Interactive Verification of Synchronous Systems

Manuel Gesell
gesell@cs.uni-kl.de

Embedded Systems Group
University of Kaiserslautern

November 7^{th}, 2014

Hypothesis

The synchronous Model of Computation (MoC) does not prohibit the application or adoption of verification techniques and tools developed for other MoCs.

Hypothesis

The synchronous Model of Computation (MoC) does not prohibit the application or adoption of verification techniques and tools developed for other MoCs.

Hypothesis

The synchronous Model of Computation (MoC) does not prohibit the application or adoption of verification techniques and tools developed for other MoCs.

Outline

1. What is a Model of Computation

Hypothesis

The synchronous Model of Computation (MoC) does not prohibit the application or adoption of verification techniques and tools developed for other MoCs.

Outline

1. What is a Model of Computation
2. Why interactive verification

Hypothesis

The synchronous Model of Computation (MoC) does not prohibit the application or adoption of verification techniques and tools developed for other MoCs.

Outline

1. What is a Model of Computation
2. Why interactive verification
3. Interactive Verification on Source-Code Level

Hypothesis

The synchronous Model of Computation (MoC) does not prohibit the application or adoption of verification techniques and tools developed for other MoCs.

Outline

1. What is a Model of Computation
2. Why interactive verification
3. Interactive Verification on Source-Code Level
4. Interactive Verification on Guarded-Action Level

Hypothesis

The synchronous Model of Computation (MoC) does not prohibit the application or adoption of verification techniques and tools developed for other MoCs.

Outline

1. What is a Model of Computation
2. Why interactive verification
3. Interactive Verification on Source-Code Level
4. Interactive Verification on Guarded-Action Level
5. Representation of Synchronous Systems for Verification

Hypothesis

The synchronous Model of Computation (MoC) does not prohibit the application or adoption of verification techniques and tools developed for other MoCs.

Outline

1. What is a Model of Computation
2. Why interactive verification
3. Interactive Verification on Source-Code Level
4. Interactive Verification on Guarded-Action Level
5. Representation of Synchronous Systems for Verification
6. Conclusion

Outline

Definition: Guarded Action

A guarded action $(\gamma \Rightarrow \alpha)$ consists of

- a Boolean guard γ and
- an atomic action α.

Example

$$\left\{ \begin{array}{l} \mathsf{true} \Rightarrow \mathtt{x=(z==0)} \\ \mathsf{true} \Rightarrow \mathtt{y=z>0} \\ \mathsf{true} \Rightarrow \mathtt{z=z+1} \end{array} \right\}$$

Sequential Model of Computation

Definition: Interleaved Guarded Actions (IGAs)

An interleaved guarded action $(\gamma \Rightarrow \alpha)$ consists of

- a Boolean guard γ and
- a set of atomic assignments α.

Behavior of IGAs (subset of Dijkstra's Guarded Commands)

- execution of a single enabled guarded actions

Sequential Model of Computation

Definition: Interleaved Guarded Actions (IGAs)

An interleaved guarded action $(\gamma \Rightarrow \alpha)$ consists of

- a Boolean guard γ and
- a set of atomic assignments α.

Behavior of IGAs (subset of Dijkstra's Guarded Commands)

- execution of a single enabled guarded actions

? What happens if more than one guarded action is enabled

Sequential Model of Computation

Definition: Interleaved Guarded Actions (IGAs)

An interleaved guarded action $(\gamma \Rightarrow \alpha)$ consists of

- a Boolean guard γ and
- a set of atomic assignments α.

Behavior of IGAs (subset of Dijkstra's Guarded Commands)

- execution of a single enabled guarded actions
- ? What happens if more than one guarded action is enabled
  - the first (found) is taken
  - use alphabetic order
  - choose one non-deterministically

  ⋮

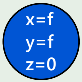
Sequential Model of Computation

Behavior of IGAs (subset of Dijkstra's Guarded Commands)

- execution of a single enabled guarded actions

Example

$$\left\{ \begin{array}{l} \mathtt{true} \Rightarrow \mathtt{x=(z==0)} \\ \mathtt{true} \Rightarrow \mathtt{y=z>0} \\ \mathtt{true} \Rightarrow \mathtt{z=z+1} \end{array} \right\}$$



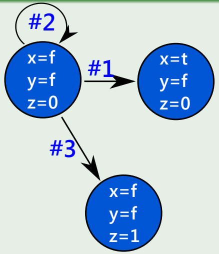
Sequential Model of Computation

Behavior of IGAs (subset of Dijkstra's Guarded Commands)

- execution of a single enabled guarded actions

Example

$$\left\{ \begin{array}{l} \text{true} \Rightarrow \texttt{x=(z==0)} \\ \text{true} \Rightarrow \texttt{y=z>0} \\ \text{true} \Rightarrow \texttt{z=z+1} \end{array} \right\}$$



Concurrent Model of Computation

Definition: Asynchronous Guarded Actions (AGAs)

An asynchronous guarded action $(\gamma \Rightarrow \alpha)$ consists of

- a Boolean guard γ and
- a set of atomic assignments α.

Behavior of AGAs

- execution of a subset of enabled guarded actions

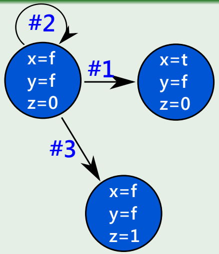
Concurrent Model of Computation

Behavior of AGAs

- execution of a subset of enabled guarded actions

Example

$$\left\{ \begin{array}{l} \texttt{true} \Rightarrow \texttt{x=(z==0)} \\ \texttt{true} \Rightarrow \texttt{y=z>0} \\ \texttt{true} \Rightarrow \texttt{z=z+1} \end{array} \right\}$$



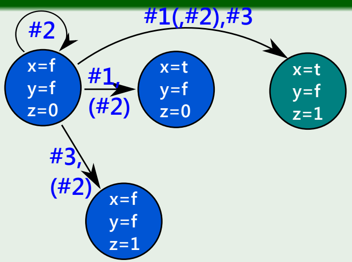
Concurrent Model of Computation

Behavior of AGAs

- execution of a subset of enabled guarded actions

Example

$$\left\{ \begin{array}{l} \text{true} \Rightarrow \texttt{x=(z==0)} \\ \text{true} \Rightarrow \texttt{y=z>0} \\ \text{true} \Rightarrow \texttt{z=z+1} \end{array} \right\}$$



Synchronous Model of Computation

Definition: Synchronous Guarded Actions (SGAs)

A synchronous guarded action $(\gamma \Rightarrow \alpha)$ consists of

- a Boolean guard γ and
- a single atomic immediate/delayed assignment α.

Behavior of SGAs

- execution of all enabled guarded actions in parallel

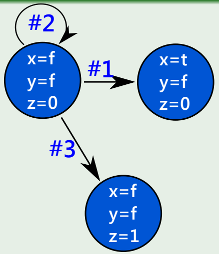
Synchronous Model of Computation

Behavior of SGAs

- execution of all enabled guarded actions in parallel

Example

$$\left\{ \begin{array}{l} \texttt{true} \Rightarrow \texttt{x=(z==0)} \\ \texttt{true} \Rightarrow \texttt{y=z>0} \\ \texttt{true} \Rightarrow \texttt{z=z+1} \end{array} \right\}$$



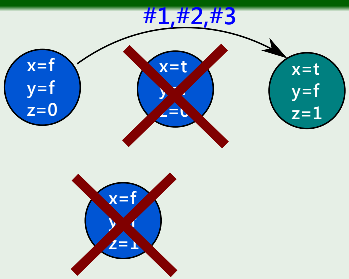
Synchronous Model of Computation

Behavior of SGAs

- execution of all enabled guarded actions in parallel

Example

$$\left\{ \begin{array}{l} \text{true} \Rightarrow \texttt{x=(z==0)} \\ \text{true} \Rightarrow \texttt{y=z>0} \\ \text{true} \Rightarrow \textbf{next}\texttt{(z)=z+1} \end{array} \right\}$$



Synchronous Model of Computation

- execution is divided into a sequence of reactions steps
- computation of WCRT
- deterministic behavior
- formal verification techniques available (i.e. model checking)
- languages: Quartz, Esterel, Signal, Lustre, etc.

Outline

Why Interactive Verification?

Model Checking

- available for synchronous languages
- fully automatic
- suffers from state-space explosion problem

Interactive Verification

- semi-automatic
- requires additional information (like invariants)
- allows abstraction from data structures and data-types
- decomposes proof goals

Combining interactive techniques with model checking is desired.

Outline

Hoare Calculus

nothing : $$\frac{}{\{\Phi\}\ \texttt{nothing}\ \{\Phi\}}$$

assign : $$\frac{}{\{[\Phi]_x^\tau\}\ x = \tau\ \{\Phi\}}$$

sequence : $$\frac{\{\Phi_1\}\ S_1\ \{\Phi_2\} \quad \{\Phi_2\}\ S_2\ \{\Phi_3\}}{\{\Phi_1\}\ S_1; S_2\ \{\Phi_3\}}$$

conditional : $$\frac{\{\sigma \wedge \Phi\}\ S_1\ \{\Psi\} \quad \{\neg\sigma \wedge \Phi\}\ S_2\ \{\Psi\}}{\{\Phi\}\ \texttt{if}(\sigma)\ S_1\ \texttt{else}\ S_2\ \{\Psi\}}$$

loop : $$\frac{\{\sigma \wedge \Phi\}\ S\ \{\Phi\}}{\{\Phi\}\ \texttt{while}(\sigma)\ S\ \{\neg\sigma \wedge \Phi\}}$$

weaken : $$\frac{\models \Phi_1 \rightarrow \Phi_2 \quad \{\Phi_2\}\ S\ \{\Phi_3\} \quad \models \Phi_3 \rightarrow \Phi_4}{\{\Phi_1\}\ S\ \{\Phi_4\}}$$

Hoare Calculus

nothing :
$$\frac{}{\{\Phi\}\ \texttt{nothing}\ \{\Phi\}}$$

assign :
$$\frac{}{\{[\Phi]_x^\tau\}\ x = \tau\ \{\Phi\}}$$

sequence :
$$\frac{\{\Phi_1\}\ S_1\ \{\Phi_2\} \quad \{\Phi_2\}\ S_2\ \{\Phi_3\}}{\{\Phi_1\}\ S_1; S_2\ \{\Phi_3\}}$$

conditional :
$$\frac{\{\sigma \wedge \Phi\}\ S_1\ \{\Psi\} \quad \{\neg\sigma \wedge \Phi\}\ S_2\ \{\Psi\}}{\{\Phi\}\ \texttt{if}(\sigma)\ S_1\ \texttt{else}\ S_2\ \{\Psi\}}$$

loop :
$$\frac{\{\sigma \wedge \Phi\}\ S\ \{\Phi\}}{\{\Phi\}\ \texttt{while}(\sigma)\ S\ \{\neg\sigma \wedge \Phi\}}$$

weaken :
$$\frac{\models \Phi_1 \rightarrow \Phi_2 \quad \{\Phi_2\}\ S\ \{\Phi_3\} \quad \models \Phi_3 \rightarrow \Phi_4}{\{\Phi_1\}\ S\ \{\Phi_4\}}$$

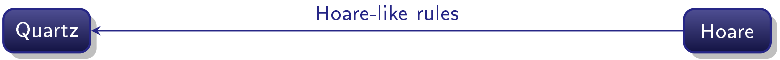
A Hoare calculus for Quartz - Naive Approach

0. synthesis to sequential code to use the classical Hoare calculus
   - destroys syntax
   - merges control and data flow
   - combines all loops to a single one
1. defining Hoare rules for each statement
2. split the verification process into two stages

A Hoare calculus for Quartz - Naive Approach

0. synthesis to sequential code to use the classical Hoare calculus
   - destroys syntax
   - merges control and data flow
   - combines all loops to a single one
1. defining Hoare rules for each statement
2. split the verification process into two stages

A Hoare calculus for Quartz - Idea 1



0. synthesis to sequential code to use the classical Hoare calculus
1. defining Hoare rules for each statement
   - local reasoning not possible
     $\Rightarrow$ rules collect assignments/identify macro step
   - using two-stage Hoare-like rules
   - default reaction (and other Quartz specific issues)
2. split the verification process into two stages

A Hoare calculus for Quartz - Idea 1

0. synthesis to sequential code to use the classical Hoare calculus
1. defining Hoare rules for each statement
    - local reasoning not possible
      $\Rightarrow$ rules collect assignments/identify macro step
    - using two-stage Hoare-like rules
    - default reaction (and other Quartz specific issues)
2. split the verification process into two stages

A Hoare calculus for Quartz - Idea 2

0. synthesis to sequential code to use the classical Hoare calculus
1. defining Hoare rules for each statement
2. split the verification process into two stages
   1. transformation that concentrates the macro-step behavior
   2. reason about code in SSTA normal form

Defining a Hoare Calculus for Quartz - Idea 2

STA Rule

$$\frac{}{\{\left[\ldots\left[[\Phi]^{\pi_1,\ldots,\pi_n}_{y'_1,\ldots,y'_n}\right]^{\tau_n}_{x_n}\ldots\right]^{\tau_1}_{x_1}\}\ (x_1,\ldots,x_m).(y_1,\ldots,y_n) = (\tau_1,\ldots,\tau_m).(\pi_1,\ldots,\pi_n)\ \{\Phi\}}$$

Pause Rule

$$\frac{}{\{\left[[\ldots\Phi\ldots]^{\tau_1\ldots\tau_n}_{i_1,\ldots,i_n}\right]^{y'_1\ldots y'_n}_{y_1\ldots y_n}\}\ \mathbf{pause}\ \{\Phi\}}$$

Defining a Hoare Calculus for Quartz - Idea 2

STA Rule

$$\frac{}{\left\{\left[\ldots\left[[\Phi]^{\pi_1,\ldots,\pi_n}_{y'_1,\ldots,y'_n}\right]^{\tau_n}_{x_n}\ldots\right]^{\tau_1}_{x_1}\right\}\ (x_1,\ldots,x_m).(y_1,\ldots,y_n) = (\tau_1,\ldots,\tau_m).(\pi_1,\ldots,\pi_n)\ \{\Phi\}}$$

Pause Rule

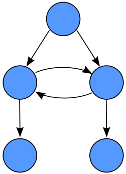
$$\frac{}{\left\{\left[[\ldots\Phi\ldots]^{\tau_1\ldots\tau_n}_{i_1,\ldots,i_n}\right]^{y'_1\ldots y'_n}_{y_1\ldots y_n}\right\}\ \textbf{pause}\ \{\Phi\}}$$

source-code transformation

- all assignment collected in a synchronous tuple assignment
- must not invent additional variables
- parallel operator must be removed

removing the parallel operator

- similar to eliminating gotos in sequential programs
- proved the impossibility without adding additional variables
- problem: representing two parallel loops may introduce an irreducible sub graph




```
module AshcroftManna(nat{3} ?i, nat{2} !o){
  {                                    {
    bool x;o = 1;                        do {
    while(!x){                             w5: pause;
      while(i==0&!x){                    } while(i!=2);
        w1: pause;               ||      w6: pause;
        o = 1;                           w7: pause;
      }                                  x = true;
      w2: pause;                       }
      o = 1;
      if(!x){
        while(i==1 & !x){
          w3: pause;
          o = 0;
        }     w4: pause;
        o = 0;
    }}
  }
}
```

module CounterExample (){

```
{ while(...){          }      { while (...){   }
{   pause;             }      {   pause;       }
{   pause;             }  ||  {   pause;       }
{   pause;             }      { }              }
{ }                    }      { pause;         }
}
```

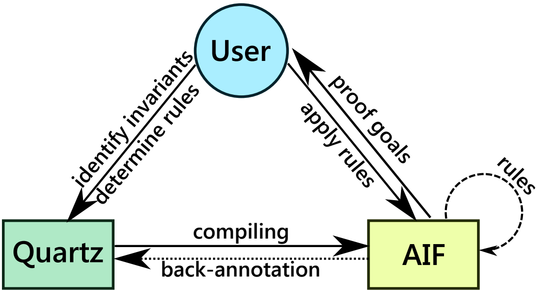
Contribution

Interactive Verification on Source-Code Level

- negative result: no Hoare rules for Quartz
- verification of SSTA programs [GS12a]
- Theorem 'Ashcroft Manna' $\Rightarrow$ transformation not possible

Outline

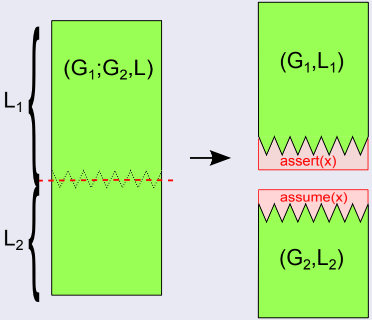
Approach


User
identify invariants
determine rules
proof goals
apply rules
rules
Quartz
compiling
back-annotation
AIF

Advantage of two Representations

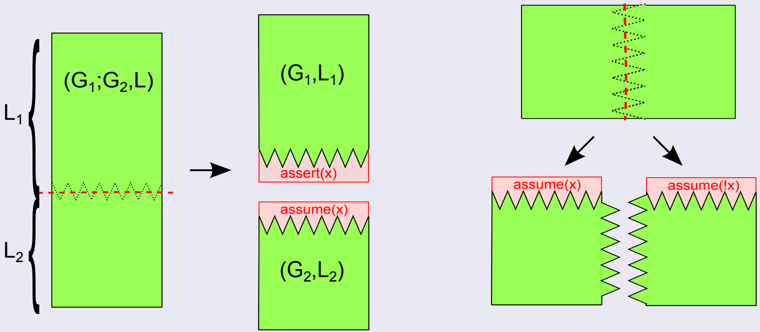
- implicit usage of SSTA normal form
- Quartz is better human readable
- AIF format is better machine-readable
- just a few simple rules are required
- schizophrenia and causality are dealt with at compile time
- flexible decompositions of proof goals (independent of syntax)
- compilation to guarded actions is verified
- AIF file contains assumption and assertions $\Rightarrow$ proof goal

Idea for the Rules

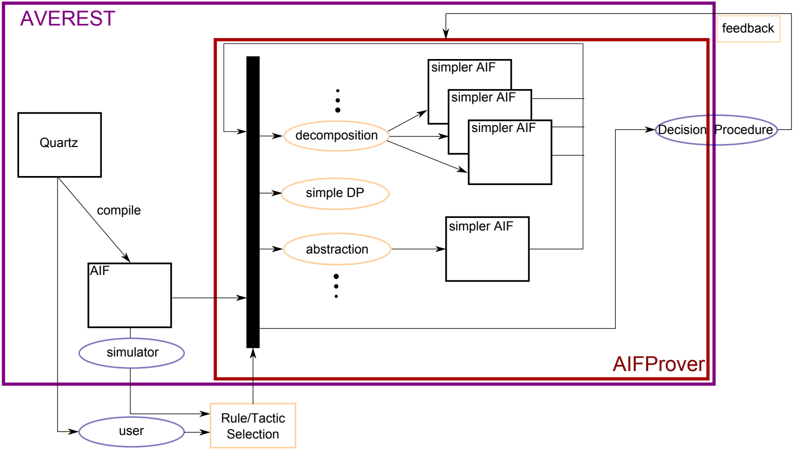


- decomposition of a sequence
- rules decompose proof goals
  $\Rightarrow$ rules split AIF files
- rules insert assumptions and assertions into AIF files

Idea for the Rules



Interactive Verification Framework - AIFProver



Contribution

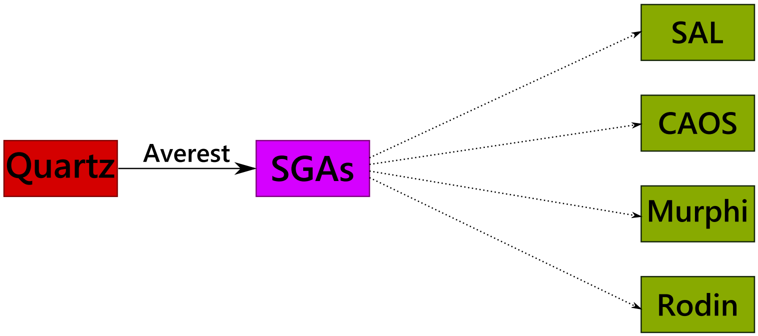
Interactive Verification on Guarded-Action Level

- interactive verification rules [GS12b]
- extension for temporal logic LTL
- rules for module calls [GS13b]
- rules for preemption context [GMS13]
- the AIFProver tool [GS12b, GS13b, GS13a]

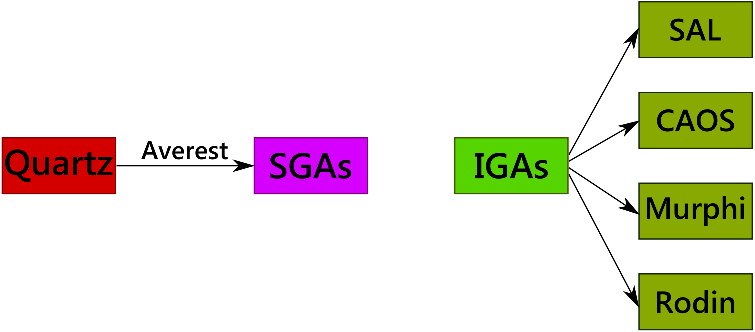
Outline

1. What is a Model of Computation?
2. Why Interactive Verification
3. Interactive Verification on Source-Code Level
4. Interactive Verification on Guarded-Action Level
5. Representation of Synchronous Systems for Verification
6. Conclusions

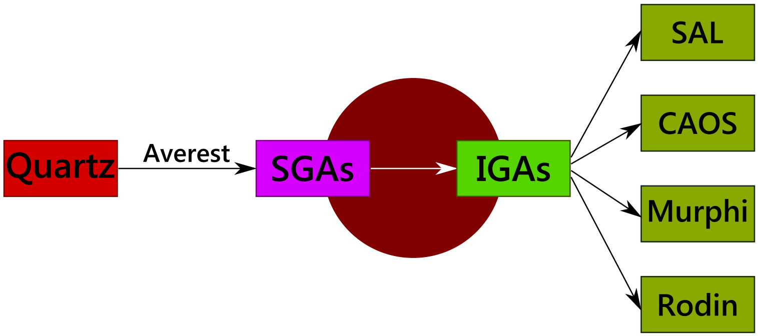
Representing Quartz in other MoCs



Representing Quartz in other MoCs



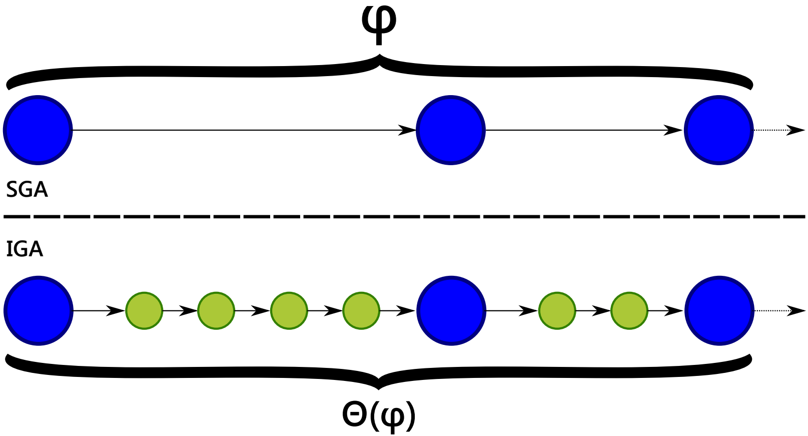
Representing Quartz in other MoCs



Summary of Identified Problems

Problems to Solve

- assignment behavior
- preservation of determinism
- execution order
- no serialization
- reaction step behavior
- temporal behavior


φ
SGA
IGA
Θ(φ)

Example

$$\left\{ \begin{array}{l} \text{true} \Rightarrow \texttt{x=(z==0)} \\ \text{true} \Rightarrow \texttt{y=z>0} \\ \text{true} \Rightarrow \texttt{\textbf{next}(z)=z+1} \end{array} \right\}$$

Example

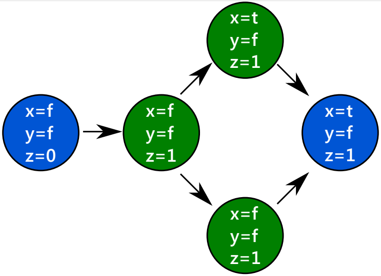
$$\left\{\begin{array}{l} \mathsf{true} \Rightarrow \texttt{x=(z==0)} \\ \mathsf{true} \Rightarrow \texttt{y=z>0} \\ \mathsf{true} \Rightarrow \textbf{next}\texttt{(z)=z+1} \end{array}\right\}$$

$$\left\{\begin{array}{l} \neg\texttt{x}_v \Rightarrow \left\{\begin{array}{l} \texttt{x=(z==0)} \\ \texttt{x}_v = \mathsf{true} \end{array}\right. \\ \neg\texttt{y}_v \Rightarrow \left\{\begin{array}{l} \texttt{y=z>0} \\ \texttt{y}_v = \mathsf{true} \end{array}\right. \\ \texttt{x}_v \wedge \texttt{y}_v \Rightarrow \left\{\begin{array}{l} \texttt{x}_v = \mathsf{false} \\ \texttt{y}_v = \mathsf{false} \\ \texttt{z} = \texttt{z+1} \end{array}\right. \end{array}\right\}$$

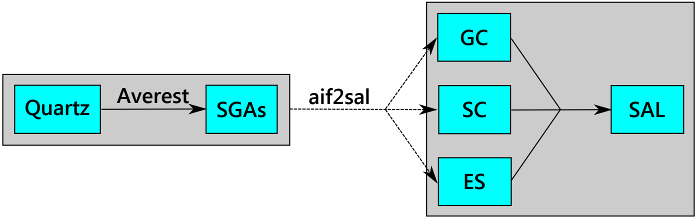
Example

$$\left\{\begin{array}{l} \mathsf{true} \Rightarrow \texttt{x=(z==0)} \\ \mathsf{true} \Rightarrow \texttt{y=z>0} \\ \mathsf{true} \Rightarrow \textbf{next}\texttt{(z)=z+1} \end{array}\right\}$$

$$\left\{\begin{array}{l} \neg \texttt{x}_v \Rightarrow \left\{\begin{array}{l} \texttt{x=(z==0)} \\ \texttt{x}_v = \mathsf{true} \end{array}\right. \\ \neg \texttt{y}_v \Rightarrow \left\{\begin{array}{l} \texttt{y=z>0} \\ \texttt{y}_v = \mathsf{true} \end{array}\right. \\ \texttt{x}_v \wedge \texttt{y}_v \Rightarrow \left\{\begin{array}{l} \texttt{x}_v = \mathsf{false} \\ \texttt{y}_v = \mathsf{false} \\ \texttt{z} = \texttt{z+1} \end{array}\right. \end{array}\right\}$$



Evaluation



P	#SGA	*GC*	*SC*	*ES*
ABRO	7	0.11	0.06	**0.05**
ABROM[M=13]	29	4.27	7.92	**3.27**
AuntAgatha	2	0.12	**0.07**	0.09
VendingMachine	23	1.14	0.15	**0.07**
LightControl	36	1.79	0.44	**0.40**
MinePumpController	42	7.60	0.22	**0.09**
RSFlipFlop	7	53.51	**1.18**	**1.18**
MemoryController	41	407.95	42.93	**3.42**
IslandTrafficControl	83	504.64	62.40	**1.94**
FischerMutex	60	0.14	0.22	**0.09**
Dekker	28	0.63	0.21	**0.17**
SingleRowNIM	15	0.06	**0.04**	**0.04**
PigeonHole	1	**0.01**	0.05	0.05
Queens	1	0.29	**0.19**	0.20
MagicSquare	29	**1.83**	65.67	9638.84

Contribution

Representation of Synchronous Systems for Verification

- by interleaved guarded actions [GS13c]
- reuse of algorithms presented in [GMS13]
- in SRI's Symbolic Analysis Laboratory [GBS14]
- The tool aif2sal [GS13c, GBS14]

Outline

Contribution

Interactive Verification of Synchronous Systems

- interactive verification techniques on source-code level:
  - verification of SSTA programs [GS12a]
  - Theorem 'Ashcroft Manna' $\Rightarrow$ transformation not possible
- interactive verification techniques on guarded-action level:
  - interactive verification rules [GS12b]
  - extension for temporal logic LTL
  - rules for module calls [GS13b]
  - rules for preemption context [GMS13]
  - the AIFProver tool [GS12b, GS13b, GS13a]
- representation of synchronous systems for verification
  - by interleaved guarded actions [GS13c]
  - in SRI's Symbolic Analysis Laboratory [GBS14]
  - The tool aif2sal [GS13c, GBS14]

Bibliography I

Gesell, M., F. Bichued, and K. Schneider: Using different representations of synchronous systems in SAL.
In MBMV, 2014.

Gesell, M., A. Morgenstern, and K. Schneider: Lifting verification results for preemption statements.
In SEFM, 2013.

Gesell, M. and K. Schneider: A Hoare calculus for the verification of synchronous languages.
In PLPV, 2012.

Gesell, M. and K. Schneider: Interactive verification of synchronous systems.
In MEMOCODE, 2012.

Bibliography II

Gesell, M. and K. Schneider: An interactive verification tool for synchronous/reactive systems.
In MBMV, 2013.

Gesell, M. and K. Schneider: Modular verification of synchronous programs.
In ACSD, 2013.

Gesell, M. and K. Schneider: Translating synchronous guarded actions to interleaved guarded actions.
In MEMOCODE, 2013.

Thank you for the attention! Any Questions?

Proof Rules

Given the AIF file ($\mathcal{G}$) and the labels ($\mathcal{L}$) of a Quartz program

Overall Task

- decompose proof goal $(\mathcal{G}, \mathcal{L})$ to $(\mathcal{G}_1, \mathcal{L}_1) \ldots (\mathcal{G}_n, \mathcal{L}_n)$
- insert assumptions and assertions representing the execution history and the user's knowledge of the program

Proof Rules

Given the AIF file ($\mathcal{G}$) and the labels ($\mathcal{L}$) of a Quartz program

Overall Task

- decompose proof goal $(\mathcal{G}, \mathcal{L})$ to $(\mathcal{G}_1, \mathcal{L}_1) \ldots (\mathcal{G}_n, \mathcal{L}_n)$
- insert assumptions and assertions representing the execution history and the user's knowledge of the program

Rule Example

$$\text{CaseDistinction} \quad \frac{\begin{array}{c}(\mathcal{G} \cup \{\text{enter}\,(\mathcal{G}, \mathcal{L}) \Rightarrow \mathtt{assume}(\sigma)\}, \mathcal{L}) \\ (\mathcal{G} \cup \{\text{enter}\,(\mathcal{G}, \mathcal{L}) \Rightarrow \mathtt{assume}(\neg\sigma)\}, \mathcal{L})\end{array}}{(\mathcal{G}, \mathcal{L}) \Leftarrow \mathrm{BoolCases}(\sigma)}$$

Rules for Temporal Logic

Given the AIF file ($\mathcal{G}$) and the labels ($\mathcal{L}$) of a Quartz program

Overall Task

- extending proof goal with specification
- decompose $(\mathcal{G}, \mathcal{L}) \models \varphi$ to $(\mathcal{G}_1, \mathcal{L}_1) \models \varphi_1 \ldots (\mathcal{G}_n, \mathcal{L}_n) \models \varphi_n$
- insert assumptions and assertions representing the execution history and the user's knowledge of the program

Rules for Temporal Logic

Idea

$$(\{(\gamma \Rightarrow \mathbf{assert}(\alpha))\} \cup \mathcal{G}, \mathcal{L}) \equiv (\mathcal{G}, \mathcal{L}) \models \mathsf{G}(\gamma \rightarrow \alpha)$$

Rule Example

$$\text{UnrollAlways} \quad \frac{(\mathcal{G}, \mathcal{L}) \models \varphi \wedge \mathsf{XG}\varphi}{(\mathcal{G}, \mathcal{L}) \models \mathsf{G}\varphi \Lleftarrow \mathrm{UnrollAlways}()}$$

Other Rules

$$\frac{(\mathcal{G},\mathcal{L}) \models \psi}{(\mathcal{G},\mathcal{L}) \models [\varphi \ \mathsf{U}\ \psi]} \qquad \frac{(\mathcal{G},\mathcal{L}) \models \psi}{(\mathcal{G},\mathcal{L}) \models [\varphi \ \underline{\mathsf{U}}\ \psi]}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models \gamma \vee \psi \wedge \mathsf{X}\,[\psi \ \mathsf{U}\ \gamma]}{(\mathcal{G},\mathcal{L}) \models [\psi \ \mathsf{U}\ \gamma] \Lleftarrow \mathrm{NextWUntil}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models \gamma \vee \psi \wedge \mathsf{X}\,[\psi \ \underline{\mathsf{U}}\ \gamma]}{(\mathcal{G},\mathcal{L}) \models [\psi \ \underline{\mathsf{U}}\ \gamma] \Lleftarrow \mathrm{NextSUntil}()}$$

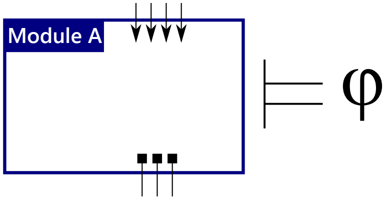
$$\frac{(\mathcal{G},\mathcal{L}) \models \varphi \wedge (\mathcal{G},\mathcal{L}) \models \varphi \rightarrow \mathsf{X}\varphi}{(\mathcal{G},\mathcal{L}) \models \mathsf{G}\varphi \Lleftarrow \mathrm{Induction}()}$$

Interactive Verification Framework - AIFProver

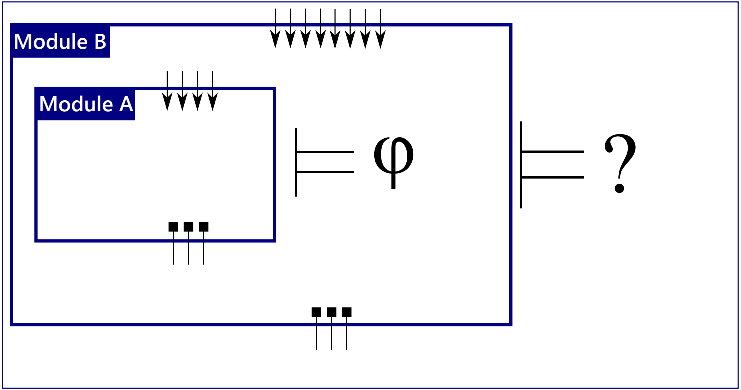
Implementation Details

- Averest is implemented in F#
- AIFProver uses same code base and is implemented in F#
- proof rules are F# functions
- proofs are F# scripts/programs
- F# interactive console allows to generate proofs

Rule for Module Calls



Rule for Module Calls



Rule for Module Calls

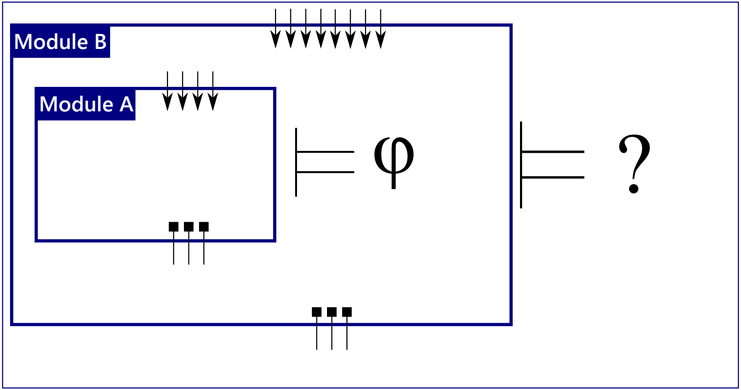
The Goal

interactive proof rule for module calls in synchronous programs

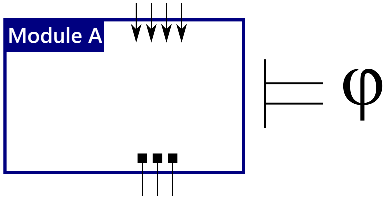
Problems Induced by Calling a Module

specific: default reaction
general: substituted behavior
specific: preemption and delayed start

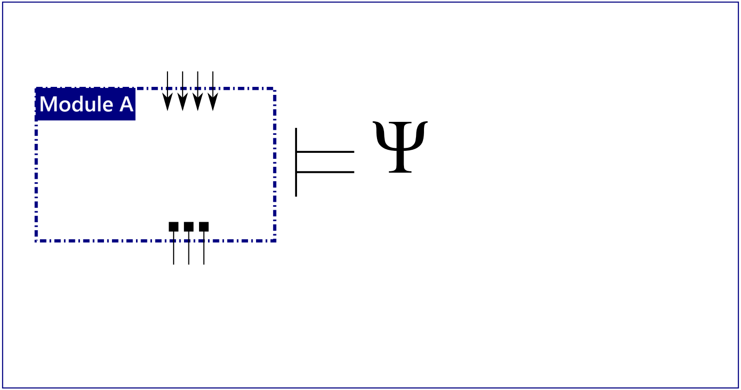
Rule for Module Calls



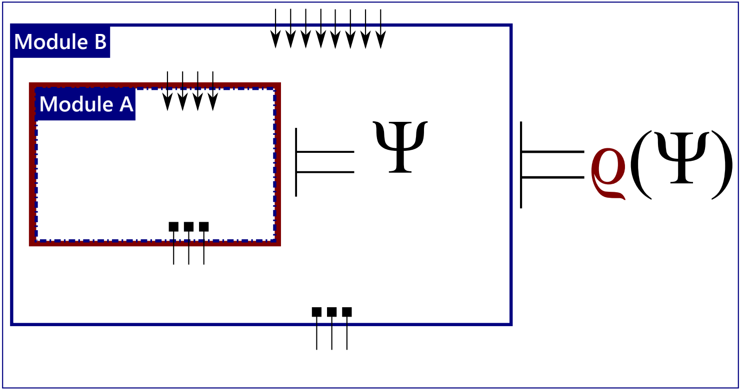
Rule for Module Calls



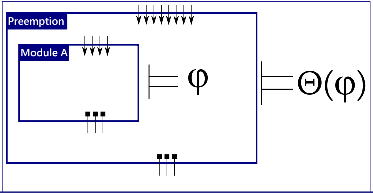
Rule for Module Calls



Rule for Module Calls



Rules for Preemption



Approach

- restriction to preemption specific behavior
- step wise application possible
- preemption-specific Θ
- specification should preserved 'as much as possible'
- correct by construction

Fibonacci Numbers

```
module Fib(nat ?i,f,event !r)

  nat k,g,n;
  n = i;
  if(n <= 0)
    f=0;
  else {
    k = 1;
    g = 0;
    f = 1;
    while(k != n) {
      next(g) = f;
      next(f) = f+g;
      next(k) = k+1;
      l: pause;
    }
  }
  emit(r);
```

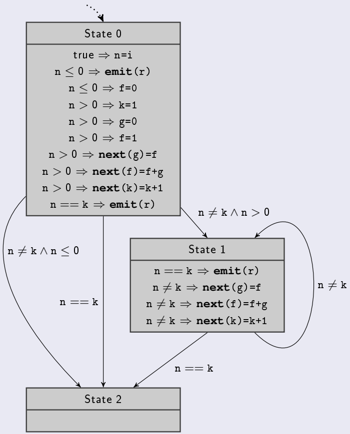
- computes Fibonacci numbers in quartz
- $\texttt{r} \rightarrow \texttt{f == FIB (i}_0\texttt{)}$

Fibonacci Numbers

module Fib(nat ?i,f,event !r)

```
nat k,g,n;
n = i;
if(n <= 0)
  f=0;
else {
  k = 1;
  g = 0;
  f = 1;
  while(k != n) {
    next(g) = f;
    next(f) = f+g;
    next(k) = k+1;
    l: pause;
  }
}
emit(r);
```

EFSM for Modul Fib



Fib in STA form (automatic-version)

```
module FSA(nat ?i,f,event r)

  nat k,g,n,l;
  do {
    case
      (l==0) do //State 0
        (n,r,k,g,f).(g,f,k,l) =
          (i,n<=0,1,0,(n>0?1:0)).
          (f,f+g,k+1,(n>0&n!=k?1:2));
      (l==1) do //State 1
        (r).(g,f,k,l) =
          (n==k).
          (f,f+g,k+1,(n!=k?1:2));
    default
      nothing;
    pause;
  } while (l!=2);
```

- structure completely destroyed
- code contains only a single loop
- same drawbacks as synthesising sequential code

Fib in STA form (handwritten-version)

```
module Fib(nat ?i,f,event !r)

  nat k,g,n;
  n = i;
  if(n <= 0)
    f=0;
  else {
    k = 1;
    g = 0;
    f = 1;
    while(k != n) {
      next(g) = f;
      next(f) = f+g;
      next(k) = k+1;
      l: pause;
    }
  }
  emit(r);
```

```
module FSH(nat ?i,f,event !r)

  nat k,g,n;
  if(n<=0) {
    (n,f,r).() = (i,0,true).();
  } else {
    (n,k,g,f,r).(g,f,k) =
      (i,1,0,1,k==n).(f,f+g,k+1);
    while(k!=n) {
      pause;
      (r).(g,f,k) = (k==n).
            (f,f+g,k+1);
    }
  }
```

Fib in STA form (handwritten-version)

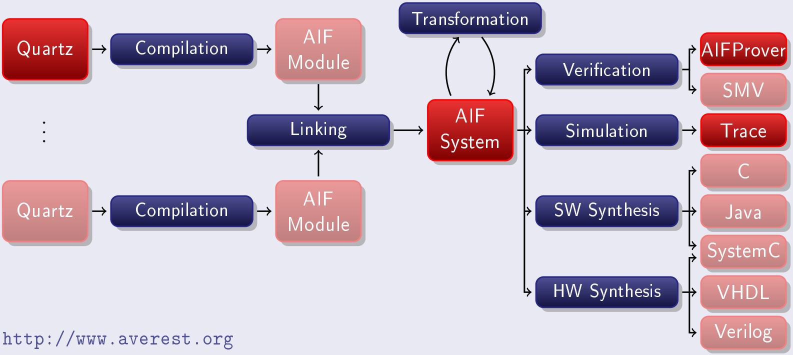
module FSH(nat ?i,f,event !r)

```
nat k,g,n;
if(n<=0) {
  (n,f,r).() = (i,0,true).();
} else {
  (n,k,g,f,r).(g,f,k) =
    (i,1,0,1,k==n).(f,f+g,k+1);
  while(k!=n) {
    pause;
    (r).(g,f,k) = (k==n).
        (f,f+g,k+1);
  }
}
```

- structure is preserved
- assignment are shifted and/or duplicated
- same invariants are usable

Averest

Averest Design Flow



http://www.averest.org

Further Work

basis for new work

- extension of rule set
- application to HybridQuartz
- improvement of the AIFProver
    - embedding in a theorem prover
    - deeper integration of existing decision procedures
    - using information from counterexamples

Sequential Model of Computation

Behavior of IGAs (subset of Dijkstra's Guarded Commands)

- execution of a single enabled guarded actions

Example

$$\left\{ \begin{array}{l} \text{true} \Rightarrow \texttt{x=(z==0)} \\ \text{true} \Rightarrow \texttt{y=z>0} \\ \text{true} \Rightarrow \texttt{z=z+1} \end{array} \right\}$$

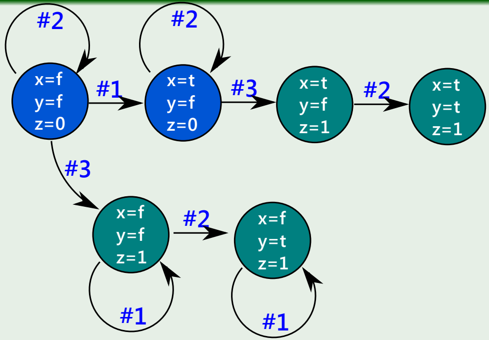
Sequential Model of Computation

Behavior of IGAs (subset of Dijkstra's Guarded Commands)

- execution of a single enabled guarded actions

Example

$$\left\{ \begin{array}{l} \mathrm{true} \Rightarrow \mathtt{x=(z==0)} \\ \mathrm{true} \Rightarrow \mathtt{y=z>0} \\ \mathrm{true} \Rightarrow \mathtt{z=z+1} \end{array} \right\}$$



Concurrent Model of Computation

Definition: Asynchronous Guarded Actions (AGAs)

An asynchronous guarded action $(\gamma \Rightarrow \alpha)$ consists of

- a Boolean guard γ and
- a set of atomic assignments α.

Behavior of AGAs

- execution of a subset of enabled guarded actions

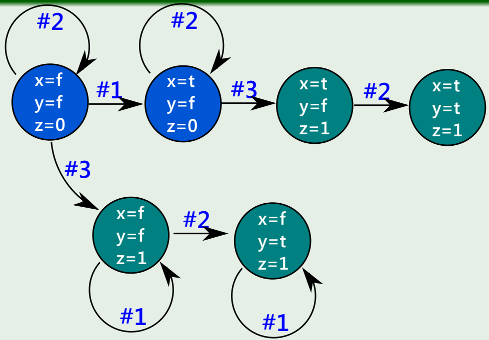
Concurrent Model of Computation

Behavior of AGAs

- execution of a subset of enabled guarded actions

Example

$$\left\{ \begin{array}{l} \mathtt{true} \Rightarrow \mathtt{x=(z==0)} \\ \mathtt{true} \Rightarrow \mathtt{y=z>0} \\ \mathtt{true} \Rightarrow \mathtt{z=z+1} \end{array} \right\}$$



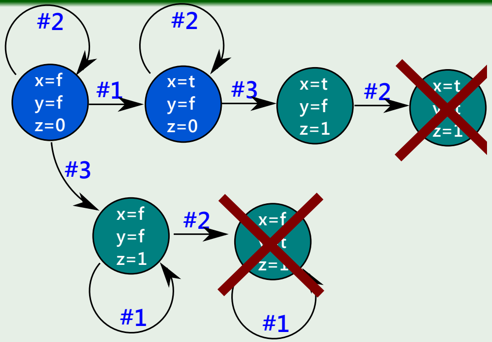
Concurrent Model of Computation

Behavior of AGAs

- execution of a subset of enabled guarded actions

Example

$$\left\{ \begin{array}{l} \mathsf{true} \Rightarrow x=(z==0) \\ \mathsf{true} \Rightarrow y=z>0 \\ \mathsf{true} \Rightarrow z=z+1 \end{array} \right\}$$



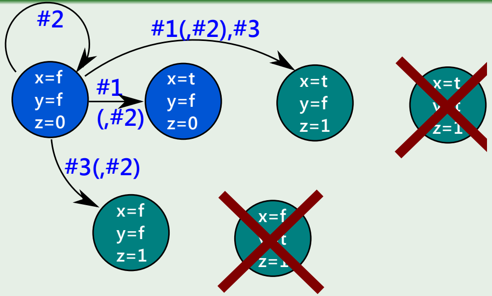
Concurrent Model of Computation

Behavior of AGAs

- execution of a subset of enabled guarded actions

Example

$$\left\{ \begin{array}{l} \mathtt{true} \Rightarrow \mathtt{x=(z==0)} \\ \mathtt{true} \Rightarrow \mathtt{y=z>0} \\ \mathtt{true} \Rightarrow \mathtt{z=z+1} \end{array} \right\}$$



Synchronous Model of Computation

Definition: Synchronous Guarded Actions (SGAs)

A synchronous guarded action $(\gamma \Rightarrow \alpha)$ consists of

- a Boolean guard γ and
- a single atomic immediate/delayed assignment α.

Behavior of SGAs

- execution of all enabled guarded actions in parallel

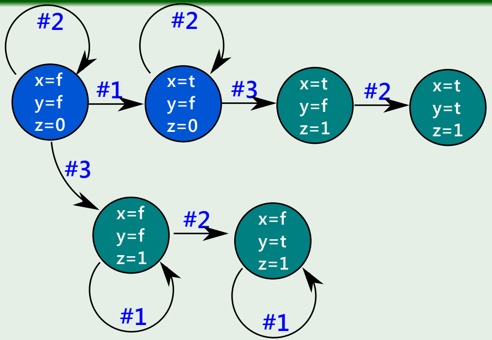
Synchronous Model of Computation

Behavior of SGAs

- execution of all enabled guarded actions in parallel

Example

$$\left\{ \begin{array}{l} \text{true} \Rightarrow \texttt{x=(z==0)} \\ \text{true} \Rightarrow \texttt{y=z>0} \\ \text{true} \Rightarrow \texttt{z=z+1} \end{array} \right\}$$



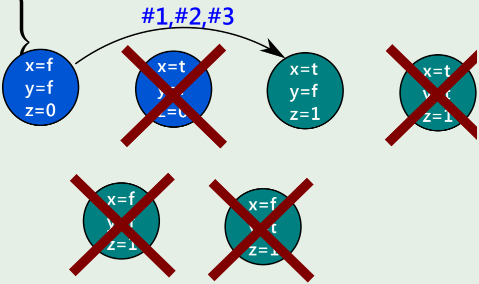
Synchronous Model of Computation

Behavior of SGAs

- execution of all enabled guarded actions in parallel

Example

$$\left\{\begin{array}{l}\text{true} \Rightarrow \texttt{x=(z==0)} \\ \text{true} \Rightarrow \texttt{y=z>0} \\ \text{true} \Rightarrow \textbf{next}\texttt{(z)=z+1}\end{array}\right\}$$



Synchronous Model of Computation

- execution is divided into a sequence of reactions steps
- computation of WCRT
- supports hard- and software synthesis
- deterministic behavior
- formal verification techniques available (i.e. model checking)
- languages: Quartz, Esterel, Signal, Lustre, etc.

Macro Step Behavior

- all inputs are read
- all outputs are produced (instantaneously)
- new internal state is determined
- each variable has a unique value