development process.

ACKNOWLEDGEMENTS

This work has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737494. This Joint Undertaking receives support from the European Union’s Horizon 2020 research and innovation programme, as well as Sweden, France, Spain, Italy, Finland, Czech Republic.

REFERENCES


