Contract-based Specification and Verification of Dataflow Programs

Jonatan Wiik Pontus Boström

Åbo Akademi University, Finland

Nordic Workshop on Programming Theory 2015

Introduction

- Modern software systems are increasingly concurrent and distributed
  - Increased number of processer cores, heterogenous systems etc.
- Developing software that efficiently exploits the capacity of such platforms is hard
  - New programming paradigms have been proposed to solve this problem
- Within the signal processing domain, the dataflow paradigm has received a lot of attention
  - A dataflow program consists of a network actors, communicating exclusively via asynchronous order-preserving channels
  - Exploits parallelism, as actors can execute concurrently whenever the required data is available on the incoming channels

Introduction

- Dataflow programs have a high level of abstraction, enabling synthesis of hardware or software implementations from the same description
- Actors can easily be mapped to different processing units
  - There are typically fewer processing units than actors, which means that actors have to be scheduled
- Scheduling has to be done dynamically in the general case, which can cause significant runtime overhead
  - Different techniques to decrease the number of runtime scheduling decisions have been investigated

Introduction

- We present an approach to contract-based specification and verification of dataflow programs
  - Contracts refer to functional specifications, consisting of preconditions and postconditions
- Fully automatic verification of correctness properties given as contracts as well as deadlock freedom
  - Only aided by annotations in the source code
  - Based on translation to the Boogie intermediate verification language
- Contracts can also be used to express and prove properties that can be utilised in compile-time scheduling
  - The use of contracts can improve both functional quality and performance

Outline

Dataflow programs

- We consider dataflow programs in a language similar to the CAL actor language
- CAL is a domain-specific language for dataflow programs
  - Has received much recent attention within the signal processing domain
  - A subset of CAL has been standardised by ISO/IEC MPEG as part of the Reconfigurable Video Coding standard

Dataflow programs

- CAL actors are allowed to have state and consist of a set of actions
  - An actor executes by firing an eligible action
  - An action is eligible depending on the tokens available on the inputs and the current state
  - Actions consume/produce a predefined amount of tokens on the inputs/outputs when firing
  - Actions written in a simple imperative programming language
- Dataflow programs considered here consist of hierarchical networks of actors
  - Networks are also actors

Basic actors

```
actor Add() int x1, int x2 ==> int y:
  action x1:[i], x2:[j] ==> y:[i+j]
end
```

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

```
actor Sum() int x ==> int y:
  int sum := 0;
  action x:[i] ==> y:[sum] do
    sum := sum+i;
  end
end
```

Basic actors

```
actor Add() int x1, int x2 ==> int y:
  action x1:[i], x2:[j] ==> y:[i+j]
end
```

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

```
actor Sum() int x ==> int y:
  int sum := 0;
  action x:[i] ==> y:[sum] do
    sum := sum+i;
  end
end
```

Basic actors

```
actor Add() int x1, int x2 ==> int y:
  action x1:[i], x2:[j] ==> y:[i+j]
end
```

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

```
actor Sum() int x ==> int y:
  int sum := 0;
  action x:[i] ==> y:[sum] do
    sum := sum+i;
  end
end
```

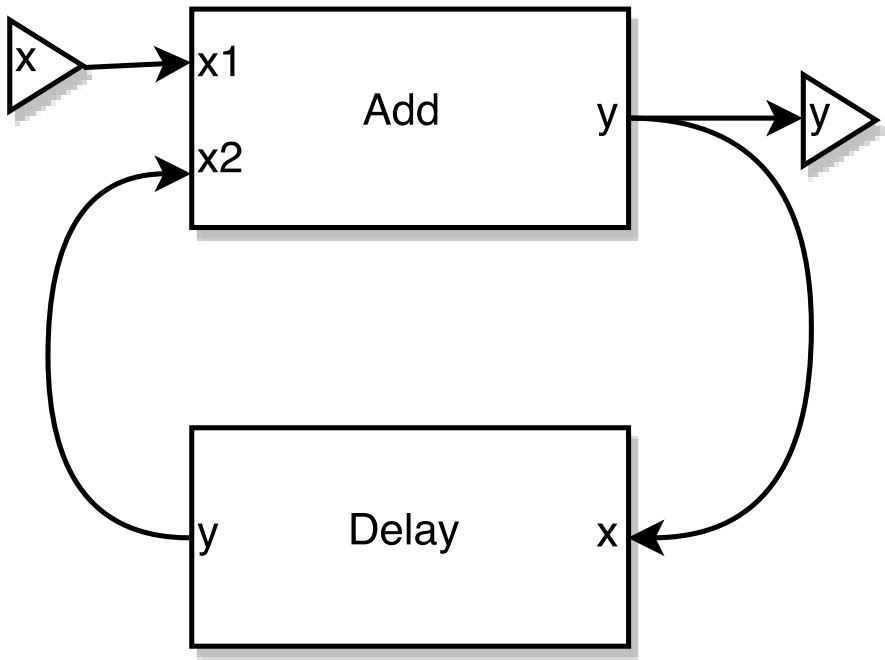
Data-dependent actors

- Data-dependent actors: the amount of tokens consumed or produced depends on the input values

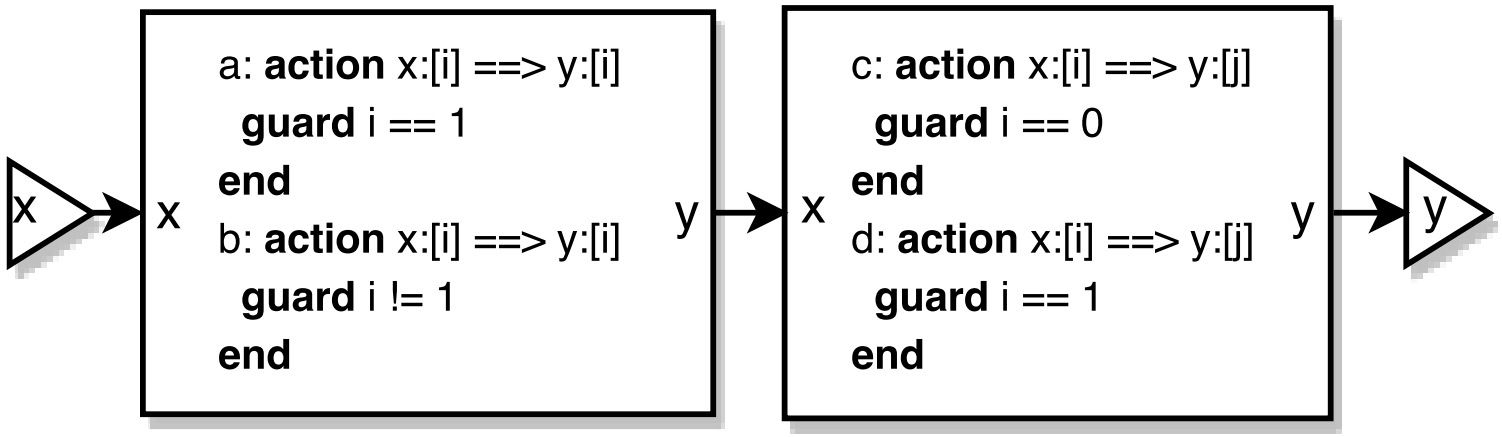
```
actor Split() int x ==> int q, int u:
  action x:[i] ==> q:[i]
    guard i < 0
  end
  action x:[i] ==> u:[i]
    guard i >= 0
  end
end
```

Actor networks

```
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
end
```



Example



- Without any restrictions on the input, the program might deadlock
- Deadlock is avoided if x is either 0 or 1. Need a precondition: `x == 0 || x == 1`
- This type of information is also useful for compile-time scheduling: Can conclude that action *a* will always be followed by action *d* and action *b* will be followed by action *c*

Specification – basic actors

- Actors and networks annotated with contracts
- Actions are annotated with preconditions and postconditions
  - Standard **requires** and **ensures** annotations
- Actor invariants for actors with state

```
actor Sum() int x ==> int y:
  inv 0 <= sum
  int sum := 0;
  action x:[i] ==> y:[sum]
    requires 0 <= i
    ensures sum == old(sum)+i
  do
    sum := sum+i;
  end
end
```

Specification – networks

- To specify networks, we give them actions with preconditions and postconditions as for basic actors
  - Networks in pure CAL do not have actions, but we use them here to describe the intended behaviour of the network
- We provide *network invariants*, which should hold before and after executing a network action
- Additionally we also provide *channel invariants*
  - Used to express the relationship between data on different channels in the network
  - Required to hold during execution of a network action
- If nothing else is specified in the network invariants, network channels should be empty after executing a network action

Specification – networks

```
network Sum() int x ==> int y:
  ...
  action x:[i] ==> y:[(0::y)[last]+i] end
  inv delay(x2,1)
  inv x2[next] == (0::y)[last]

  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
```

Specification – networks

```
network Sum() int x ==> int y:
  ...
  action x:[i] ==> y:[(0::y)[last]+i] end
  inv delay(x2,1)
  inv x2[next] == (0::y)[last]

  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
```

Verification

- Automatic verification with respect to contracts of both basic actors and networks
- Verification based on translation to the Boogie language
- Boogie is a program verifier and programming language
  - Designed to bridge the gap between programs with specifications and verification conditions suitable for an SMT solver
  - The Boogie verifier generates verification conditions and discharges them with the Z3 SMT solver

Verification – basic actors

- Each action of a basic actor is verified separately
  - Assume that the invariant, guard and precondition hold
  - Check that the postcondition and invariant hold after executing the action

```
actor A() int x ==> int y:
  inv I
  action x:[i] ==> y:[j]
    guard G
    requires P
    ensures Q
  do
    S;
  end
end
```

```
assume I;
assume G;
assume P;
trans(S);
assert Q;
assert I;
```

Verification – networks

- Networks can be verified by checking that firing any eligible actor in the network preserves the channel invariants
- For a network with network invariants I, channel invariants C and postcondition Q, where $F_1, \ldots, F_n$ are the firing rules of all actions $A_1, \ldots, A_n$ of every actor in the network we:
  - Assume that C hold and check that C hold again after executing any action A_i for which F_i evaluates to true
  - If no action can be fired, the postcondition Q and the network invariants I must hold: $\neg F_1 \wedge \ldots \neg F_n \wedge C \implies Q \wedge I$

Verification – networks

```
network N() int x ==> int y:
  entities A_1, A_2, ... end
  structure ... end
  inv I
  chinv C
  action x:[i] ==> y:[j]
    requires P
    ensures Q
  end
end
```

```
assume I;
assume P;
assert C;
```

```
assume C;
assume F_i;
actor(A_i);
assert C;
```

```
assume C;
assume ¬F_1 ∧ ··· ∧ ¬F_n;
assert I;
assert Q;
```

Verification – networks

```
network N() int x ==> int y:
  entities A_1, A_2, ... end
  structure ... end
  inv I
  chinv C
  action x:[i] ==> y:[j]
    requires P
    ensures Q
  end
end
```

assume I;
assume P;
assert C;

assume C;
assume F_i;
$actor(A_i)$;
assert C;

assume C;
assume $\neg F_1 \wedge \cdots \wedge \neg F_n$;
assert I;
assert Q;

Verification – networks

```
network N() int x ==> int y:
  entities A_1, A_2, ... end
  structure ... end
  inv I
  chinv C
  action x:[i] ==> y:[j]
    requires P
    ensures Q
  end
end
```

```
assume I;
assume P;
assert C;
```

```
assume C;
assume F_i;
actor(A_i);
assert C;
```

```
assume C;
assume ¬F_1 ∧ ··· ∧ ¬F_n;
assert I;
assert Q;
```

Future work

- Tool support, complete support for the CAL language
- Invariant inference
  - To make the approach usable in practice, channel invariants should be inferred automatically whenever possible
  - We plan to investigate automatic inference of invariants for special classes of actors
- Dynamic networks
  - The approach is now limited to static networks
  - We plan to investigate if the approach can be extended to also consider networks that are created dynamically

Conclusions

- Presented an approach to specification and verification of dataflow programs
- Actors are specfied by giving actions preconditions and postconditions
- Verification by translation to the Boogie language
- Contracts useful both to ensure correctness and for compile-time scheduling