Runtime Verification

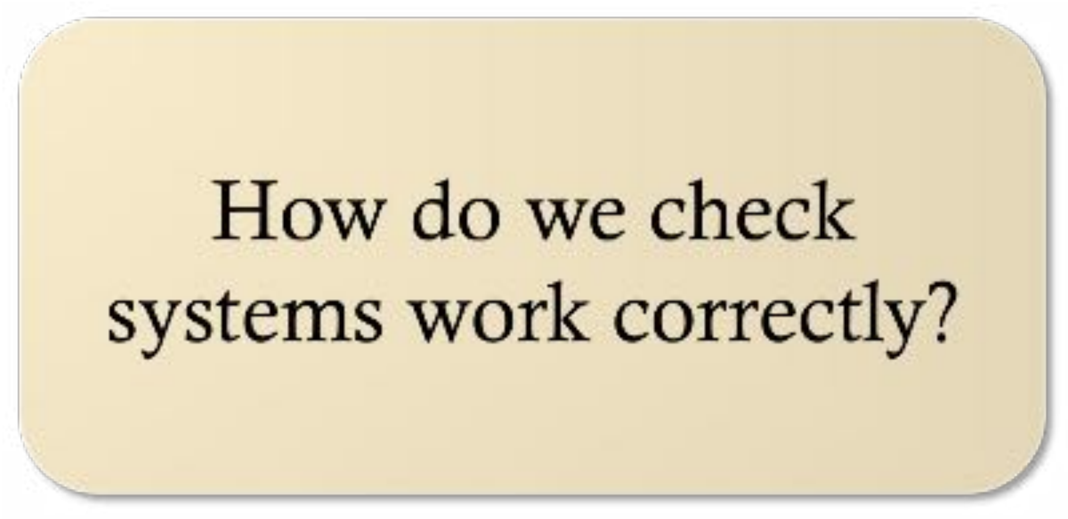
Gordon J. Pace
University of Malta

May 2015

Why Verify?

- Systems are growing:
  - … in size
  - … in complexity
  - … in the ways they interact with real-life objects
- Leading to more opportunities for failure,
- and making the consequences of failure more serious
- Meaning we need to ensure our systems work correctly.

The Question


How do we check systems work correctly?

Let’s Start with Testing

- Before deployment (and ideally throughout the development phase)…
- try the program along different execution paths to see whether the program works correctly.

Testing: The Challenges… (1)

- What constitutes a correct program?
  - Black listed users may not start a transaction.
  - Account balances may never go below zero.
  - No transaction may last longer than 3 hours.
- Oracles used to check for correct functional behaviour

Testing: The Challenges… (2)

- What paths to test?
  - Choices over inputs
  - Non-determinism

Testing: The Challenges… (2)

```
function transfer(act_source, act_dest, amount) {
       balance = act_source.getBalance();
       if (act_source.isWhiteListed() &&
           balance >= amount) {
              act_dest.deposit(amount);
              act_source.withdraw(amount);
       }
}
```

Testing: The Challenges… (2)

```
if (webinterface.getTransferRequest()) {
       if (webinterface.fromField.valid &&
           webinterface.toField.valid &&
           webinterface.amountField > 0) {
              transfer(
                     webinterface.fromField,
                     webinterface.toField,
                     webinterface.amountField);
       }
}
```

Testing: The Challenges… (2)

```
if (webinterface.getTransferRequest()) {
       if (webinterface.fromField.valid &&
           webinterface.toField.valid &&
           webinterface.amountField > 0) {
              transfer(
                     webinterface.fromField,
                     webinterface.toField,
                     webinterface.amountField);
       }
}
```

- **Question 1:** What values and accounts should we test the function on?

Testing: The Challenges… (2)

```
thread1.execute(transfer(joe, peter, 1000));
thread2.execute(transfer(joe, peter, 5000));
```

- **Question 2:** What about non-determinism?

Testing: The Challenges… (2)

transfer
(joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

transfer
(joe, peter, 5000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

Testing: The Challenges… (2)

transfer (joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

transfer (joe, peter, 5000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

Testing: The Challenges… (2)

transfer
(joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amo
                act_source.withdraw(
        }
}
```

getBalance returned $5600

transfer
(joe, peter, 5000)

```
transfer(act_source, act_dest, amount)
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

Testing: The Challenges… (2)

transfer (joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

transfer (joe, peter, 5000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

Testing: The Challenges… (2)

transfer
(joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

transfer
(joe, peter, 5000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

Testing: The Challenges… (2)

transfer
(joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() &      lance >= amount) {
                act_dest.deposit(amo
                act_source.withdraw(
        }
}
```

getBalance returned $5600

transfer
(joe, peter, 5000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

Testing: The Challenges… (2)

transfer (joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

transfer (joe, peter, 5000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

Testing: The Challenges… (2)

transfer
(joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

transfer
(joe, peter, 5000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

Testing: The Challenges… (2)

transfer
(joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {


        }
}
```

transfer
(joe, peter, 5000)

```
transfer(ac
        bal
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

The result is that the account balance goes below zero.

Testing: The Challenges… (2)

transfer
(joe, peter, 1000)

```
transfer(act_source, act_dest, amount) {
        balance = act_source.getBalance();
        if (act_source.isWhiteListed() && balance >= amount) {



        }
}
```

transfer
(joe, peter, 5000)

```
transfer(ac
        bal
        if (act_source.isWhiteListed() && balance >= amount) {
                act_dest.deposit(amount);
                act_source.withdraw(amount);
        }
}
```

Intelligent test case generation is challenging.

Summary: Testing

- Pros:
  - (Relatively) easy to set up
  - Can be performed throughout the development phase
  - (Usually) tests can be rerun across versions
- Cons:
  - Difficult to generate paths intelligently
  - Can only talk about what happens, not about what may happen
  - We can (normally) never say “Correct”

Verification Summary

- In an ideal world, we would like to ensure that our systems cannot fail…
- … but testing leaves out paths which may occur in practice.

Some Observations…

- Observation 1:
  a. Checking a whole system is usually computationally expensive…
  b. But checking a single path is usually not (hence oracles in testing).
- Observation 2:
  a. Generating a representative set of paths is tough…
  b. because the system may go through some other path which we may not have checked before.

A Logical Conclusion…

- So why not:
  - check properties only on certain execution paths
  - but continue checking after deployment to ensure that any execution paths followed by the live system do not violate the property,
  - and if they do fix the system or just stop it.

A Logical Conclusion…

runtime monitoring

- So why not:
  - check properties only on releva execution paths
  - but continue checking after deployment to ensure that any execution paths followed by the live system do not violate the property,
  - and if they do fix the system or just stop it.

A Logical Conclusion…

- So why not:
  - check properties only on relevant execution paths
  - but continue checking after deployment to ensure that any execution paths followed by the live system do not violate the property,
  - and if they do fix the sy

runtime verification

So Finally, Runtime Verification

- Monitor what the system is doing…
- verifying that it does not violate any property.
- If it does, stop the system or fix it.

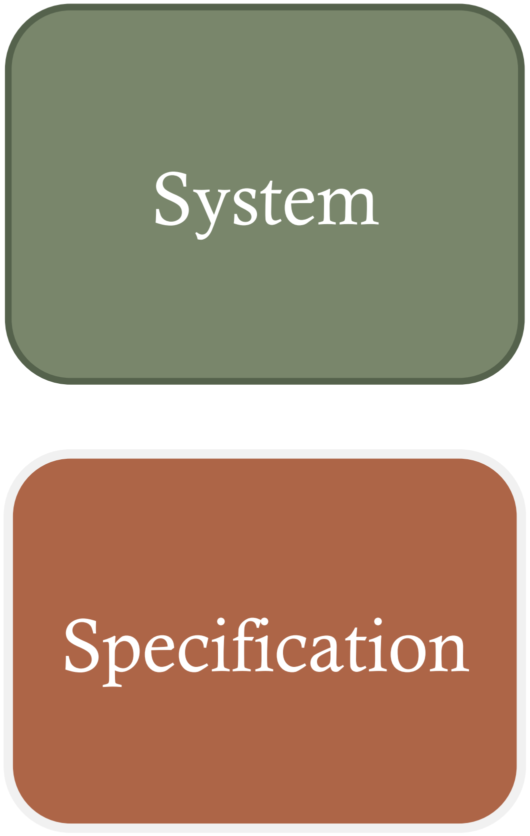
Summary: Runtime Verification

- Pros:
  - We can say “The system never continues after failure.”
  - Scales up.
  - Easy to adopt.
- Cons:
  - We can never say “The system is correct.”
  - Overheads (time and memory) may be prohibitive.
  - Can only talk about finite traces.
  - Can only be performed at runtime.
  - Does not remove bugs, but stops their consequences.

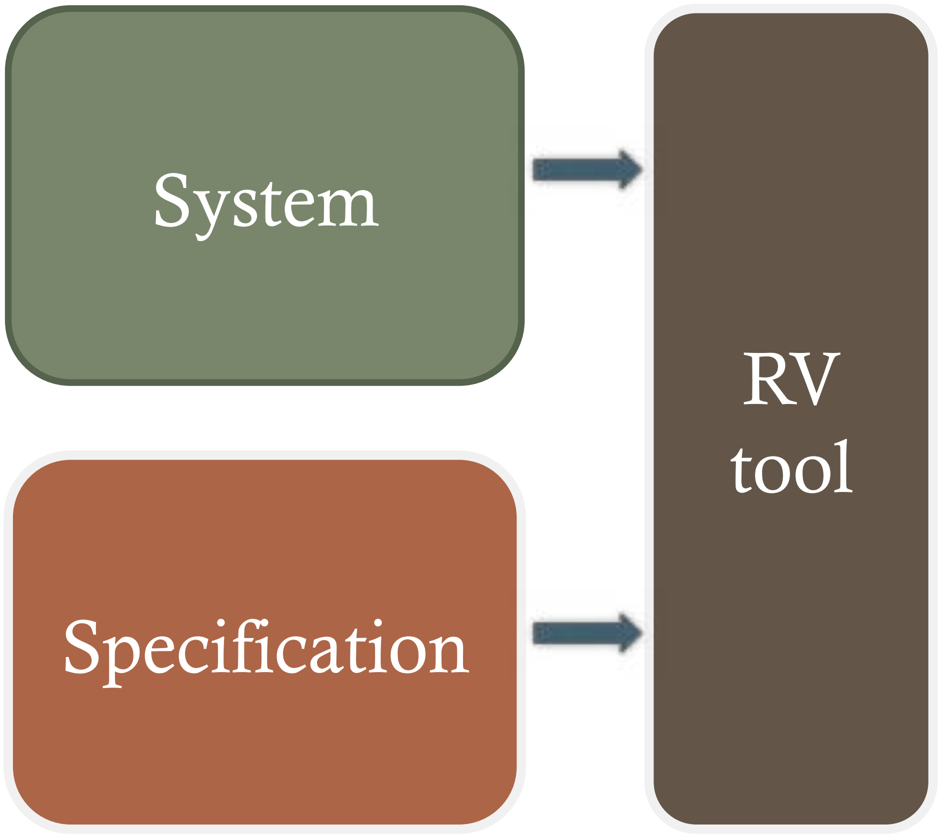
Part II

An Overview of Runtime Verification

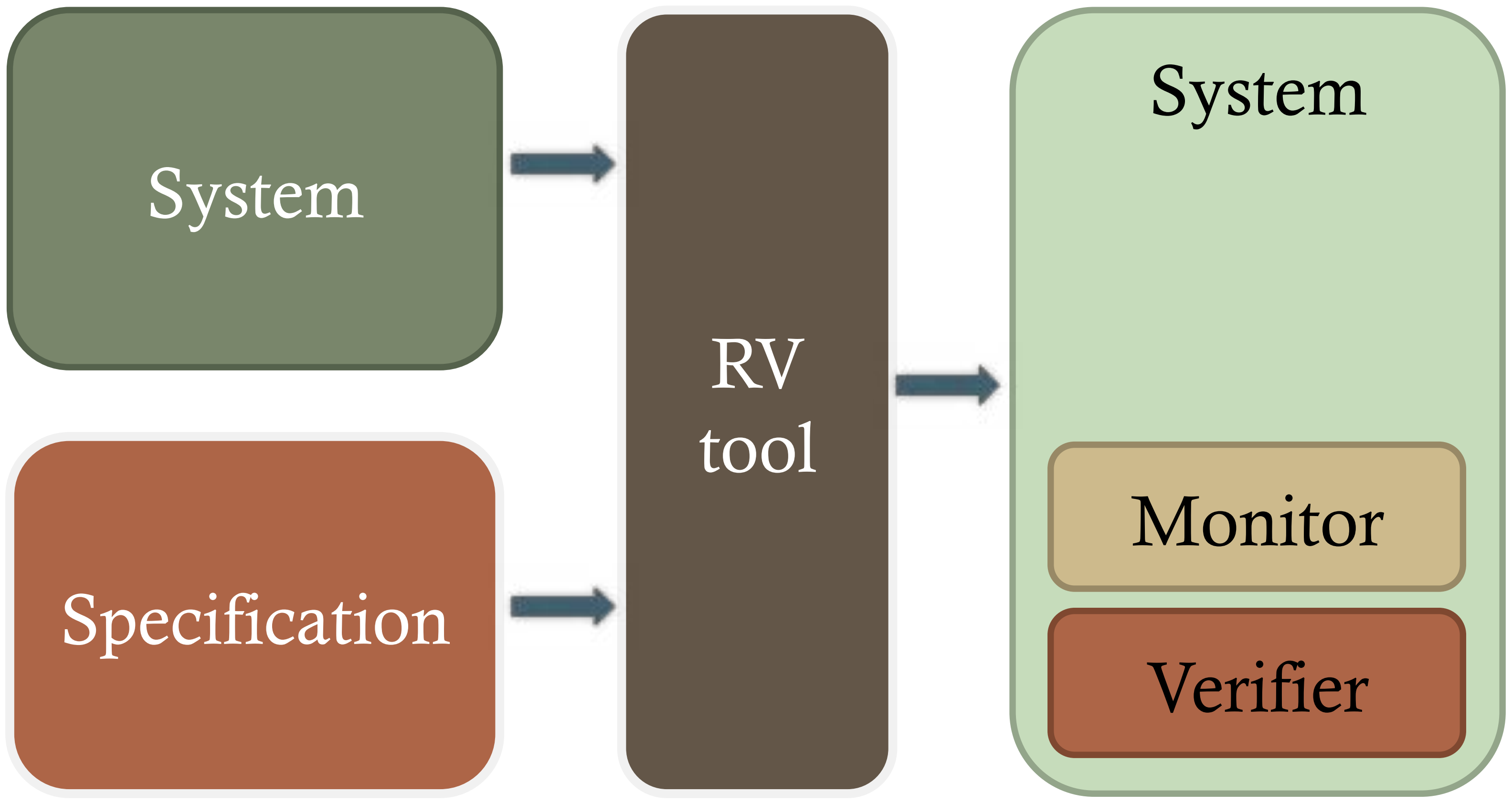
Runtime Verification



Runtime Verification



Runtime Verification



Some Issues and Choices

- Where do we write the properties?
- How do we express the properties?
- Where does the verification code go?
- How does the verification code communicate with the main system?

Writing Specifications

- Two primary options:
  - **Option 1:** Inlining monitors
  - **Option 2:** Separate specification from system description

Inlining Specifications: Assertions

```
function startTransaction(User u) {
      assert(!u.isBlacklisted());
      Transaction t = createTransaction();
      if (u.isDormant()) {
            t.makePending();
      }
      assert(t.isActive());
      return(t);
}
```

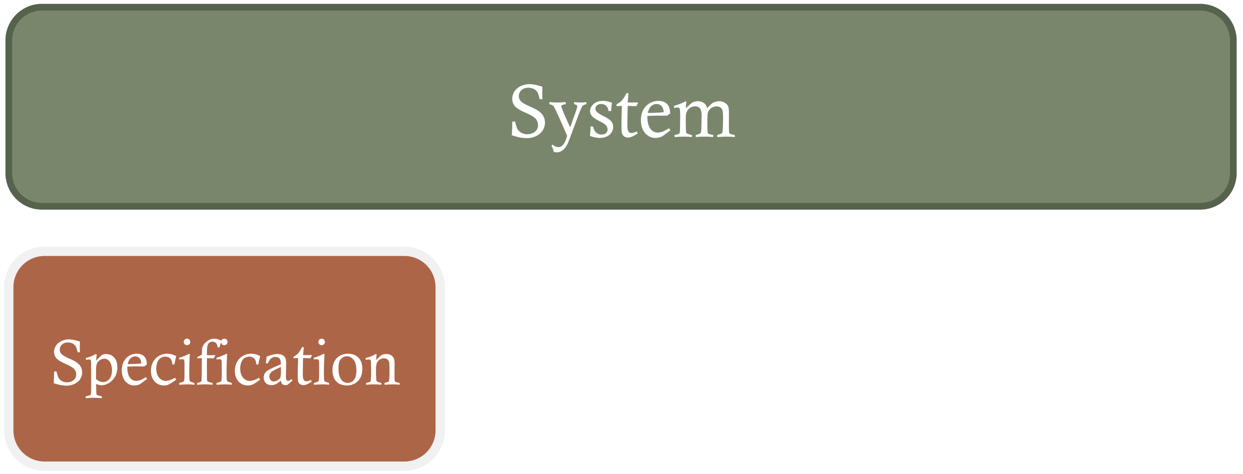
Inlining Specifications: Assertions with Additional Logic

```
function beginTransaction(User u) {
        Transaction t = createTransaction();
        …
        ongoingTransactions.add(t);
}

function endTransaction(User u, Transaction t) {
        assert(ongoingTransactions.hasItem(t));
        …
        ongoingTransactions.removeItem(t)
        …
}
```

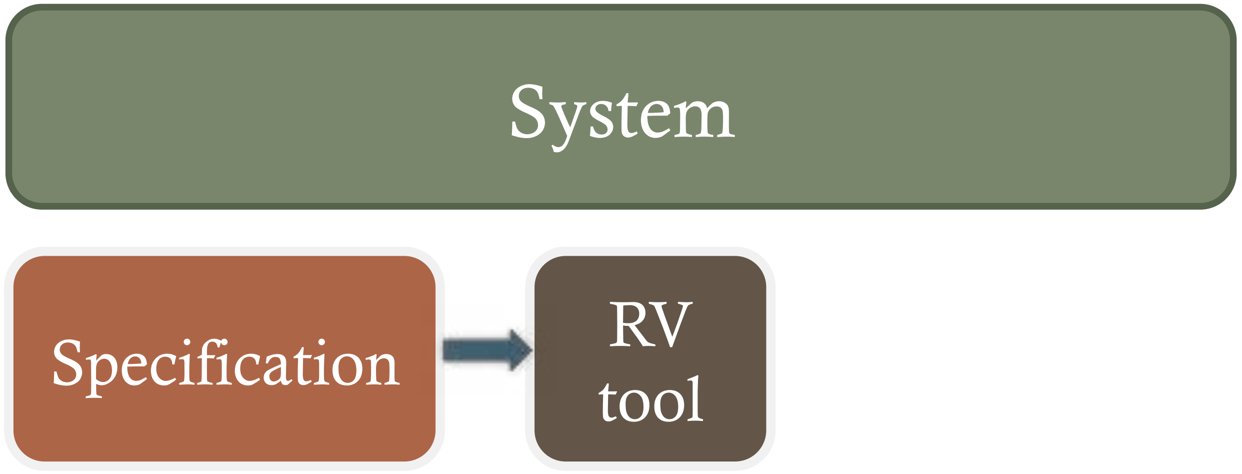
Separation of Concerns

- Separating specification from the system requires additional work to *weave* the two together.



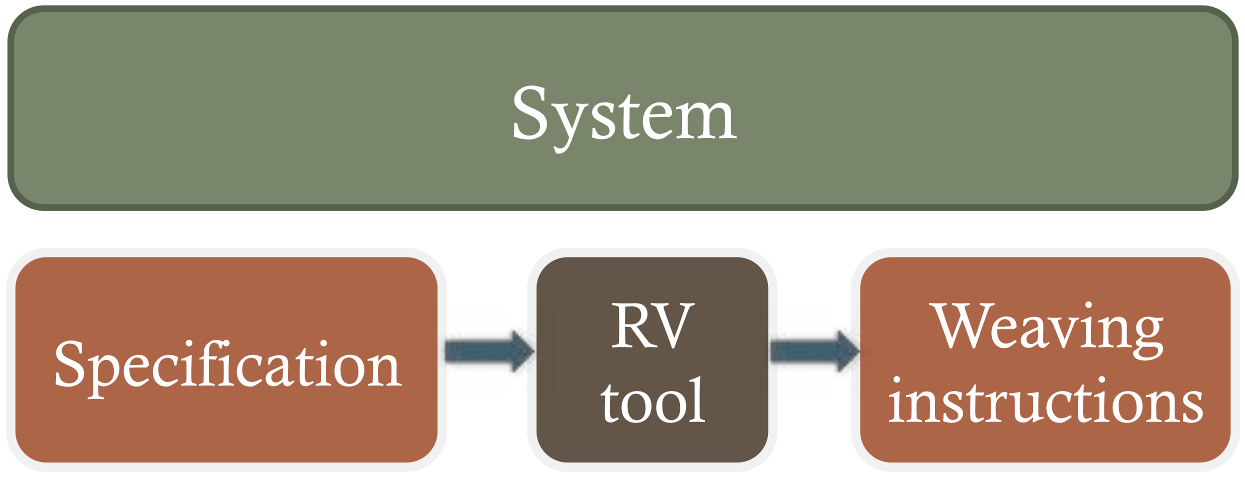
Separation of Concerns

- Separating specification from the system requires additional work to *weave* the two together.



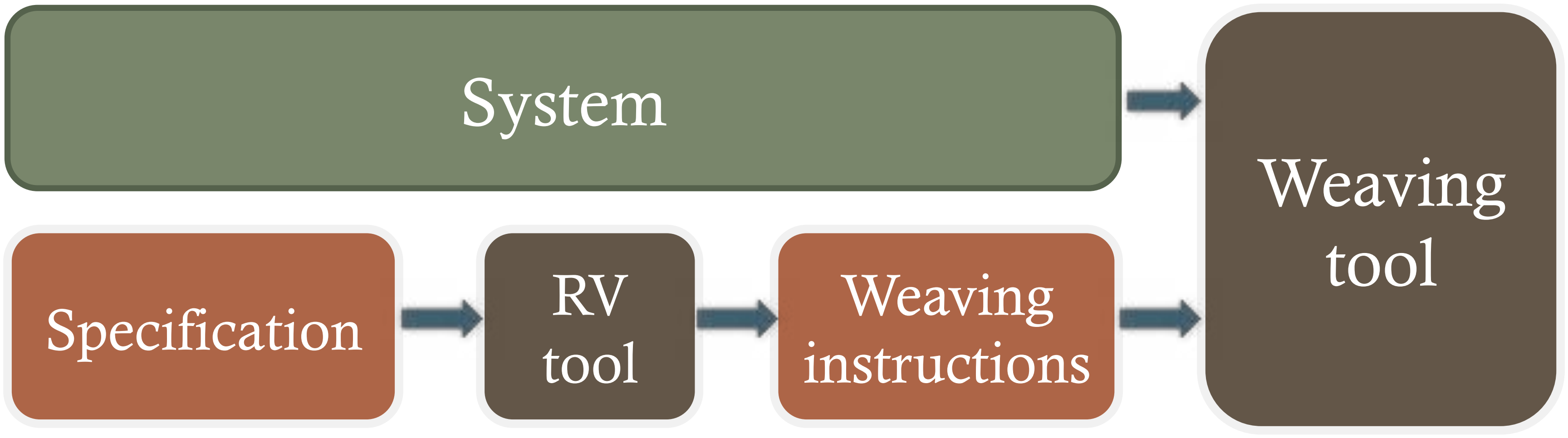
Separation of Concerns

- Separating specification from the system requires additional work to *weave* the two together.



Separation of Concerns

- Separating specification from the system requires additional work to *weave* the two together.



Separation of Concerns: Program Transformations

- Tools exist to transform a program into another program in the same language
- Typically such tools access the abstract syntax tree of the source program, and produce the abstract syntax tree of the destination program.

Separation of Concerns: Aspect-Oriented Programming

- Modular abstraction techniques e.g. object-oriented design aims at abstracting models – sharing common parts across the implementation.
- Sometimes, one modularisation strategy foregoes another, requiring code replication and the merging of business-logic and support code.
- Aspect-oriented programming gives ways of changing code across a whole system in a modular way.

Separation of Concerns: Aspect-Oriented Programming

- Consider adding a logging feature for all types of money transfers, and which can be toggled or or off.
- Either we add a line to each transfer method:

```
void transfer() {
        if (log.enabled) { log.write("…"); }
        ...
}
```

- Or we can use an AOP tool and write:

```
@before call(*.transfer(..))
        log.write("…");
```

Property Specification Languages

- Writing properties as assertions at particular points in the program is very restrictive:
  - Choice of points is purely syntactic, not semantic e.g. specifying *"y should be 0 when x becomes 42"*.
  - Any reasoning about context has to appear as additional code intermingling with the system code e.g. *"**startTransaction** should be called before **endTransaction**"*.
  - Richer logics enable implicit specification of such context.

Property Specification Languages

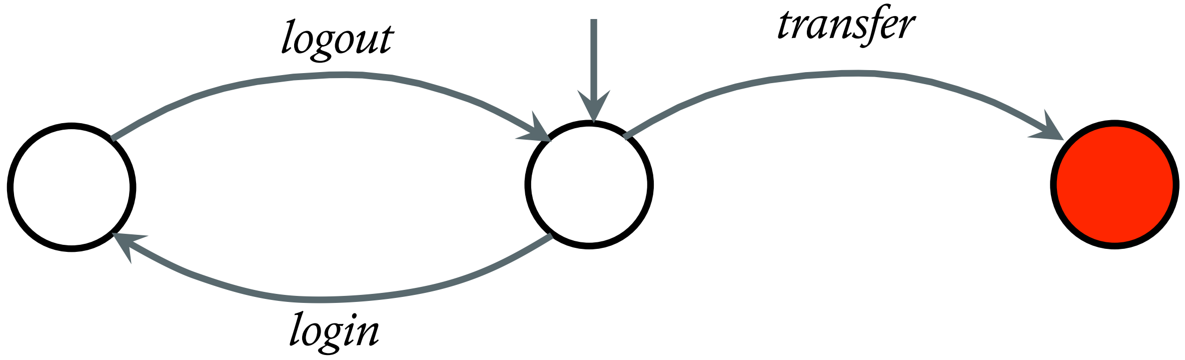
- Temporal logics enable reasoning about events as they happen over time.
- Basic events usually still control-flow (syntactic) rather than data-flow (semantic) due to overheads:
  - upon starting a method call
  - at the end of a method call
  - when an exception is raised

Property Specification Languages: Automata

- Finite state automata (e.g. in the form of UML diagrams) can be a good way of allowing the specification of consequentiality behaviour.

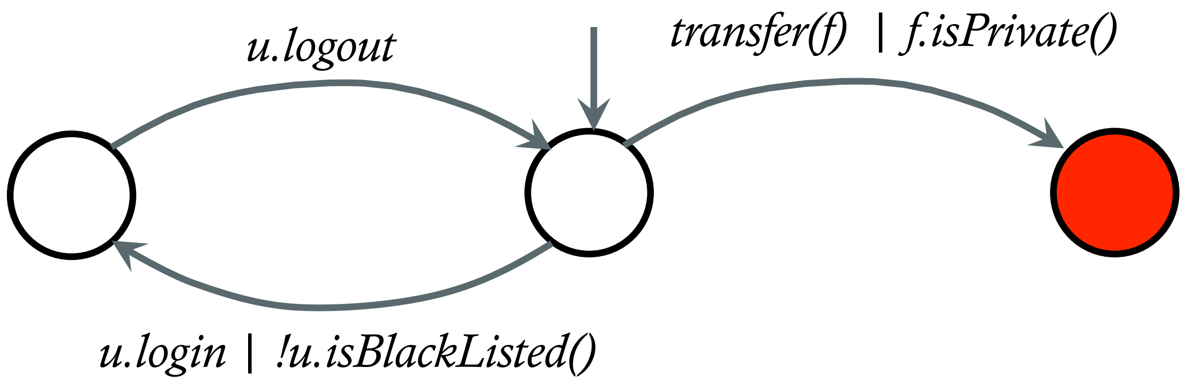
Property Specification Languages: Automata

- Finite state automata (e.g. in the form of UML diagrams) can be a good way of allowing the specification of consequentiality behaviour.



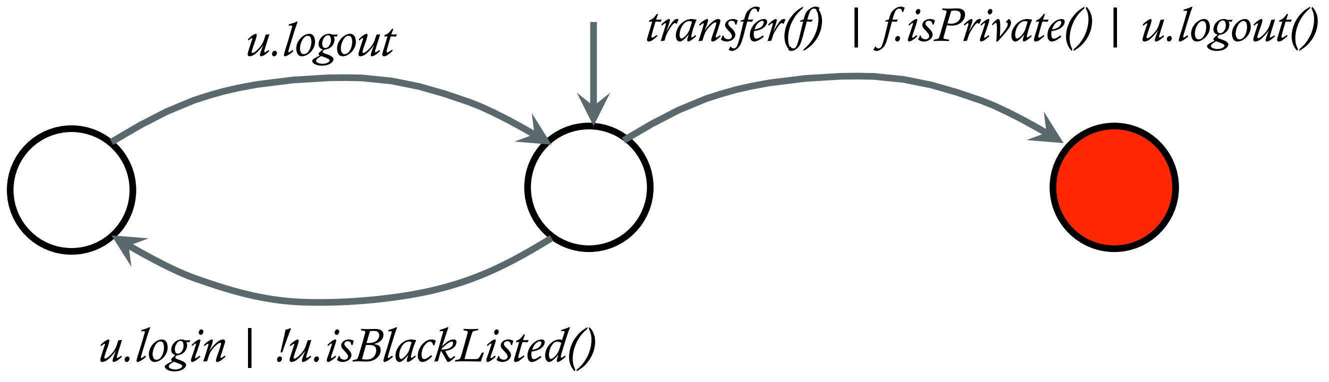
Property Specification Languages: Automata

- Access to system data enhances expressiveness…



Property Specification Languages: Automata

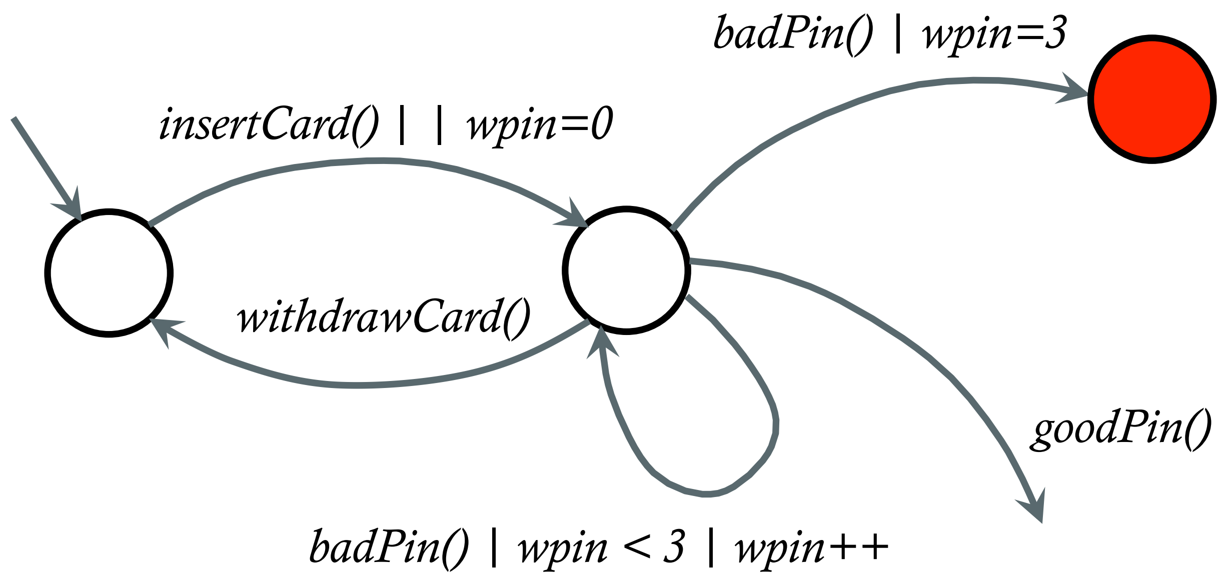
- Access to system data enhances expressiveness and interaction with the system…



u.logout
transfer(f) | f.isPrivate() | u.logout()
u.login | !u.isBlackListed()

Property Specification Languages: Automata

- Access to data enhances expressiveness and allows for more compact descriptions:



Property Specification Languages: Logic-Based

- Textual logics, with a grammar, can be more effective in some settings.
- For instance, consider **Regular Expressions**…

Property Specification Languages: Logic-Based

- Regular expressions can also be used for temporal specifications:

$$\begin{array}{lcl} \psi & ::= & \text{event} \\ & | & \overline{\text{event}} \\ & | & ? \\ & | & 1 \\ & | & 0 \\ & | & \psi + \psi' \\ & | & \psi\psi' \\ & | & \psi^* \end{array}$$

Property Specification Languages: Logic-Based

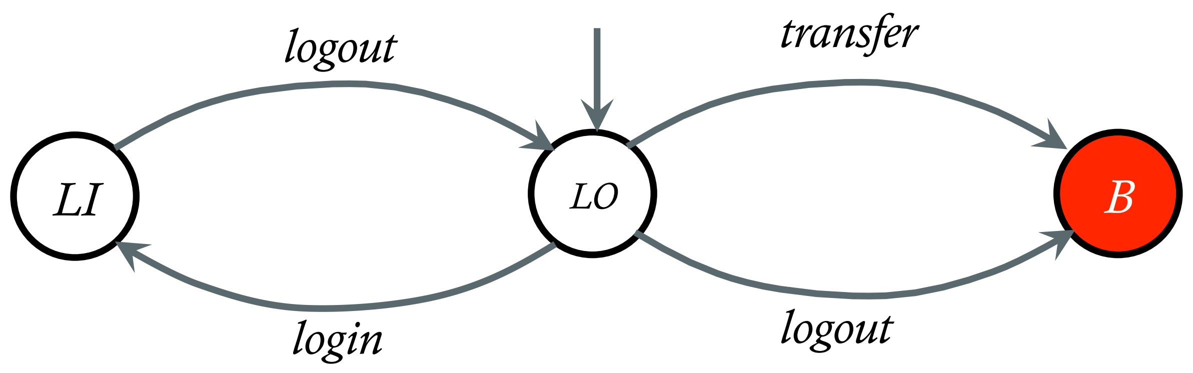
- Regular expressions can also be used for temporal specifications.
- Examples:
  - ?* login logout
  - (login (write + read)* logout)* (read + write)

Verification Algorithms

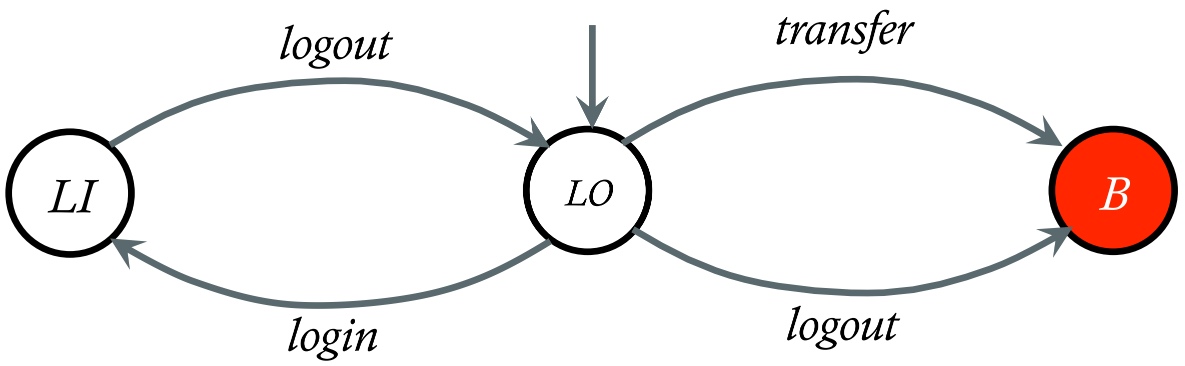
- Given a specification written in a particular logic, how can we automatically check whether a trace matches it?
- For online monitoring, the algorithm should ideally be *incrementally computable* from left to right.
- For offline monitoring this need not be the case, and more efficient parsing algorithms can be used.

Verification Algorithms for Automata

- Consider the use of automata for RV:



Verification Algorithms for Automata



- Instrumentation is straightforward:

```
@before call(*.login(..))
      if (state==LO) { state = LI; }
@before call(*.logout(..))
      if (state==LI) { state = LO; } else
      if (state==LO) { state = B; reportError(); }
@before call(*.transfer(..))
      if (state==LO) { state = B; reportError(); }
```

Verification Algorithms for Logics

- Let us consider regular expressions again:

$$
\begin{array}{rcl}
\psi & ::= & \mathrm{event} \\
 & | & \overline{\mathrm{event}} \\
 & | & ? \\
 & | & 1 \\
 & | & 0 \\
 & | & \psi + \psi' \\
 & | & \psi\psi' \\
 & | & \psi^*
\end{array}
$$

Verification Algorithms for Logics

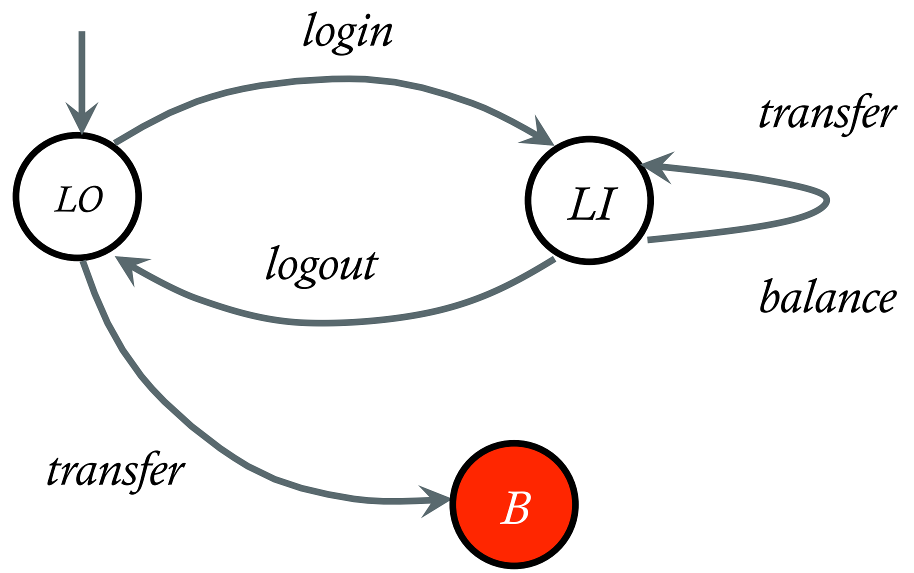
- One possibility is to translate regular expressions into automata.

$$(\text{login}(\text{transfer} + \text{balance})^*\text{logout})^*\text{trasfer}$$

Verification Algorithms for Logics

- One possibility is to translate regular expressions into automata.

$$(\text{login}(\text{transfer} + \text{balance})^*\text{logout})^*\text{trasfer}$$



Verification Algorithms for Logics

- Given a regular expression *e* to match, and the first event *a*, identify a regular expression *f* such that after *a*, we can equivalently match *f*.
- More precisely: Given property *e*, find *e'* such that any trace *aw* matches *e* if and only if *w* matches *e'*.
- This together with an algorithm to check whether the empty trace matches property *e* suffices.

Verification Algorithms for Logics

- Another possibility is to define a *residual algorithm* for regular expressions.
- Starting from a regular expression to match *e*, what remains to be matched after receiving a particular event *e?*

Verification Algorithms for Logics

(login (transfer + balance)* logout)* transfer

→login→

(transfer + balance)* logout
(login (transfer + balance)* logout)* transfer

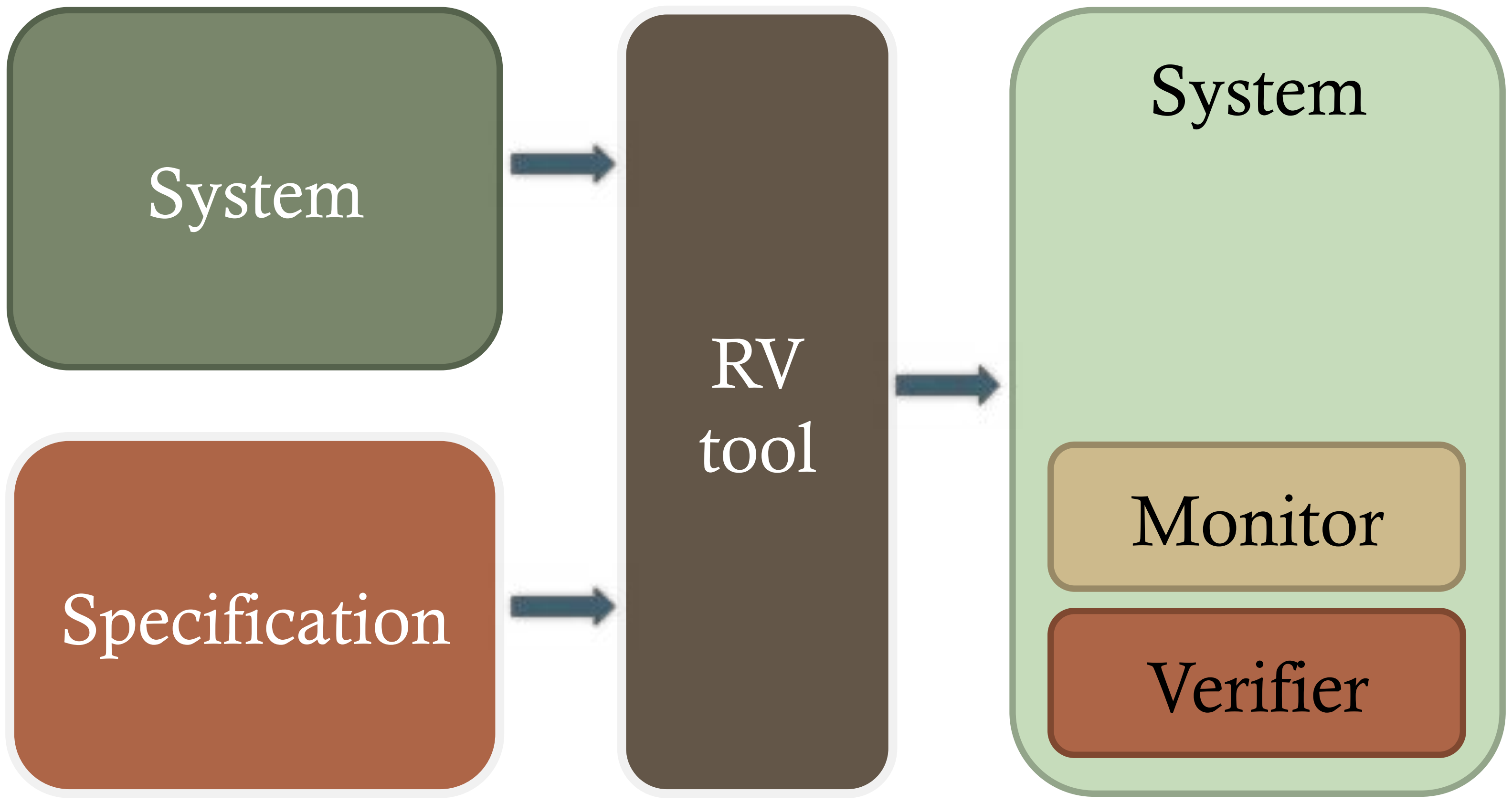
→logout→

(login (transfer + balance)* logout)* transfer

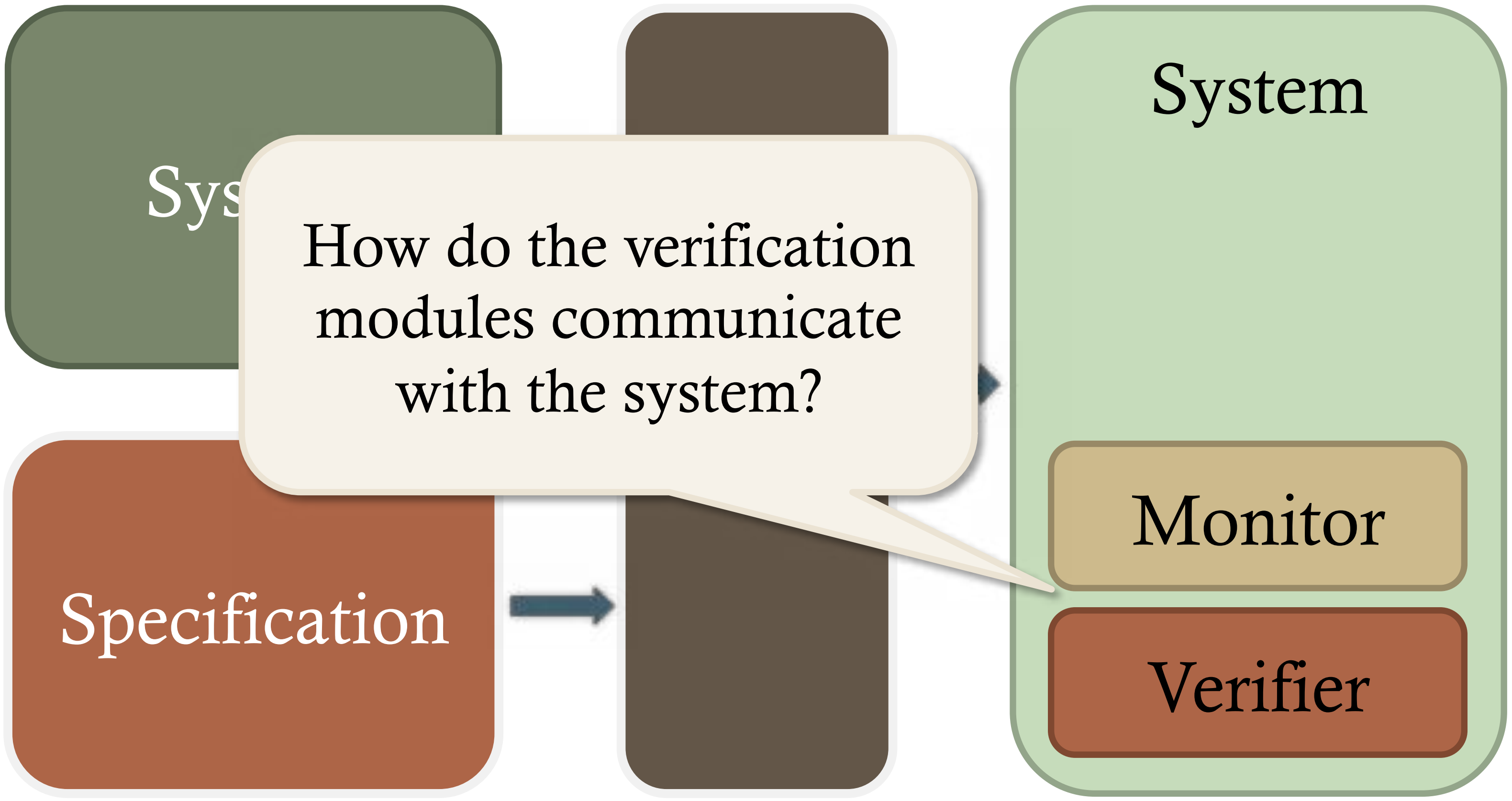
→transfer→

MATCH!

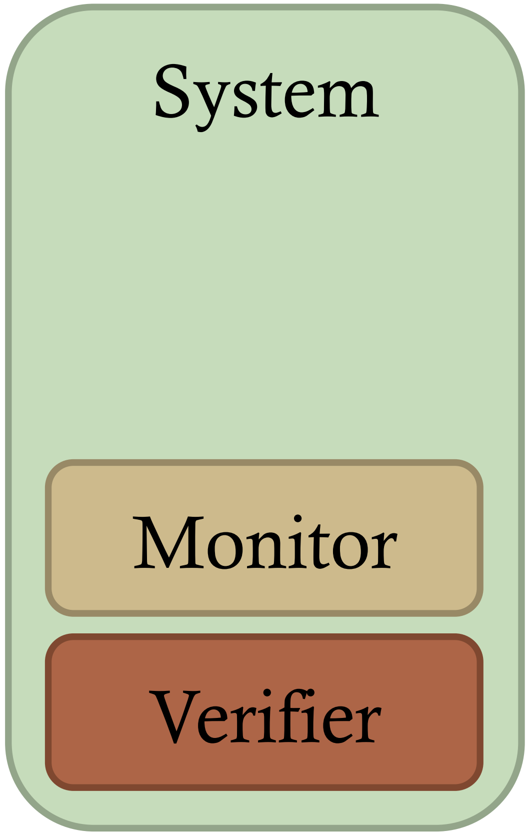
RV Architecture



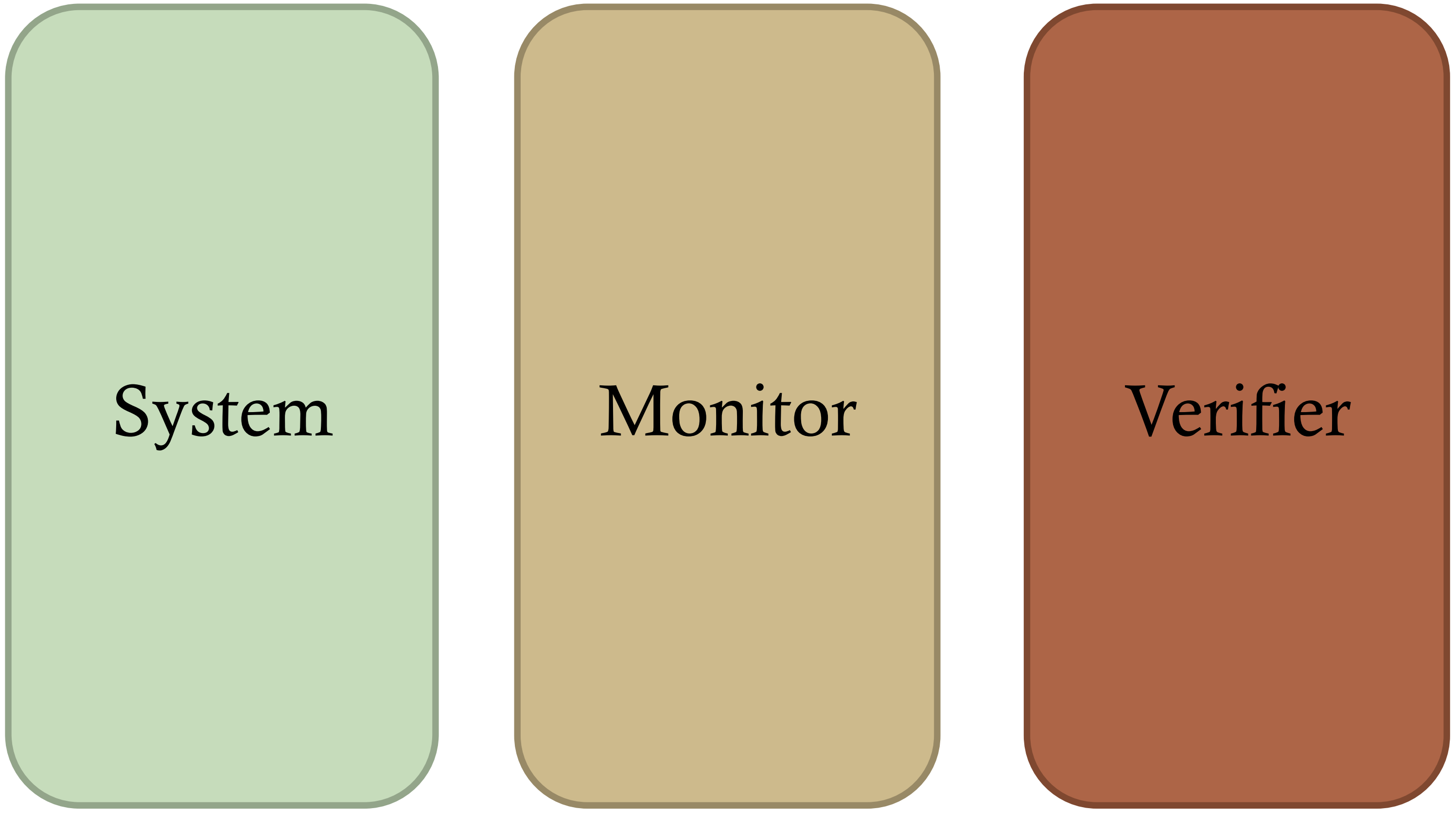
RV Architecture


Sys
Specification
System
Monitor
Verifier
How do the verification modules communicate with the system?

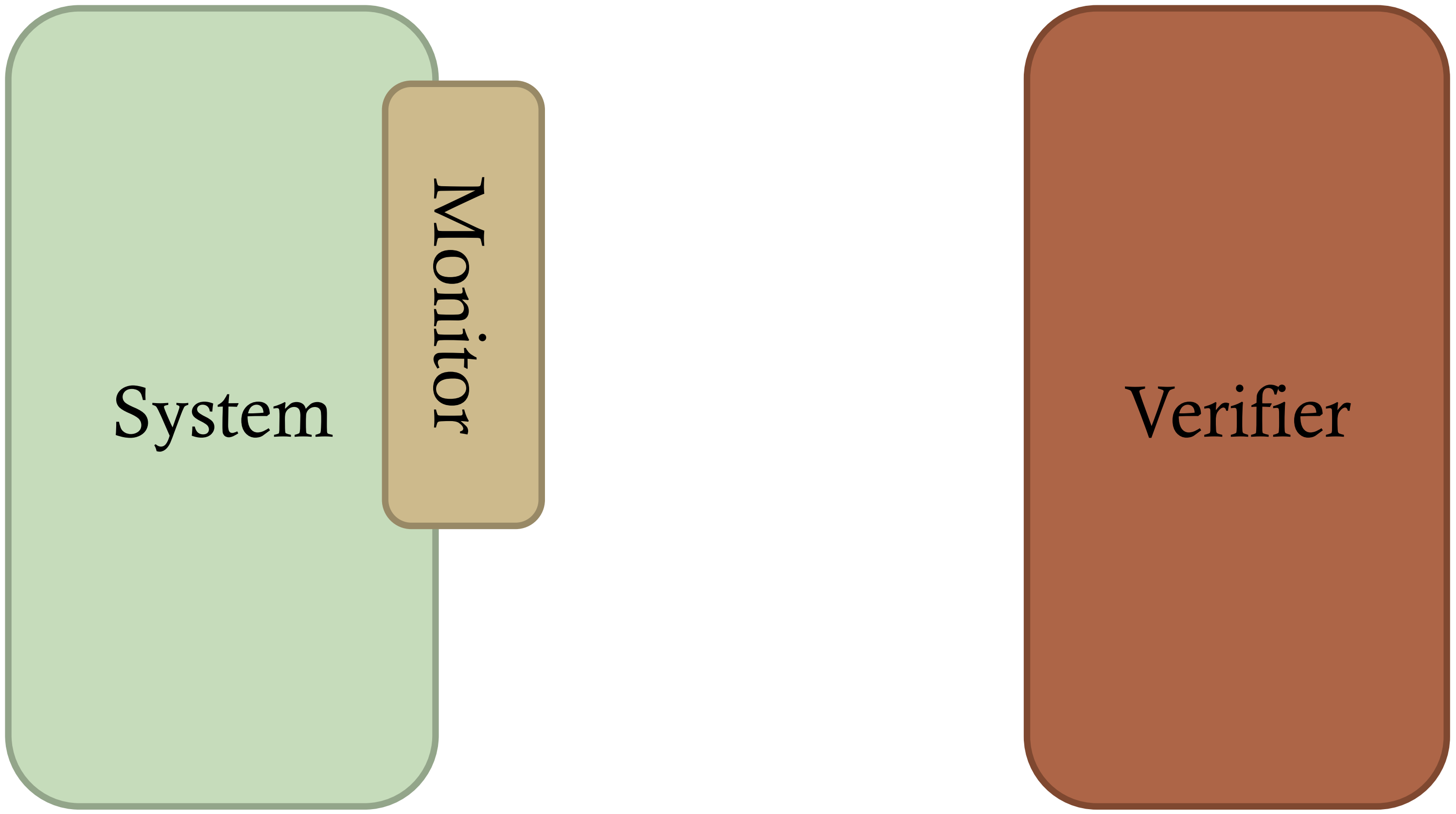
RV Architecture



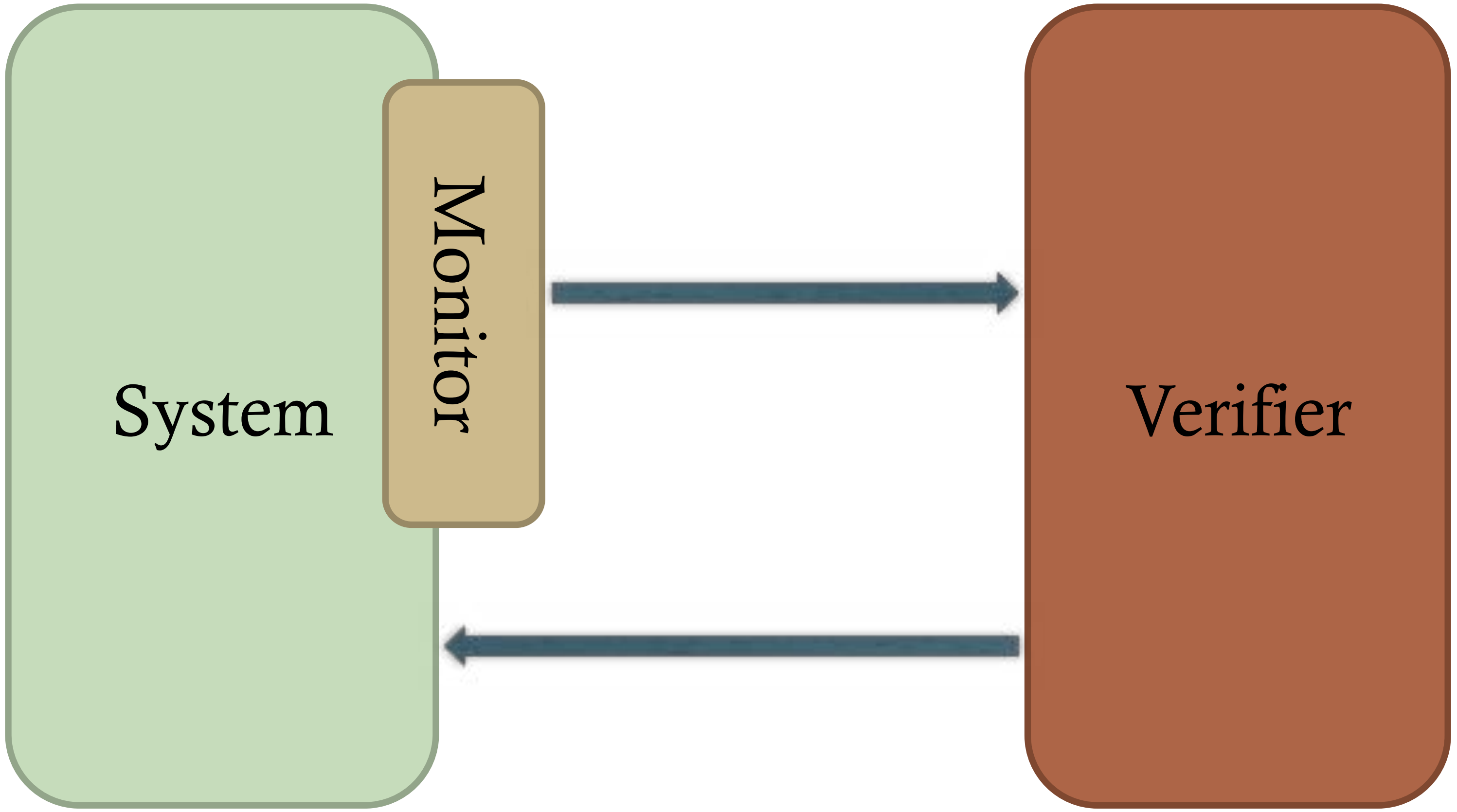
RV Architecture



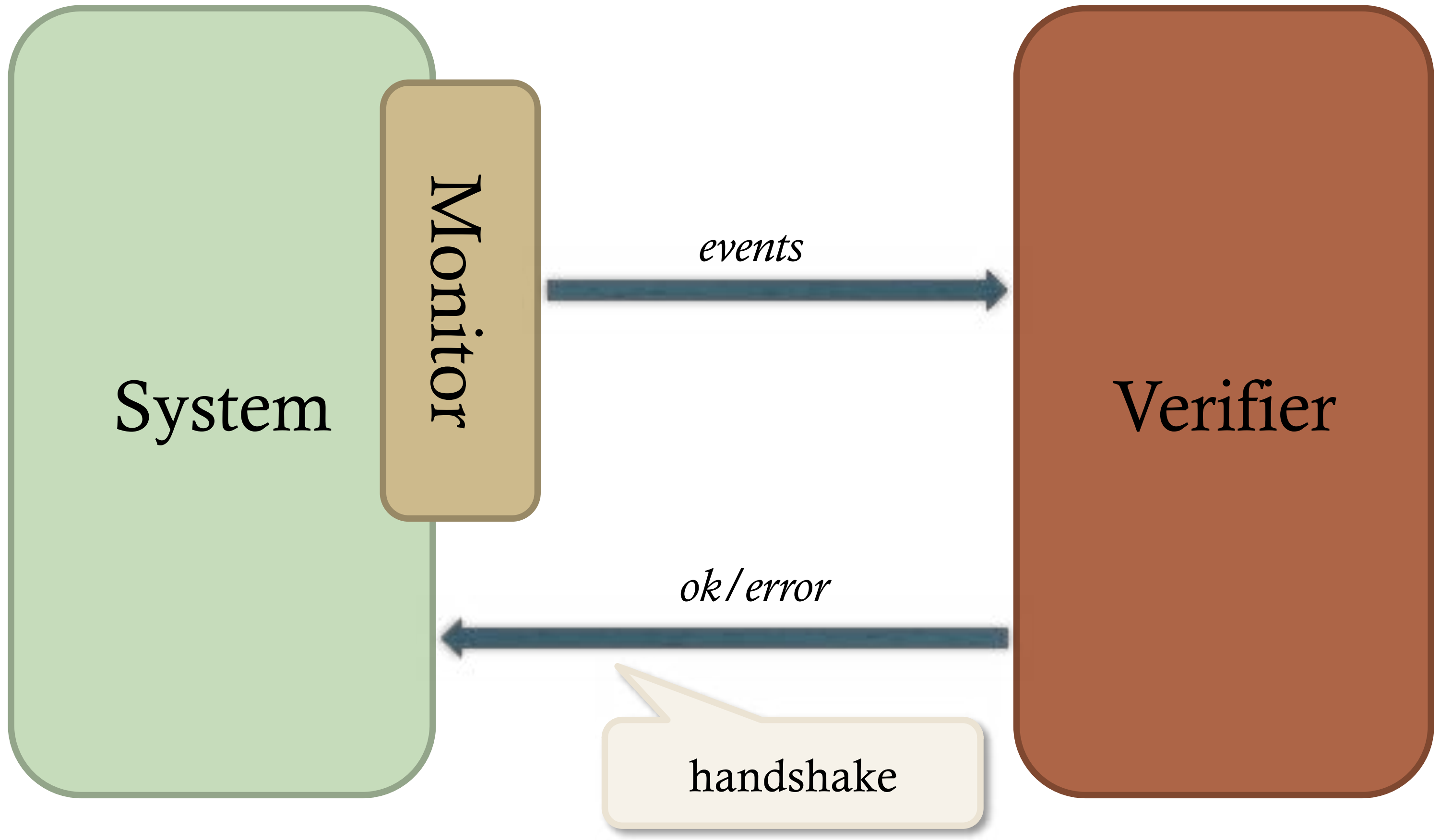
RV Architecture



RV Architecture



RV Architecture: Synchronous



RV Architecture: Synchronous

```
function login(…) {
      // main code
      …
}
```

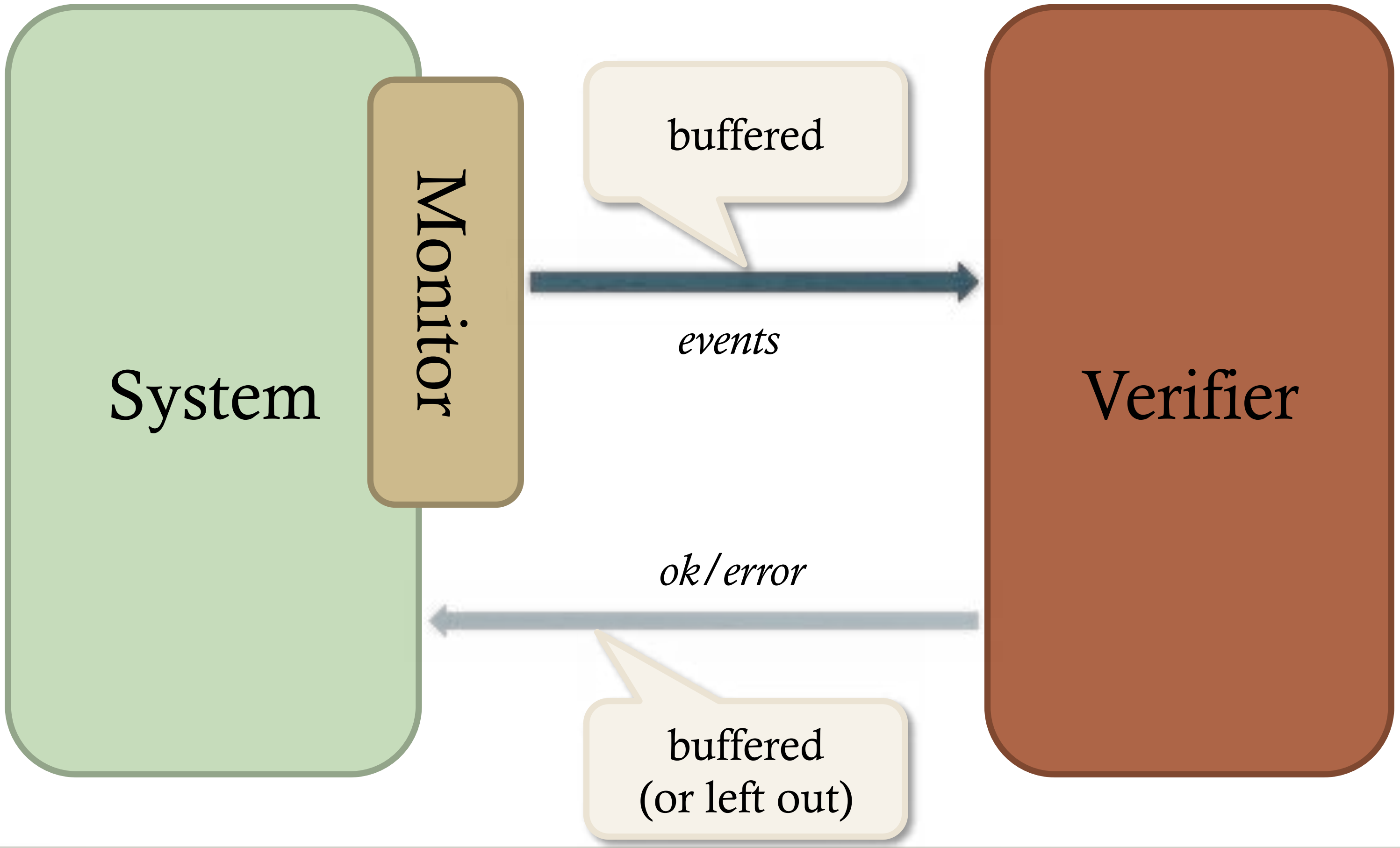
RV Architecture: Synchronous

```
function login(…) {
     Verifier.send(login, …);
     result = Verifier.receive();
     if (result == OK) {
          // main code
          …
     }
}
```

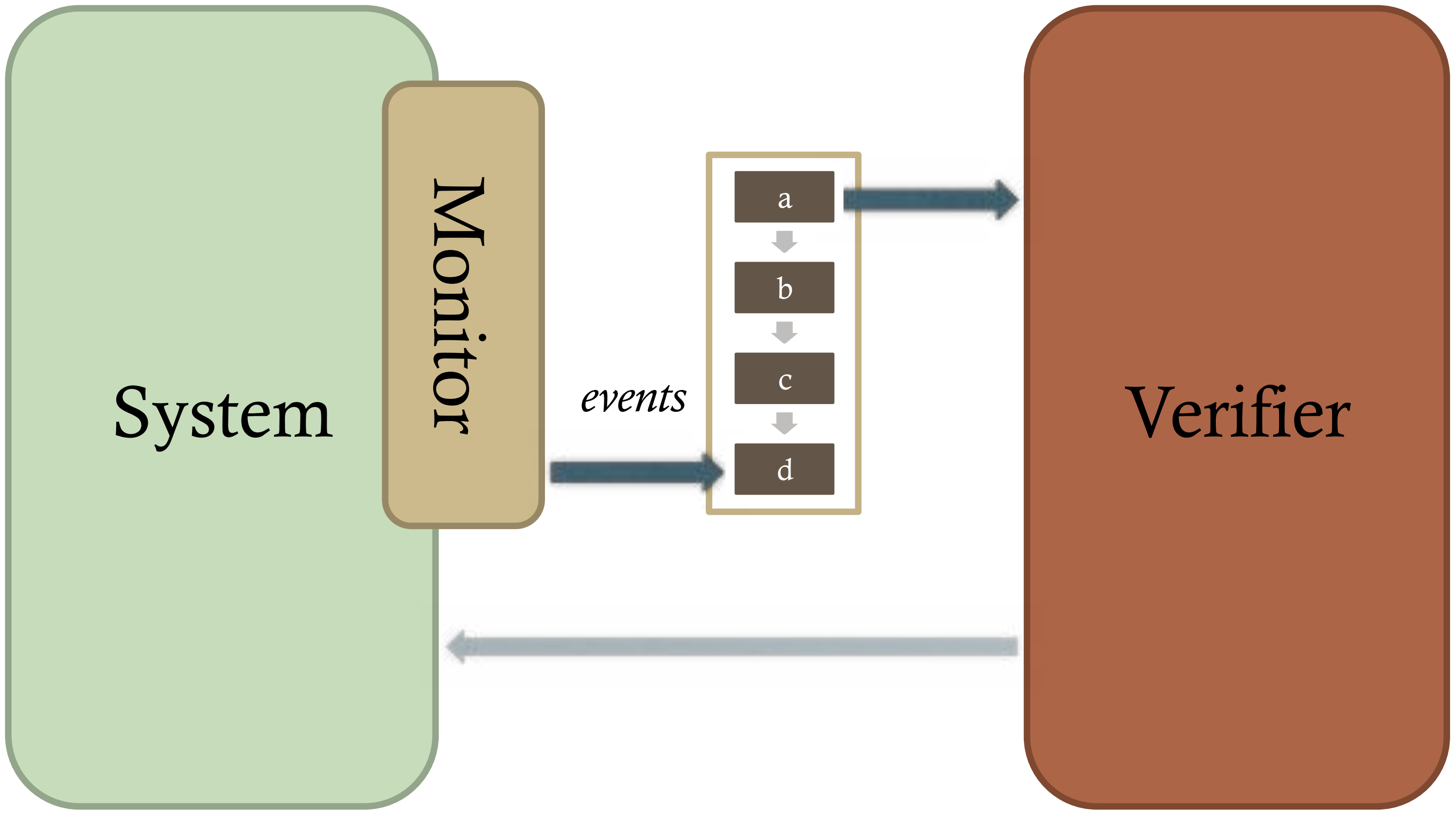
RV Architecture: Synchronous

- Synchronous monitoring guarantees that the system never advances beyond failure.
- Upon failure one can attempt to fix the problem since we are guaranteed to stop immediately after the failure.
- The cost in overheads can be prohibitive if the volume of monitored events is large.

RV Architecture: Asynchronous



RV Architecture: Asynchronous



RV Architecture: Asynchronous

```
function login(…) {
     // main code
     …
}
```

RV Architecture: Asynchronous

```
function login(…) {
      Verifier.send(login, …);
      // main code
      …
}
```

RV Architecture: Asynchronous

- Overheads limited to the cost of logging events.
- If there is complete independence between runtime and verification time, we can postpone the verification till after the whole trace has been generated and parse it in any way we want (e.g. for some past time logics, backward analysis is more efficient).
- Once failure is identified, we cannot do much more than notify the developers.

RV Overview

- There are many design choices in RV:
  - How to monitor (extract events)
  - How to specify what to verify
  - Where to specify what to verify
  - How to verify
  - How to connect the monitor with the verifier
  - When to verify
  - What to do with the outcome of the verification

Runtime Verification & Testing

Gordon J. Pace
University of Malta

May 2015

RV and Testing

- Testing has been established as *de rigeur* practice in industry, with quality assurance being a crucial part of the development cycle.
- Runtime verification is still far from being assimilated in software engineering development processes for a number of reasons:
    - Overheads can be prohibitive
    - Requires the adoption of new tools and development processes
    - Requires re-training of developers

RV and Testing

- Testing has been established as *de rigeur* practice in industry, with quality
of the development c
- Runtime verification
assimilated in softwa
processes for a numb
  - Overheads can be prohibi
  - Requires the adoption of new tools and development processes
  - Requires re-training of developers

Can we hinge on the success of testing for software quality assurance to introduce runtime verification "for free"?

Testing

- Recall some of the major challenges testing faces:
  - **Generation:** How to generate representative traces to test on?
  - **Validation:** How to check that the system works well on these traces?
  - **Coverage:** How to assess the coverage of the traces over traces that will happen at runtime?

Testing

- Recall some of the major challenges testing faces:
  - **Generation:** How to generate representative traces to test on?
  - **Validation:** How to check that the system works well on these traces?
  - **Coverage:** How to ass over traces that will ha

This is a concern shared with runtime verification

Generation and Validation

- Different approaches solve the *generation* and *validation,* typically combining the issues in a single framework e.g.
  - **Unit testing:** Test units (e.g. modules) separately, by mocking other parts of the system – using *test scripts* specifying (i) the test cases; and (ii) oracles to check the results.
  - **Model-based testing:** An abstract model of the system under test is built and abstract test cases defined (manually or automatically) from which concrete test cases are generated and run against the real system to compare the output with that of the model.

Generation and Validation

- We have two languages implicitly defined:
  - The language $\mathcal{T}$ of testable traces from which we will generate elements.
  - The language $\mathcal{Y}$ (or $\mathcal{N}$) of correct (or bad) traces.

Generation and Validation

- **Example:**
  - Testable traces $\mathcal{T}$ consist of traces in which the user will trigger a sequence of *login*, *logout*, *read* and *write* events but no *reconnect* events.
  - The set of bad traces $\mathcal{N}$ consists of all those in which the user tries to read or write without first logging in.

Generation and Validation

- **Example:**
  - Testable traces $\mathcal{T}$ consist of traces in which

Is this trace a violation?

login . read . logout . reconnect . write

  - in which the user tries to read or write without first logging in.

Generation and Validation

which

gin,

Is this trace a violation?

login .

It is in $\mathcal{N}$, but…

not in $\mathcal{T}$…

i.e. the testing tool would never identify this as a violation

- The
  in w
  with

RV and Testing

- **Question:** How can we use the testing specification (i.e. $\mathcal{T}$ and $\mathcal{N}$ or $\mathcal{Y}$) to construct a runtime verifier to match bad traces (i.e. $\mathcal{T} \cap \mathcal{N}$ or $\mathcal{T} \setminus \mathcal{Y}$)?

RV and Testing

- **Question:** How can we use the testing specification (i.e. $\mathcal{T}$ and $\mathcal{N}$ or $\mathcal{Y}$) to construct a runtime verifier to match bad traces (i.e. $\mathcal{T} \cap \mathcal{N}$ or $\mathcal{T} \setminus \mathcal{Y}$)?
- We will be doing this for two particular technologies: QuickCheck and LARVA.

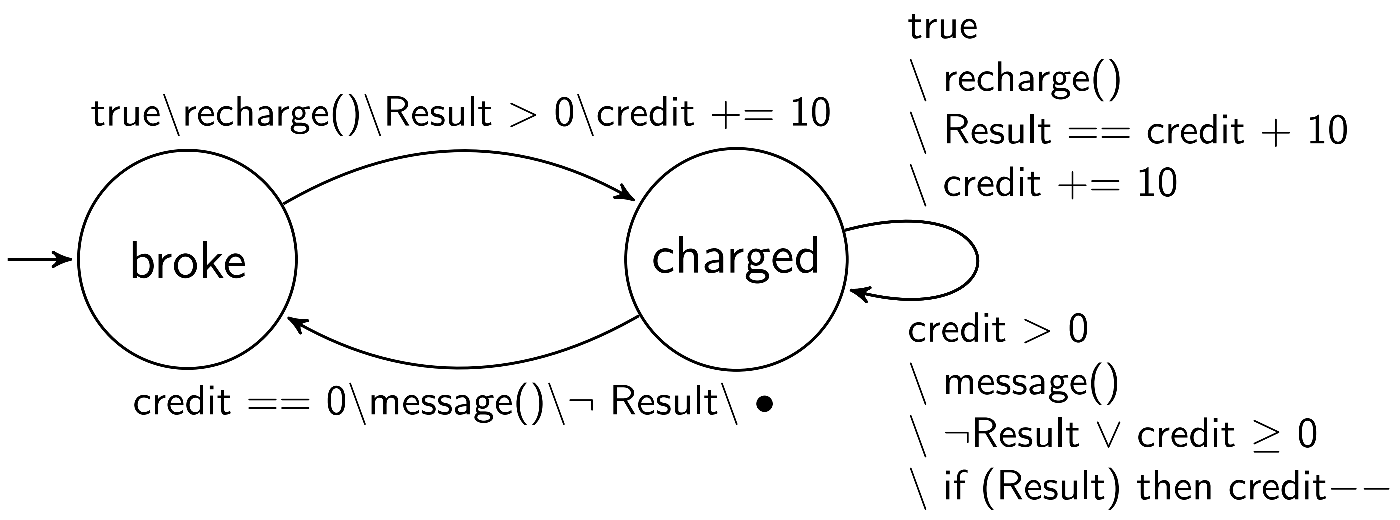
QuickCheck

- QuickCheck is a model-based testing tool originally built for Haskell, now available for Erlang, C, etc.
- Works through random test case generation using QuickCheck Finite State Automata (QCFSA).
- QCFSAs serve both for *generation* and *validation* of traces:
  - Generation through directed (semi-random) traversal
  - Validation through specification of postconditions associated with each transition

QuickCheck

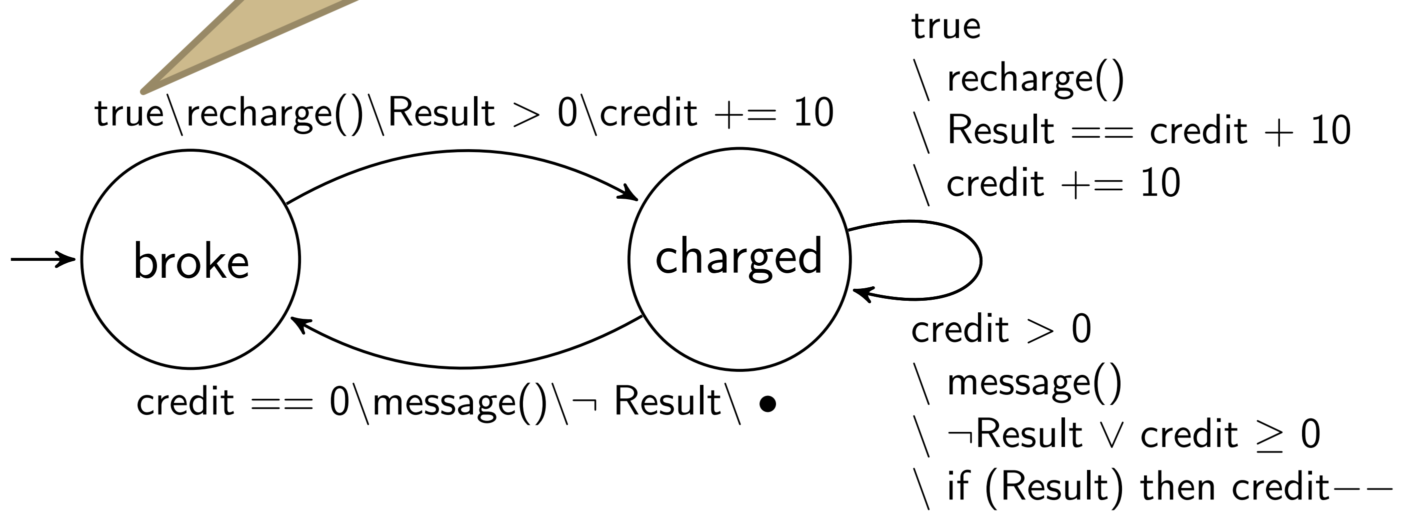
- **Example:** Consider a mobile phone credit top-up system
  - `message()` costs 1 credit, returns false on failure.
  - `recharge()` adds 10 credits to the account, returns new credit.

QuickCheck Automata



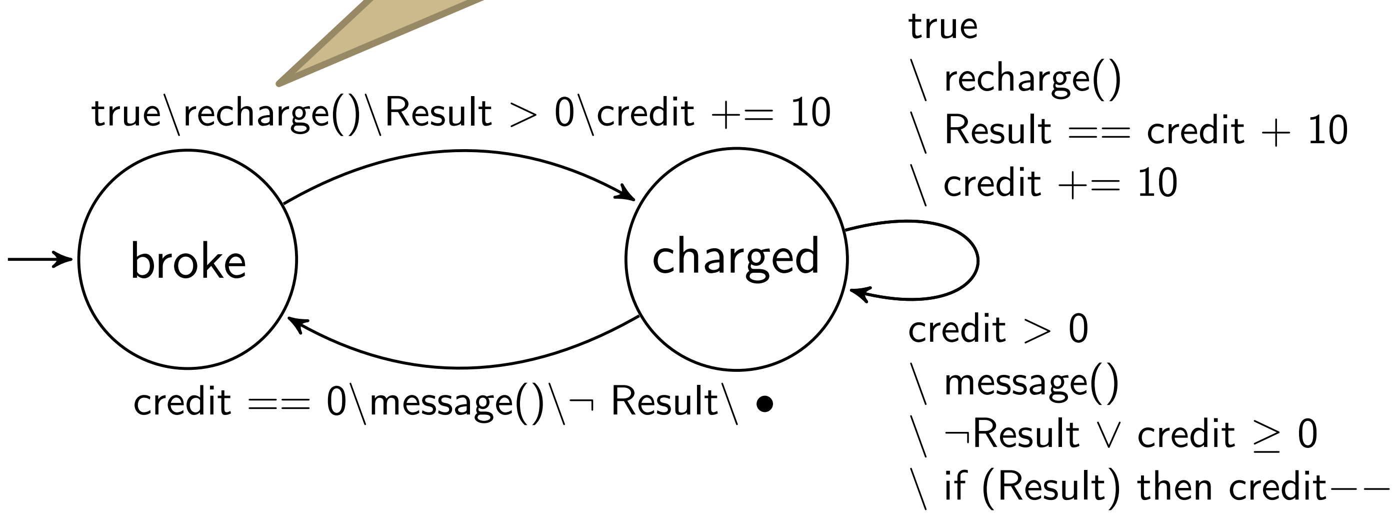
Q utomata

The precondition: the transition can only be taken if the precondition is true



Qui mata

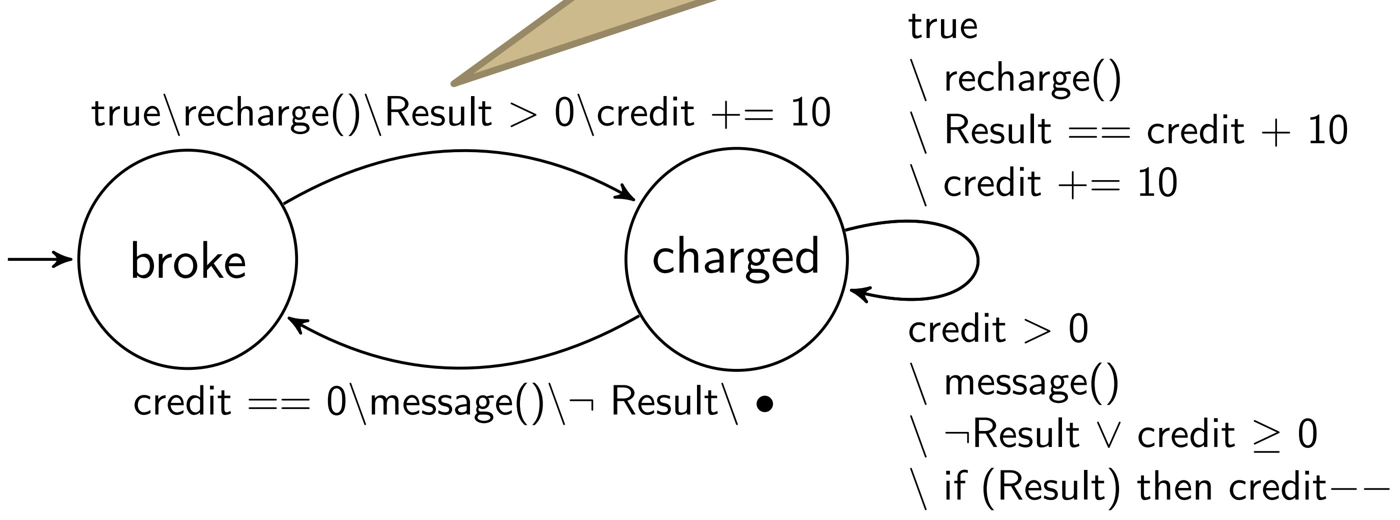
The invocation: A function to invoke when taking the transition



true\recharge()\Result > 0\credit += 10
broke
charged
credit == 0\message()\¬ Result\ •
true \ recharge() \ Result == credit + 10 \ credit += 10
credit > 0 \ message() \ ¬Result ∨ credit ≥ 0 \ if (Result) then credit−−

QuickC

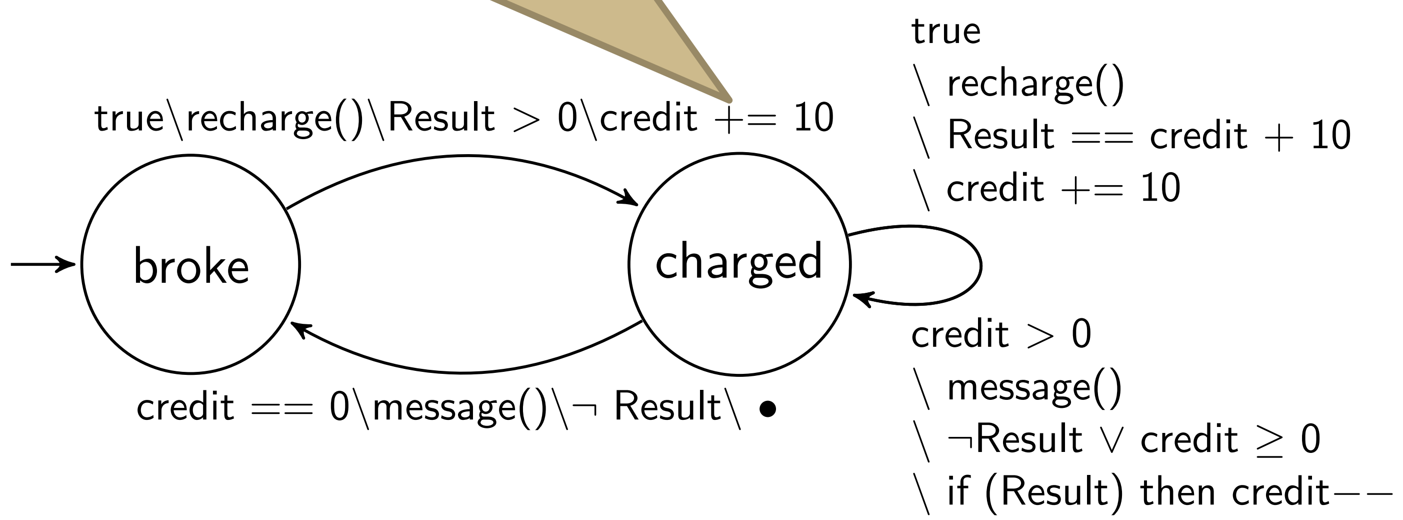
The postcondition: Will be checked upon termination of the invocation – will cause a failure if it returns false



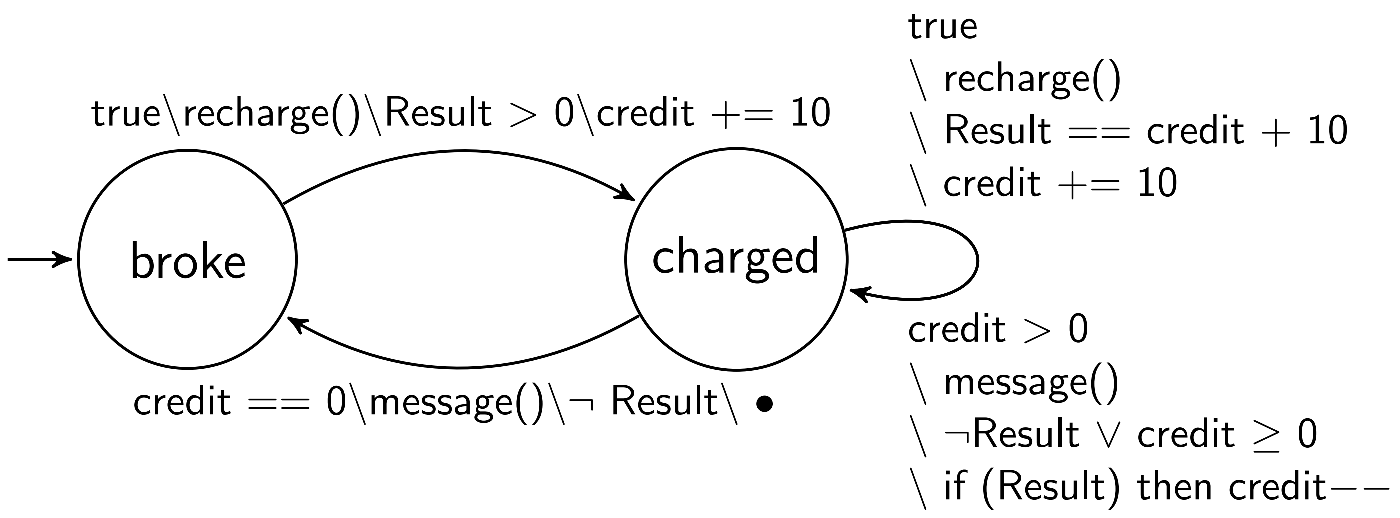
true\recharge()\Result > 0\credit += 10
broke
charged
true \ recharge() \ Result == credit + 10 \ credit += 10
credit == 0\message()\¬ Result\ •
credit > 0 \ message() \ ¬Result ∨ credit ≥ 0 \ if (Result) then credit−−

Automata

The action: The action is taken after checking the postcondition, before proceeding further with the automaton.



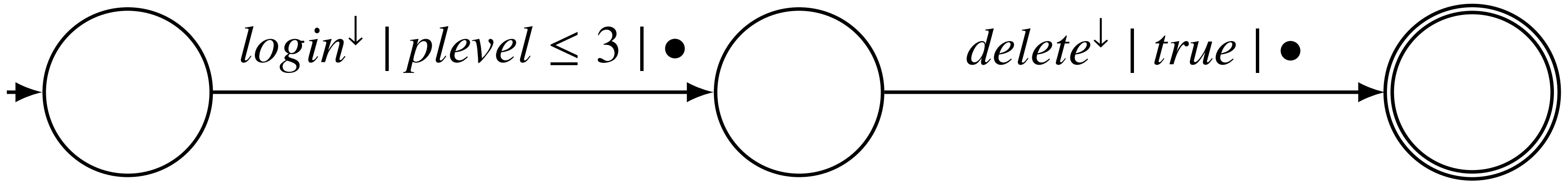
QuickCheck Automata



LARVA

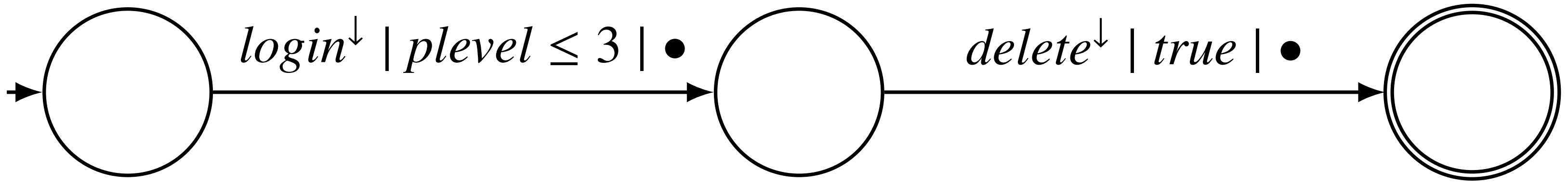
- LARVA is a synchronous runtime verification tool, originally built for Java, but with a version also available for Erlang.
- Uses DATEs (Dynamic Automata with Timers and Events) to give a specification.

LARVA Automata


$login^{\downarrow} \mid plevel \leq 3 \mid \bullet$
$delete^{\downarrow} \mid true \mid \bullet$

LARVA Automata

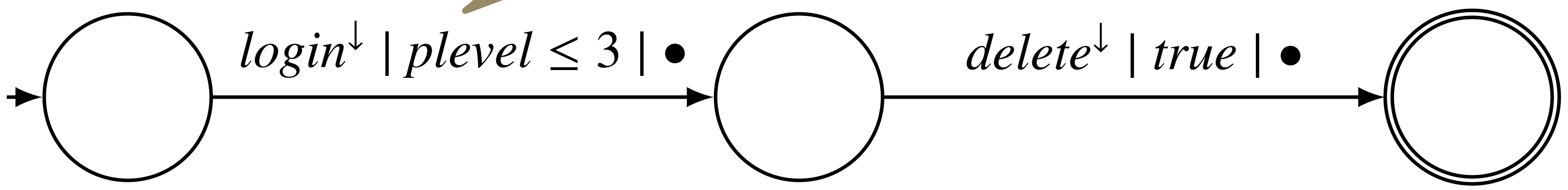
Event: The transition can only be taken when the system triggers the event.



$login^{\downarrow} \mid plevel \leq 3 \mid \bullet$
$delete^{\downarrow} \mid true \mid \bullet$

LARVA Automata

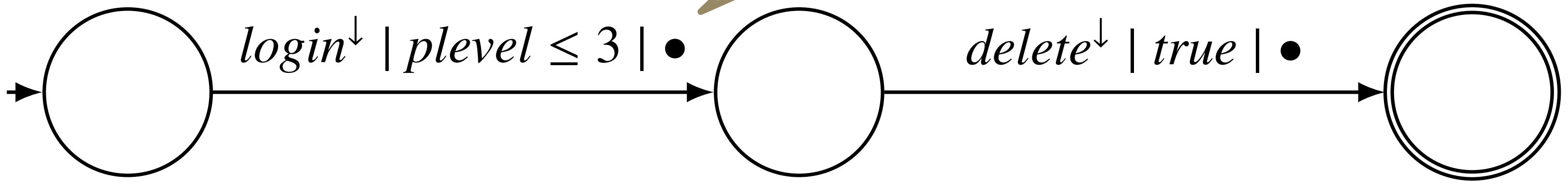
Condition: When the event occurs, the condition has to be satisfied for the transition to be taken.



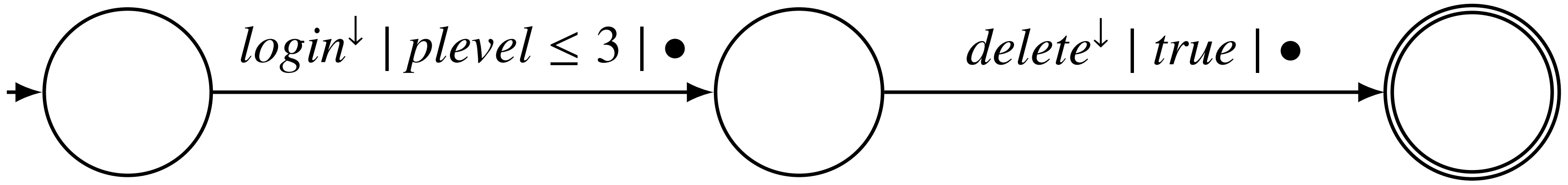
$login^{\downarrow} \mid plevel \leq 3 \mid \bullet$
$delete^{\downarrow} \mid true \mid \bullet$

LARVA Automata

Action: When the transition is taken, the action is executed.

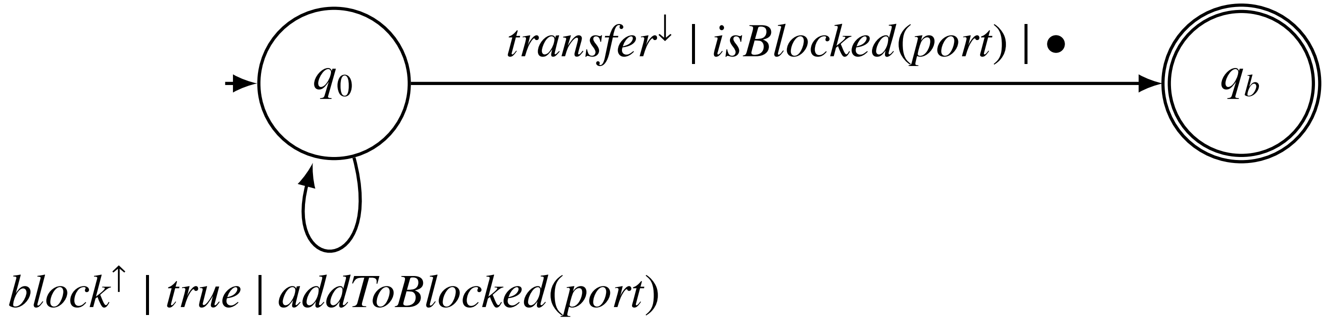

$login^{\downarrow} \mid plevel \leq 3 \mid \bullet$
$delete^{\downarrow} \mid true \mid \bullet$

LARVA Automata


$$\begin{tikzcd}[column sep=huge] \arrow[r] & \bigcirc \arrow[r, "login^{\downarrow} \mid plevel \leq 3 \mid \bullet"] & \bigcirc \arrow[r, "delete^{\downarrow} \mid true \mid \bullet"] & \circledcirc \end{tikzcd}$$

LARVA Automata

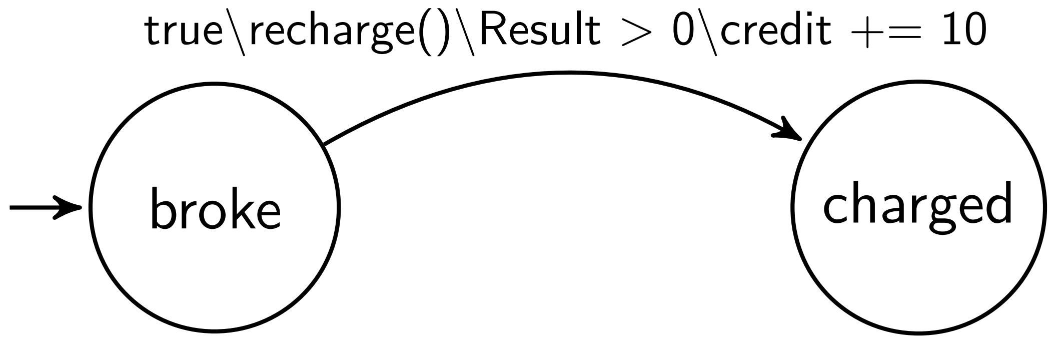
After blocking port p, no data transfer may occur on that port.



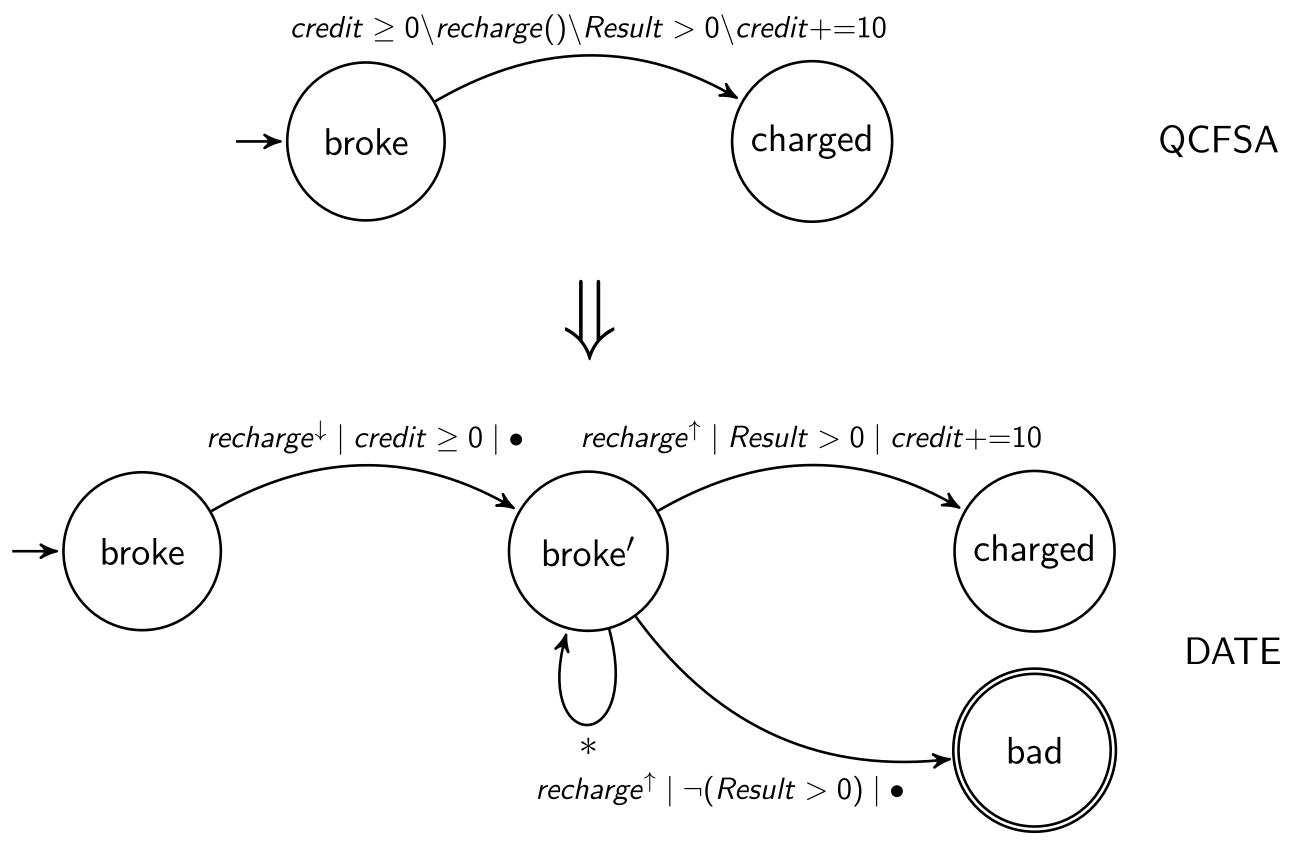
q_0
$transfer^{\downarrow} \mid isBlocked(port) \mid \bullet$
q_b
$block^{\uparrow} \mid true \mid addToBlocked(port)$

Since blocking a port and transfering data may be concurrently accessed, *transfer* may not be entered only once *block* terminates.

QuickCheck Specifications to LARVA Automata



QuickCheck Specifications to LARVA Automata



$credit \geq 0 \backslash recharge() \backslash Result > 0 \backslash credit {+}{=} 10$
broke
charged
QCFSA
$\Downarrow$
$recharge^{\downarrow} \mid credit \geq 0 \mid \bullet$
$recharge^{\uparrow} \mid Result > 0 \mid credit {+}{=} 10$
broke
broke′
charged
DATE
$*$
$recharge^{\uparrow} \mid \neg(Result > 0) \mid \bullet$
bad

QuickCheck Specifications to LARVA Automata

Theorem: Given a QuickCheck automaton *M*, there exists a DATE *D* which will recognise all and only the negative traces which can be generated by *M*.

Limitations

- Test cases are sometimes very specific and not generalised.
- Tests sometimes give high importance to borderline cases increasing overheads for rare instances.
- In some cases, the approach would fail:
  - *Equivalence partitioning:* Partitions test space into parts such that testing one trace in each partition – other cases, although relevant, are not caught by the monitor.
  - *Regression testing:* Focuses on finding faults on code changes. Tests may be localised to changes in code, thus severely limiting coverage.

Conclusions

- Oracles in runtime verification and testing serve an identical purpose…
- But generation comes for free in runtime verification making testing-to-runtime verification easier.
- Still, tests have to be built with runtime verification in mind to maximise return.
- Similar work combining JUnit and runtime verification identified similar issues.

Some papers to read

- Martin Leucker, Christian Schallhart, *A brief account of runtime verification*. J. Log. Algebr. Program. 78(5): 293-303, 2009.
- Kevin Falzon, Gordon J. Pace, *Combining Testing and Runtime Verification Techniques*. MOMPES 2012: 38-57, 2012.
- Normann Decker, Martin Leucker, Daniel Thoma, *jUnitRV-Adding Runtime Verification to jUnit.* NASA Formal Methods 2013: 459-464, 2013.