Specification and Automated Verification of Dynamic Dataflow Networks

Jonatan Wiik   Pontus Boström

Faculty of Science and Engineering, Åbo Akademi University, Finland

Introduction

- Modern software systems are increasingly concurrent and distributed
  - Many-core processors, heterogenous systems etc.
- Developing software that efficiently exploits the capacity of such platforms is hard
- Within the signal processing domain, the dataflow paradigm has received a lot of attention as a possible solution
  - A program is modelled as a static network of actors, communicating exclusively via asynchronous FIFO channels.
  - Actors can execute concurrently whenever the required data is available on the incoming channels.
  - Provides a high level of abstraction, enabling synthesis of hardware or software implementations for different platforms from the same description.

Introduction

- We present an approach to contract-based specification and verification of *dynamic* dataflow programs (DDF)
    - In DDF, the number of tokens consumed/produced by an actor can depend on the value of input tokens and the actor state
    - More general model of computation than, e.g., synchronous dataflow (SDF) and synchronous languages (Lustre, Simulink)
- Fully automatic verification of correctness properties given as contracts as well as deadlock freedom
    - Only aided by annotations in the source code
- Verification by encoding in the intermediate verification language Boogie
    - Efficient verification conditions for the SMT solver Z3
    - Actors contain imperative code that can be directly translated to Boogie

Introduction

- Main contributions:
  - A method to specify the behaviour of networks based on the reaction of the network to individual tokens
  - An encoding of actors, networks and their specifications in a guarded command language
  - A method to automatically generate invariants needed for verification of a common type of actors

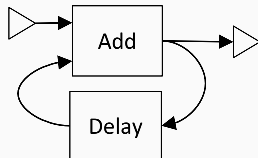
Outline

Dataflow programs

- An actor is a stateful operator consisting of a set of: *inports*, *outports*, *state variables* and *actions*
- An actor evolves by executing (firing) enabled actions. When an action fires, it consumes tokens on its input ports, updates its state and outputs tokens on its output ports
  - The number of tokes consumed/produced on a port during a firing is called *rate*
- We use a language closely resembling the RVC-CAL dialect of the CAL Actor Language to describe our actors

Dataflow programs – example

- A network consisting of one instance of actors `Add` and `Delay` respectively, and computes the accumulated sum of the input
- `Add` has two inports `x1` and `x2` and consumes one token on each inport and outputs their sum each time it fires
- `Delay` delays its input with one token. It has a special **`initialize`** action which is only fired once when the actor is initialized

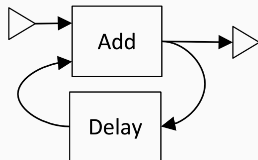



```
actor Add int x1, int x2 ⟹ int y:
  action x1:[i], x2:[j] ⟹ y:[i+j] end
end

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

Dataflow program – example

- Network declarations consist of an **`entities`** block, declaring the actor instances of the network, and a **`structure`** block, declaring the interconnection of the actor instances




```
network SumNW int in ⟹ int out:
  entities
    add = Add();
    del = Delay(0);
  end
  structure
    a: in ⟶ add.x1;
    b: del.y ⟶ add.x2;
    c: add.y ⟶ out;
    d: add.y ⟶ del.x;
  end
end
```

Network contract

$$\texttt{\textbf{contract}}\ x\colon n \Longrightarrow y\colon m\ \ \textbf{guard}\ G\ \ \texttt{\textbf{requires}}\ P\ \ \textbf{ensures}\ Q\ \ \textbf{end}$$

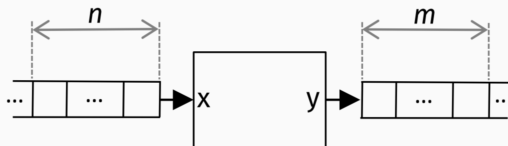
specifies that, given n input tokens on port x conforming to $G \wedge P$, the network outputs m tokens on port y that conforms to Q.

The contract is enabled when G is satisfied. The precondition P is required to hold whenever the contract is enabled.

A network can have more than one contract, but the guards have to be mutually exclusive.

Specification – networks

- The contract describes the response of the network to a finite sequence of n input tokens
- Networks are verified for a finite window. The length of the window is given by the contract.
- Network channels are required to be empty between windows, if nothing else is specified in invariants
- Hence, we essentially verify that the network has a periodic behaviour, where the period (window) is described by the contracts



Specification – networks

- In specifications we can refer to the i:th token produced on a channel using `c[i]`
- Useful to refer to the tokens produced and consumed during a contract window

Bullet

$\bullet(c)$ is the total number of tokens that had been consumed on the channel c before the current contract window

Specification – networks

```
network SumNW int in ⟹ int out:
  contract in:1 ⟹ out:1
    requires 0 ≤ in[•]
    ensures in[•] ≤ out[•]
    ensures 0 < •(out) ⇒ out[•] = out[• − 1] + in[•]
  end
  ...
end
```

- The contract specifies that the network takes 1 input token on `in` and outputs 1 token on `out`
- `in[•]` refers to the consumed input token and `out[•]` refers to the produced output token

- `SumNet` contains a loop, and an initial token has to be produced to avoid deadlock. There will always be an unread token between contract windows.
- We can use the construct **`tokens(c,n)`** in an invariant to state that there has to be n unread tokens on channel c
  - An invariant **`tokens(c,0)`** is implicitly assumed for each channel c not mentioned explicitly
- There are two types of invariants: *network invariants* declared using keyworkd **`invariant`** and *channel invariants* declared using keyword **`chinvariant`**
  - Network invariants are required to hold between contract windows
  - Channel invariants are additionally required to hold between sub-actor firings

```
network SumNW int in ⟹ int out:
  ...
  invariant tokens(b,1)
  chinvariant b[0] = 0 ∧ 0 ≤ b[•]
  ...
end
```

Specification – actors

- Actors can have state and/or dynamic rates
- Preconditions and postconditions can be provided as annotations to actions
- Restrictions on the state variables can be expressed using invariants, which actions are required to maintain
- In the example, an invariant is needed to prove the action postcondition

```
actor Sum int x ⟹ int y:
  int sum;
  invariant 0 ≤ sum
  initialize ⟹ do sum := 0; end
  action x:[i] ⟹ y:[sum]
    requires 0 ≤ i
    ensures i ≤ sum
    do sum := sum + i;
  end
end
```

Specification – actors

- To prove that interconnected actors in a network are compatible, the relationship between input tokens and output tokens need to be expressed as invariants
- Can be done using channel invariants on the network level, but we can also provide such information as actor invariants which are locally proven
  - The actor invariants are locally proven on the actor level
  - Can be used directly as assumption on the network level above
- **tot**(x) and **rd**(x) is the total amount of tokens produced and consumed on a channel, respectively

```
actor Sum int x ⟹ int y:
  ...
  invariant tot(y) = rd(x)
  invariant ∀ int j · every(y,j,1) ⇒ y[j] = y[j−1]+x[j]
  ...
end
```

Specification – data-dependent actors

For a data-dependent actor, the amount of tokens consumed or produced is not known at compile time.

```
actor Split int input ⟹ int positive, int negative:
  invariant rd(input) = tot(positive) + tot(negative)
  action input:[i] ⟹ positive:[i] guard i ≥ 0 end
  action input:[i] ⟹ negative:[i] guard i < 0 end
end
```

For data-dependent actors it is often challenging to give invariants that completely describes the relationship between input and output tokens.

Boogie encoding

- The Boogie encoding is based on tracking content of network channels trough a number of global map variables

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

- $\mathcal{I}[c]$ is the number of tokens consumed on c when the contract window started
- $\mathcal{R}[c]$ is the total number of consumed on c
- $\mathcal{C}[c]$ is the number of tokens produced on c
- $\mathcal{M}[c, i]$ is the value of the i:th token produced on c
- Based on these definitions, we can define the encoding $\llbracket _ \rrbracket$ of the most common specification constructs:
    - $\llbracket \bullet(c) \rrbracket \equiv \mathcal{I}[c]$
    - $\llbracket \mathtt{rd}(c) \rrbracket \equiv \mathcal{R}[c]$
    - $\llbracket \mathtt{tot}(c) \rrbracket \equiv \mathcal{C}[c]$
    - $\llbracket \mathtt{tokens}(c, e) \rrbracket \equiv \mathcal{C}[c] - \mathcal{R}[c] = \llbracket e \rrbracket$
    - ...

Boogie encoding

- The resulting encoding is a set of proof obligations in the form of imperative Boogie procedures
- For actors:
  - The initialisation establishes the invariants
  - Each action establishes its postcondition and maintains the invariants
- For networks:
  - Network initialisation establishes the channel and network invariants
  - Executing the network for a contract window establishes the contract and maintains channel and network invariants
  - Channel invariants are maintained by sub-actor firings and when new network input is received

- Want to decrease the number of user-provided invariants needed
- We show how this can be done for static-rate actors
- Consider again the actor **Add**: We want to find invariants describing the relationship between input tokens and output tokens for this actor

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

- The following invariants can describe the behaviour of this actor:

$$\mathtt{tot}(y) = \mathtt{rd}(\mathit{x1})$$
$$\mathtt{tot}(y) = \mathtt{rd}(\mathit{x2})$$
$$\forall\ \mathtt{int}\ j \cdot 0 \leq j < \mathtt{tot}(y) \Rightarrow y[j] = \mathit{x1}[j] + \mathit{x2}[j]$$

Invariant generation

- Consider a general static-rate actor with of the form described below, where each d_i and e_i are expressions:

$$\texttt{initialize} \implies y\colon [d_1, \ldots, d_r]\ \textbf{end}$$
$$\texttt{action}\ x\colon [i_1, \ldots, i_n] \implies y\colon [e_1, \ldots, e_m]\ \textbf{guard}\ g\ \textbf{end}$$

- The following invariants can describe the behaviour of this actor:

$$(n \times \texttt{tot}(y)) = (m \times \texttt{rd}(x)) + r \tag{1}$$
$$\forall\ \texttt{int}\ j \cdot \mathsf{R} \wedge \mathsf{G} \Rightarrow y[j] = \mathsf{E}_k \tag{2}$$

  where
  - R restricts the range of j
  - G is the guard g transformed to refer to positions on the input channel x
  - E_k is the output expression e_k transformed to refer channel positions on x
- We get an invariant of form (1) for each inport-outport pair. We get an invariant of the form in (2) for each output expression e_k.
- This can also be generalised to actors with state by considering the state as a feedback channel to the actor.

Evaluation

- The verification approach has been implemented in a prototype tool and evaluated on a number of examples to assess the practical usability
- Some of the examples were realistic networks, e.g. digital filters, based on examples available as part of the Orcc compiler infrastructure RVC-CAL programs

Evaluation

- For networks including mostly static-rate actors (e.g. `SumNet`, `IIR`, `FIR`) the number of user-provided invariants were fairly low
- For networks containing dynamic actors, invariants have to be manually provided (e.g. `ZigBee`)
- `LMS` took roughly 25s to verify and other examples were verified in less than 10s
- Decomposing networks into smaller sub-networks is important for scalability

Name	LOC	Actors	User invs.	Gen. invs.	Asserts	Boogie LOC
`SumNet`	41	2	3	17	31	585
`DataDep`	49	2	6	5	61	619
`Nested`	107	6	15	26	101	1,184
`IIR`	52	6	2	23	28	1,245
`FIR`	86	13	5	33	68	4,506
`LMS`	213	45	9	128	312	46,300
`ZigBee`	399	6	36	24	288	2,814

Conclusions

- Presented an approach to specification and automated verification of dataflow programs.
- Ensures functional correctness with respect to contracts as well as deadlock freedom.
- Verification based on checking networks for a window of finite length given by the contract.
- The approach has been implemented in a prototype tool and has been used to verify a number of pre-existing programs.
- All required invariants are automatically generated for static-rate actors.