



Software
Systems
Engineering

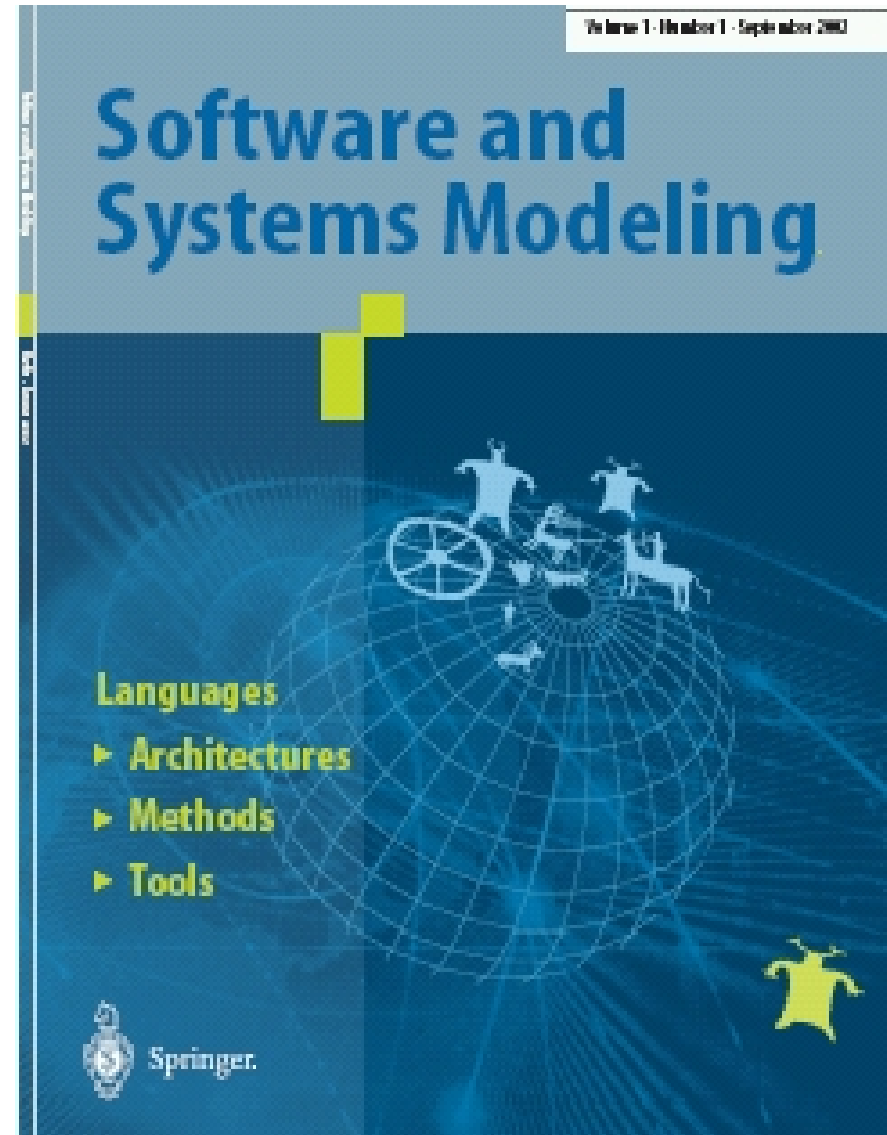
Model Driven Testing of Time Sensitive Distributed Systems

Bernhard Rumpe , Boris Gajanovic, Hans Grönniger
Software Systems Engineering
Braunschweig University of Technology, Germany
<http://www.sse.cs.tu-bs.de/>

The premier journal for Modelling ...

- www.sosym.org
- Modeling,
- Meta-Modeling,
- Languages,
- UML,
- Domain Specific Things,
- ...

- Usages of these



Model Driven Software Engineering

- Promote **models as primary artifacts** for software development
- If use of models should have an impact, an adequate development method is needed
 - **Model Driven Software Engineering**
- **Goals** of the MDSE approach
 - Improve developer's **productivity**
 - Enhance **quality** of the product
 - Reduce project's **costs**

Core Elements of an Agile Model Driven Method

- Intensive **use of models** in a project for
 - architecture modeling, code generation, test definition, analyses, ...
- **Incremental development** of architectures, functionalities and tests
- **Model driven and automated tests**
- **Code generation** for production code and tests
- Small incremental releases
- Intensive **simulation** with customer feedback
- **Evolutionary transformations** for incremental extensions and optimizations

One important aspect: Model Driven Testing

- Use models to **define tests**
- **Variants**
 - Test cases **derived from analysis or design models**
 - Models used to **measure quality of tests** (“coverage”)
 - The model itself can **describe (parts of) test cases**
- Important characteristics
 - **Precise, compact notation** with underlying semantic framework
 - **Automated tests** and efficient **simulation**
 - Usable in **early phases** to reduce costs
 - Deal with incomplete models (**underspecification**)

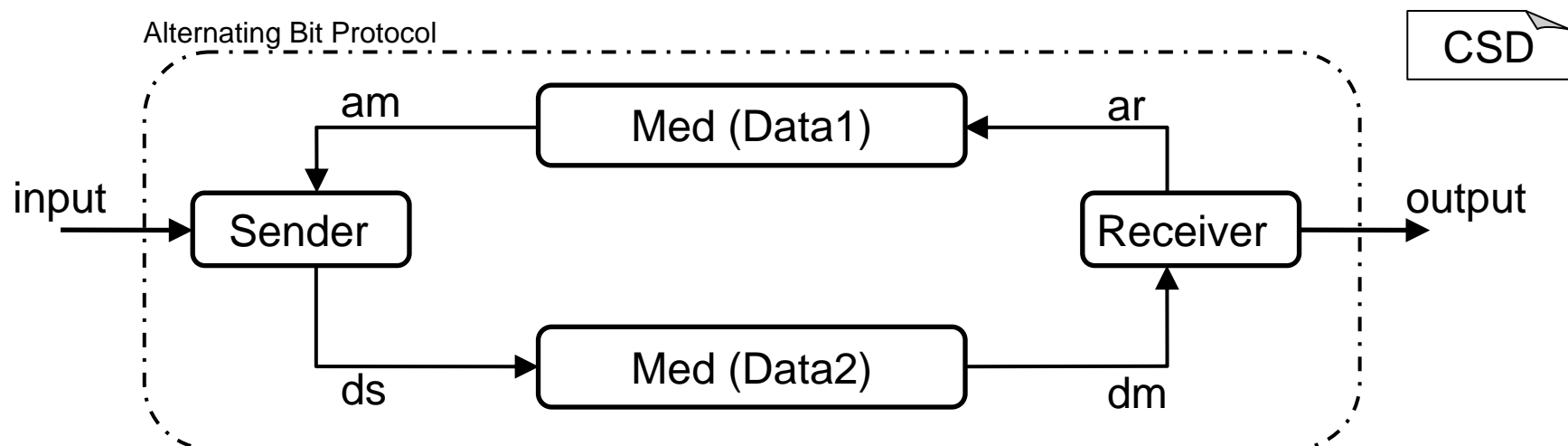
Time sensitive distributed systems

- We assume a
 - logically or physically **distributed system**
 - no shared data space
 - explicit communication using **asynchronous messages** via **buffered, directed channels**
 - **time** matters (e.g. timeouts)

- Some examples:
 - Telecommunication systems
 - Internet connected business applications
 - Control devices in automotive systems
 - Manufacturing lines

Example Application: Alternating Bit Protocol

- Protocol to **transmit data over unreliable media**
- Media can
 - **delay or lose messages** but cannot invent or modify them
 - eventually transmits a message correctly
- **Challenge:**
 - 1) specify media as “given”
 - 2) find and specify sender and receiver to allow safe transmits
 - 3) ensure correctness of implementation through proper tests



Semantic Framework: System Model

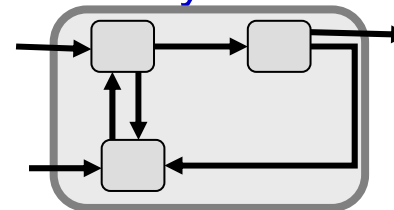
- A system consists of
 - components with explicit interfaces (“ports”)



- directed communication channels

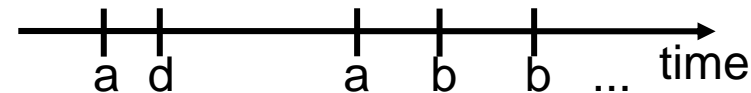


- and can be decomposed hierarchically



Messages and Channels

- We have sets of **Messages** as typed sets M, I, O
- Dataflow of a **channel** is observed through
 - a finite or infinite stream of messages $M^\omega = M^* \cup M^\infty$
 - example: $[a,d,a,b,b]$:

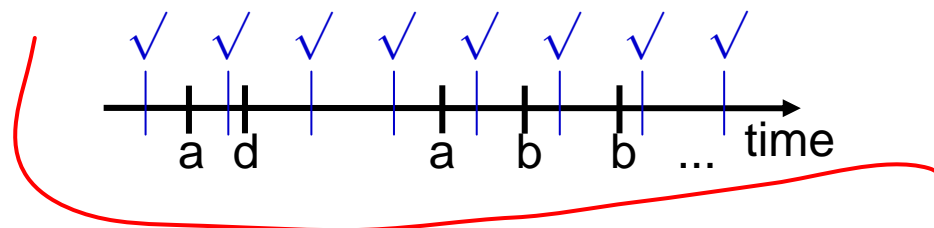


- Math functions:
- **Append element**
- **Concatenation**
- **First element, rest**
- **Prefix relation**
- **Size**
- **Empty observation**

$$\begin{aligned}
 & \underline{a:s} \\
 & \underline{s++t} \\
 & \underline{\text{head}(s) : \text{tail}(s) = s} \quad \text{if } s \neq \varepsilon \\
 & \underline{s \sqsubseteq t} \Leftrightarrow \exists r \in M^\omega: s++r = t \\
 & \#s \in \mathbb{N} \cup \{\infty\} \\
 & \underline{\varepsilon} \in M^\omega
 \end{aligned}$$

Timed Streams

- Concept: using ticks \surd to model that time passes
 - \surd model equidistant passing of time (e.g. milliseconds)
- Timed dataflow of a channel is an observation
 - a finite or infinite stream of messages and ticks, where
 - $M^\surd = \{ s \in (M \cup \{\surd\})^\omega \mid \forall n \in \mathbb{N}: \text{nth}(n,s) \neq \surd \Rightarrow \text{nth}(n+1,s) = \surd \}$
 - observation intervals are detailed:
 - at maximum one message per time interval
 - sufficiently fine granular view on the channel
- Example: $[\surd, a, \surd, d, \surd, \surd, a, \surd, b, \surd, b, \surd, \dots]$:

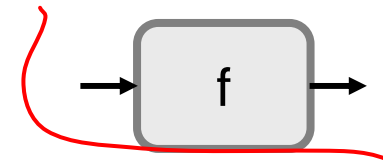


Components

- A **component** is defined through a function f :

- from Input I to Output O :

$$f: I^\omega \rightarrow O^\omega$$



- and can have several channels:

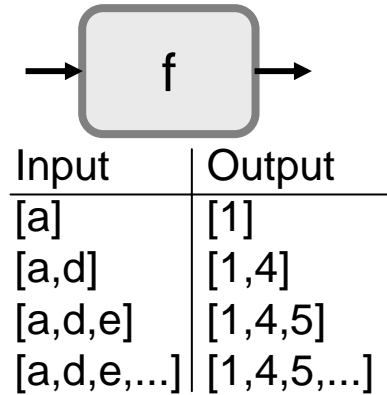
$$g: I1^\omega \times I2^\omega \times I3^\omega \rightarrow O1^\omega \times O2^\omega$$



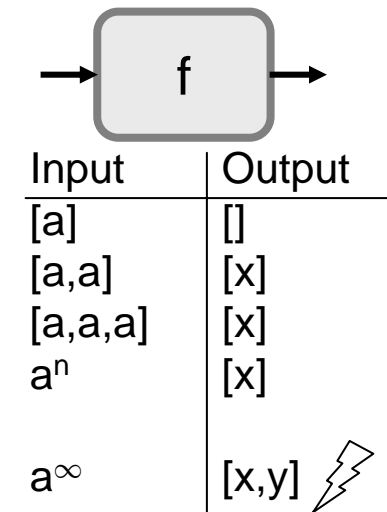
- with sets of messages usually denoted by $I, O, I1, I2, \dots$

Properties of Component-Functions

- Given a component-function $f: I^\omega \rightarrow O^\omega$ we expect to describe a **true behavior**:
 - output once emitted cannot be taken back, i.e.
 - longer input lead to longer output



- Furthermore a component-function $f: I^\omega \rightarrow O^\omega$ **cannot look into the future**:
 - “end of input” is not detectable
 - and “infinite input” is not detectable

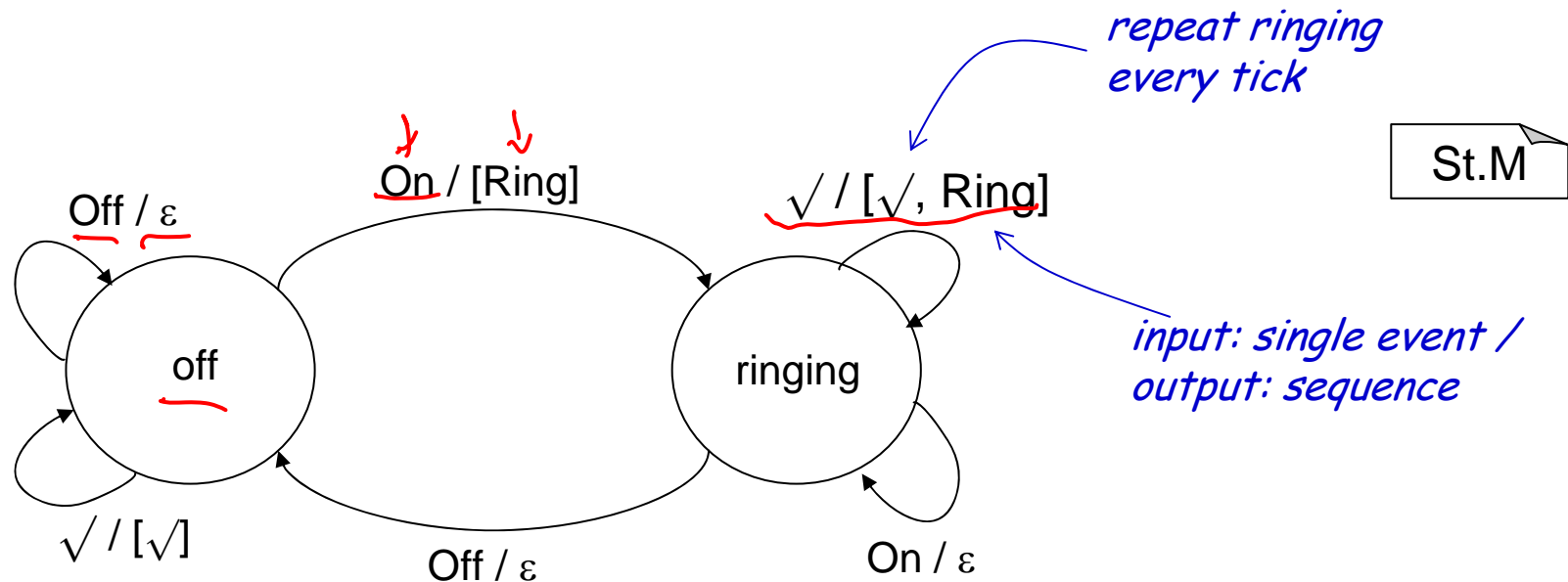


- Formally: f is monotonic and continuous vs. \sqsubseteq

Specification through state machines -1

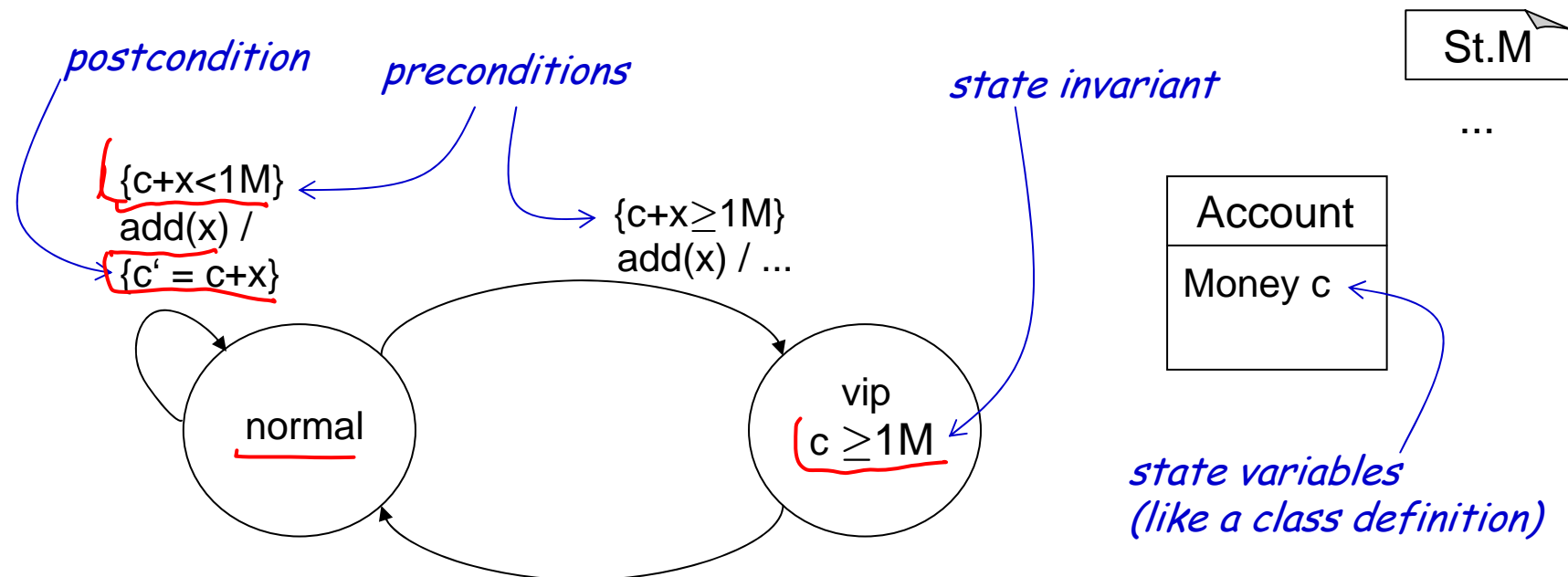
- Idea:
 - Store the relevant abstraction of the history in the automaton
 - React on an incoming event through some output and a state change

- Example: Signal : $\{\text{On}, \text{Off}, \checkmark\}^\omega \rightarrow \{\text{Ring}, \checkmark\}^\omega$



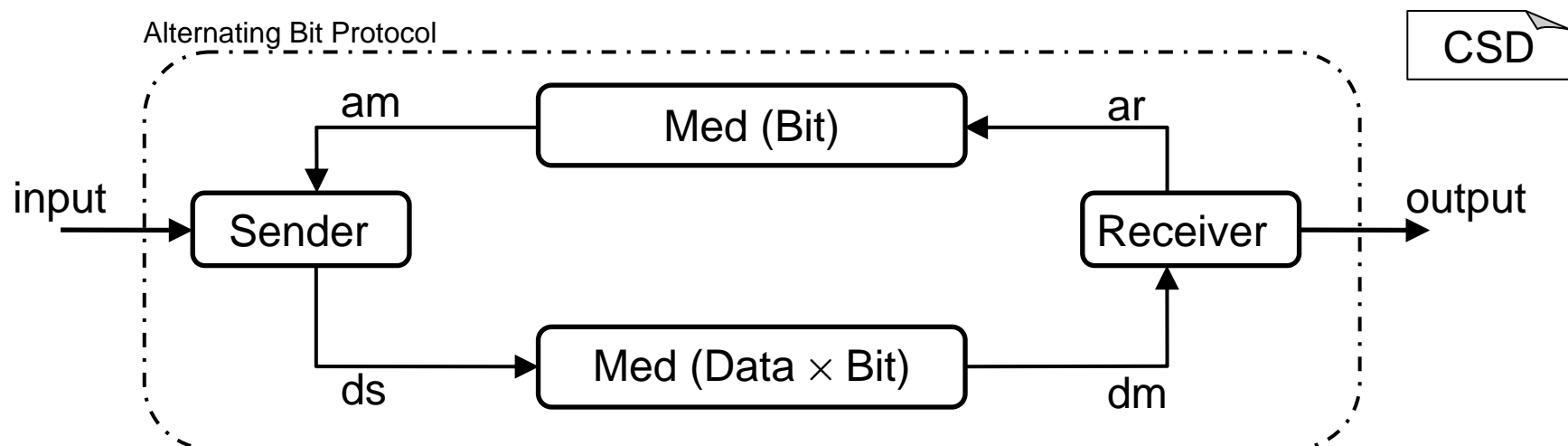
Specification through state machines -2

- Additional concepts:
 - state variables,
 - state invariants and
 - transition pre-/postconditions
- Example: account with VIP-status beyond 1M
- **Underspecification:** (1) nothing said or (2) several alternatives



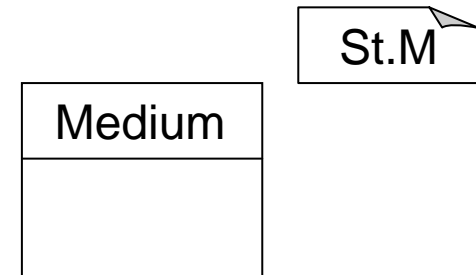
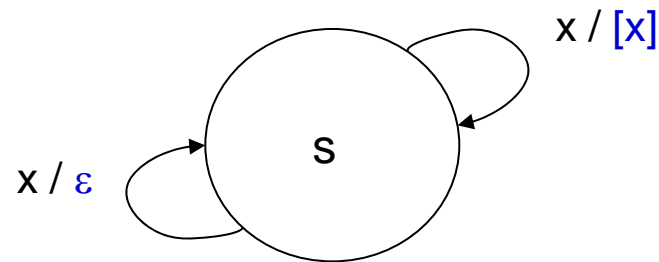
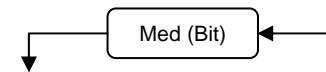
Alternating Bit Protocol

- Media can
 - delay or lose messages but cannot invent or modify them
 - eventually transmits a message correctly
- Idea:
 - sender awaits acknowledgment
 - missing acknowledgment leads to retransmission
 - one bit acknowledgment suffices



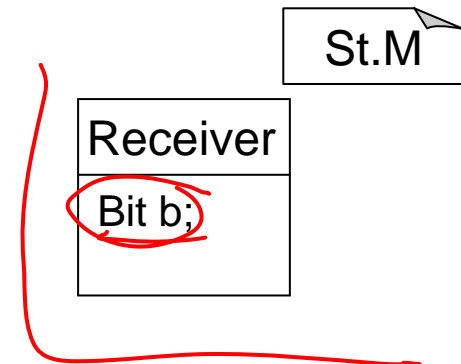
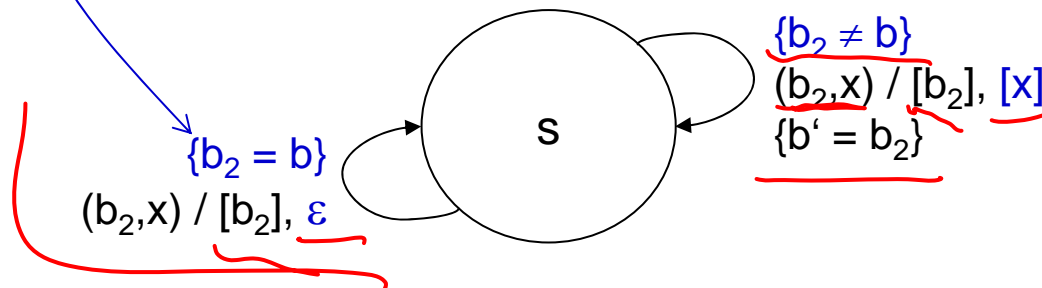
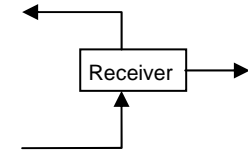
Specification of the ABP: 1) the Medium

- **Medium:** either transmits or forgets the message
- One a single state “s”
- Additional constraint: a “**fairness**” property:
 - eventually a message will pass



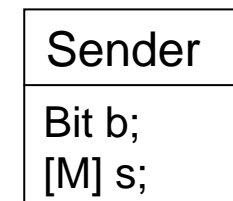
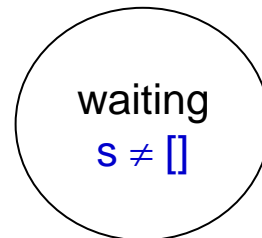
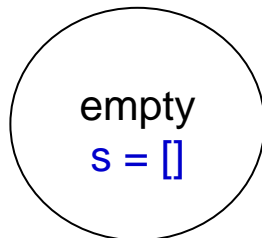
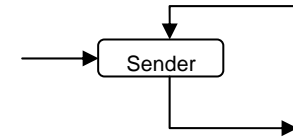
Specification of Receiver

- If a new message arrives (the alternative bit)
 - acknowledgment and data are sent
- otherwise,
 - only the acknowledgment is sent again (may have been lost)



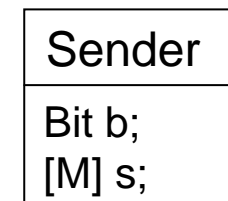
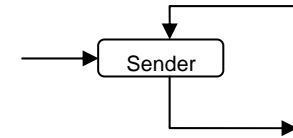
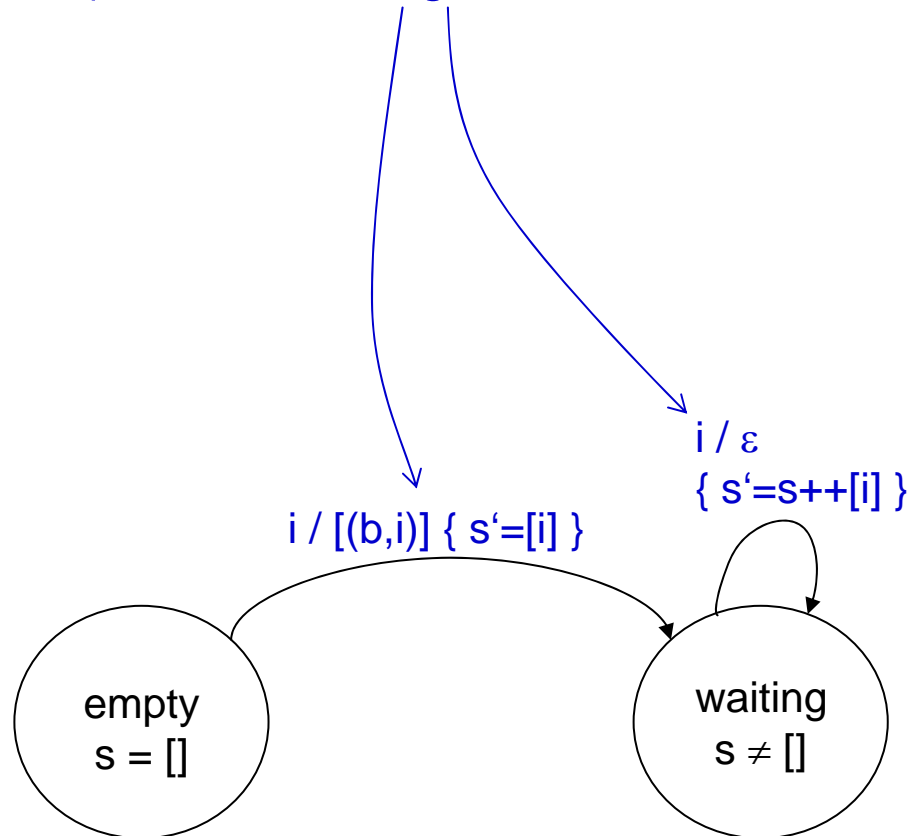
Stream-based Specification of the ABP : Sender -1

- Sender is a **time-sensitive** component: use of \surd
- **Component state space:**
 - buffer s and the alternating bit b
- **Visual state space:**
 - empty vs. nonempty buffer s
 - empty buffer also characterizes: currently nothing in transmission



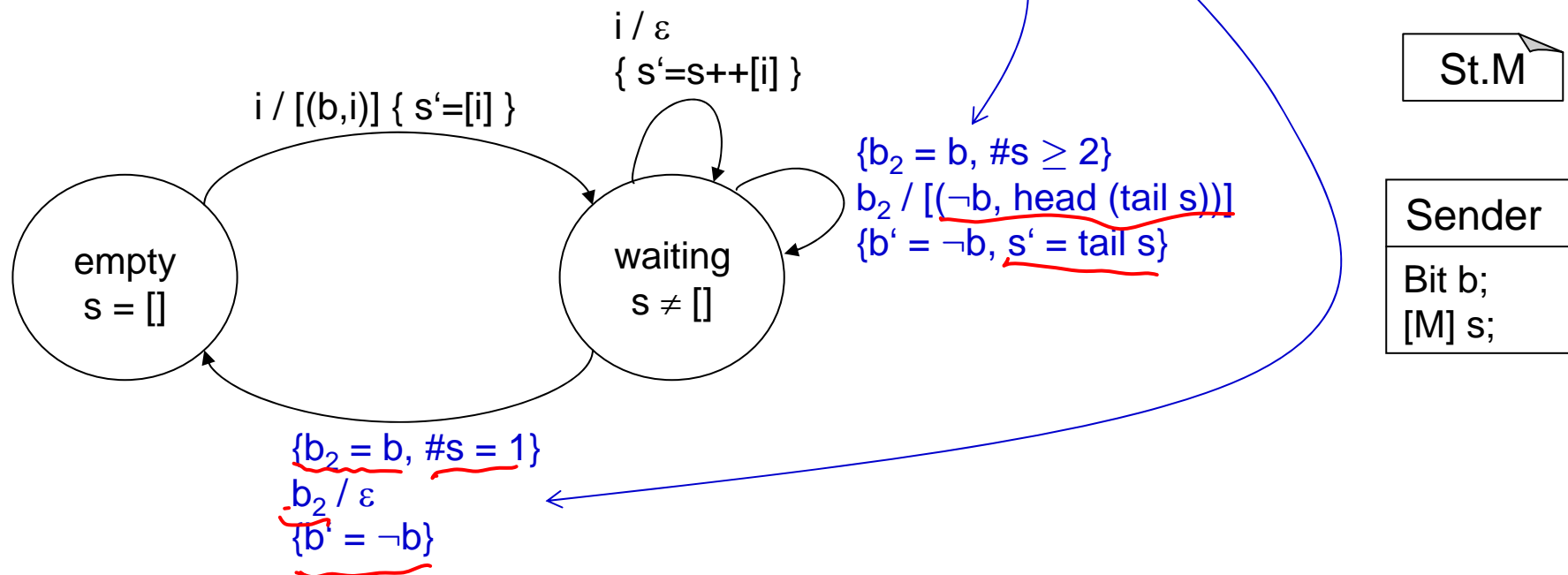
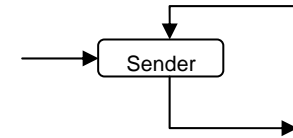
Stream-based Specification of the ABP : Sender -2

1) New message arrives



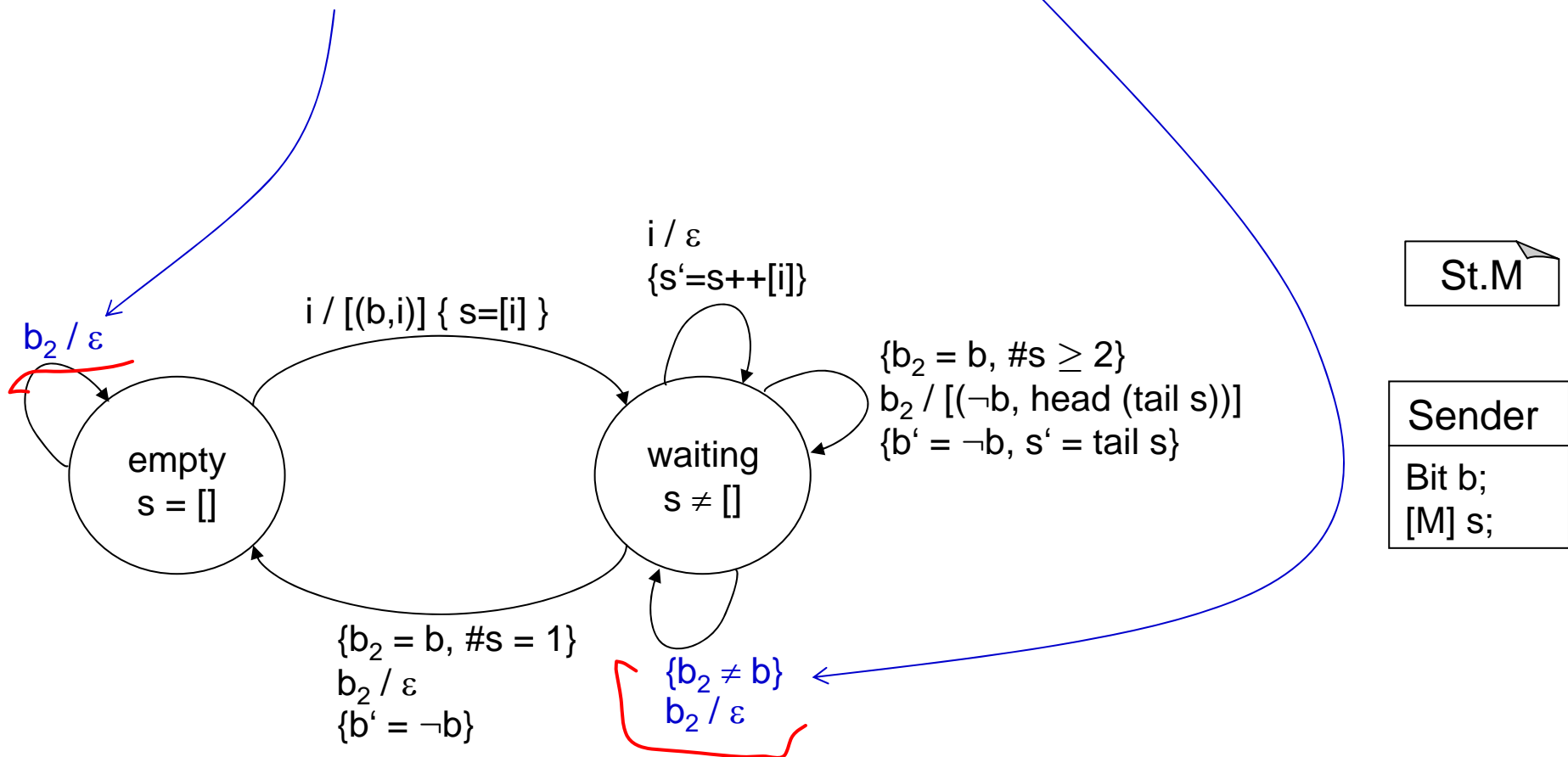
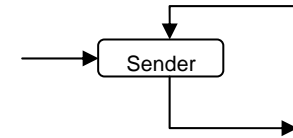
Stream-based Specification of the ABP : Sender -2

- 1) New message arrives
- 2) Correct acknowledgement arrives



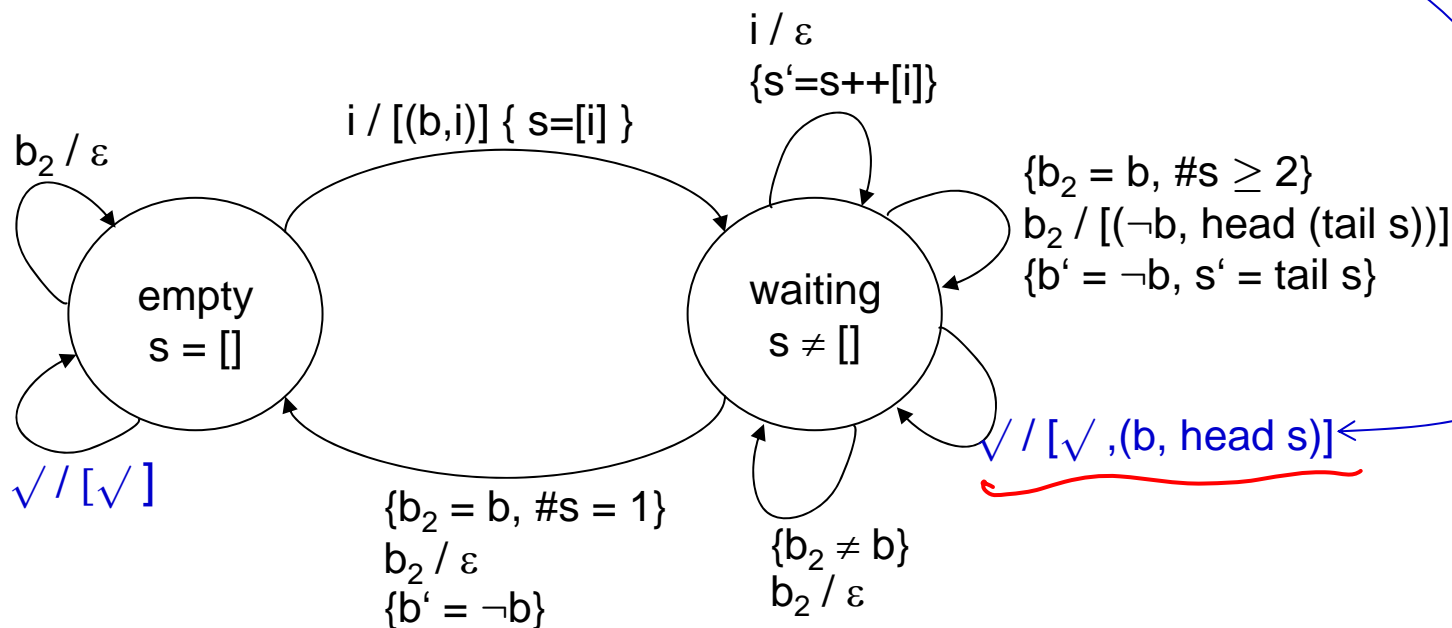
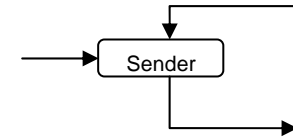
Stream-based Specification of the ABP : Sender -3

- 1) New message arrives
- 2) Correct acknowledgement arrives
- 3) Wrong acknowledgement arrives (ignored)



Stream-based Specification of the ABP : Sender -4

- 1) New message arrives
- 2) Correct acknowledgement arrives
- 3) Wrong acknowledgement arrives (ignored)
- 4) Timeout: resend last message



St.M

Sender
 Bit b;
 [M] s;

Model Driven Testing

- Along with inspections, reviews and verification techniques testing is an important **element of software quality management**
- **Large portfolio** of testing techniques
 - functional tests, black box tests, stress tests, random tests
 - glass box tests, ...
- Originally “code-based” but also applicable on the modeling level allowing for **early quality assessment**
- **Goals**
 - Demonstrate the product’s quality
 - Find errors
 - Basis for an incremental development

Model Driven Testing

- Requirements on testing
 - Automated tests
 - tests can be replayed easily to check whether important properties still hold
 - Deterministic tests with determined results
 - No side effects (to be able to repeat)
 - Good coverage of the system
(also of border cases, special cases, ...)

- Here:
 - 1) Test the model: Does it do what we want?
 - 2) Testing an implementation vs. the model?

Model Driven Testing

- 1) **Test the model:** Does it do what we want?
 - Manually find test cases that check the model
 - Use brain as oracle

- 2) **Testing an implementation** vs. the model?
 - Generate test cases from the model
 - Use model as oracle

- Both cases:
 - Check coverage of the model

- But at first:
 - “Animate the model” to be able to run tests

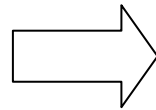
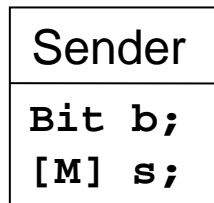
Simulation: A Mapping to Haskell

- Mapping of the model to **Haskell**
 - to run tests based on the specification
 - to simulate the system from the specification
- **Light-weight method**, no complex implementation framework needed
- **Haskell resembles the underlying communication primitives**
 - Haskell type of lists can be taken to represent streams
- **Some care must be taken**: Stream-processing functions and automata correspond to a certain style of programming
 - Input must be processed step-wise
 - Output must not be taken back
 - End-of-Input-Detection must not be used
(e.g. function “length” must not be used on streams)

Simulation: A Mapping to Haskell

- Mapping of a single component

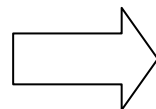
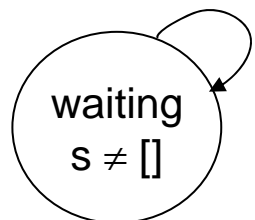
- Type definition according to the “class-like” box



type SenderState m = (Bit, [m])

- Behaviors described through state machine by **pattern matching**
 - for each state and input define a corresponding pattern
 - use the transition actions, postconditions and the target state to determine the next state and output

i / ε
 {s' = s ++ [i]}



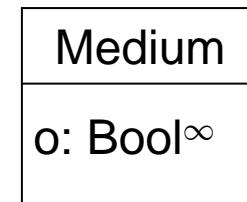
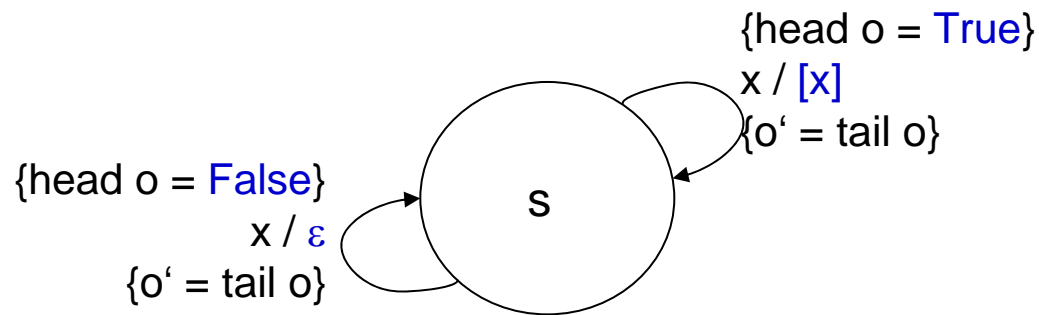
senderDelta (b, s) (MsgI i) = ((b, s ++ [i]), [])

Handwritten annotations:
 - Above (b, s): *S₁* with two downward arrows pointing to 'b' and 's'.
 - Above (MsgI i): *T_{2y}* with a bracket underneath.
 - Above ((b, s ++ [i]), []): *State* with a bracket underneath.
 - Above the final []: *Out* with a downward arrow pointing to it.

(code is simplified --> see paper)

Simulation of underspecification: Oracle for the Medium

- **Medium:** either transmits or forgets the message
- We add an **oracle** $o: \text{Bool}^\infty$ to resolve underspecification
- Controlling the oracle allows to test various situations.

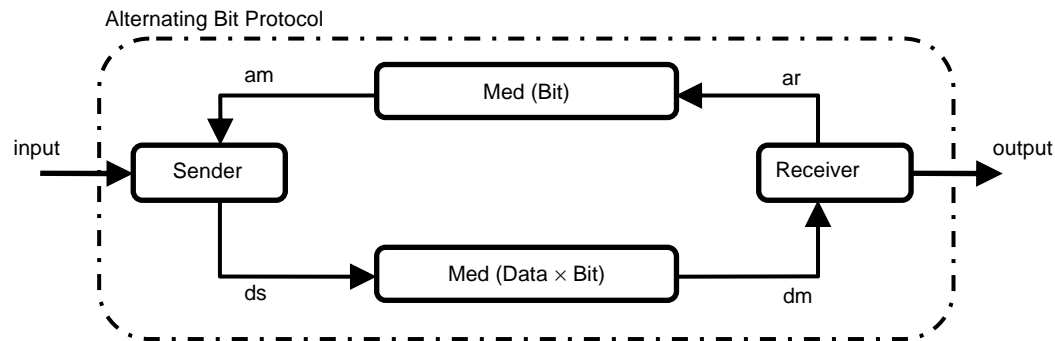


- Furthermore: predicate on oracle describes “eventually transmission happens”:

$$\#(\text{filter } \{ \text{True} \} o) = \infty$$

Simulation: A Mapping to Haskell

- Mapping of a composite component
 - composition in CSD = functional composition



```

abp (oracle1,oracle2) input = output
  where (ar,output) = receiver dm
         dm          = medium oracle1 ds
         am          = medium oracle2 ar
         ds          = sender is am
  
```

input

First attempt: Testing Functional Programs

- The result can be tested as ordinary functional program
 - Functions are **free of side effects**
 - We concentrate on **input / output behavior**
- Rules from **equivalence class testing** are applicable
 - Identify uniformly treated input values and pick representatives
 - Corner cases and special cases like empty lists, zero etc.
 - Examine **pattern matching** to indicate interesting input values
- Avoid **lambda abstractions** (better to test)
- Prefer **curried form** over tuple notation (more compact)
- Testing can be refined when recalling where functions come from (state-based models, composition)

Second attempt: Testing State-Based Models

- Component behavior specified as **state machines**
- Therefore: **Reuse well-known test strategies** for state machines
 - State coverage, Transition coverage
 - Examine pre-/postconditions, invariants
 - Minimal path coverage

- Although the system includes a **“run-time” part** (standard functionalities to execute state machines, to add time etc.)
 - this can be ignored for testing: reduces complexity
 - focus on **specified state machines**, i.e. their transition functions

Second attempt: Testing State-Based Models

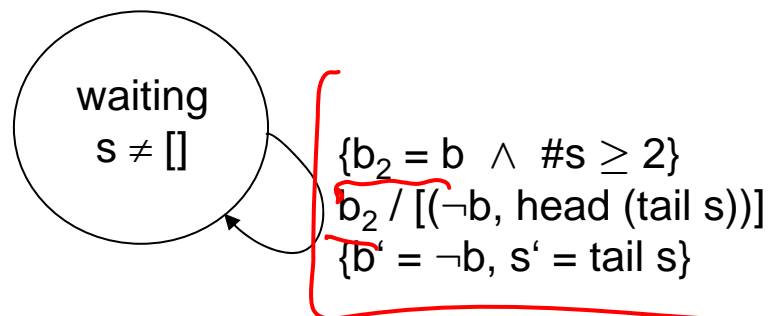
- For each component's state machine fulfill **transition coverage**
 - Every transition is executed at least once
 - Determine concrete values for start state, input, output and destination state

- Basis for transition coverage
 - Finitely many states (and transitions) obtained by grouping the possibly infinitely many states into **equivalence classes**

- **Consider pre- and postconditions**
 - Conditions can further constrain or determine possible values for states, inputs or outputs
 - Every sub-expression in a disjunction is evaluated to True/False at least once

Testing State-Based Models: Example

- The ABP's sender component has 8 transitions
 - leads to at least 8 test cases for transition coverage
- Consider the following transition of the sender component which originates and ends at a state with an arbitrary bit b and a non-empty buffer s



St.M

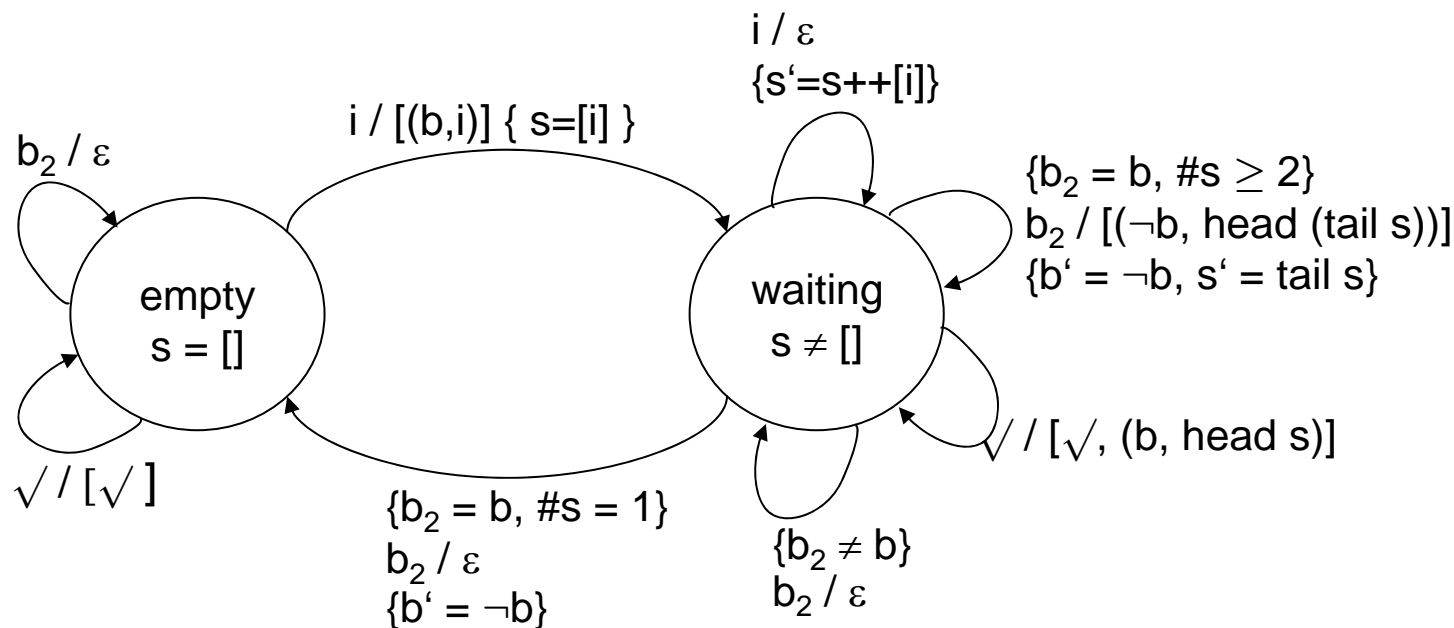
- Examination of the precondition constraint on the buffer to contain at least two messages (no restriction on b)
- A possible test initial state: $(\text{True}, [3, 4, 6])$
- Automation possible: derivation of good test cases as “usual”!

Testing State-Based Models: Example

- Given the start state $(\text{True}, [3, 4, 6])$
- precondition $\{b_2 = b\}$ leads us to **input** $b_2 = \text{True}$
- Case 1: If we check the spec and act as oracle ourselves:
 - without considering the model, we identify output and new state!
- Case 2: We check an implementation against the spec
 - we derive the transition's **output**:
$$[(\neg b, \text{head}(\text{tail } s))] \mapsto [(False, 4)]$$
 - and the **destination state** according to the postcondition
$$\{b' = \neg b, s' = \text{tail } s\} \mapsto (False, [4, 6])$$
- Automation possible:
 - execution of model gives desired output (easy if executable)
 - determining data to fulfill preconditions:
undecidable in theory / hard in practice

Testing State-Based Models

- Beyond transition coverage: **Path coverage**
 - interesting to check combined behavior of transitions
 - e.g. minimized path coverage that checks every path where loops are handled once
 - costly but can detect errors that occur in “unusual” situations
- Try (minimal) path coverage here:



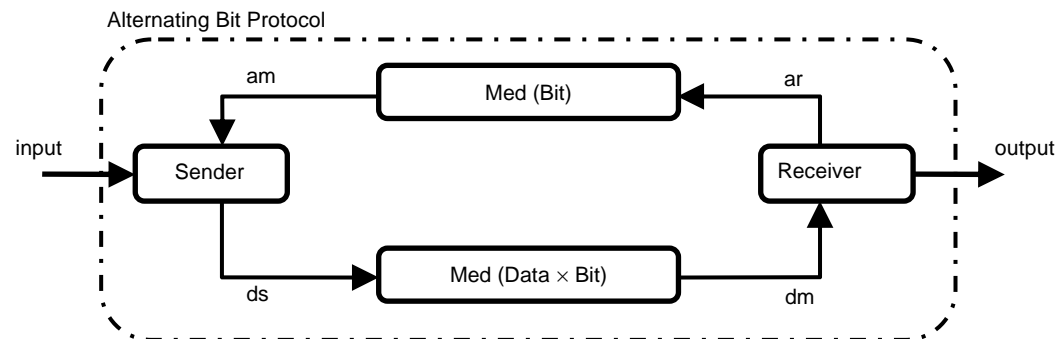
St.M

Sender

Bit b ;
 $[M] s$;

Testing State-Based Models

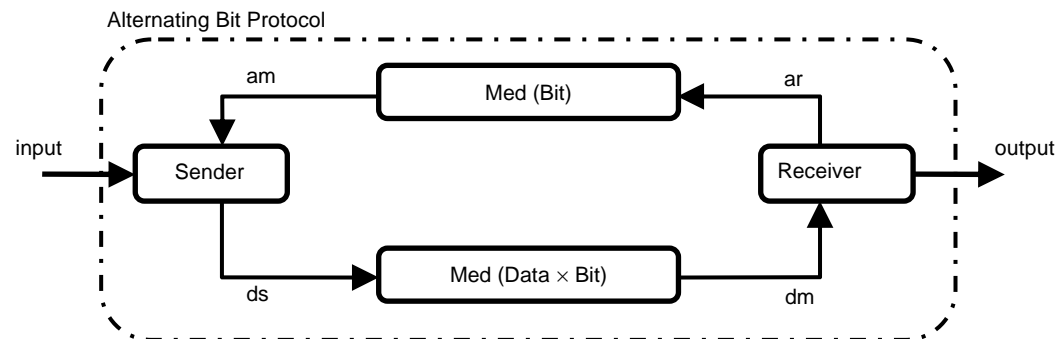
- Consider input streams
 - examine interesting message sequences and combinations (e.g. same message arrives twice etc.)
 - partition input messages into equivalence classes (if different from state partition)
 - may lead to further refinement of tests
- E.g. test ABP on sequences with repeating messages



CSD

Testing State-Based Models

- Consider oracles
 - examine the internal behavior based on oracles and identify, which oracles might define special cases
- E.g. test what happens if both media are somehow related (only one bus) and together forget each second message?



CSD

Implementing Tests in Haskell

- Main goals
 - Efficient generation and execution
 - Collect tests systematically (**test suites**)
 - Support **automated** test execution
 - Provide a light-weight test infrastructure

Light-Weight Test Infrastructure

- Only a few generic functions needed (similar to a runtime system)
- Allows us to write tests in a concise way
- Similar to unit tests for other languages

Transition Tester

- takes a transition function, state, input, expected state and output and returns a Boolean:

```
transT/deltafct (s, i, expS, expO) =  
                ((delta s i) == (expS, expO))
```

Handwritten annotations in red:
- A bracket underlines the function name `transT/deltafct`.
- A bracket underlines the parameter list `(s, i, expS, expO)`.
- A bracket underlines the expression `((delta s i) == (expS, expO))`.
- A handwritten `fct` is written below the expression, with an arrow pointing to the `delta` function.

Path Tester

- a few variants of path testers available that execute complete paths

Test Case Organization

- Assume all (sender's) transition **test cases** are given as a list „senderTransitionTests“
- Together with the test infrastructure (and possible path tests not shown here) a **test suite** is defined as

```
senderTestSuite = map (transT senderDelta)  
                    senderTranitionTests
```

- The **short cut definition**

```
allTests = and (senderTestSuite ++  
               receiverTestSuite ++ ...)
```

resembles the „green/red“-signal from unit testing

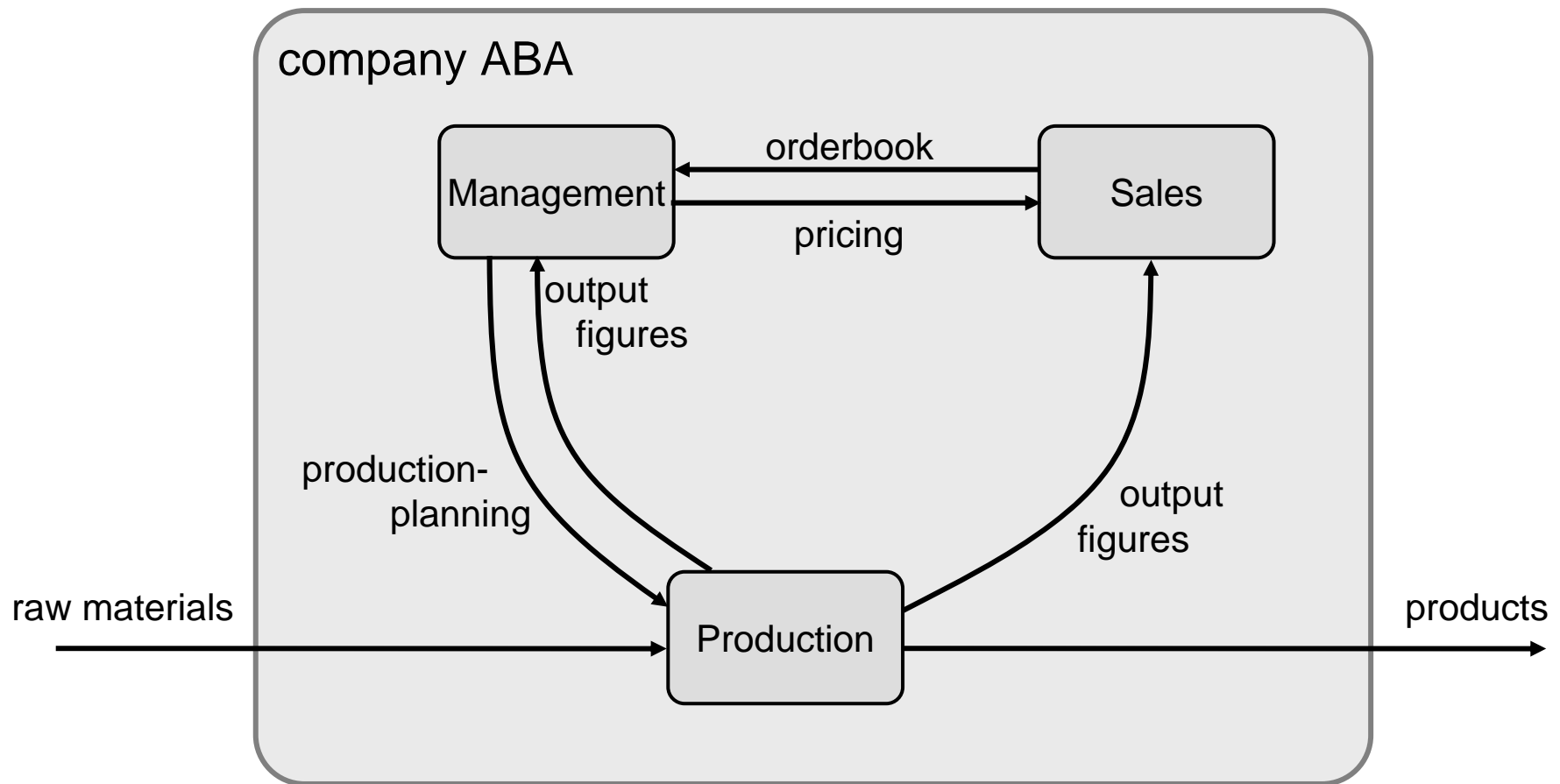
Conclusion

- Approach to **model structure and behavior** of distributed, time sensitive, asynchronously communicating systems
- Mapped to an executable form using Haskell for **early simulation and testing**
- **Systematically derived tests** from behavior specifications
- Two possible alternatives
 - in case of code generation:
 - no complete automatic tests
 - code and specification help to find tests
 - manually added expected test results
 - in case of manual implementation:
 - also derive test results from the specification used as test oracles for the actual implementation
- **Light-weight but semantically sound** approach that proves to be **effective for developers**

Thank you.

Backup

Example: dataflow in a company



Standard operators for specifying streams -1

■ Constant stream (finite and infinite):

- $\text{copy}: \mathbb{N}_\infty \times M \rightarrow M^\omega$
 - $\text{copy}(0, a) = \varepsilon$
 - $\text{copy}(n, a) = a : \text{copy}(n-1, a)$ for $n \in \mathbb{N}_\infty, n \neq 0$

■ Repeat stream n-times:

- $\text{scopy}: \mathbb{N}_\infty \times M^\omega \rightarrow M^\omega$
 - $\text{scopy}(0, s) = \varepsilon$
 - $\text{scopy}(n, s) = s \& \text{scopy}(n-1, s)$ for $n \in \mathbb{N}_\infty, n \neq 0$

■ Shorthands:

- $a^n = \text{copy}(a, n) = [a, a, \dots]$ with length n
- $s^n = \text{scopy}(s, n) = s \& s \& \dots$ with length $n \cdot \#s$
resp. 0 if $n=0$ or $s=\varepsilon$

Standard operators for specifying streams -2

- **Take n first elements:**
 - $\text{take}: \mathbb{N}_{\infty} \times M^{\omega} \rightarrow M^{\omega}$
 - $\text{take}(0,s) = \varepsilon$
 - $\text{take}(n+1,\varepsilon) = \varepsilon$
 - $\text{take}(n+1,a:s) = a:\text{take}(n,s)$

- **Drop n first elements:**
 - $\text{drop}: \mathbb{N} \times M^{\omega} \rightarrow M^{\omega}$
 - $\text{drop}(0,s) = s$
 - $\text{drop}(n+1,\varepsilon) = \varepsilon$
 - $\text{drop}(n+1,a:s) = \text{drop}(n,s)$

- **n-th Element** ($n \neq \infty, n < \#s$):
 - $\text{nth}: \mathbb{N} \times M^{\omega} \rightarrow M$
 - $\text{nth}(0,s) = \text{fst}(s)$
 - $\text{nth}(n+1,a:s) = \text{nth}(n,s)$

Standard operators for specifying streams -3

- **Map function on streams:**
 - $\text{map}: (M \rightarrow N) \times M^\omega \rightarrow N^\omega$
 - $\text{map}(f, \varepsilon) = \varepsilon$
 - $\text{map}(f, a:s) = (f\ a):\text{map}(f, s)$
- **Iterate function** repeatedly on streams: $[a, f(a), f(f(a)), \dots]$
 - $\text{iterate}: (M \rightarrow M) \times M \rightarrow M^\omega$
 - $\text{iterate}(f, a) = a:\text{iterate}(f, f(a))$
- Notation for **function definition**:
 - $(\lambda \text{ params: body})$ (like $f = (\lambda a: 2*a)$)
- **Examples:**
 - $\text{map}((\lambda a: 2*a), [1,2,3,4]) = [2,4,6,8]$
 - $\text{iterate}((\lambda a: 2*a), 1) = [1,2,4,8,16,32, \dots]$

Monotonicity of component-functions

- Given a component-function $f: I^\omega \rightarrow O^\omega$ we expect to describe a **true behavior**:
 - output once emitted cannot be taken back, resp.
 - enlarged input lead to enlarged output



Input	Output
[a]	[1]
[a,d]	[1,4]
[a,d,e]	[1,4,5]
[a,d,e,...]	[1,4,5,...]

- Formally:
- A **component-function** $f: I^\omega \rightarrow O^\omega$ must be **monotonic vs. prefixing** \sqsubseteq :
 - $\forall s, t \in I^\omega: s \sqsubseteq t \Rightarrow f(s) \sqsubseteq f(t)$
- (similar with several channels).

Continuity of component-functions

- Furthermore a component-function $f: I^\omega \rightarrow O^\omega$ cannot look into the future:

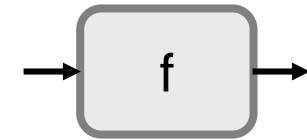
- “end of input” is not detectable
- and f cannot react on “end of input”

- Formally:

- A **component-function** $f: I^\omega \rightarrow O^\omega$ must be **continuous vs. prefixing** Ξ :

- $\forall \text{ chain } (k_i) \subseteq I^\omega: f(\sqcup (k_i)) = \sqcup (f(k_i))$

- (similar with several channels).
- Continuity implies monotonicity. Why?



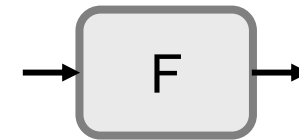
Input	Output
[a]	[]
[a,a]	[x]
[a,a,a]	[x]
a^n	[x]
a^∞	[x,y]

... this f is not continuous

Specifications & Refinement

- A function $f: I^\omega \rightarrow O^\omega$ defines exactly one behavior
- But when developing, we want to use under-specification
 - a) some behavior is yet unknown
 - b) Implementation may be nondeterministic

- **Component-specification** is therefore a set of functions



$F: \text{Set}(I^\omega \rightarrow O^\omega)$

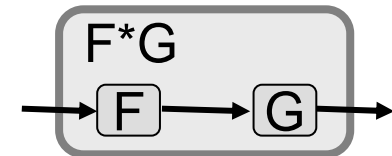
- **Refinement** of behaviors then is simply a subset relation

$$F \subseteq G$$

- with specific adaptations:
 - structural composition,
 - interface refinement, ...

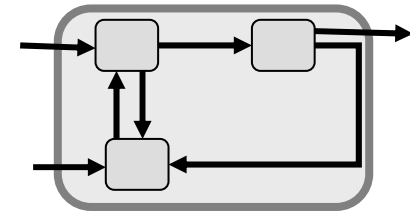
A technical extension of function application to sets

- Using sets of functions is similar to using functions:
- For $F: \text{Set}(I^\omega \rightarrow O^\omega)$, $G: \text{Set}(O^\omega \rightarrow Q^\omega)$ and $i: I^\omega$
- Pointwise applying sets of functions:
 - $F(i) = \{ f(i) \mid f \in F \} \subseteq O^\omega$
- Function composition on sets:
 - $F^*G = \{ h: I^\omega \rightarrow Q^\omega \mid \exists f \in F, g \in G: h(i) = g(f(i)) \}$
 - $= \{ h: I^\omega \rightarrow Q^\omega \mid \exists f \in F, g \in G: h = f^*g \}$

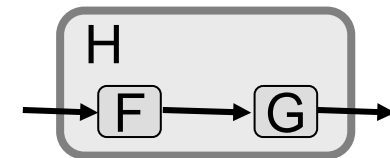


Composition

- **Composition** is given by function composition



- **Sequential composition:** $H = F * G$



- further variants: parallel, feedback
- effects: channels are function arguments and are **encapsulated**:
 - **internal channels**
 - und **explicit interfaces**

Parallel composition

- Parallel composition

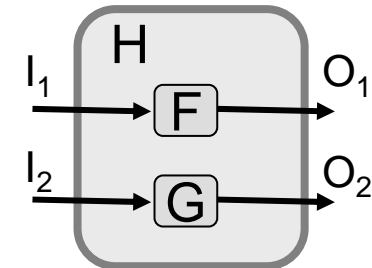
- $H = F \parallel G$

- formally:

- $H = \{ h: I_1^\omega \times I_2^\omega \rightarrow O_1^\omega \times O_2^\omega \mid$

- $\exists f \in F, g \in G:$

- $\forall s_1 \in I_1^\omega, s_2 \in I_2^\omega: \quad h(s_1, s_2) = (f(s_1), g(s_2)) \}$

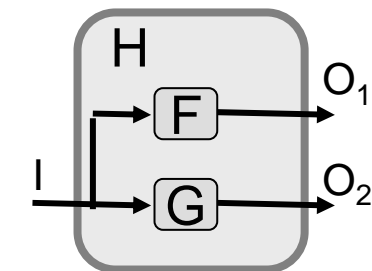


- Elementwise composition of sets
 - Various forms with more inputs and outputs
 - Sharing of inputs are allowed, e.g.:

- $H = \{ h: I^\omega \rightarrow O_1^\omega \times O_2^\omega \mid$

- $\exists f \in F, g \in G:$

- $\forall s \in I^\omega: \quad h(s) = (f(s), g(s)) \}$



Feedback

- Feedback on channel J

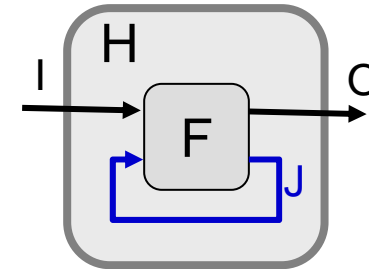
- $H = \mu_J F$

- formally:

- $H = \{ h: I^\omega \rightarrow O^\omega \mid$

- $\exists f \in F:$

- $\forall s \in I^\omega: \exists j \in I^\omega: (h(s), j) = f(s, j)) \}$



- Feedback (with encapsulation of an internal channel)

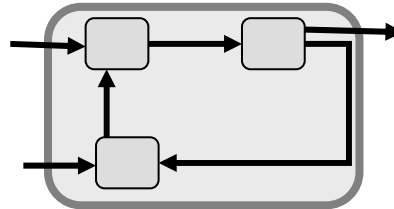
- effects: channels are function arguments and are **encapsulated**:

- internal channels**

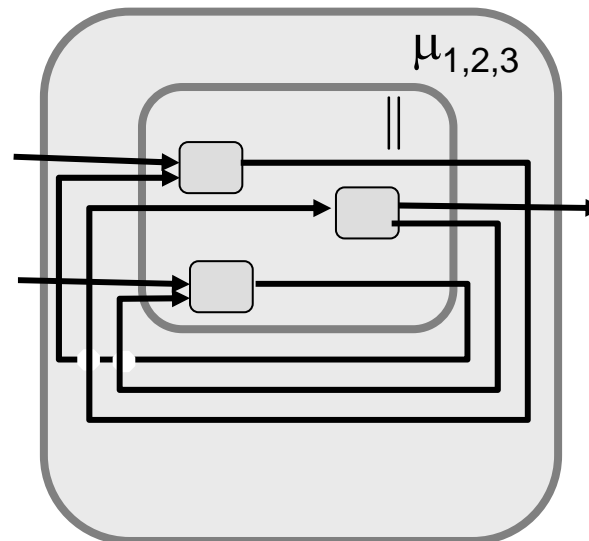
- and **explicit interfaces**

General Composition

- All general forms of composed systems can be defined using these primitives \parallel , μ_J :
 - Example:



- is equal to



Refinement & Composition

- Refinement of behaviors F to G is a subset relation

$$F \subseteq G$$

- Refinement is compatible with composition:

- Formally:

- $F \subseteq G \Rightarrow$
 - $F \parallel H \subseteq G \parallel H$
 - $\mu_j F \subseteq \mu_j G$
- and also:
 - $F^*H \subseteq G^*H$
 - $H^*F \subseteq H^*G$

- Proof? See Definitions!

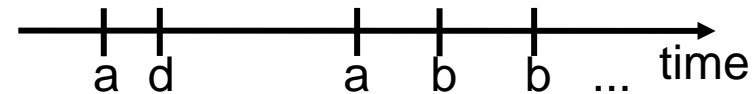
- This is a very important property:

- If a part is refined, then also the composed whole!

Messages and Channels

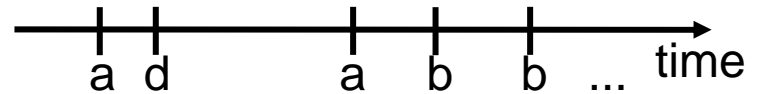
- Dataflow of a channel is observed through

- a finite or infinite stream of messages $M^\omega = M^* \cup M^\infty$
- example: $[a,d,a,b,b]$:



- Messages are flowing over a channel
 - There is a precedence in the messages
 - Nothing is said about timing!
- But: Distributed systems often have **real-time constraints**:
 - E.g. need to react if a message does not arrive in time
- Solution: Modeling time within the stream

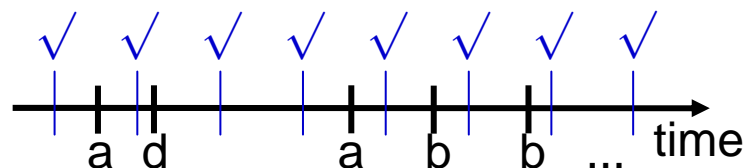
Timed observations on channels



- Ideas:
- a) attach a time stamp to each message
 - like: [(a,13ms), (d,17ms), (a,35ms), ...]
 - Problem: context conditions on the timing + awkward to handle
- b) improved: attach timing difference between messages
 - like: [+13ms, a, +4ms, d,+18ms, a, ...]
 - Problem: still awkward to handle, very detailed information
- c) more abstract: Using **Ticks** ✓ to model equidistant frames of time
 - like: [✓, a, ✓, d, ✓, ✓, a, ✓, b, ✓, b, ✓, ...]:
- Variants:
 - c1) **exactly one message** per time frame (Schaltnetze)
 - c2) **not more than one message** per time frame
 - c3) **finitely many messages** per time frame

Timed Streams: Definition

- Concept: using ticks \surd to model that time passes
 - \surd model equidistant passing of time (e.g. milliseconds)
- Timed dataflow of a channel is an observation
 - a finite or infinite stream of messages and ticks, where
 - $M^\surd = \{ s \in (M \cup \{\surd\})^\omega \mid \forall n \in \mathbb{N}: \text{nth}(n,s) \neq \surd \Rightarrow \text{nth}(n+1,s) = \surd \}$
 - observation intervals are detailed:
 - at maximum one message per time interval
 - sufficiently fine granular view on the channel
- Example: $[\surd, a, \surd, d, \surd, \surd, a, \surd, b, \surd, b, \surd, \dots]$:



Timed Streams: Consequences

- Timed dataflow of a channel is an observation
 - $M^\surd = \{ s \in (M \cup \{\surd\})^\omega \mid \forall n \in \mathbb{N}: \text{nth}(n, s) \neq \surd \Rightarrow \text{nth}(n+1, s) = \surd \}$
 - observation intervals are detailed:
 - at maximum one message per time interval
 - It follows: infinite streams have infinitely many \surd :
 - $\forall s \in M^\surd: \#s = \infty \Rightarrow \#\text{filter}(\{\surd\}, s) = \infty$
 - Thus infinite observations cover the complete time axis
 - Todo: Write a function to check above property in Hugs
 - $\text{check} :: \mathbb{N} \rightarrow (M \cup \{\surd\})^\omega \rightarrow \mathbb{B}$



Example: Timer

- Timer receives initialization value, counts down to zero and emits an alert “!”
 - $\text{timer} :: \mathbb{N}^\vee \rightarrow \text{String}^\vee$
 - $\text{timer} = \text{timH } 0$

 - -- helper function:
 - $\text{timH} :: \mathbb{N} \rightarrow \mathbb{N}^\vee \rightarrow \text{String}^\omega$
 - $\text{timH } 1 (\sqrt{:}xs) = \sqrt{:}!"":(\text{timH } 0 \text{ } xs)$ -- alert
 - $\text{timH } 0 (\sqrt{:}xs) = \sqrt{:} \text{ timH } 0 \text{ } xs$ -- inactive state “0”
 - $\text{timH } n (\sqrt{:}xs) = \sqrt{:} \text{ timH } (n-1) \text{ } xs$ for $n > 0$ -- count down
 - $\text{timH } n (c:xs) = \text{timH } c \text{ } xs$ for $c \neq \sqrt{}$ -- initialize with c

 - Important: whenever $\sqrt{}$ is consumed, also $\sqrt{}$ is produced!

 - Todo: mapping to hugs?



Example: Merge

- The merge combines the values of two channels
 - $\text{Merge} \subseteq M^\vee \times M^\vee \rightarrow M^\vee$
- Specified by:
 - $\text{Merge} = \{ \text{merge} \mid \exists \text{merge2} \in \text{Merge} :$
 - $\forall xs, ys \in M^\vee, a \in M :$
 - $\text{merge} (\sqrt{:}xs) (\sqrt{:}ys) = \sqrt{:}(\text{merge2 } xs \text{ } ys) \wedge \text{-- timestep}$
 - $(\text{merge } (a:xs) \quad ys = a :(\text{merge2 } xs \text{ } ys) \vee$
 - $\text{merge} \quad xs \text{ } (a: ys) = a :(\text{merge2 } xs \text{ } ys) \quad) \}$
- Observations:
 - Merge is **underspecified**:
Choice of message order if both channels have one
 - Problem: If both channels have a message result has two messages in one time unit
- Todo:
 - Mapping to hugs (deterministic variant of merge)

Example: Merge with delay

- A **merge** component **with delay** to prevent several messages within a time frame
 - $\text{MergeD} \subseteq M^\vee \times M^\vee \rightarrow M^\vee$
 - $\text{MergeDH} \subseteq \text{List}(M) \times M^\vee \times M^\vee \rightarrow M^\vee$ -- **helper** with state

- Specified by:
 - $\text{MergeD} = \text{MergeDH} []$ -- read equation element wise
 - $\text{MergeDH} = \{ \text{mergeDH} \mid \exists \text{mergeDH2} \in \text{MergeDH}:$
 - $\forall xs, ys \in M^\vee, a \in M, b \in M, bs \in \text{List}(M):$
 - $\text{mergeDH} [] (\sqrt{:xs}) (\sqrt{:ys}) = \sqrt{:(\text{mergeDH2} [] xs ys) \wedge}$
 - $\text{mergeDH} (b:bs) (\sqrt{:xs}) (\sqrt{:ys}) = b \text{ :}(\text{mergeDH2} bs xs ys) \wedge$
 - $(\text{mergeDH} bs (a :xs) ys = \text{mergeDH2} (bs\&[a]) xs ys) \vee$
 - $\text{mergeDH} bs xs (a :ys) = \text{mergeDH2} (bs\&[a]) xs ys) \}$

- Observations:
 - Again mergeD is **underspecified**

- Todo: Implement in Hugs (deterministic variant)



Underspecification in Hugs

- Problem: Sets of functions cannot be implemented in Hugs.
- Example: **Lossy transmission**
(internet occasionally loses messages)
 - $\text{Transmit} \subseteq M^\vee \rightarrow M^\vee$
 - $\text{Transmit} = \{ f \mid \exists g \in \text{Transmit}: \forall x \in M, xs \in M^\vee:$
 - $f(\sqrt{:}xs) = \sqrt{:}g(xs) \quad \wedge \quad (f(x:xs) = g(xs) \vee f(x:xs) = x:g(xs)) \quad \}$
- Solution: Simulate underspecification through an Oracle!
 - **type Oracle** = [Bool] -- Infinite list
- Example revisited:
 - $\text{transmit} \in \text{Oracle} \times M^\vee \rightarrow M^\vee$
 - $\text{transmit} \quad \text{os} \quad (\sqrt{:}xs) = \sqrt{:}:(\text{transmit os xs})$
 - $\text{transmit} (\text{True :os}) \quad (x:xs) = x :(\text{transmit os xs})$
 - $\text{transmit} (\text{False:os}) \quad (x:xs) = \quad \text{transmit os xs}$

- Todo: Implement transmit and timed fair merge in Hugs

Introducing time in untimed channels

- Given: untimed observation $S \subseteq M^\omega$
- Needed: timed observations $T \subseteq M^\vee$
- Idea:
 - Pair of mappings
 - **untimed** : $M^\vee \rightarrow M^\omega$
 - **timed** : $M^\omega \rightarrow \text{Set}(M^\vee)$
- where
 - $\text{untimed } xt = \text{filter } M \text{ } xt$
 - $\text{timed } xs = \{ xt \mid \text{untimed } xt = xs \}$

 - $\text{untimed } T = \{ \text{filter } M \text{ } xt \mid xt \in T \}$ -- extended to sets
 - $\text{timed } S = \bigcup_{s \in S} \text{timed } s$
- Todo: untimed and oracle based implementation of timed operator
- Later more on the properties of oracles!

