Contract-based specification and verification of dataflow programs

Jonatan Wiik and Pontus Boström

Åbo Akademi University, Turku, Finland
{jonatan.wiik, pontus.bostrom}@abo.fi

1 Introduction

Modern software systems are increasingly concurrent as the computational capacity of modern CPUs is improved mainly by increasing the number of processor cores. Writing software that efficiently exploits the capacity of such CPUs is hard. For this purpose new programming paradigms have been proposed. One such paradigm, which has gained a lot of attention within the signal processing domain is the dataflow paradigm. A dataflow program consists of a network of actors connected via asynchronous channels that describe the flow 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 the general case dataflow programs have to be scheduled dynamically at runtime when run on general purpose hardware, which can cause significant runtime overhead. Consequently, different techniques to reduce the number of dynamic scheduling decisions have been investigated, e.g. [3].

In this work we present an approach to specification and automatic verification of dataflow programs based on assume-guarantee reasoning. The approach is based on annotating actors and networks with contracts stating functional properties which the actor or network should adhere to. The goal of the approach is to ensure functional correctness with respect to the contracts as well as deadlock freedom for dataflow networks. The contracts can potentially also be used to specify and prove properties which can be utilised to make scheduling decisions at compile-time. The work presented is a generalisation of previous work on verification of Simulink models [2], in which Simulink models are translated to synchronous data flow [5] (SDF) networks for verification. SDF is a subset of the the dataflow programs considered here, which can be statically scheduled.

2 Dataflow programs

We here consider dataflow programs in a language similar to the CAL actor language [4]. CAL has gained recent attention within the signal processing domain, and a subset of the language, named RVC-CAL, has also been standardised by ISO/IEC MPEG as part of the Reconfigurable Video Coding standard [6].

The dataflow programs considered here consist of a set of actors communicating over order-preserving channels of infinite size. Actors, which are allowed to have state, consist of a set of actions. The actor executes by firing an eligible action. An action is eligible depending on the number of tokens available on the incoming channels, the values of the tokens and the current state of the actor. Some examples of basic actors are listed in Fig. 1a. The actor *Add* has two input ports *x1* and *x2* and one output port *y*. The actor has one action, which reads one token from each of the input ports and outputs the sum of the read tokens. The actor *Delay*

```
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

actor Split() int x ==> int q, int u:
  action x:[i] ==> q:[i]
    guard i < 0
  end
  action x:[i] ==> u:[i]
    guard i >= 0
  end
end

actor Sum() int x ==> int y:
  inv sum == (0::y)[last]

  int sum := 0;

  action x:[i] ==> y:[sum]
    sum := sum+i;
  end
end
```

(a)

```
network Sum() int x ==> int y:
  entities
    a = Add();
    d = Delay(0);
  end
  structure
    x1: x --> a.x1; x2: d.y --> a.x2;
    y: a.y --> y; z: a.y --> d.x;
  end

  inv delay(x2,1)
  inv x2[next] == (0::y)[last]

  action x:[i] ==> y:[(0::y)[last]+i] end

  chinv total(y) == read(x1)
  chinv total(y) == read(x2)
  chinv total(z) == read(x1)
  chinv total(z) == read(x2)
  chinv total(x2) == read(z)+1
  chinv forall int i . 0 <= i && i < total(y)
    ==> y[i] == x1[i]+x2[i]
  chinv forall int i . 0 <= i && i < total(z)
    ==> z[i] == x1[i]+x2[i]
  chinv forall int i . 1 <= i && i < total(x2)
    ==> x2[i] == z[i-1]
end
```

(b)

Figure 1: (a) Implementations of some basic actors. (b) A dataflow network consisting of two basic actors of type *Add* and *Delay*.

delays the data on the input channel with one token. The actor has a special initialisation action outputting an initial token on the output port. This action is run only once when the actor is initialised. The actor *Split* is an example of a data-dependent actor, as its behaviour depends on the value of the incoming token. It outputs negative input tokens on port q and non-negative input tokens on port u. The actor *Sum* is an example of an actor with state. It accumulates the sum of the inputs it has received. A network consisting of one *Add* actor and one *Delay* actor is listed in Fig. 1b. It implements the same functionality as the actor *Sum* in Fig. 1a. It should be noted that the syntax used for specifying the network is not standard CAL. For instance RVC-CAL uses an XML-based format for networks. Networks in pure CAL do not have actions, but we use them here to describe the intended behaviour of the network. Hence a network action describes how the network should react when it receives input tokens.

3 Verification

For verification we encode the dataflow programs in the intermediate verification language Boogie [1]. An actor is verified by checking each action against its contract. Action contracts state preconditions and postconditions relating tokens on the input and output channels. For an

actor with invariant I and an action with precondition P, guard condition G and a postcondition Q, we need to check that Q and I hold after executing the action, assuming that P, G and I hold before executing the action.

Verification of a network means checking that the network has the behaviour described by its network actions. To do this we need to express the relations between data on the channels in the network. We call these relations channel invariants. In the example in Fig. 1b, channel invariants are declared using the **chinv** keyword. Channel invariants are required to hold during the execution of a network action, while network invariants, declared using the **inv** keyword, are required to hold before and after a network action is executed, but not necessarily while the action is executed. The channel invariants provided in the example in Fig. 1b express relations both between the number of tokens on the channels and between the data on the different channels. We for instance have the property that the total number of tokens (written as well as read) on the channel y during execution of the action is equal to the number of read tokens on channel *x1*. This type of properties are needed to ensure that the amount of tokens specified in the network action is available on the output channels after executing the action. Channel invariants also relate data on the different channels, for instance that the i:th token on channel y should be equal to the sum of the i:th tokens on channels *x1* and *x2*.

A network can be verified to be correct with respect to its contract in the following way: Assume that we have a network with network invariant I and an action with postcondition Q. Additionally assume that $F_1, \ldots, F_n$ are the firing conditions of each action $A_1, \ldots, A_n$ of every actor in the network and that $C_1, \ldots, C_m$ are the channel invariants of the network. Assuming that $C_1, \ldots, C_m$ hold, we check that $C_1, \ldots, C_m$ hold again after executing any actor A_i for which F_i evaluates to true. We additionally also check that the postcondition of the network action holds when no actor can be fired: $\neg F_1 \wedge \cdots \wedge \neg F_n \wedge C_1 \wedge \cdots \wedge C_m \Rightarrow Q \wedge I$.

4 Conclusions

We have outlined an approach to contract-based specification and verification of dataflow programs. The work is still in progress. To make the approach more usable in practice it would be important to infer as many as possible of the channel invariants for a network. We plan to investigate automatic inference of invariants for special classes of actor networks. We also plan to investigate the use of contracts to aid the compile-time scheduling of dataflow programs.

References

[1] 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.

[2] P. Boström and J. Wiik. Contract-based verification of discrete-time multi-rate Simulink models. *Software & Systems Modeling*, 2015.

[3] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silvén. Actor merging for dataflow process networks. *IEEE Transactions on Signal Processing*, 63(10):2496–2508, 2015.

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

[5] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987.

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