Design and Verification of Reactive Real-Time Systems

Prof. Dr. Klaus Schneider

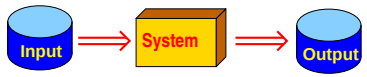
Reactive Systems Group
Department of Computer Science
University of Kaiserslautern

Kaiserslautern, August 4th, 2006

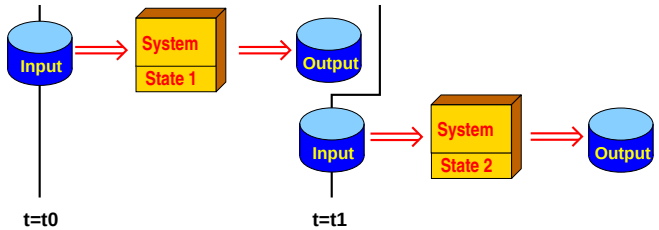
Outline

What are Reactive Systems?

Transformational Systems



Interactive/Reactive Systems



Reactive Systems

- request for interaction
  - interactive systems: by the system itself
  - reactive systems: by the environmental system

⇝ reactive systems must '*always*' be ready for interaction

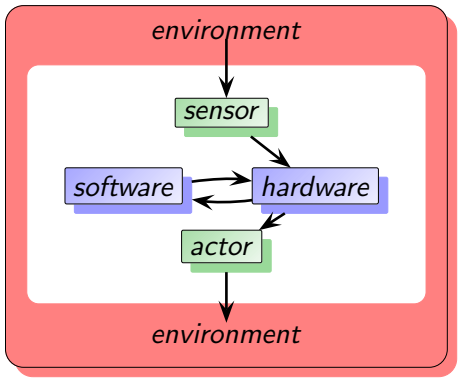
⇝ **reactive systems are real-time systems**

- basic computations are interactions with environment

⇝ interactions determine logical points of time

- interactions (= macro steps) are divided into micro steps
- *micro steps are executed in 'zero time'*

Technical Realization: Embedded Systems

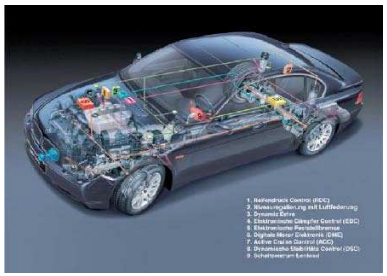


- *direct* interaction with environment (no human user involved)
- environment: mechanic, electronic, pneumatic,...
- sensor/actor often analog hardware
- software runs on one or more microprocessors
- multithreaded software
- application-specific hardware/processors
- reconfigurable systems are emerging

Embedded Systems

- embedded systems
  - used 98% of the microprocessors shipped in 1995
  - caused up to 40% of development costs of modern cars
  - appear in consumer electronics, automotive & avionics industries
  - growing field of applications
- growing impact on competition
  - more 'intelligent' systems
  - product distinction often due to embedded systems
  - 90% of new development in automotive is software

Example: Automotive Industry



- up to 100 embedded systems in modern cars
- connected with busses like CAN, TT-CAN, FlexRay, MOST
- powerful digital signal processors used

Advantages of Embedded Systems

- decrease of production costs
- decreased size and power consumption
- increased comfort/simplifying the usage
- increase of safety: ABS, EPS ...
- automatic maintainance/fault detection
- optimization of functions like fuel injection
- personalization of products

Research Problems

- heterogeneous design flows for hardware and software
- difficult design decisions in advance:
  - choice of modules to be implemented in software
  - choice of microprocessors
  - choice and design of application-specific hardware
- estimation of real-time behavior required
- time-to-market is decreasing $\rightsquigarrow$ less development time
- safety-critical applications $\rightsquigarrow$ guarantee absence of errors
- complexity of systems is dramatically increasing
  $\rightsquigarrow$ test/simulation no longer sufficient

Example: Therac-25 Incident (1986/1987)

- Therac-25 is a electron accelerator
- it is used in radiation therapy in many hospitals
- the device was approved by government agencies
- forerunners Therac-20 and Therac-6 used many hardware interlocks to prevent wrong usage:
  e.g. microwave oven should not operate when door is open
- for Therac-25, designers chose to removed many of the interlocks, but reused part of Therac-20 software
- software bugs caused it to fail: MUTEX was not given due to wrong test and setroutines that became wrong because of removed hardware interlocks

$\rightsquigarrow$ some patients died

Example: PATRIOT Weapon System (1991)

- accident during Gulf War in Dharan, Saudi Arabia in 1991
- Patriot weapon system to intercept Scud missiles, but failed
- result: 28 soldiers died; what was the reason for the failure?
- counting time in tenth of seconds
- 0.1 as floating point number (IEEE 754 standard):

$\Rightarrow$

0	01111011	10011001100110011001101

- is not exact; error is about $1.4901161138 \times 10^{-09}$
- after 100 hours of activation: ≈ 5.36 msec difference
- speed of Scud missiles: 1676 $\frac{m}{s}$

Example: Pentium Microprocessor (1994)

- November 1994: Th. Nicely detects error in Pentium processor
- example:
  - $x - (\frac{x}{y}) \times y$ should be 0,
    but for $x = 4195835$ and $y = 3145727$, Pentium computes 256
- why? an error in the arithmetic unit of the processor
- December 1994: Intel offered to exchange all processors

$\Rightarrow$ 470 000 000 € financial damage

Example: Ariane Space Rocket (1996)

- June 4th, 1996: Ariane 5 explodes
- reason was erroneous software:
  64 bit floating point number converted to 16 bit integer

$\Rightarrow$ resulted in number overflow

$\Rightarrow$ raised exception to initiate self-destruction of the rocket

- estimated development cost: 7 000 000 000 €
- financial damage: 500 000 000 €

Example: Cobalt 60 Machine (2001)

- Panama, 2001
  - Cobalt 60 machine used in radiation therapy
  - the machine delivered 20-100?% more radiation than required
  - 8-12 patients died
- who was at fault?
- inspections of the software manufacturer (Multidata Systems) revealed that there was poor specification and documentation, inadequate testing, no verification
- in November 2004, engineers were sent to prison for four years...

Example: Tempomat in Renault (2004)

- France, October 2004
  - Tempomat in Renault Vel Satis went wrong
  - increased speed up to 190 $\frac{km}{h}$
  - braking was useless
  - ignition could not be stopped since Renault no longer uses keys
- police opened Maut offices between Vierzon and Riom (France)
- Renault is currently investigating the incident
- in 2004, more than 50% of car problems were caused by electronics/software

...and it is still not over

- most compilers use IEEE 754 floating point numbers
- but many of them have problems with that
- example in ANSI-C:

```
float q = 3.0/7.0;
if q == 3.0/7.0 printf("no problem.");
else printf("problem!");
```

- try it, and you will see that C has a problem!
- reason: expressions in C computed in double precision, but the float q has only single precision

. . . and it is still not over

- no solution: avoid tests on equality
- instead, check if difference is very small:

$$\textbf{float } q = 3.0/7.0;$$
$$\textbf{if } |q - 3.0/7.0| \leq \varepsilon \textbf{ printf}(\text{“no problem.“});$$
$$\textbf{else printf}(\text{“problem!“});$$

- but this „equality“ is no equivalence relation
- you may have $x = y$ and $y = z$, but not $x = z$

Outline

Design Problems of Software Systems

- why are there so many errors in computer systems?
- there are not more errors than anywhere else, but they have more consequences
- discrete data domains:
  - can you really buy 1 kilo of something?
  - normally you receive a bit more or less
  - but computer systems may fail due to such inaccuracies!
- complexity is rapidly growing
- development under pressure of time-to-market
- but: is that a theoretical/philosophical issue? unfortunately not, we already saw serious incidents

Formal Verification

- formal verification technology is one of the success stories of computer science
- nowadays, it allows one to automatically verify large systems even systems with 10^{100} states have been verified
- but: it requires a formal semantics of the system
- unfortunately, not available for most programming languages
- but available for synchronous programming languages
- but still: what properties can be verified?
- how are the properties presented and verified?

Aims and Limits of Verification

- given a formal specification, and a precise system description
- verification can check, whether the system satisfies the spec
- this is done by generating some sort of mathematical proof
- thus, the verification is mathematically sound
- and therefore, a lot formalisms are used
- however, not all problems can be solved by mathematical logic

. . .

Aspects of Dependable Systems

- **reliability**
  - system works all the time
  - continuously available, graceful degradation
  - ⇝ todo: fault tolerance, diversity of design
- **security**
  - no non-authorized usage
  - ⇝ todo: cryptography, passwords, . . .
- **correctness**
  - no design errors
  - ⇝ todo: design flows with test, validation, and verification

Different Classes of Faults

- specification faults: wrong/incomplete/vacuous specification
- design errors: faulty system does not satisfy given spec
- faulty design tools: compiler generates wrong code
- fabrication faults: faults on chips or other hardware parts
- usage errors: aging of systems; use of incompatible versions

what do you suggest for preventing these problems?

Limits of Formal Verification

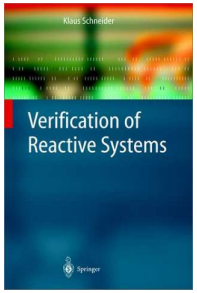
- given a formal specification, and a precise system description
- verification checks, whether the system satisfies the spec
- thus, checking absence of design errors, but
  - **was the specification right?**
    - often given in natural language, thus imprecise
    - and if formally given, often hard to read
    - simulate/verify the specification? against what?
  - **completeness of specification**
    - were all important properties specified
    - hard to tell, coverage metrics are used nowadays

Different Tools and Techniques

- requirements engineering
- risk analysis
- theorem proving
- abstract interpretation
- model checking
- symbolic simulation
- validated code generation
- proof carrying code

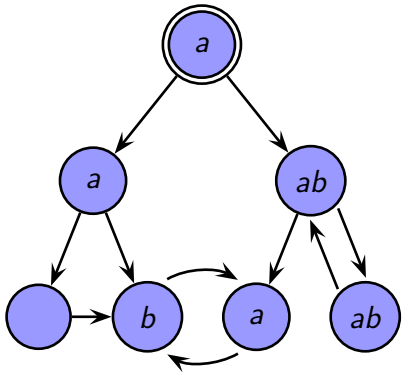
Specification Logics

- many formalisms and logics have been considered to specify temporal properties of systems
- in particular:



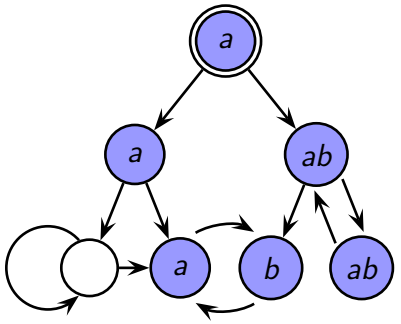
  - temporal logics like LTL, CTL or CTL*
  - ω-automata on words and trees
  - μ-calculus on transition systems
  - first and second order monadic predicate logics on words and trees
- the μ-calculus turned out to be the most powerful logic

Labeled Transitions Systems = Kripke Structures



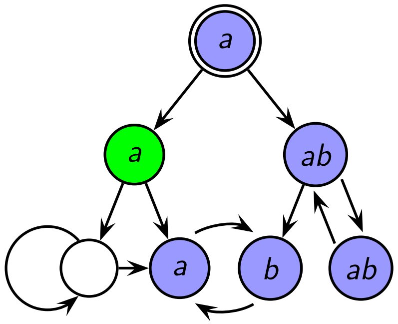
- transition systems model system's behavior
- states are program states
- transitions are atomic actions
- paths are computations
- labels of states are content of memory
- each transition may be viewed to take time

Example: Model Checking AG($a \vee b$)



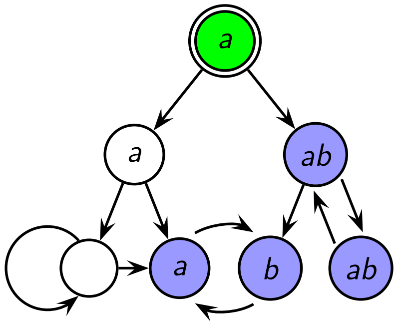
start with states satisfying $a \vee b$

Example: Model Checking AG$(a \vee b)$



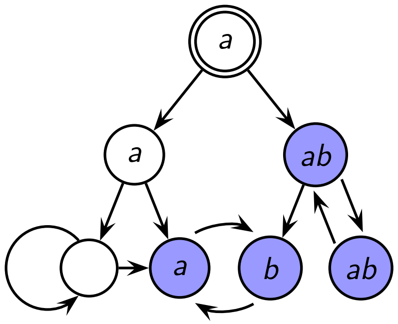
remove states that have a successor outside the current set;
repeat this until set becomes stable

Example: Model Checking AG$(a \vee b)$



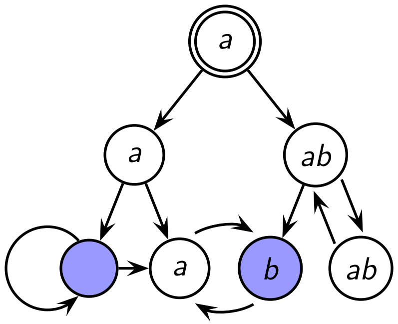
remove states that have a successor outside the current set;
repeat this until set becomes stable

Example: Model Checking AG$(a \vee b)$



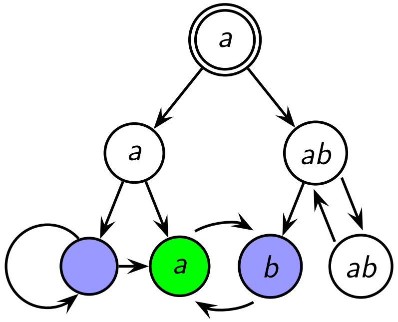
remove states that have a successor outside the current set;
repeat this until set becomes stable

Example: Model Checking AF$\neg a$



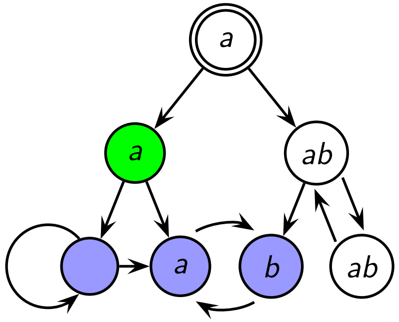
start with states satisfying $\neg a$

Example: Model Checking AF¬*a*



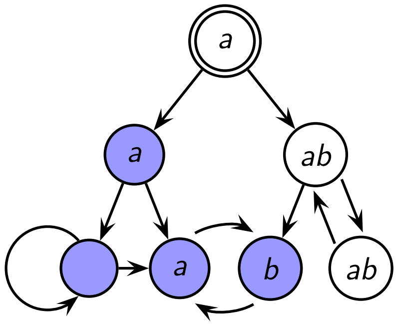
add *states where all successors belong to the current set; repeat this until set becomes stable*

Example: Model Checking AF¬*a*



add *states where all successors belong to the current set; repeat this until set becomes stable*

Example: Model Checking AF¬*a*



add *states where all successors belong to the current set; repeat this until set becomes stable*

Current Research Activities

- symbolic model checking allows us to handle very large, but finite systems
- infinite states model checking
- quantitative temporal logics
- different kinds of model reductions
- program (syntax) directed verification

Synthesis versus Verification

- **verification**
  - given a labeled transition system $\mathcal{K}$ and a specification φ, check whether $\mathcal{K} \models \varphi$ holds
  - efficient tools for symbolic model checking are available
- **supervisor synthesis**
  - given a labeled transition system $\mathcal{K}$ and a specification φ, check whether there is a system $\mathcal{C}$ such that $\mathcal{K} \parallel \mathcal{C} \models \varphi$ holds
  - only a few preliminary tools are available
  - many formalisms are discussed
- **supervisor synthesis can be reduced to verification**

Alternating Time μ-Calculus

- introduced in 1997 by Alur, Henzinger, and Kupferman
- to specify open systems
- playing a game between system and its environment
- expressing properties like:
  there is a strategy such that a certain property holds
- address problems like:
  - receptiveness
  - realizability (program synthesis)
  - supervisory control
  - module checking

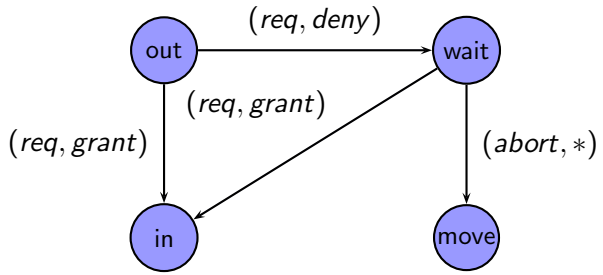
Concurrent Game Structures

- concurrent game structures are an extension of labeled transition systems
- two players A and B are considered
- each player can choose actions of a set $\mathcal{A}$
- each transition $(s, s') \in \mathcal{R}$ is labeled with a pair (α, β) where α is the action of player A and β is the action of player B
- players choose their actions concurrently and independently of each other
- a generalization of turn-based games

Concurrent Game Structures

- formally, a concurrent game structure $\mathcal{G} = (\mathcal{I}, \mathcal{S}, \delta, \Gamma_A, \Gamma_B, \mathcal{L})$ over the variables $\mathcal{V}$ and the actions $\mathcal{A}$ consists of
  - a finite set of states $\mathcal{S}$
  - initial states $\mathcal{I} \subseteq \mathcal{S}$
  - a partial transition relation $\delta \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{A} \times \mathcal{S}$
  - actions $\Gamma_A : \mathcal{S} \to 2^{\mathcal{A}}$ of player A
  - actions $\Gamma_B : \mathcal{S} \to 2^{\mathcal{A}}$ of player B
  - label function for the states $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{V}}$
- a transition from state s to state s' is labeled with $(\alpha, \beta) \in \Gamma_A(s) \times \Gamma_B(s)$
- clearly, a generalization of Kripke structures

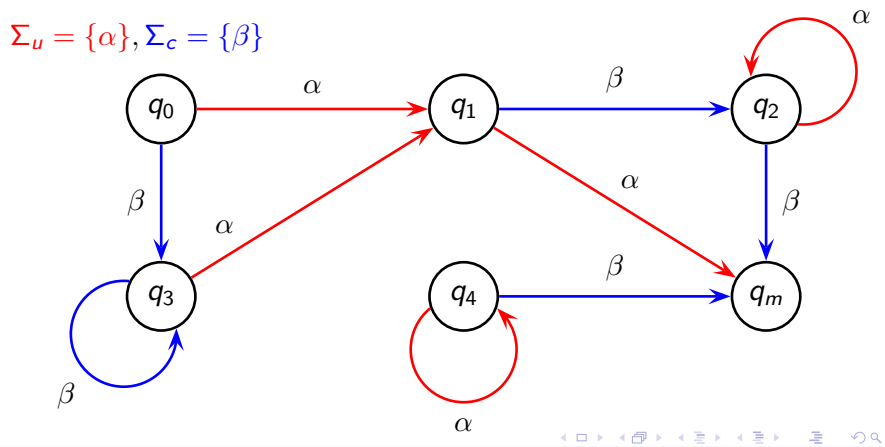
Concurrent Game Structures



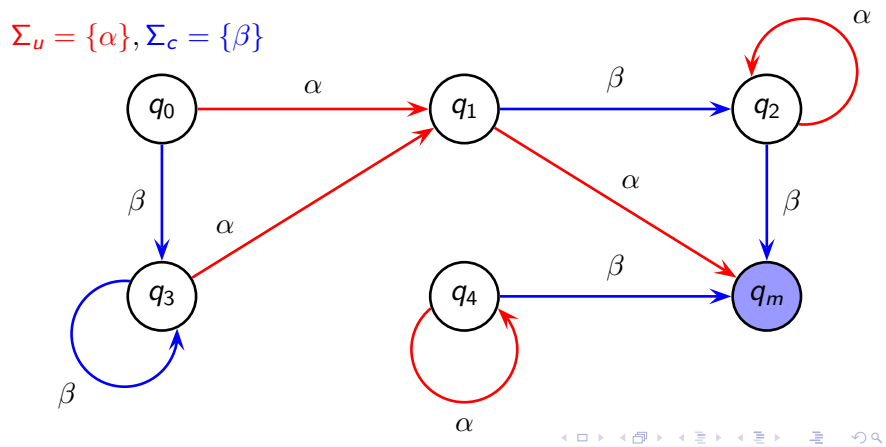
Supervisory Control

- given a finite state automaton $\mathcal{A} = \langle Q, \Sigma, q_0, \delta, Q_m \rangle$ with
  - state set Q
  - input alphabet Σ, partitioned into
    - controllable events Σ_c
    - uncontrollable events Σ_u
  - transition function $\delta : Q \times \Sigma \to Q$
  - initial state $q_0 \in Q$
  - final (marked) states Q_m
- **question: is there a strategy $\zeta : Q \to 2^{\Sigma_c}$ such that all reachable states can reach marked states?**

Example

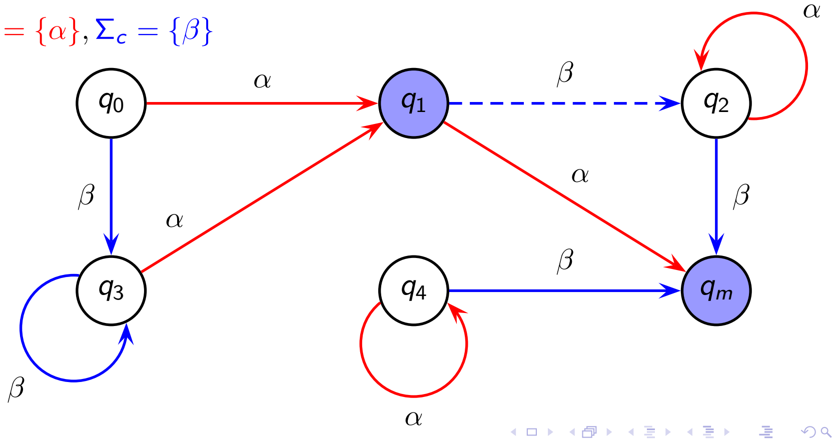


Example



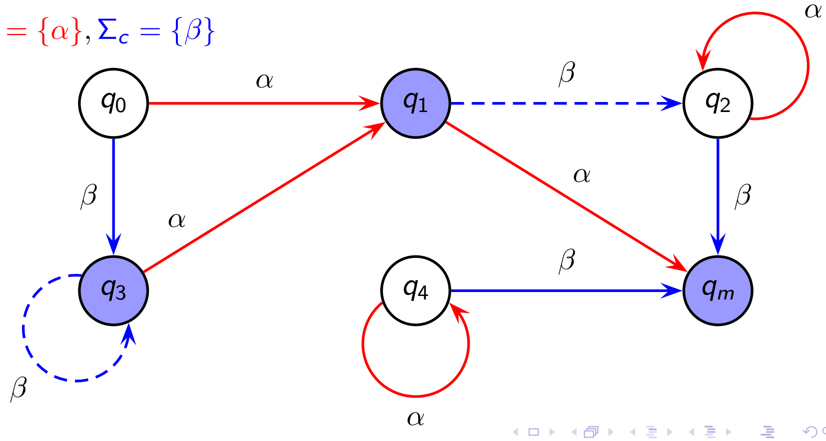
Example

$\Sigma_u = \{\alpha\}, \Sigma_c = \{\beta\}$



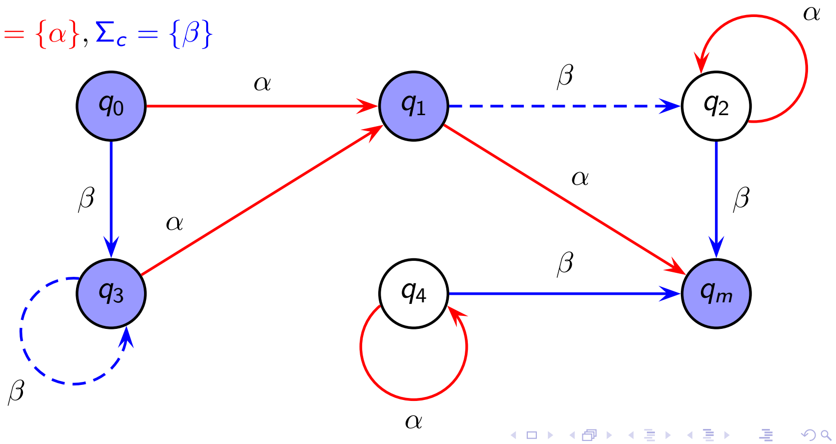
Example

$\Sigma_u = \{\alpha\}, \Sigma_c = \{\beta\}$



Example

$\Sigma_u = \{\alpha\}, \Sigma_c = \{\beta\}$



Supervisory Control

- supervisory control allows to compute the controllable states
- generating a controller requires a bit more effort, but is also feasible
- supervisory control can also be used to debug systems:
  - if verification fails, since there is an error
  - then, for each module, ask supervisor synthesis if modules can be repaired
  - **error localization**

Outline

Synchronous Languages

- several languages have been developed since 1992
  - Esterel
    - imperative language
    - well-suited for complex controllers
  - Lustre
    - data flow language
    - well-suited for signal processing
  - Statecharts
    - graphical state-based language
    - well-suited for complex controllers
    - many incompatible variants exist

Paradigm of Synchronous Languages

- **distinguish between micro and macro steps**
- micro steps
  - execution does not take time (in the programmer's model)
  - immediate effect
- macro steps
  - consist of finitely many micro steps
  - number of micro steps can be estimated at compile time
  - $\rightsquigarrow$ estimation of reaction time possible
- deterministic systems
  - due to synchronous concurrency
  - due to precise semantics

Basic Statements of Quartz

nothing	(empty statement)
ℓ : **pause**	(separation of macro step)
emit x, **emit next**(x)	(signal emission)
$x := \tau$, **next**$(x) := \tau$	(assignment)
if σ **then** S_1 **else** S_2 **end**	(conditional)
choose $S_1 \,[]\, S_2$ **end**	(nondeterministic choice)
S_1**;** S_2	(sequence)
$S_1 \parallel S_2$ and $S_1 \,\|||\, S_2$	(synch./asynch. concurrency)
do S **while** σ	(loop)
[weak] abort S **when [immediate]** σ	(process abortion)
[weak] suspend S **when [immediate]** σ	(process suspension)
local $x : \alpha$ **in** S **end**	(local declarations)

Micro and Macro Steps

- **pause** is the only statement that takes time
- each **pause** statement consumes one logical unit of time

⇒ *threads synchronize at their next* **pause** *statements*

- synchronization done by semantics

⇒ *deterministic multi-threaded programs!*

- precomputation of thread interaction at compile time

⇒ *no process management at runtime*

⇒ *fast programs and small code*

Example Program

- example:

$$\left[\begin{array}{l}\textbf{emit } b;\\ p : \textbf{pause};\\ \textbf{if } a \textbf{ then emit } b \textbf{ end};\\ r : \textbf{pause}\end{array}\right] \;\|\|\; \left[\begin{array}{l}q : \textbf{pause};\\ \textbf{if } \neg b \textbf{ then emit } c \textbf{ end};\\ \textbf{emit } a;\\ s : \textbf{pause}\end{array}\right]$$

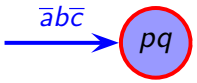
-

Example Program

- example: time step 0 $\Rightarrow (a, b, c) = (0, 1, 0)$

$$\left[\begin{array}{l}\textbf{emit } b;\\ p : \textbf{pause};\\ \textbf{if } a \textbf{ then emit } b \textbf{ end};\\ r : \textbf{pause}\end{array}\right] \;\|\|\; \left[\begin{array}{l}q : \textbf{pause};\\ \textbf{if } \neg b \textbf{ then emit } c \textbf{ end};\\ \textbf{emit } a;\\ s : \textbf{pause}\end{array}\right]$$

- automaton:

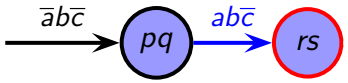


Example Program

- example: time step 1 $\Rightarrow (a, b, c) = (1, 1, 0)$

$$\left[\begin{array}{l}\textbf{emit } b;\\ p : \textbf{pause};\\ \textbf{if } a \textbf{ then emit } b \textbf{ end};\\ r : \textbf{pause}\end{array}\right] \;\|\|\; \left[\begin{array}{l}q : \textbf{pause};\\ \textbf{if } \neg b \textbf{ then emit } c \textbf{ end};\\ \textbf{emit } a;\\ s : \textbf{pause}\end{array}\right]$$

- automaton:

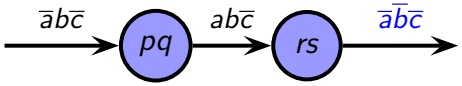


Example Program

- example: time step 2 $\Rightarrow (a, b, c) = (0, 0, 0)$

$$\left[\begin{array}{l} \textbf{emit } b; \\ p : \textbf{pause}; \\ \textbf{if } a \textbf{ then emit } b \textbf{ end}; \\ r : \textbf{pause} \end{array}\right] \;\|\|\; \left[\begin{array}{l} q : \textbf{pause}; \\ \textbf{if } \neg b \textbf{ then emit } c \textbf{ end}; \\ \textbf{emit } a; \\ s : \textbf{pause} \end{array}\right]$$

- automaton:

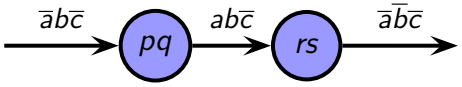


Example Program

- example:

$$\left[\begin{array}{l} \textbf{emit } b; \\ p : \textbf{pause}; \\ \textbf{if } a \textbf{ then emit } b \textbf{ end}; \\ r : \textbf{pause} \end{array}\right] \;\|\|\; \left[\begin{array}{l} q : \textbf{pause}; \\ \textbf{if } \neg b \textbf{ then emit } c \textbf{ end}; \\ \textbf{emit } a; \\ s : \textbf{pause} \end{array}\right]$$

- automaton:



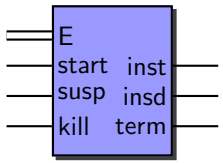
Hardware Synthesis

- synchronous hardware circuits and synchronous programs share the same paradigm of logical time

$\rightsquigarrow$ hardware synthesis is very simple

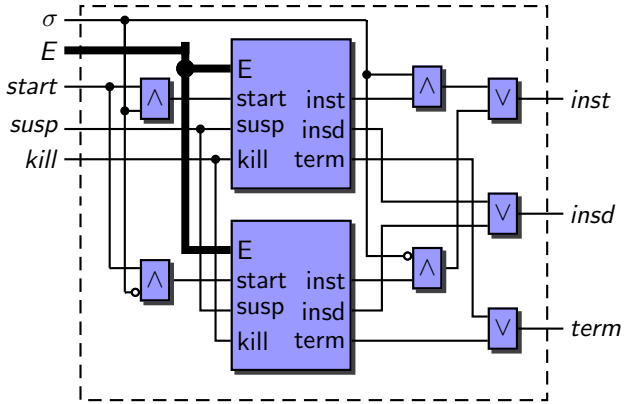
- **define a template circuit for every statement**
- allows hardware translation of program with n statements to circuit with $O(n^2)$ gates
- many optimizations like retiming known from hardware synthesis can be applied

Circuit Template for Control Flow Circuits

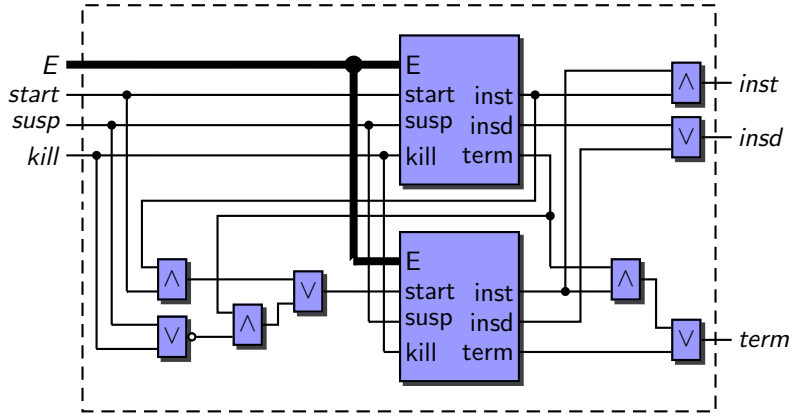


- E represents all wires for inputs and outputs
- *start* is used to start the computation
- *susp*: suspend the computation (freeze the control)
- *kill*: abort the computation (higher priority than susp)
- *inst* is true iff the computation would now be instantaneous
- *insd* is true iff control is currently inside the circuit
- *term* is true iff execution currently terminates

if σ then S_1 else S_2 end



$S_1; S_2$



Software Synthesis

- cycle-based code
  - idea: simulate circuit by C program
  - very small code size, but slow reaction time
- explicit automaton based code
  - idea: generate C-function for each reachable state that implements the behavior
  - state changes are covered by calling the right C-functions
  - very fast code, but code size can grow exponentially
- event-driven code
  - idea: propagate changes of inputs via data flow dependencies
  - medium sized code, reaction time quite fast

Cycle Based Code Generation

```
for i := 1 to k do
  ℓ_i := false
end
do
  for i := 1 to m do
    x_i := ReadInput(i)
  end;
  for i := 1 to n do
    y_i := g_i(x_1, ..., x_m, ℓ_1, ..., ℓ_k)
  end;
  for i := 1 to k do
    ℓ'_i := f_i(x_1, ..., x_m, ℓ_1, ..., ℓ_k)
  end;
  for i := 1 to k do
    ℓ_i := ℓ'_i
  end
end
```

- the left hand side can be sequentially evaluated to simulate the following equation system:

$$\begin{cases} \vec{\ell}^{(t+1)} & := & \vec{f}(\vec{x}^{(t)}, \vec{\ell}^{(t)}) \\ \vec{y}^{(t)} & := & \vec{g}(\vec{x}^{(t)}, \vec{\ell}^{(t)}) \end{cases}$$

- in general, this is not possible with cyclic equation systems:

$$\begin{cases} \vec{\ell}^{(t+1)} & := & \vec{f}(\vec{x}^{(t)}, \vec{y}^{(t)}, \vec{\ell}^{(t)}) \\ \vec{y}^{(t)} & := & \vec{g}(\vec{x}^{(t)}, \vec{y}^{(t)}, \vec{\ell}^{(t)}) \end{cases}$$

Hardware-Software Codesign

- application-specific instruction set processor
  - generate profiling data
  - frequent operations in macro steps yield special instructions
  - generate special processor
- application-specific coprocessor
  - generate hardware of some modules
  - generate hardware of some modules
  - interaction via interrupts (difficult)
- transaction-based communication with hardware
  - generate hardware of some modules
  - generate hardware of some modules
  - interaction via shared memory (simple)

Advantages of Synchronous Languages

- simplified execution time analysis
  - macro steps must not contain loops
  - estimating execution time of macro steps is simple!
- static analysis at compile time
  - guarantees freedom of deadlocks
  - guarantees deterministic behavior
- formal semantics allows application of formal methods
  - estimation of reaction time
  - verification of temporal properties
  - verification of functional properties

Compilation Problems

- synchronous languages have many advantages
- but there are also disadvantages
  - difficult compilation due to causality problems
  - difficult compilation due to reincarnation problems
- new research problems
  - GALS: (globally asynchronous locally synchronous)
  - use of multi-core processors and systems/networks on chip
  - distributed systems
  - code for multiprocessors

Causality Cycles

- **micro steps are executed in zero time**

⇝ output reaction is generated immediately with input action

- **problem: causality cycles may arise**
  if trigger condition of an action is influenced by the action
- programs may loose reactivity or become nondeterministic
- hence, unique existence of program behavior has to be checked
- either logically: **unique satisfiability** for all inputs
- but usually, by means of **causality analysis**

Causality Cycles may cause Problems

module P_3:
output o;
if o **else emit** o **end**
end module

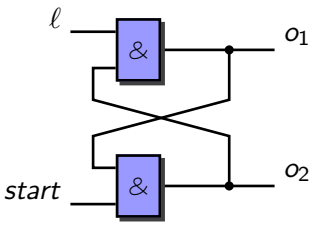
module P_4:
output o;
if o **then emit** o **end**
end module

- P_3 **is not reactive:**
  - if $o =$ true, it would not be emitted $\rightsquigarrow$ contradiction
  - if $o =$ false, it will be emitted $\rightsquigarrow$ contradiction
- P_4 **is not deterministic:**
  - if $o =$ true, it would be emitted $\rightsquigarrow$ okay
  - if $o =$ false, it would not be emitted $\rightsquigarrow$ okay

Causality Cycles may be okay

module P_{14} :
output o_1, o_2;
if o_1 **then emit** o_2 **end**;
ℓ : **pause**;
if o_2 **then emit** o_1 **end**
end

- hardware circuit



- okay, since ℓ is false at starting time
  - *start* $=$ true $\rightsquigarrow \ell = o_1 = o_2 =$ false
  - $\ell =$ true $\rightsquigarrow$ *start* $= o_1 = o_2 =$ false

Cyclic Programs can be smaller

- cyclic programs can be smaller

$$\begin{aligned} y &= \textbf{if } c \textbf{ then } y_f \textbf{ else } y_g; \\ y_f &= f(x_f); \\ y_g &= g(x_g); \\ x_f &= \textbf{if } c \textbf{ then } y_g \textbf{ else } x; \\ x_g &= \textbf{if } c \textbf{ then } x \textbf{ else } y_f; \end{aligned}$$

- implements **if** c **then** $f(g(x))$ **else** $g(f(x))$ with only one instance of f and g
- **impossible without cycles**

Results on Causality Analysis

- beautiful equivalences:
  - **synchronous program is constructive**
  - **corresponding circuit stabilizes for all gate delays**
  - **corresponding software never runs into deadlocks**
- cyclic programs/circuits can be smaller than every equivalent acyclic program/circuit

$\rightsquigarrow$ **causality analysis guarantees safe and small systems**

Summary

- current work
  - implementation of reactive systems by synchronous languages
  - model checking to avoid design errors
  - theorem proving to assure compiler correctness
  - supervisory control for error location
  - Averest system www.averest.org
- future work
  - hybrid systems (continuous and discrete time/data flow)
  - adaptive systems
  - abstraction to high-level system models