Programatica

Integrating
Programming,
Properties, and
Certification

# Tutorial

Mark P. Jones
Pacific Software Research Center (PacSoft)
OGI School of Science & Engineering
Oregon Health & Science University

# Current Team Members:

- James Hook
- Mark Jones
- Dick Kieburtz
- John Matthews
- Andrew Tolmach
- Peter White
- Bill Harrison
- Thomas Hallgren
- Amber Telfer

# Programatica Goals:

- Develop architecture and tools to support construction and certification of high-assurance systems

- Integrate a broad (and open) spectrum of assurance techniques (code review, testing, formal methods, ...)

- Application focus: assurance of security properties (e.g., separation) in complex software artifacts of engineering significance.

# Building High-assurance Software:

There are many ways to increase assurance:

- Test programs on specific cases
- Test programs on randomly generated test cases derived from expected properties
- Peer review
- Use algorithms from published papers
- Reason about equational properties
- Reason about meta-properties (e.g., using types)
- Use theorem provers to validate (translated) code
- …

Each one can contribute significantly to increased reliability, security, and trustworthiness

# Evidence: A Unifying Feature

- There are significant differences in the applicability, assurance, and technical details of each of these techniques.

- But there is a common feature:
  - Each one results in some **tangible** form of **evidence** that provides a **basis for trust**

# Examples of Evidence:

There are many kinds of evidence:

- An (input, expected output) pair for a test case
- A property statement, and heuristics for guiding the selection of "interesting" random test cases
- A record of a code review meeting
- A citation/URL for a published paper or result
- An equational proof
- A type and the associated derived property
- A translation of the source program into a suitable theory and a user-specified proof tactic
- …

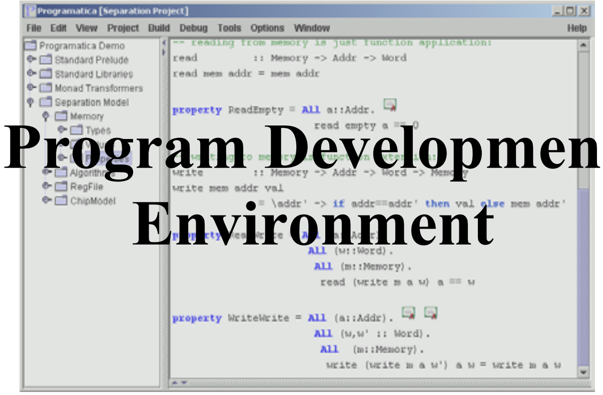
In Programatica, each different kind of evidence is stored with the program as a **certificate**

# Evidence and Certificates:

The certificate abstraction allows users to:

- **Capture** evidence of validity (in many different forms) and **Collate** it with source materials
- **Combine** of evidence from different sources
- **Track** dependencies and **detect** when evidence needs to be revalidated because of changes in the source code
- **Manage** evidence by analyzing and reporting on what has been established, identifying weaknesses, guiding further effort, etc...

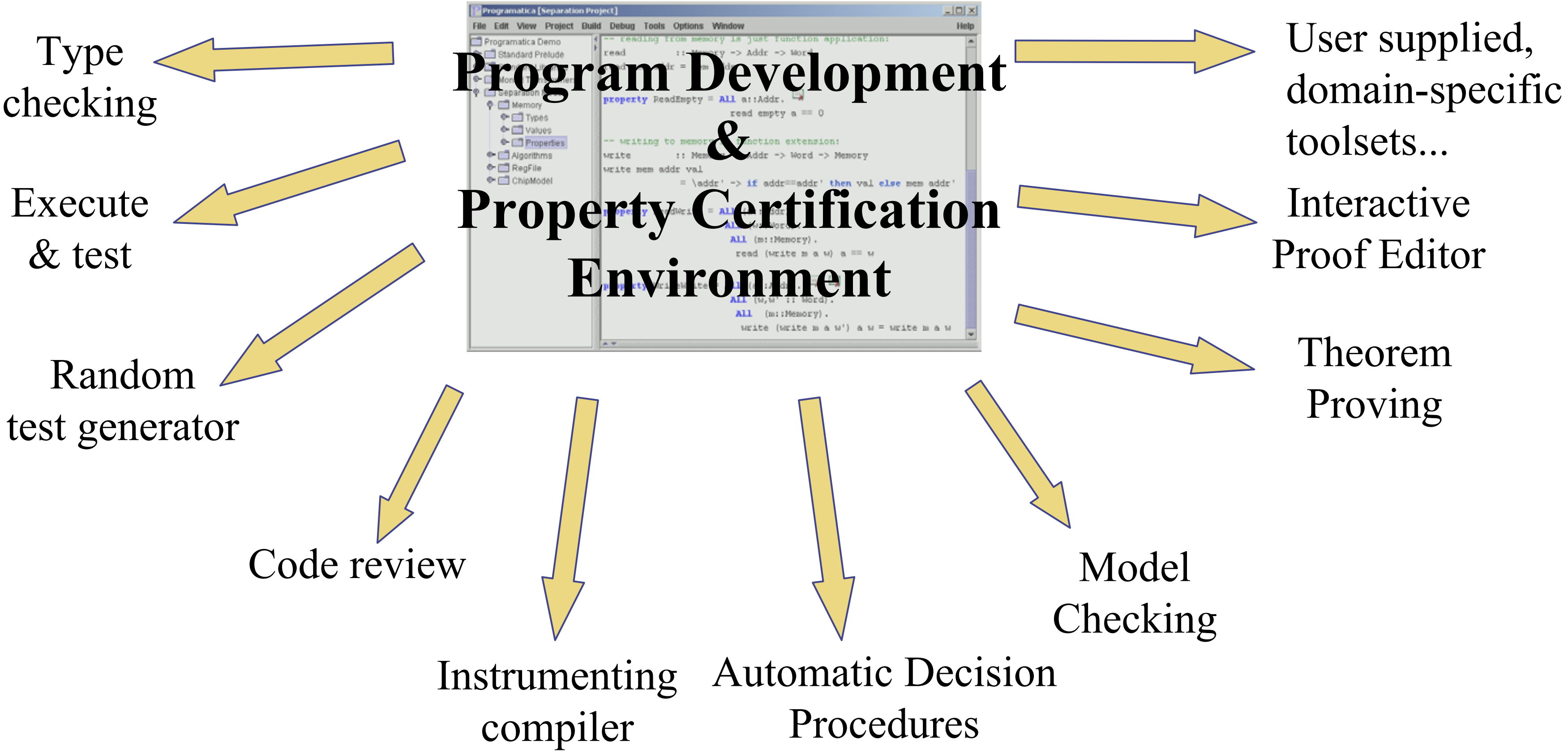
# The Programatica Vision:



```
-- reading from memory is just function application:
read         :: Memory -> Addr -> Word
read mem addr = mem addr

property ReadEmpty = All a::Addr.
                          read empty a == 0

-- writing to memory is function extension:
write        :: Memory -> Addr -> Word -> Memory
write mem addr val
             = \addr' -> if addr==addr' then val else mem addr'

property ReadWrite = All (a::Addr).
                      All (w::Word).
                       All (m::Memory).
                        read (write m a w) a == w

property WriteWrite = All (a::Addr).
                       All (w,w' :: Word).
                        All  (m::Memory).
                         write (write m a w') a w = write m a w
```

**Program Development Environment**

# The Programatica Vision:



```
-- reading from memory is just function application:
read          :: Memory -> Addr -> Word
read mem addr = mem addr

property ReadEmpty = All a::Addr.
                       read empty a == 0

-- writing to memory is function extension:
write         :: Memory -> Addr -> Word -> Memory
write mem addr val
              = \addr' -> if addr==addr' then val else mem addr'

property ReadWrite = All (a::Addr).
                       All (w::Word).
                       All (m::Memory).
                        read (write m a w) a == w

property WriteWrite = All (a::Addr).
                       All (w,w' :: Word).
                        All  (m::Memory).
                         write (write m a w') a w = write m a w
```

**Program Development & Property Certification Environment**

Type checking

Execute & test

Random test generator

Code review

Instrumenting compiler

Automatic Decision Procedures

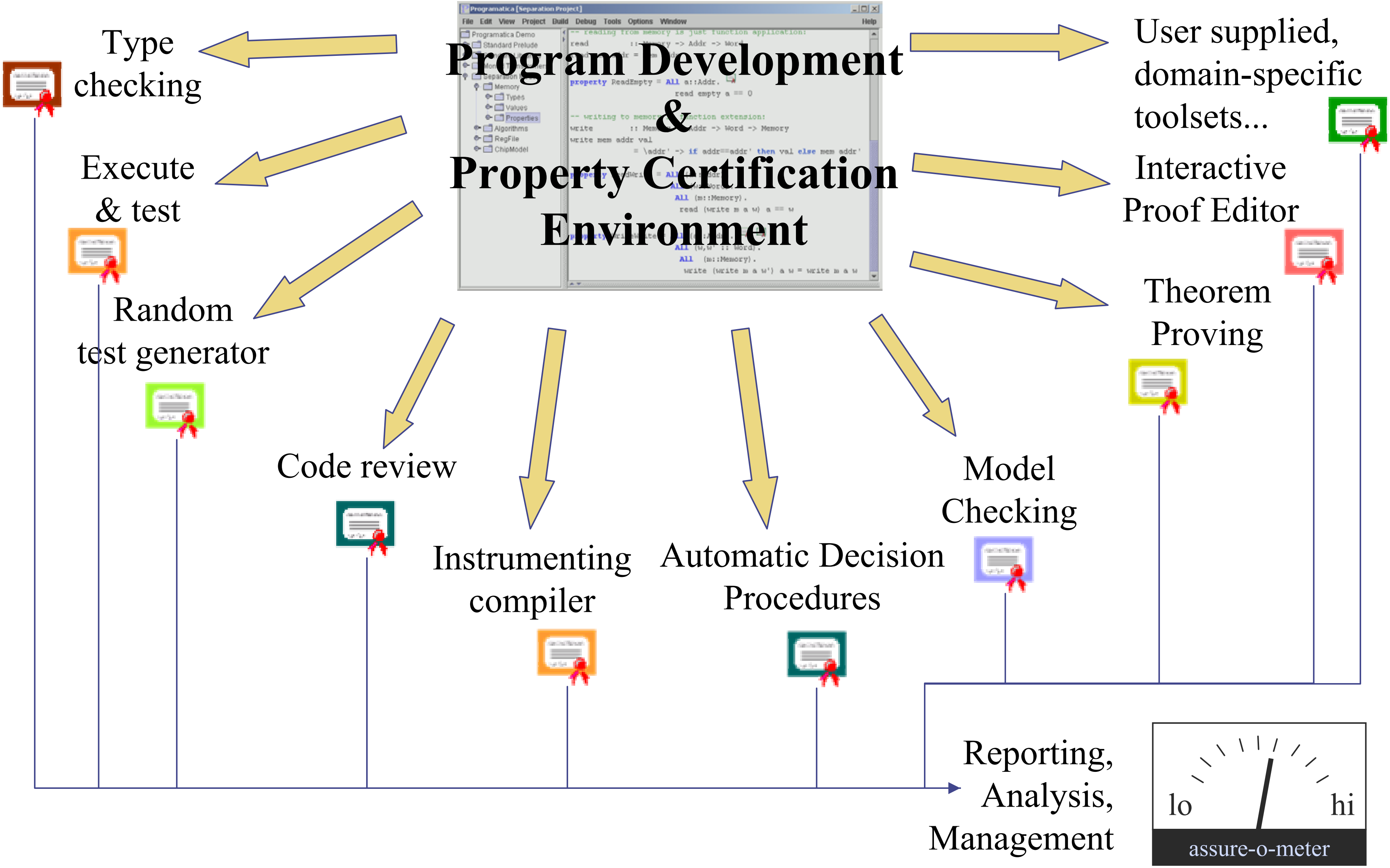
Model Checking

Theorem Proving

Interactive Proof Editor

User supplied, domain-specific toolsets...

# The Programatica Vision:



```
-- reading from memory is just function application:
read          :: Memory -> Addr -> Word
read mem addr = mem addr

property ReadEmpty = All a::Addr.
                       read empty a == 0

-- writing to memory is function extension:
write         :: Memory -> Addr -> Word -> Memory
write mem addr val
          = \addr' -> if addr==addr' then val else mem addr'

property ReadWrite = All (a::Addr).
                     All (w::Word).
                     All (m::Memory).
                      read (write m a w) a == w

property WriteWrite = All (a::Addr).
                      All (w,w' :: Word).
                       All (m::Memory).
                        write (write m a w') a w = write m a w
```

Program Development & Property Certification Environment

Type checking

Execute & test

Random test generator

Code review

Instrumenting compiler

Automatic Decision Procedures

Model Checking

Theorem Proving

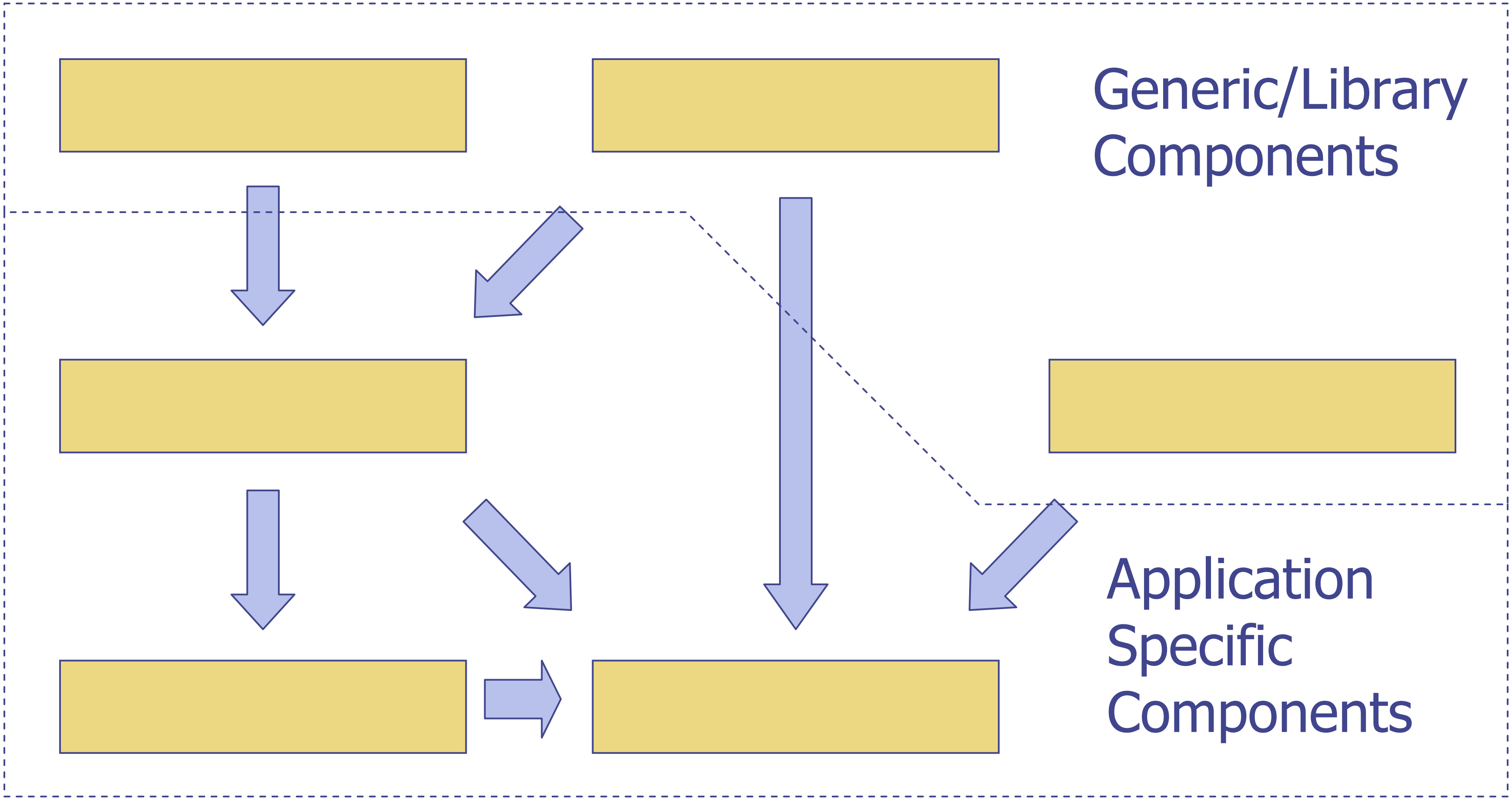
Interactive Proof Editor

User supplied, domain-specific toolsets...

Reporting, Analysis, Management

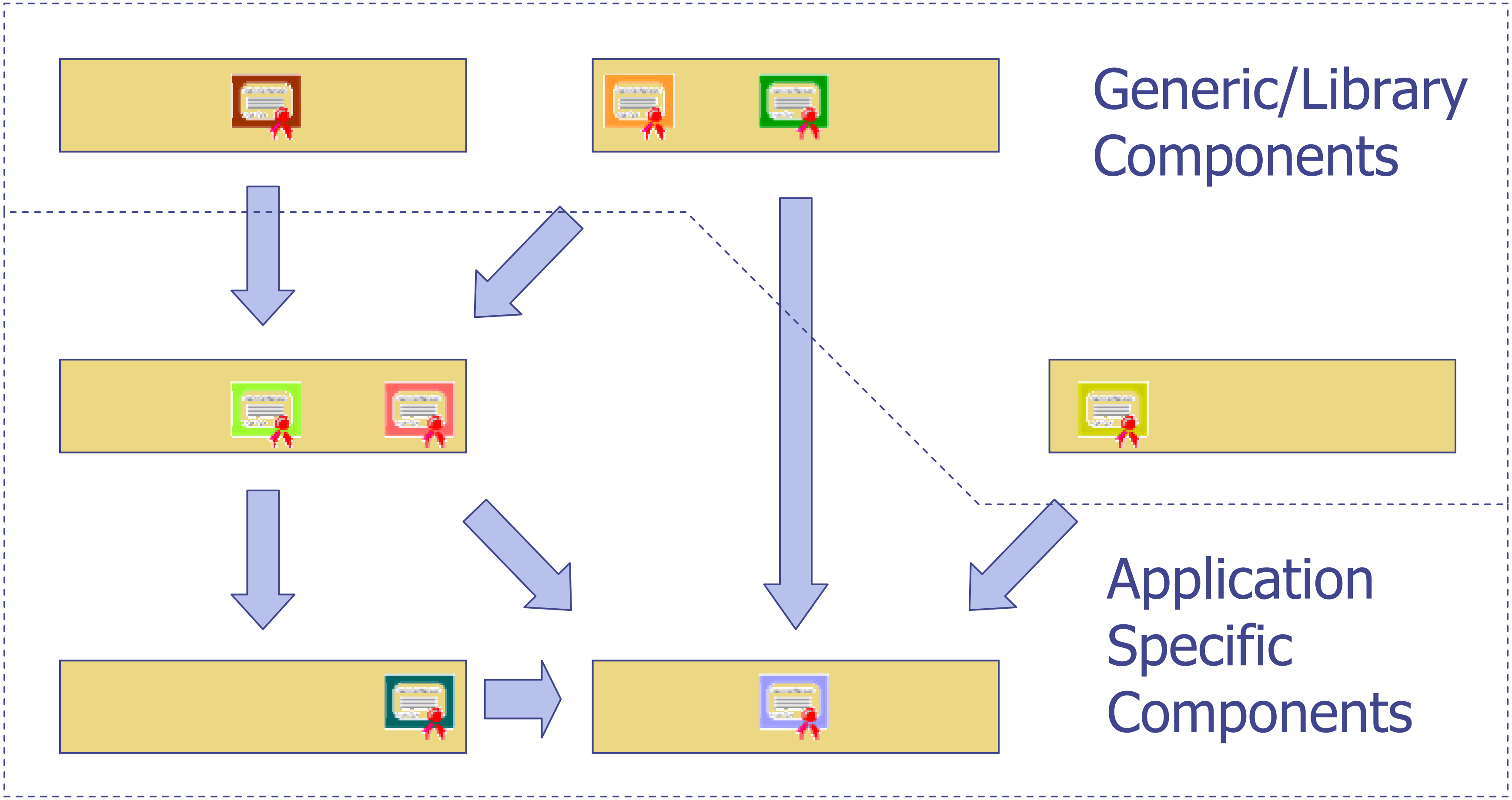
# Modular Construction

The modular design and construction of computer systems ...



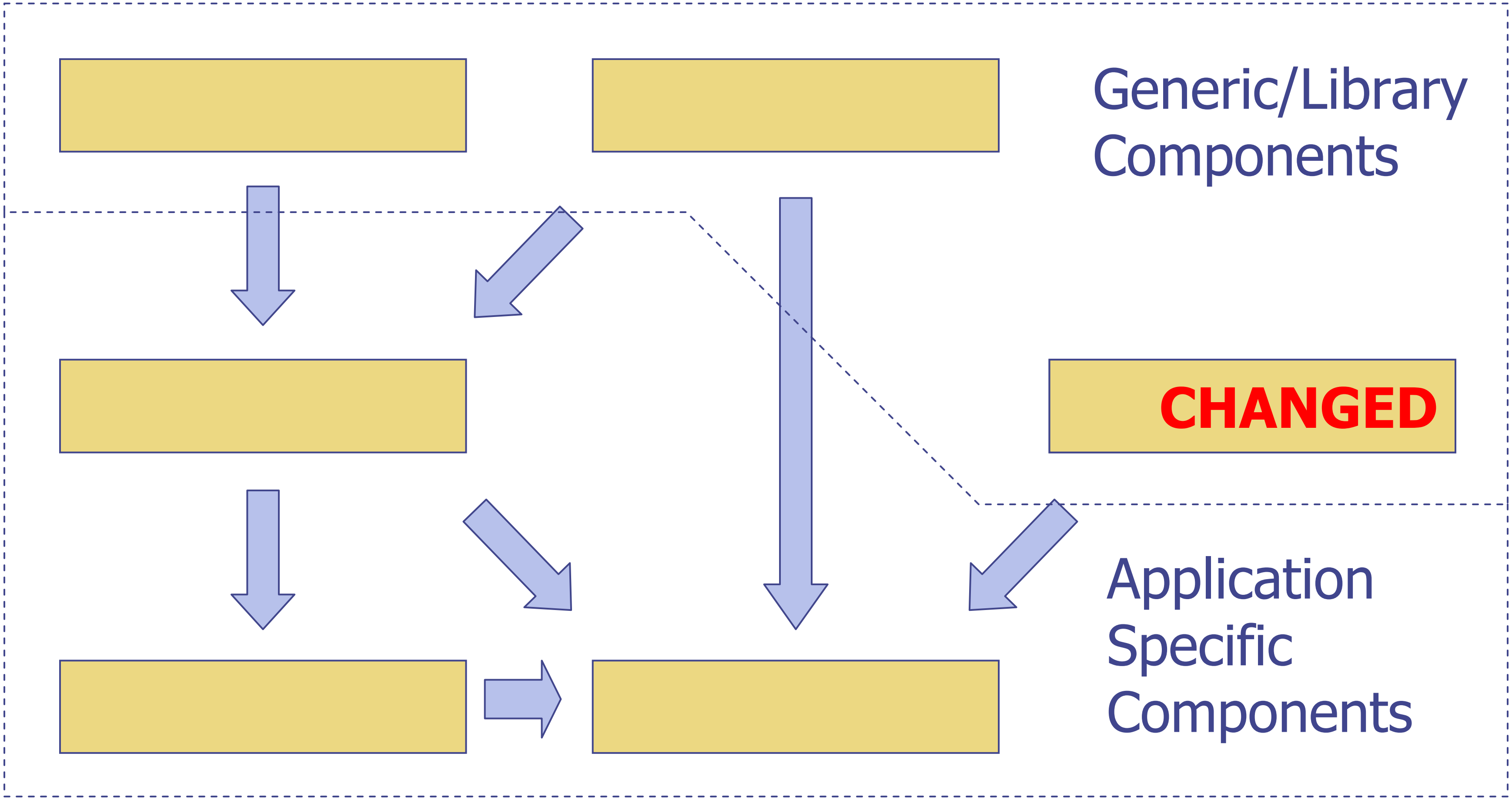
# Modular, Automated Certification

… should be reflected in modular certification processes that are used to validate them:



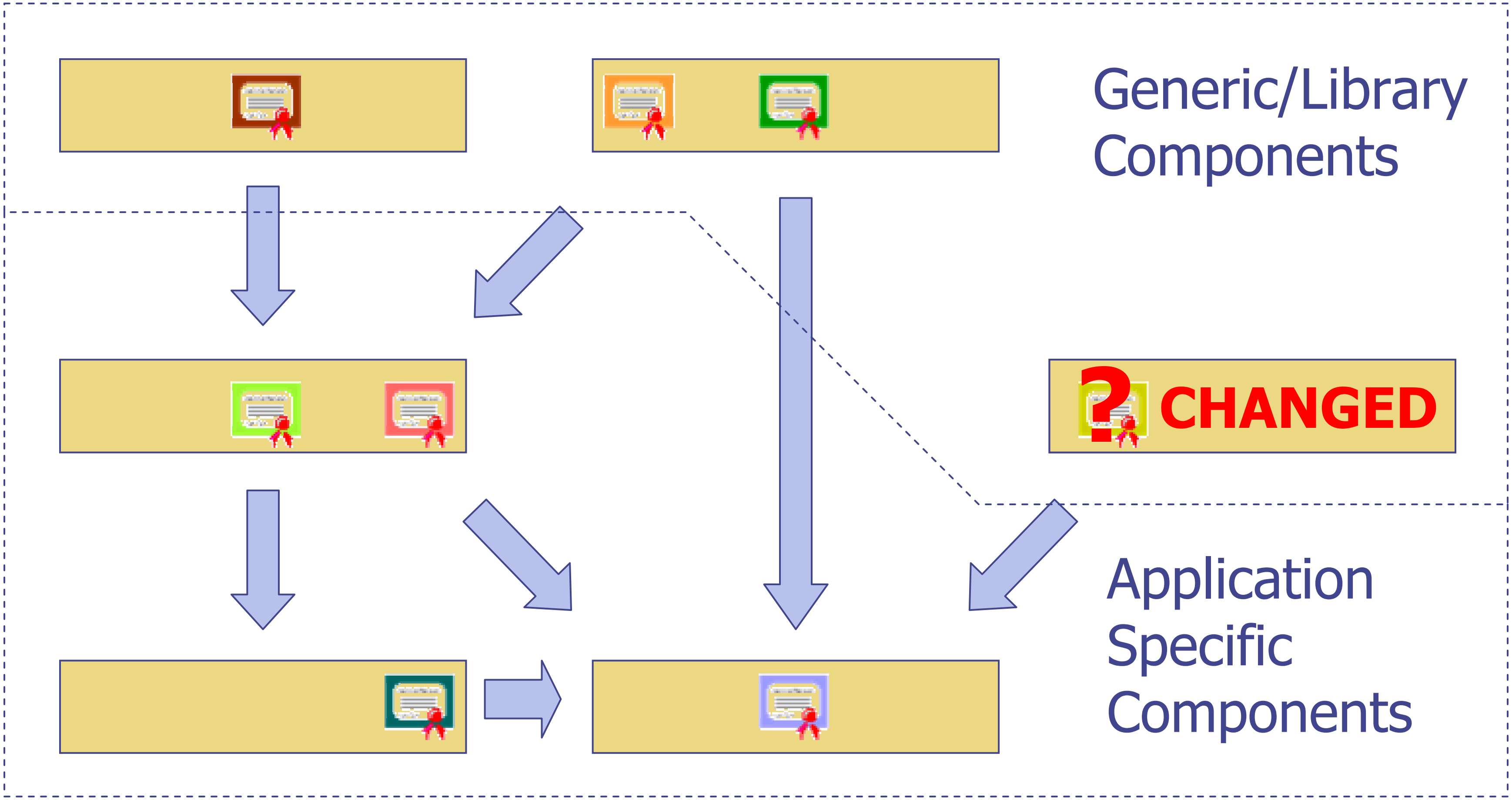
# Systems Change:

Modularity minimizes the **impact** of change



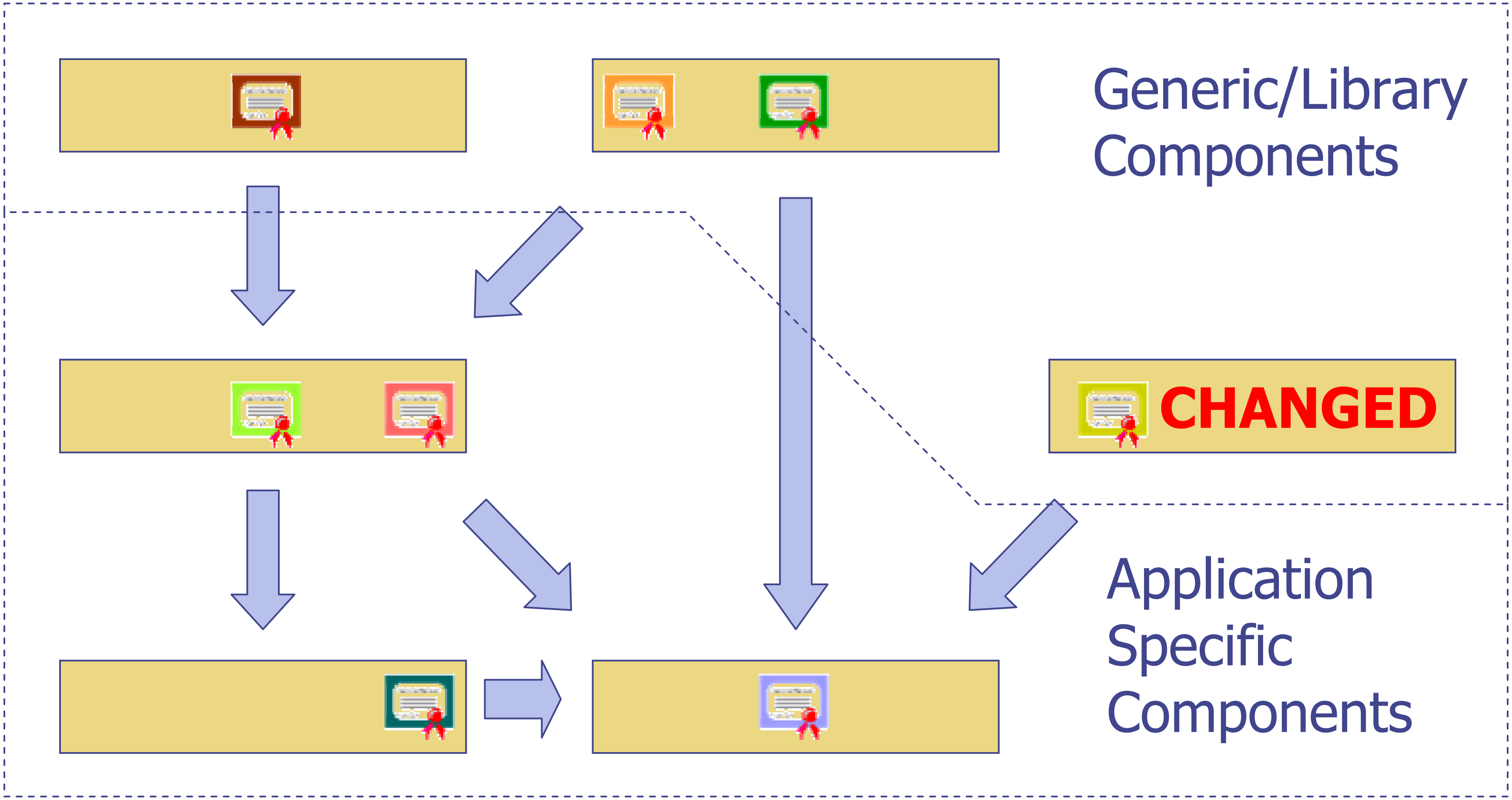
# Systems Change:

Modularity minimizes the **cost** of recertification



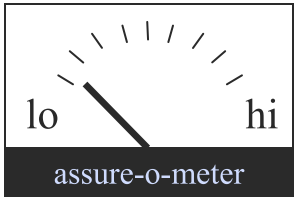
# Systems Change:

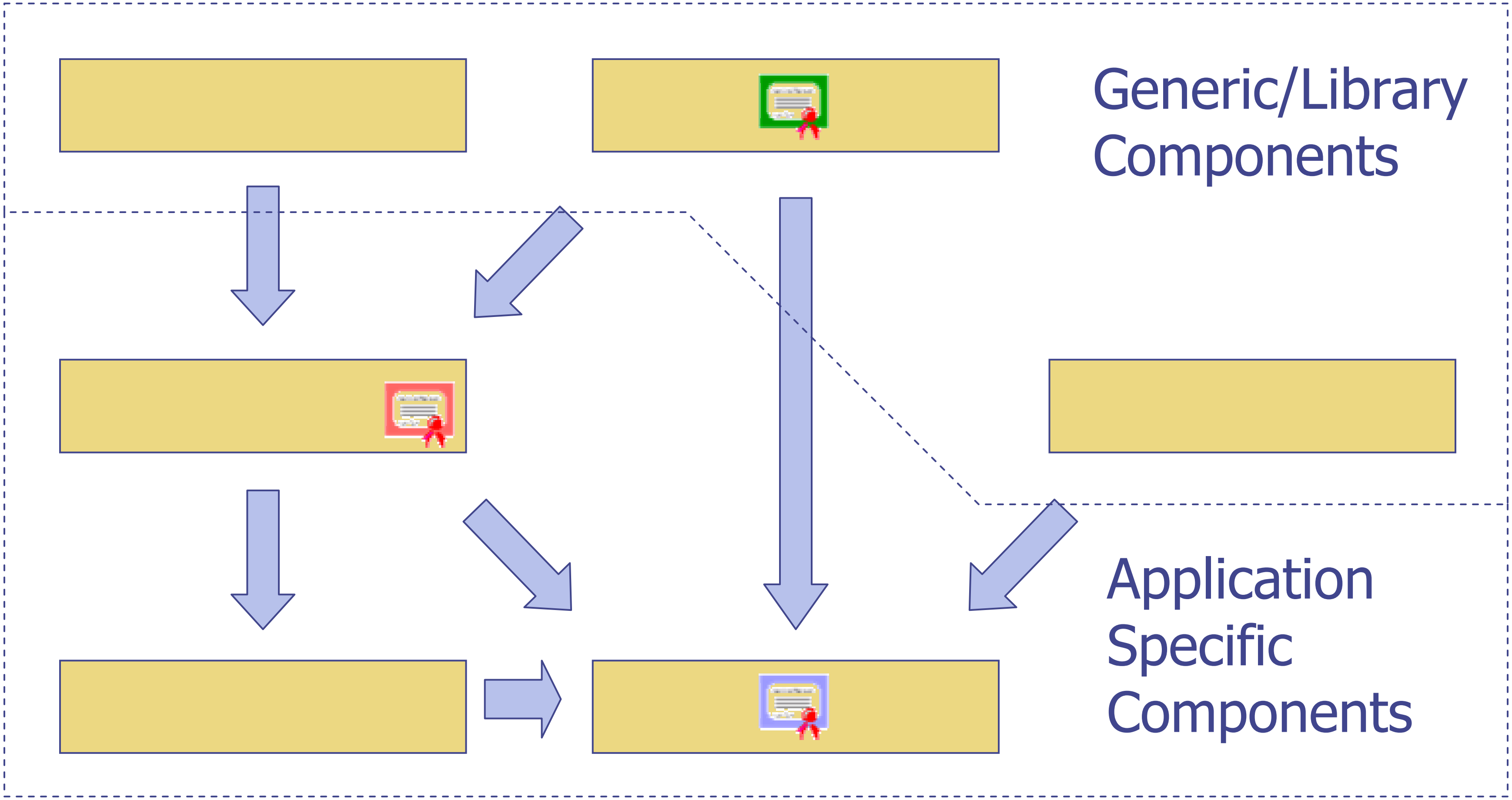
Modularity minimizes the **cost** of recertification (**automation** helps too …)



# Assurance Requirements Change:

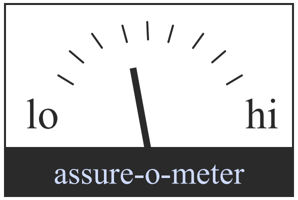
Minimize cost during early stages of development

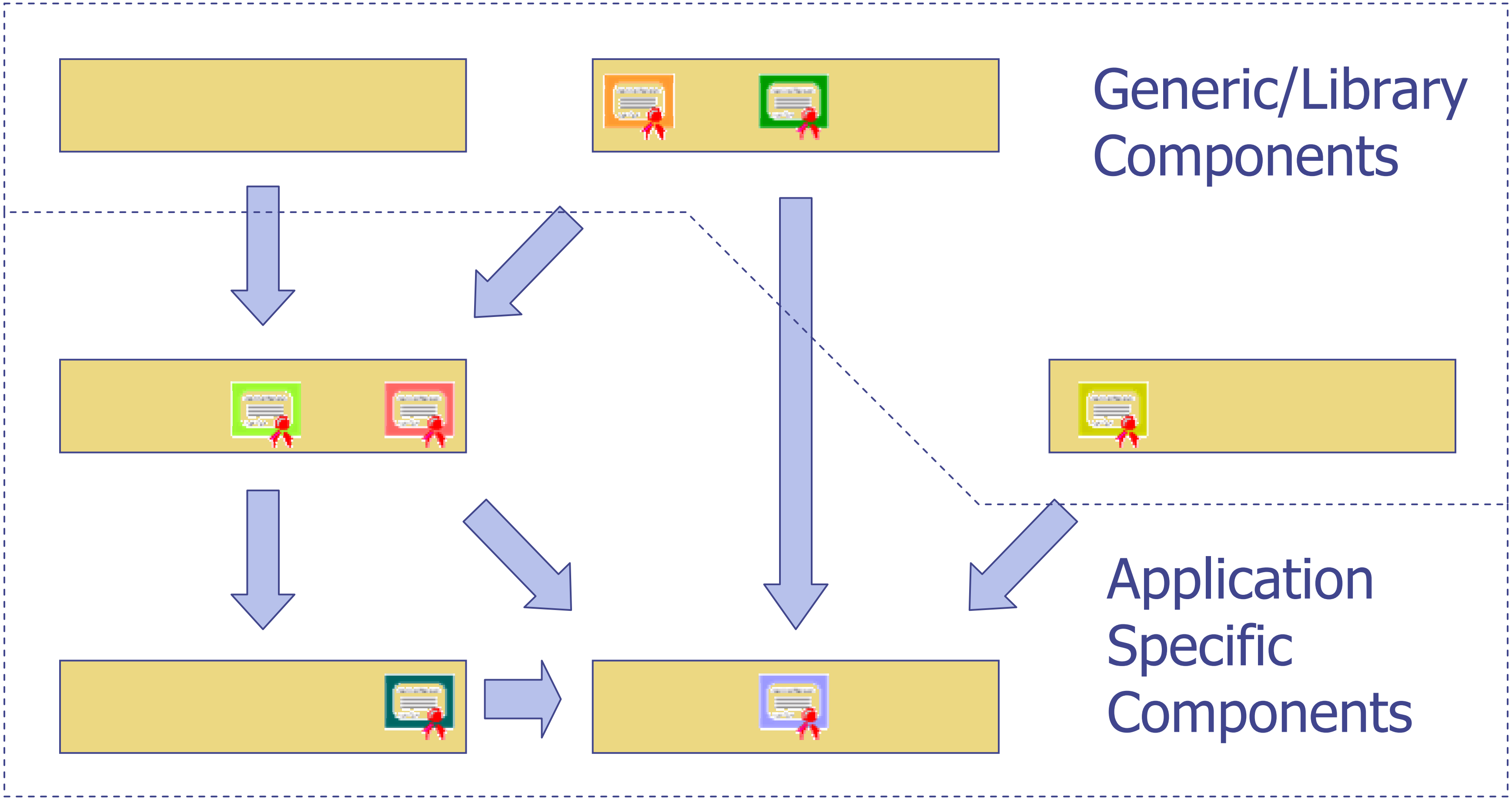




# Assurance Requirements Change:

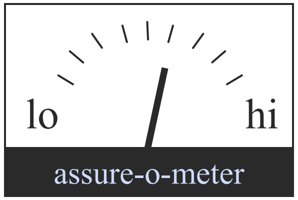
Invest more in validation as overall design begins to stabilize

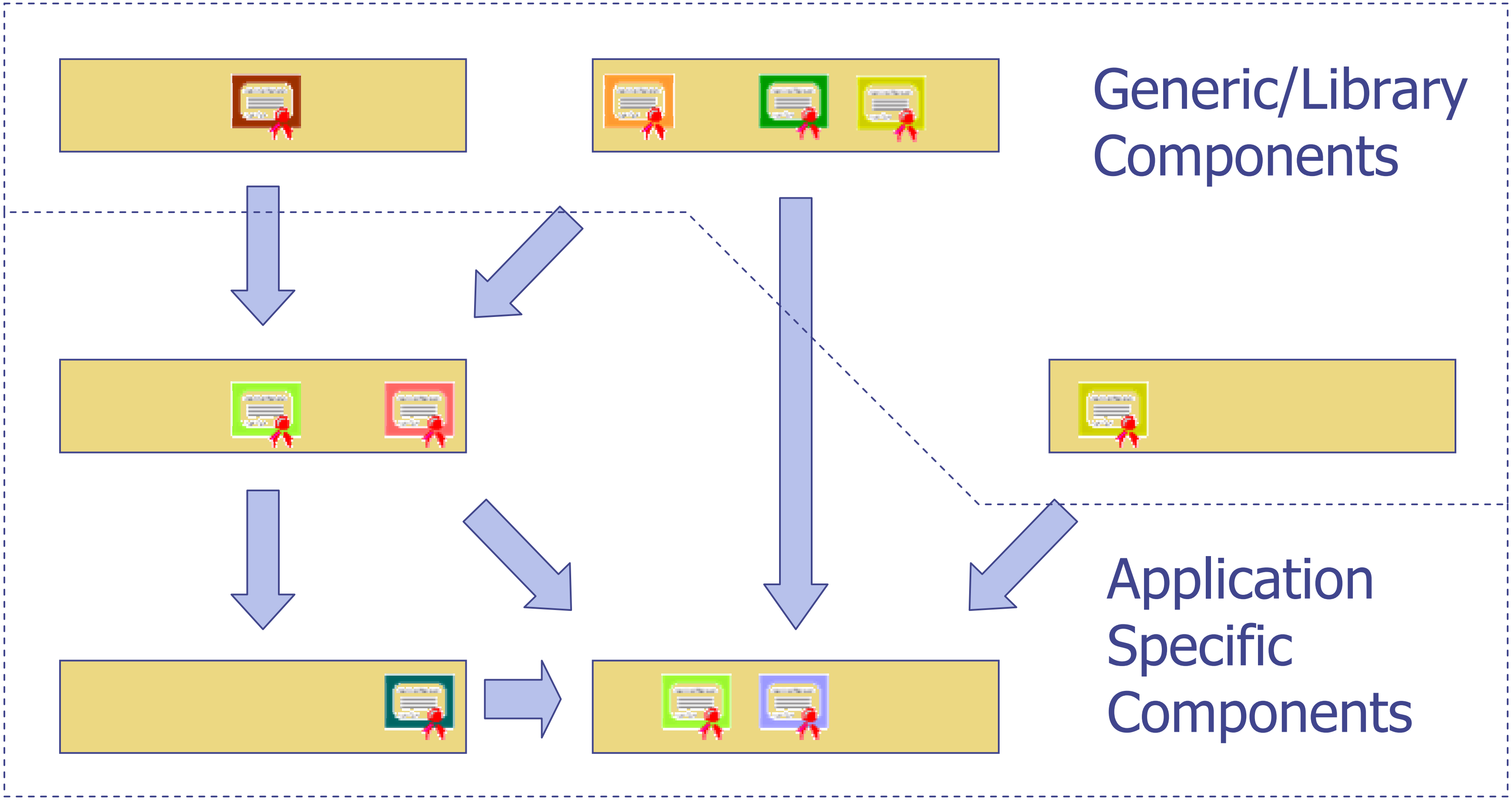




# Assurance Requirements Change:

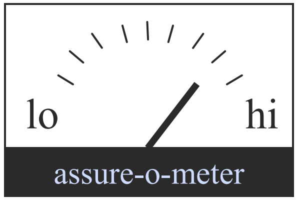
Increase assurance as development begins to mature

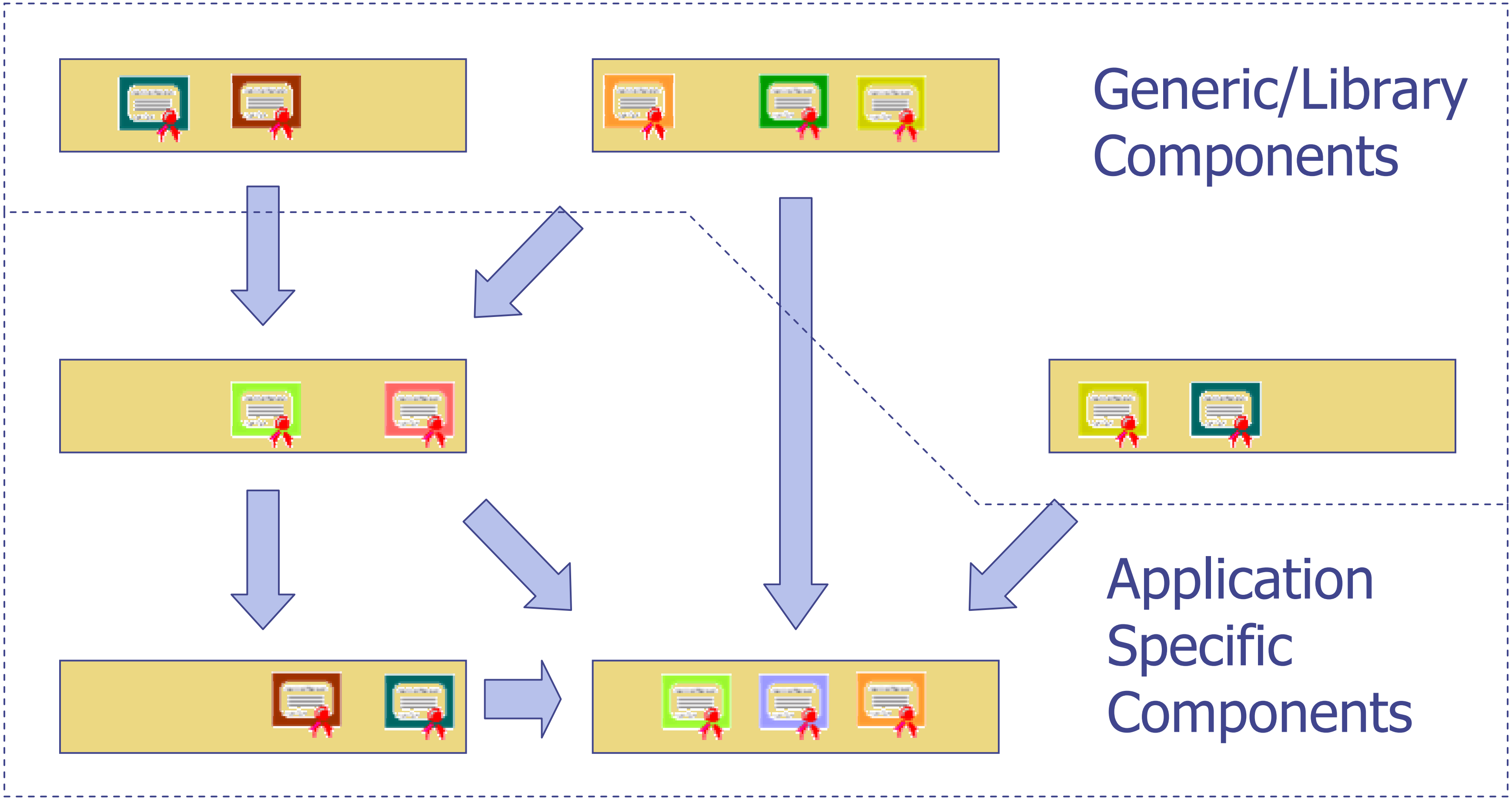




# Assurance Requirements Change:

Maximize assurance before final deployment





# Programatica Components:

- A semantically rich, formal modeling language (Haskell)
- An expressive programming logic that can be used to capture critical program properties (P-logic)
- A toolset for creating, maintaining, and auditing the supporting evidence (pfe,cert,...)

# Sample Applications:

- Channel separation for a (hypothetical) crypto chip design

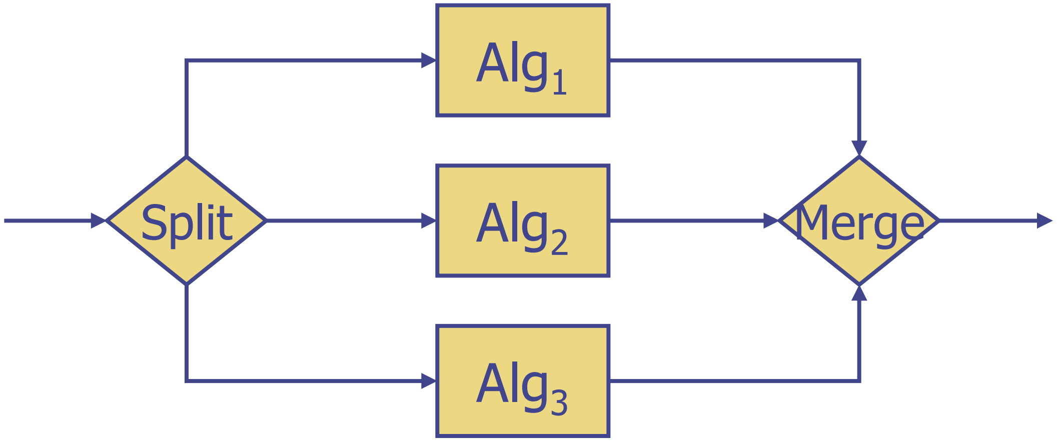
Running example in this talk

- Domain/process separation in Osker, the "Oregon Separation Kernel"

- Preliminary Experiments in the context of Trusted Web Server work at Galois Connections

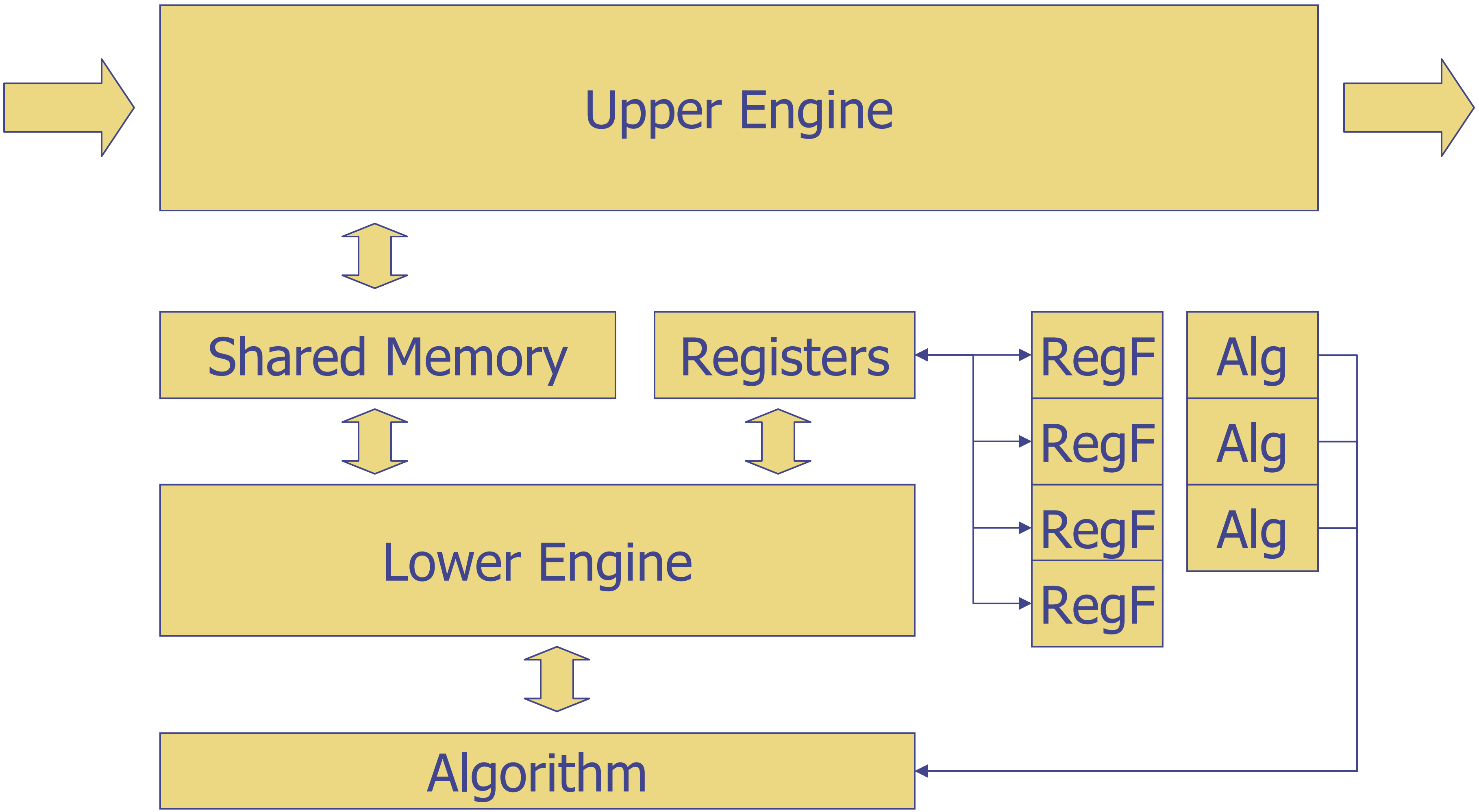
# Example: Modeling a Crypto-Chip

- An example based on a hypothetical crypto-chip design
- Conceptual view:



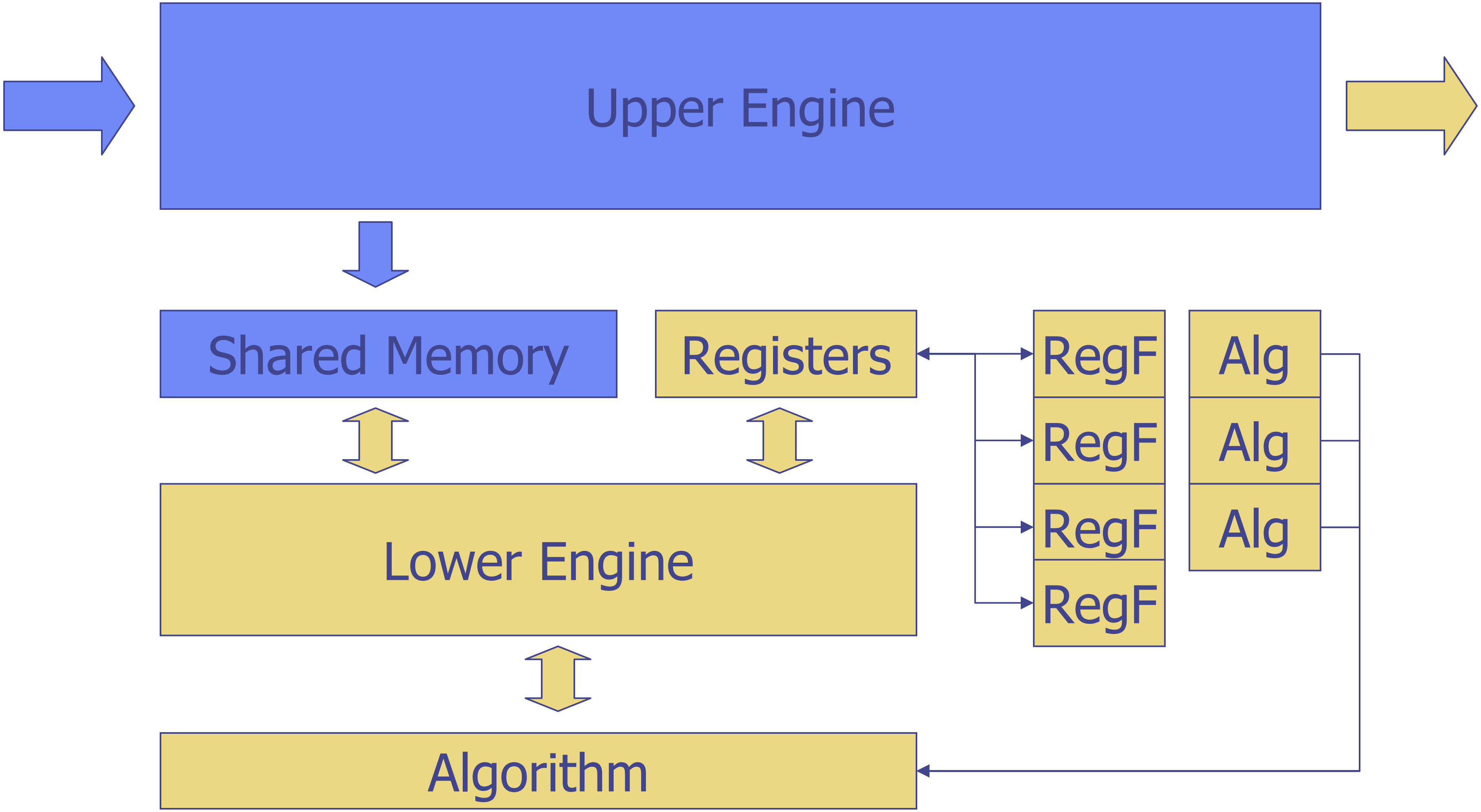
- One chip, multiple channels
- Channels may use different algorithms
- GUARANTEED separation between channels

# Basic architecture:



# Basic architecture:

Receive packets, save in shared memory.


Upper Engine

Shared Memory

Registers

RegF

RegF

RegF

RegF

Alg

Alg

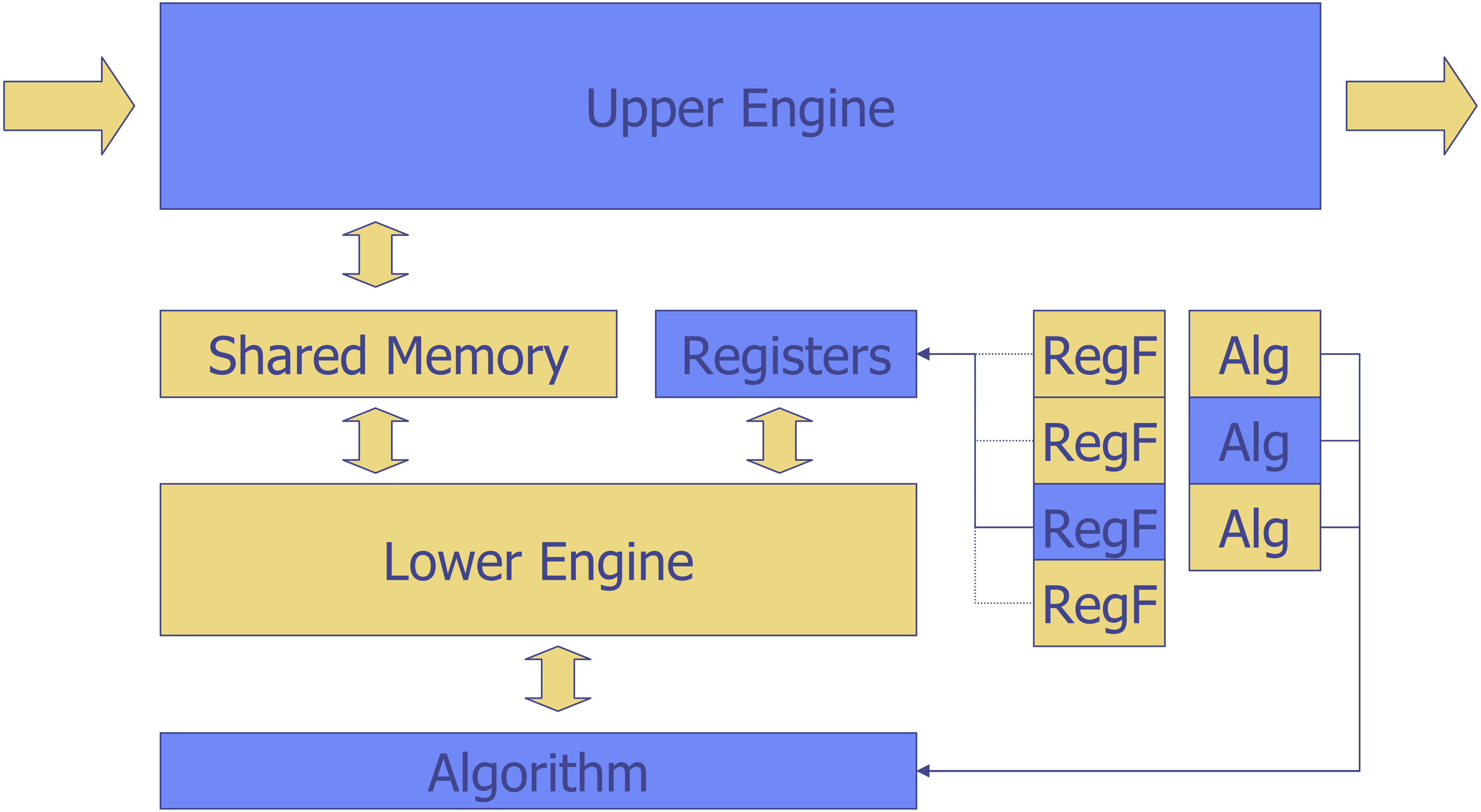
Alg

Lower Engine

Algorithm

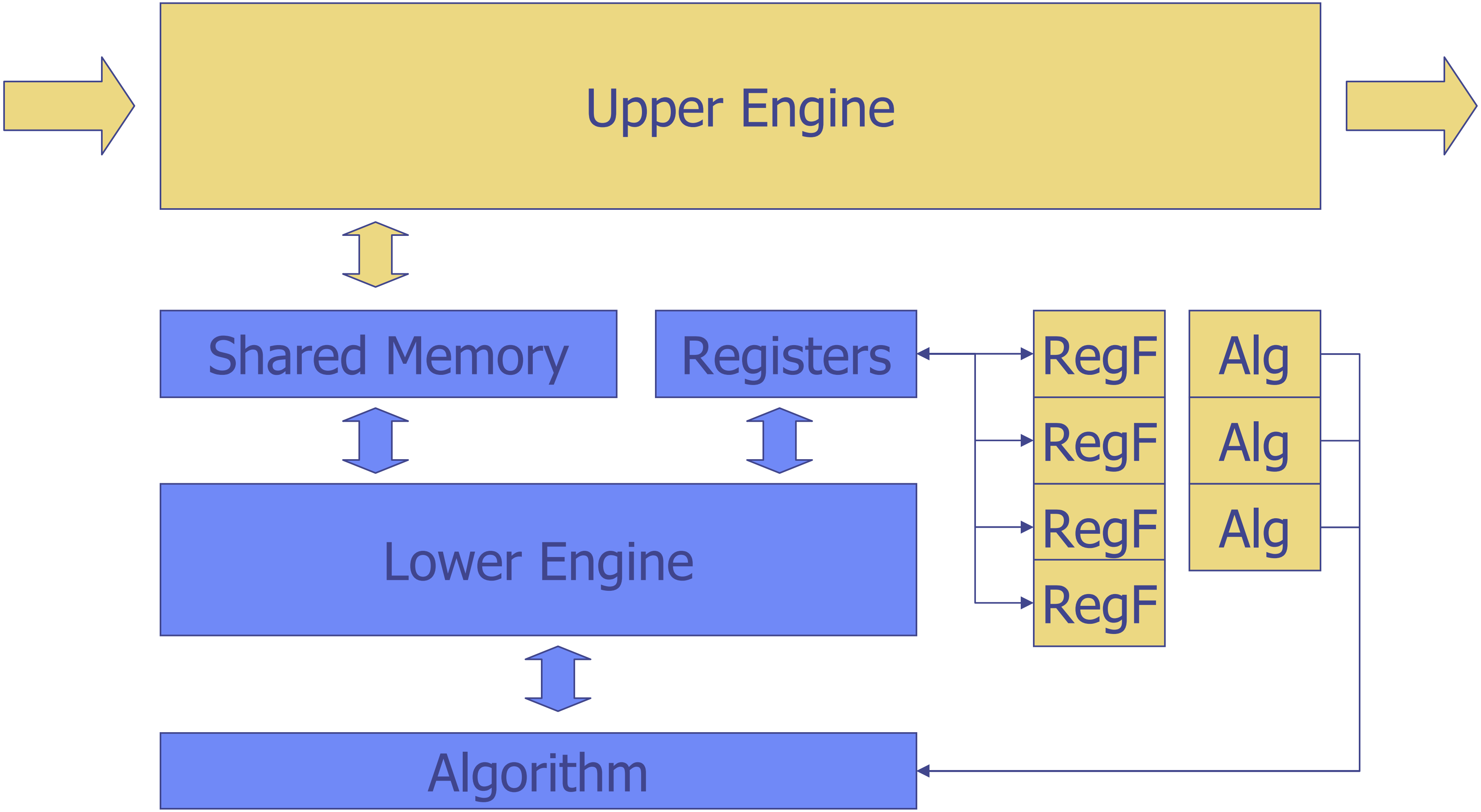
# Basic architecture:

Load saved registers & algorithm for channel.



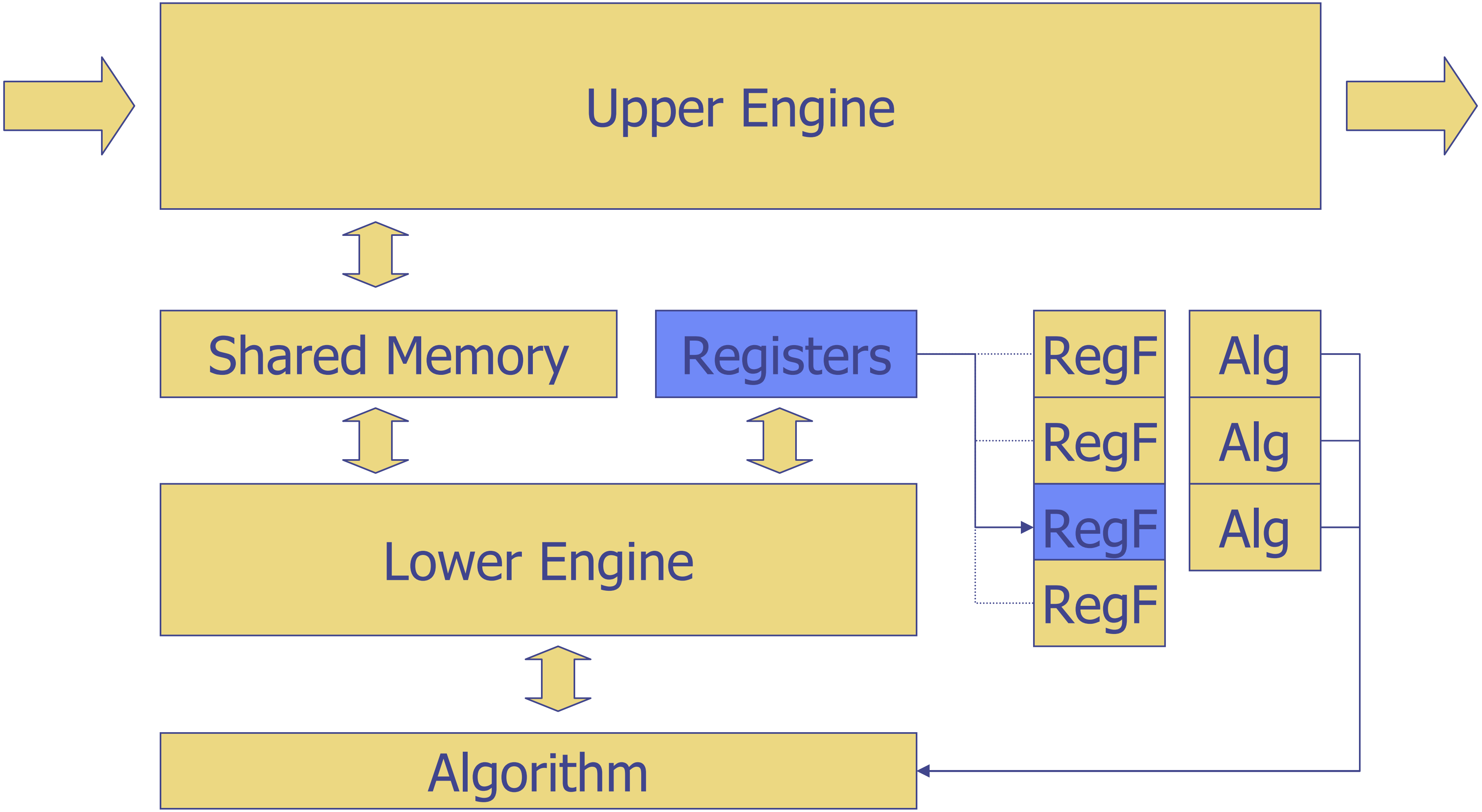
# Basic architecture:

Invoke lower engine to process packet.



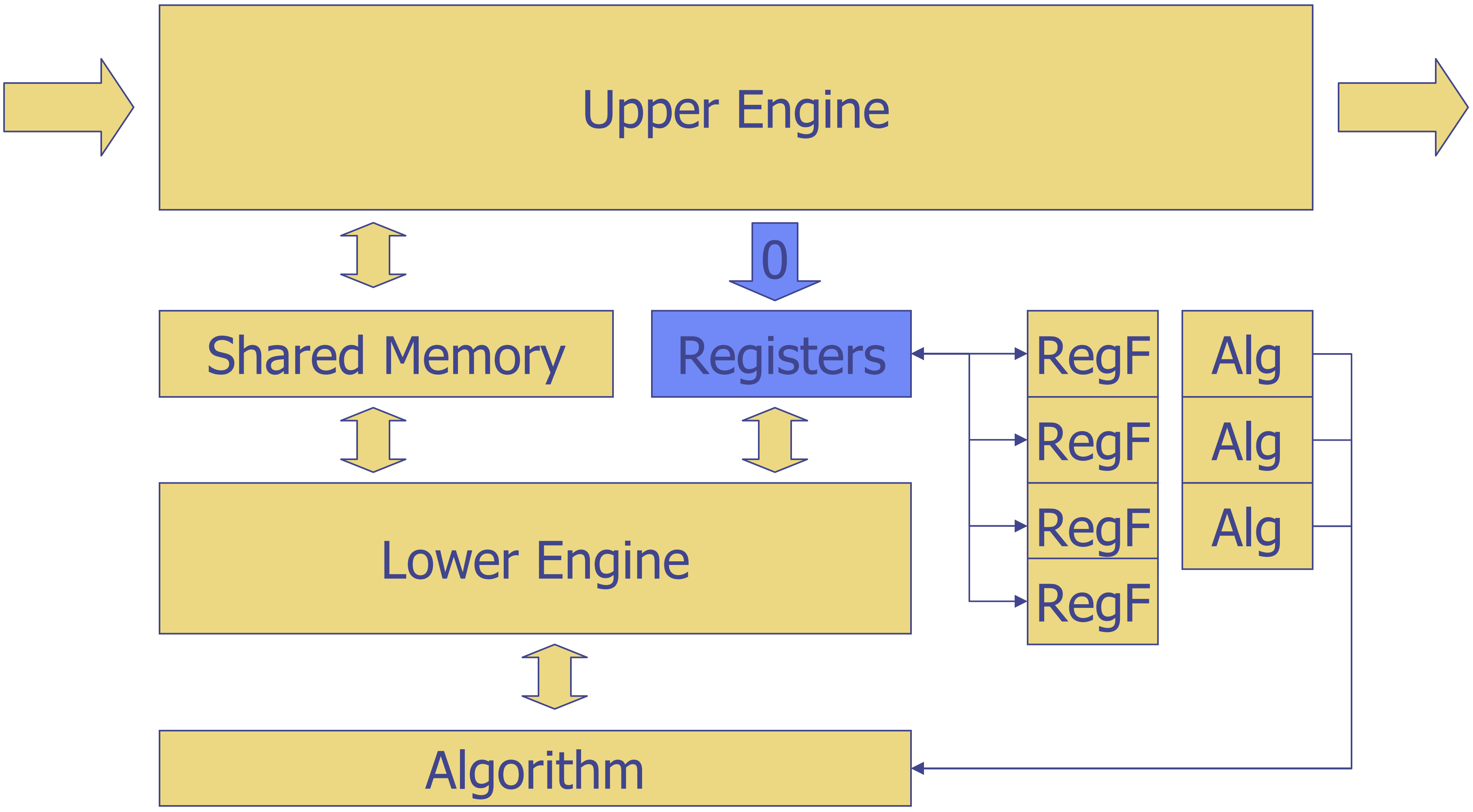
# Basic architecture:

Save register set, if lower engine completes successfully.



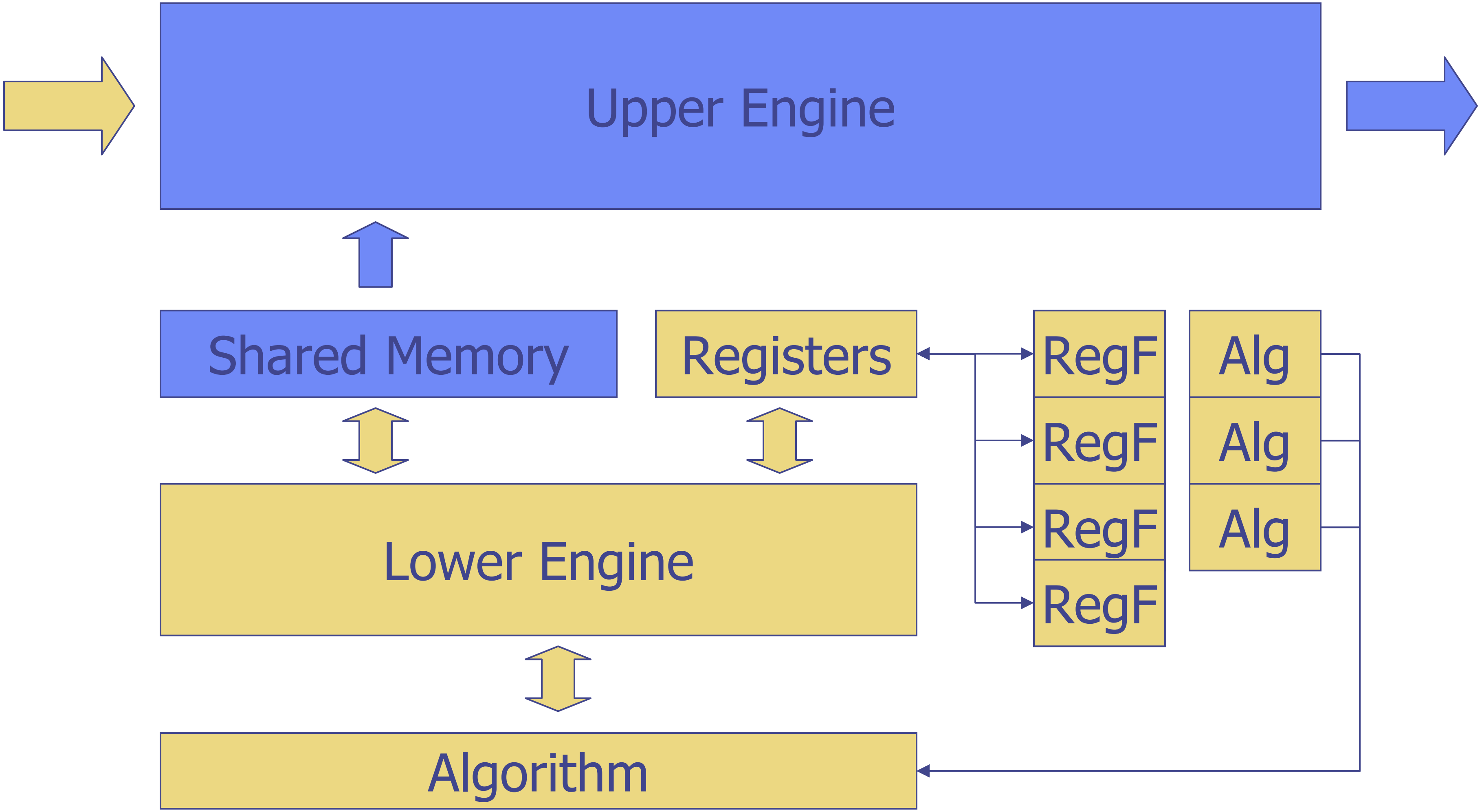
# Basic architecture:

Zero out shared register set.



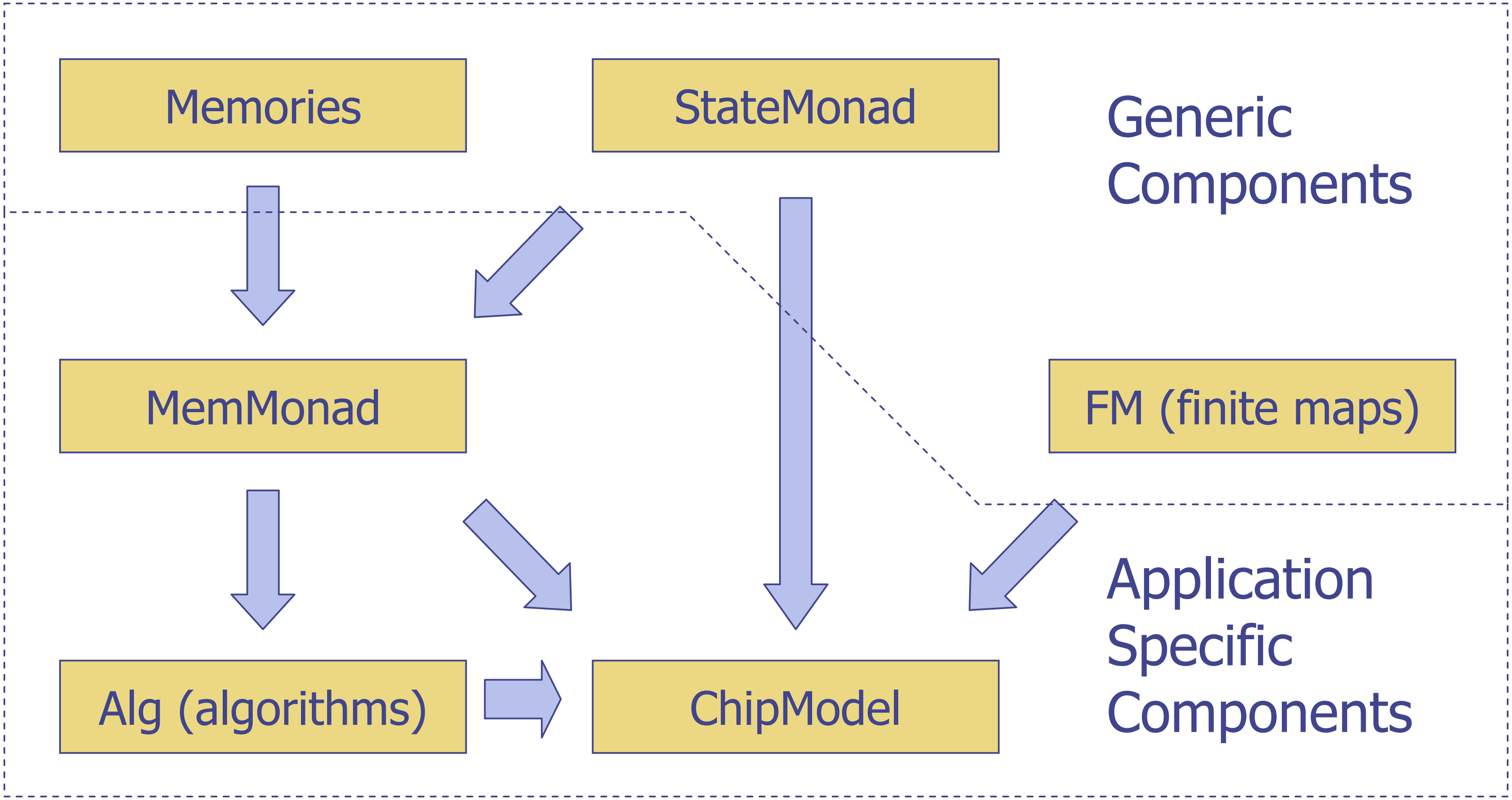
# Basic architecture:

Pass processed packet data to output.



# Building the Model:

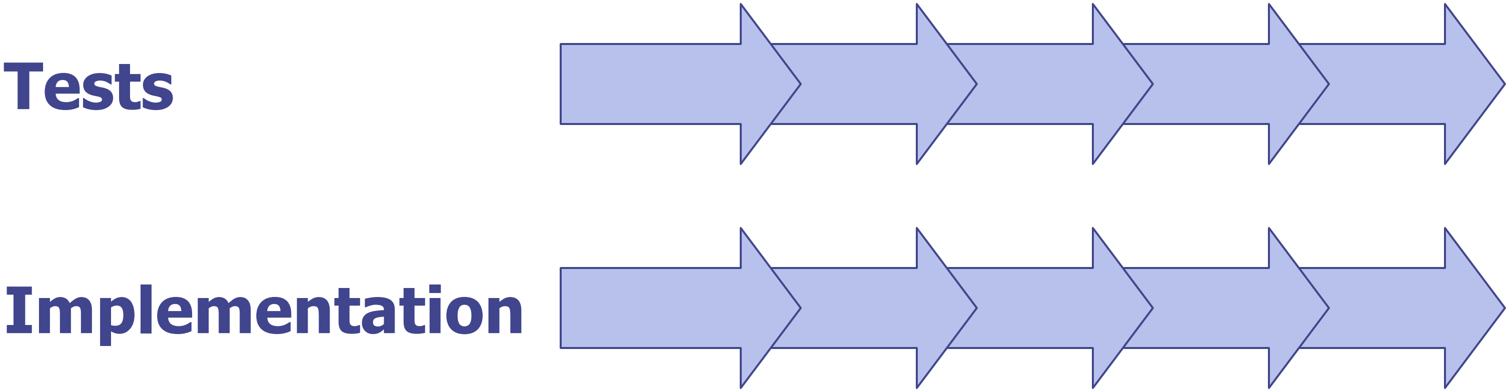
We developed an executable model of the `chip` as a Haskell program: (~260 LOC)



# "Programming as if Properties Matter"

- Properties are written, parsed, analyzed, and type-checked as an integral part of source text
- Maintains consistency between code and properties
- Captures programmer expectations/intentions as part of the programming process
- Our experience: Just writing down properties heightens thinking about correctness

# Extreme Programming

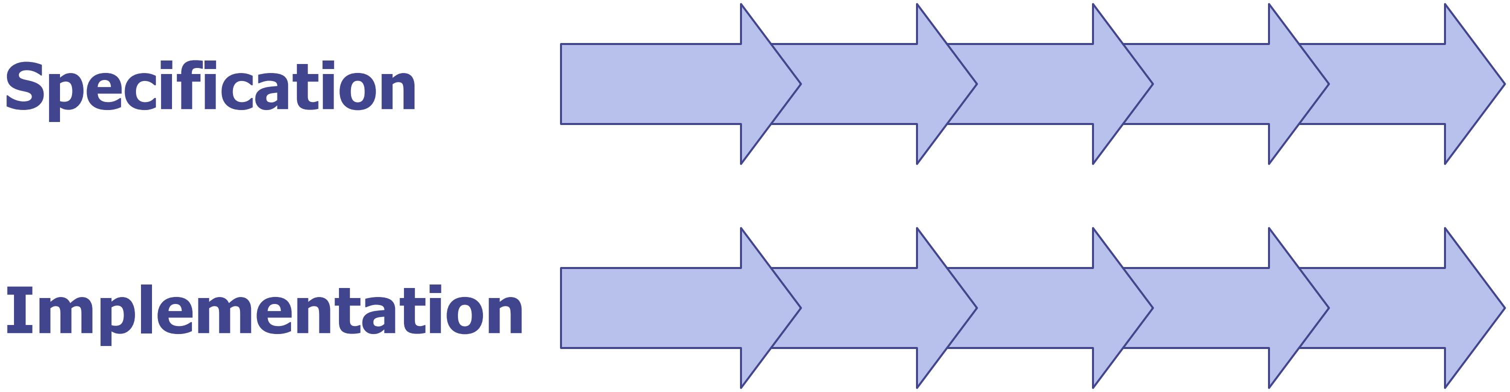


**Tests**

**Implementation**

- Testing and Programming proceed hand in hand
- Testing reveals errors in the program
- Programming reveals errors in the test cases

# "Extreme Formal Methods"



**Specification**

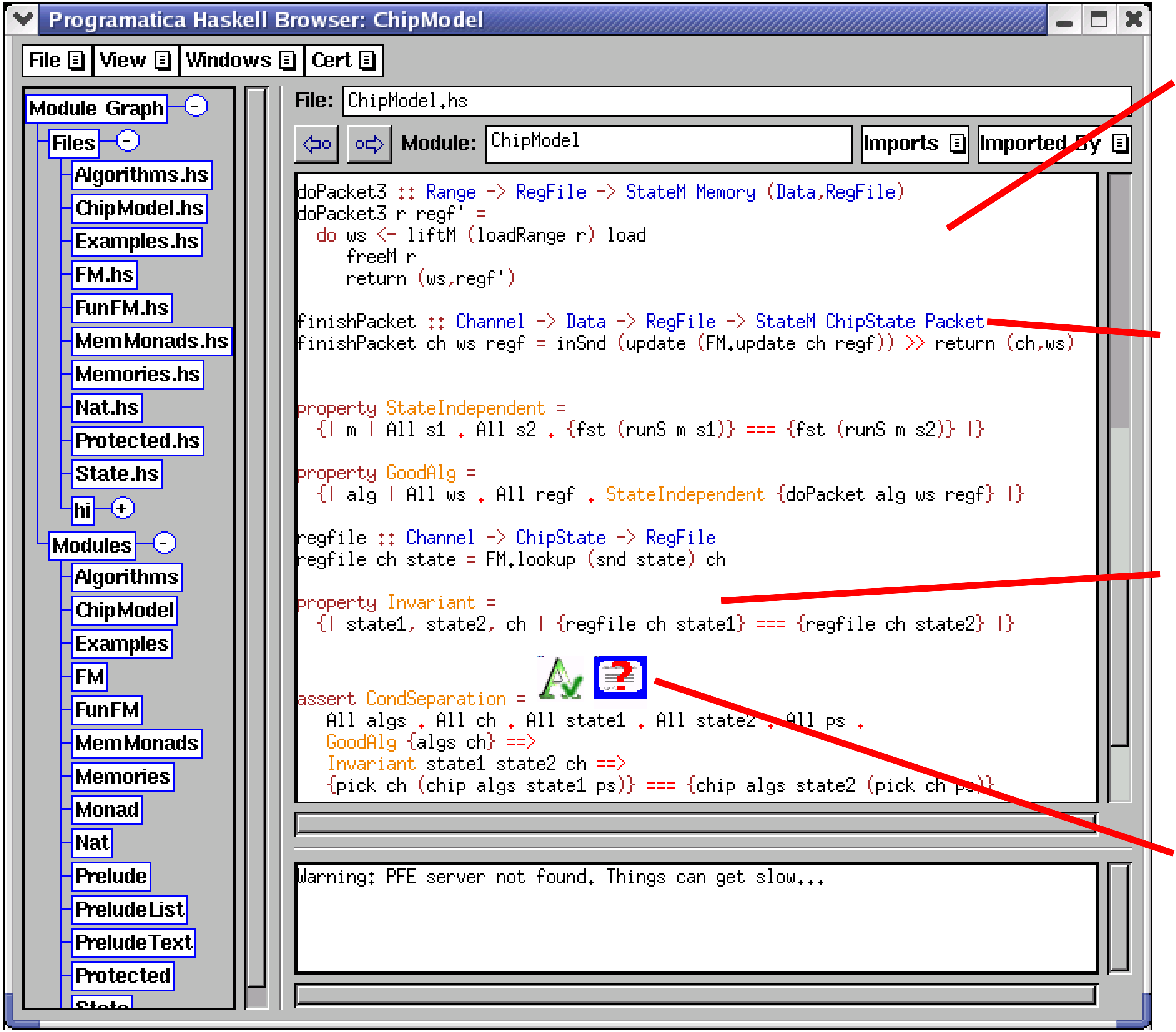
**Implementation**

- Programming and Validation proceed hand in hand
- Validation reveals errors in the program
- Programming reveals errors in the specification

# Demo:

- Programatica as a Modeling and Development Environment

(At this point in the talk, I started switching back and forward between the slides and a demo of the Programatica toolset. The next few slides show screenshots from that demo with a few additional annotations that I hope will convey the key ideas ...)



File: ChipModel.hs

Module: ChipModel

```
doPacket3 :: Range -> RegFile -> StateM Memory (Data,RegFile)
doPacket3 r regf' =
  do ws <- liftM (loadRange r) load
     freeM r
     return (ws,regf')

finishPacket :: Channel -> Data -> RegFile -> StateM ChipState Packet
finishPacket ch ws regf = inSnd (update (FM.update ch regf)) >> return (ch,ws)


property StateIndependent =
  {| m | All s1 . All s2 . {fst (runS m s1)} === {fst (runS m s2)} |}

property GoodAlg =
  {| alg | All ws . All regf . StateIndependent {doPacket alg ws regf} |}

regfile :: Channel -> ChipState -> RegFile
regfile ch state = FM.lookup (snd state) ch

property Invariant =
  {| state1, state2, ch | {regfile ch state1} === {regfile ch state2} |}


assert CondSeparation =
   All algs . All ch . All state1 . All state2 . All ps .
   GoodAlg {algs ch} ==>
   Invariant state1 state2 ch ==>
   {pick ch (chip algs state1 ps)} === {chip algs state2 (pick ch ps)}
```

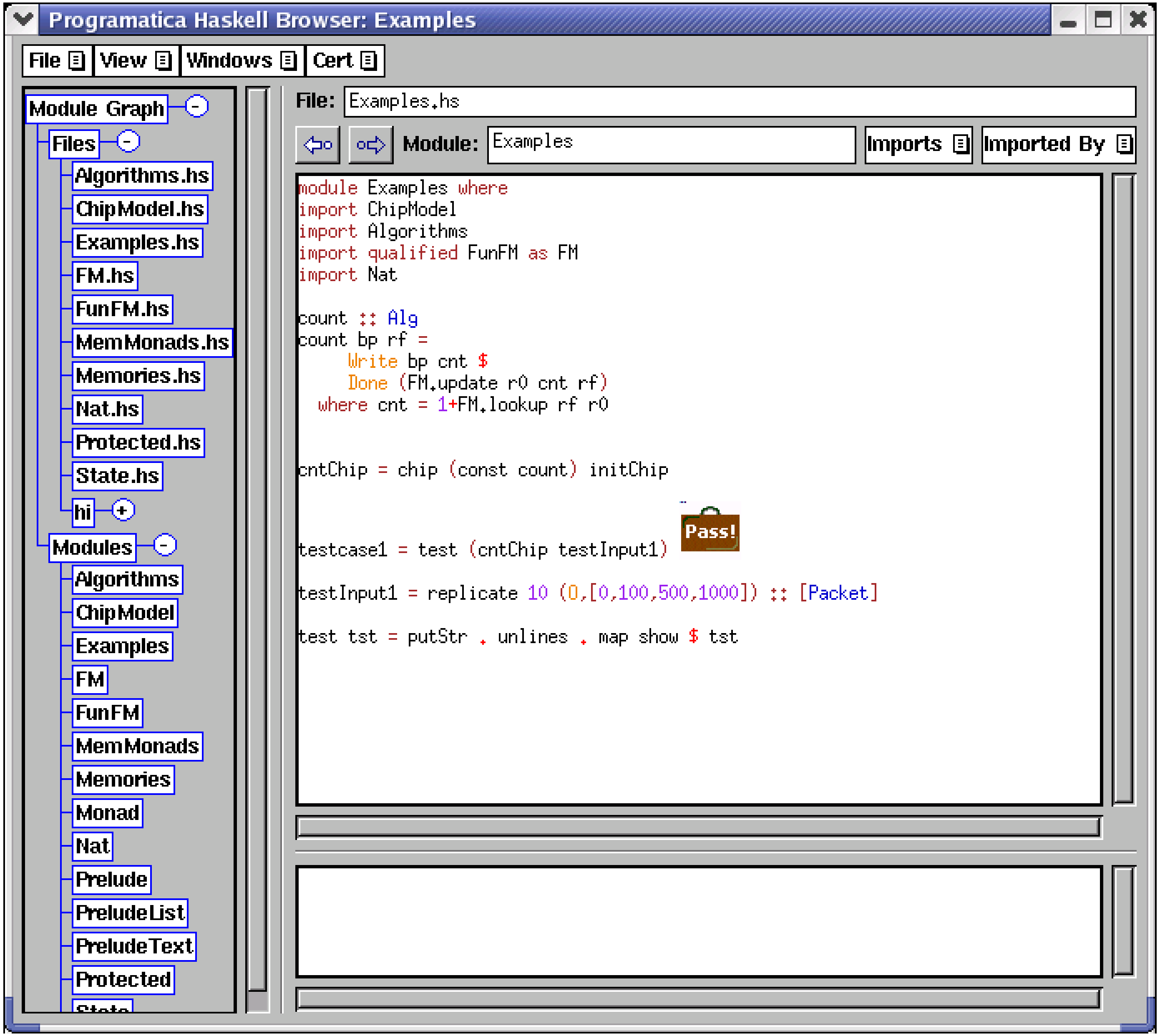
```
Warning: PFE server not found. Things can get slow...
```

A program development environment

Syntax coloring and hyper linking

Embedded property definitions and assertions

Embedded certificates



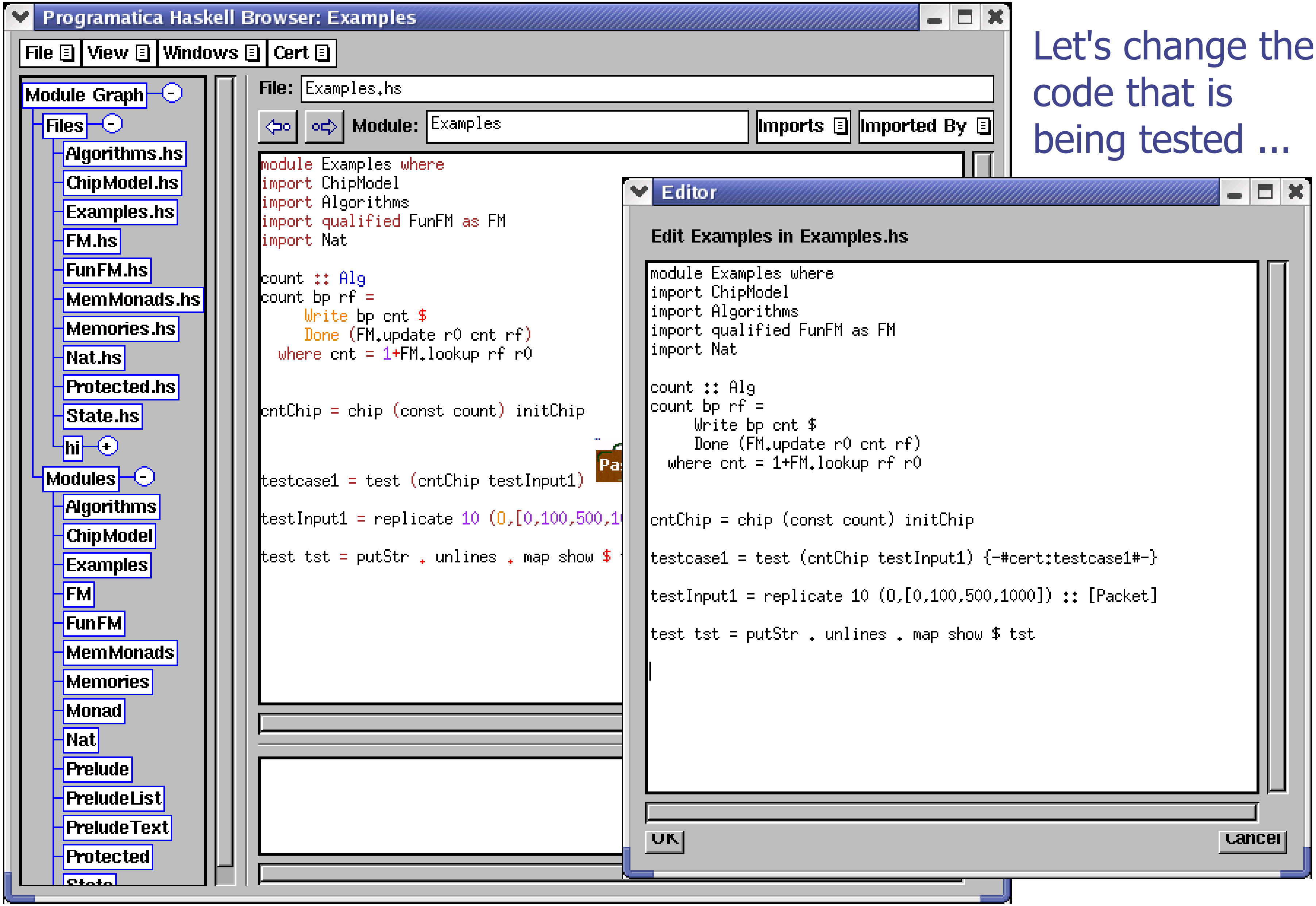
File: Examples.hs

Module: Examples

```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
     Write bp cnt $
     Done (FM.update r0 cnt rf)
  where cnt = 1+FM.lookup rf r0


cntChip = chip (const count) initChip



testcase1 = test (cntChip testInput1)

testInput1 = replicate 10 (0,[0,100,500,1000]) :: [Packet]

test tst = putStr . unlines . map show $ tst
```

Here's a program that contains a simple test case certificate ...

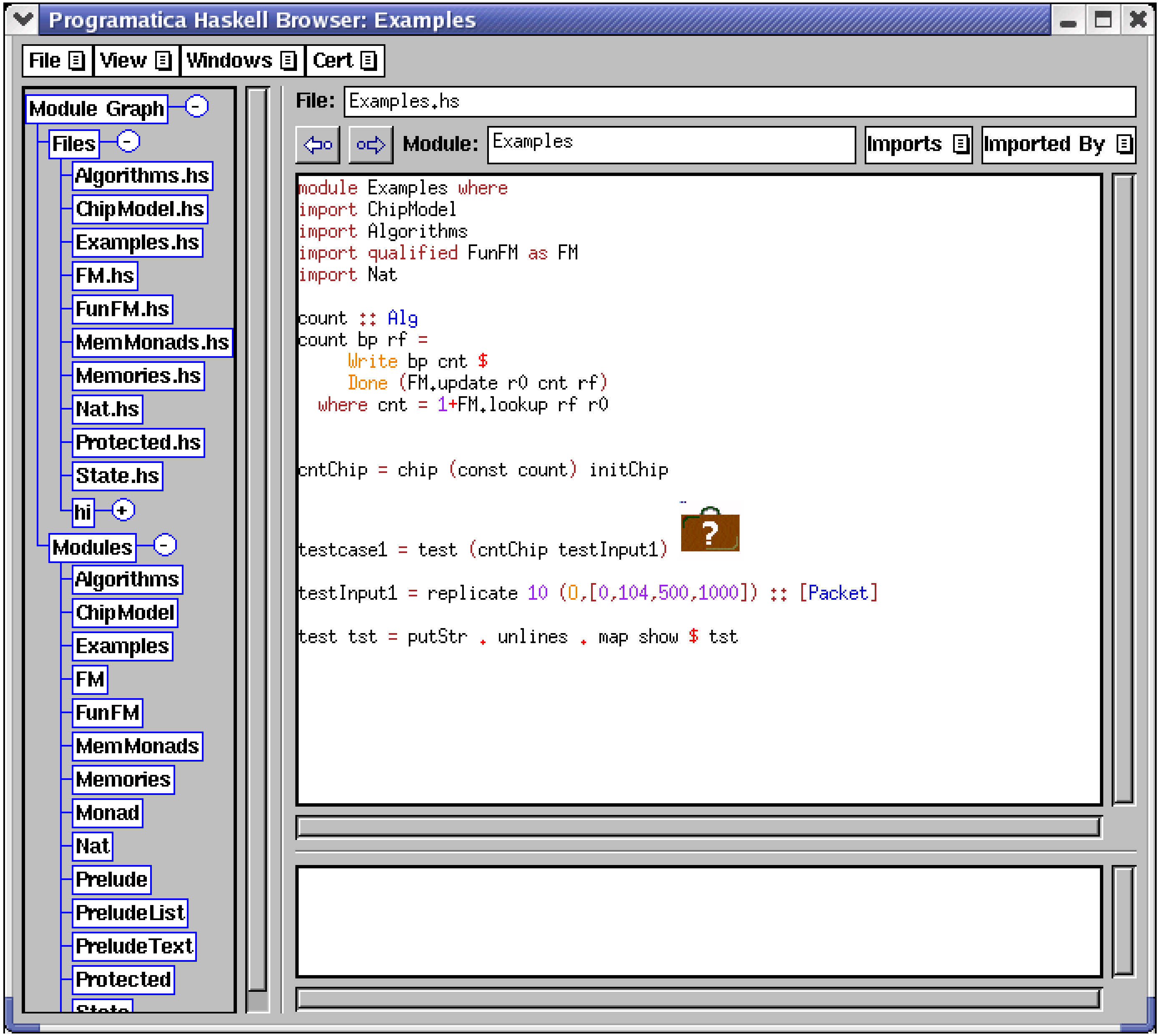
Let's change the code that is being tested ...



```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
     Write bp cnt $
     Done (FM.update r0 cnt rf)
  where cnt = 1+FM.lookup rf r0


cntChip = chip (const count) initChip


testcase1 = test (cntChip testInput1)

testInput1 = replicate 10 (0,[0,100,500,1

test tst = putStr . unlines . map show $ t
```

Edit Examples in Examples.hs

```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
     Write bp cnt $
     Done (FM.update r0 cnt rf)
  where cnt = 1+FM.lookup rf r0


cntChip = chip (const count) initChip

testcase1 = test (cntChip testInput1) {-#cert:testcase1#-}

testInput1 = replicate 10 (0,[0,100,500,1000]) :: [Packet]

test tst = putStr . unlines . map show $ tst
```



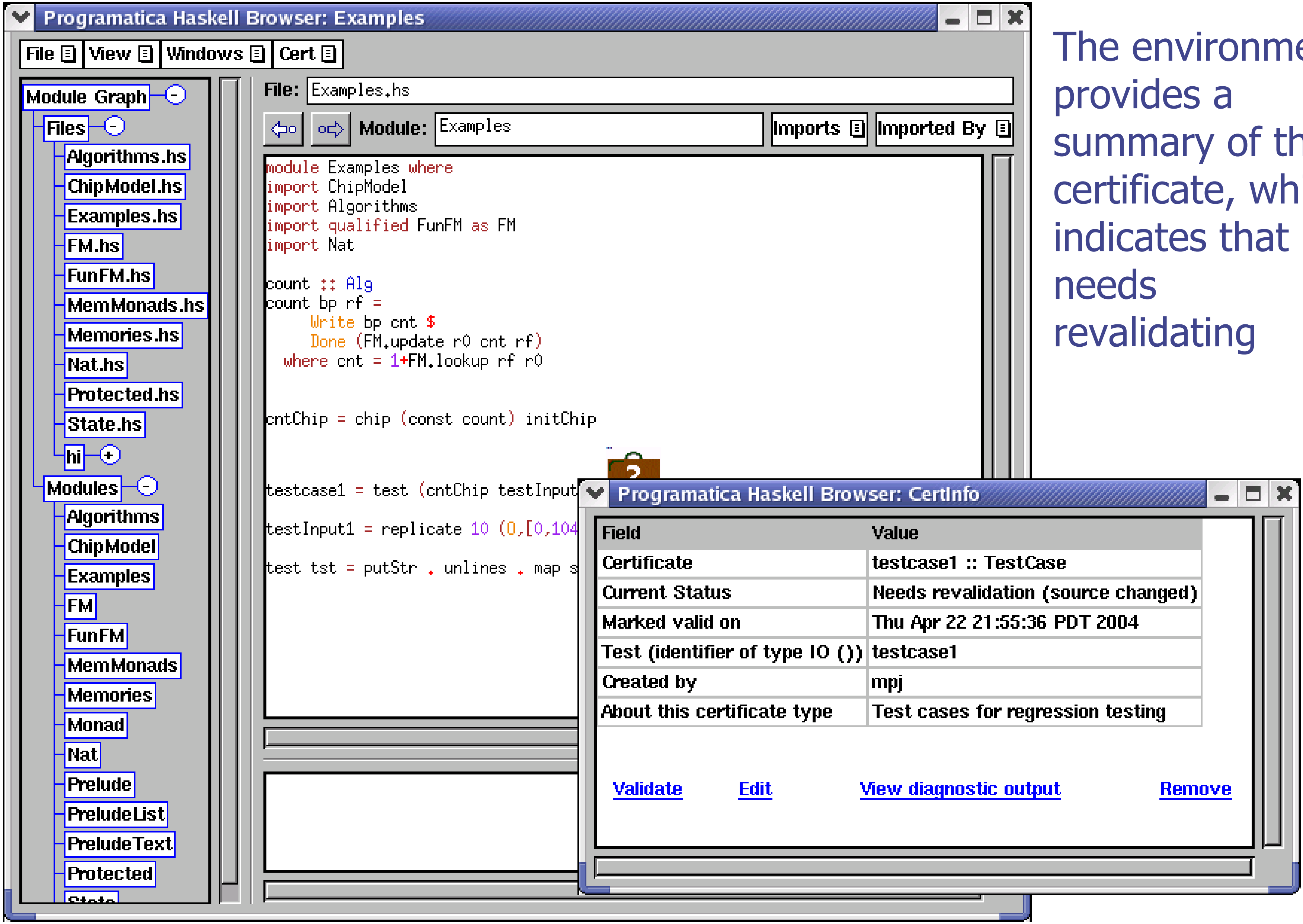
File: Examples.hs

Module: Examples

```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
     Write bp cnt $
     Done (FM.update r0 cnt rf)
  where cnt = 1+FM.lookup rf r0


cntChip = chip (const count) initChip


testcase1 = test (cntChip testInput1)

testInput1 = replicate 10 (0,[0,104,500,1000]) :: [Packet]

test tst = putStr . unlines . map show $ tst
```

Programatica's dependency checking mechanisms detect that there have been changes in parts of the program that might affect the validity of the certificate.

So it is marked with a "?" ...

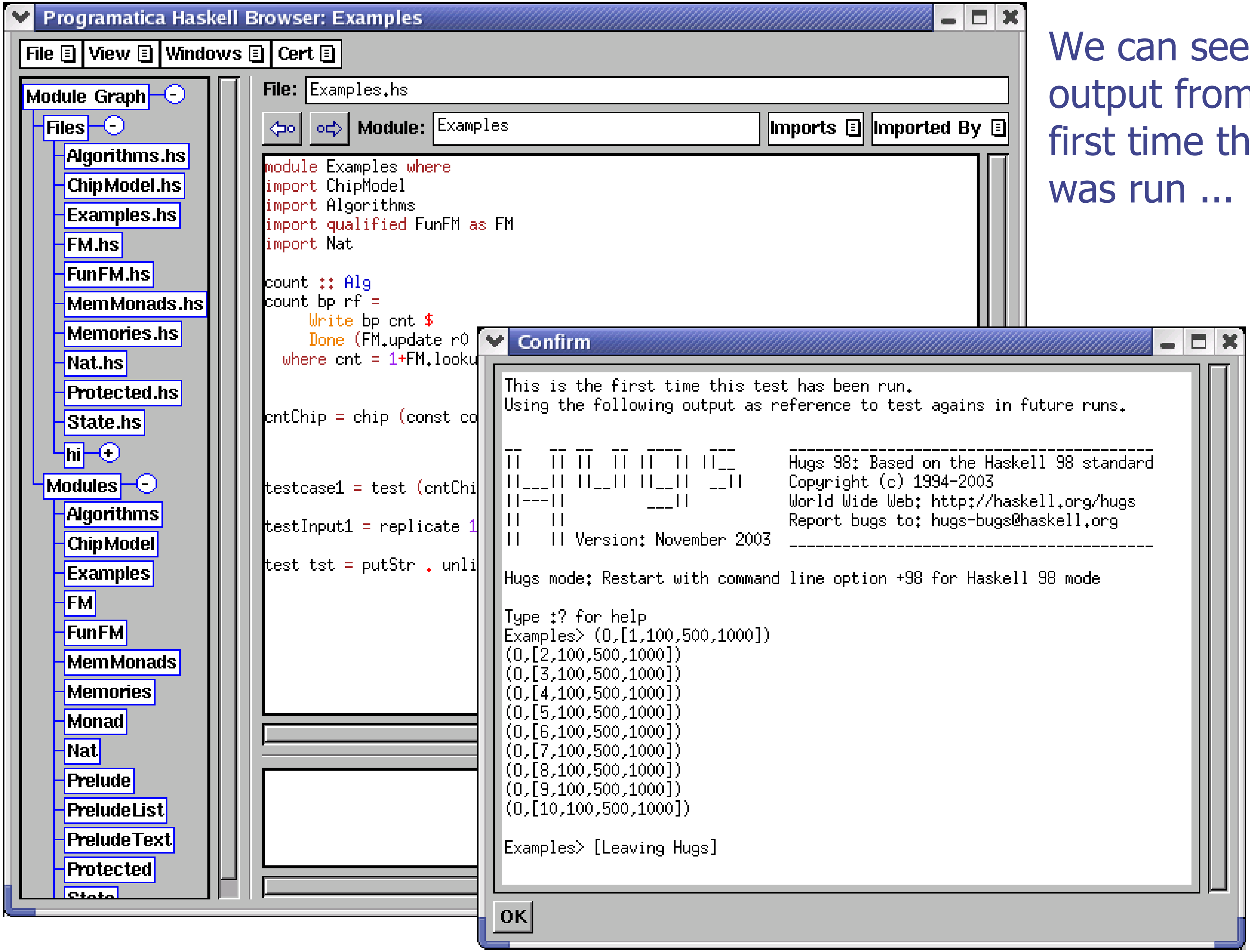


```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
      Write bp cnt $
      Done (FM.update r0 cnt rf)
  where cnt = 1+FM.lookup rf r0


cntChip = chip (const count) initChip



testcase1 = test (cntChip testInput

testInput1 = replicate 10 (0,[0,104

test tst = putStr . unlines . map s
```

| Field | Value |
|---|---|
| Certificate | testcase1 :: TestCase |
| Current Status | Needs revalidation (source changed) |
| Marked valid on | Thu Apr 22 21:55:36 PDT 2004 |
| Test (identifier of type IO ()) | testcase1 |
| Created by | mpj |
| About this certificate type | Test cases for regression testing |

The environment provides a summary of the certificate, which indicates that it needs revalidating

We can see the output from the first time the test was run ...



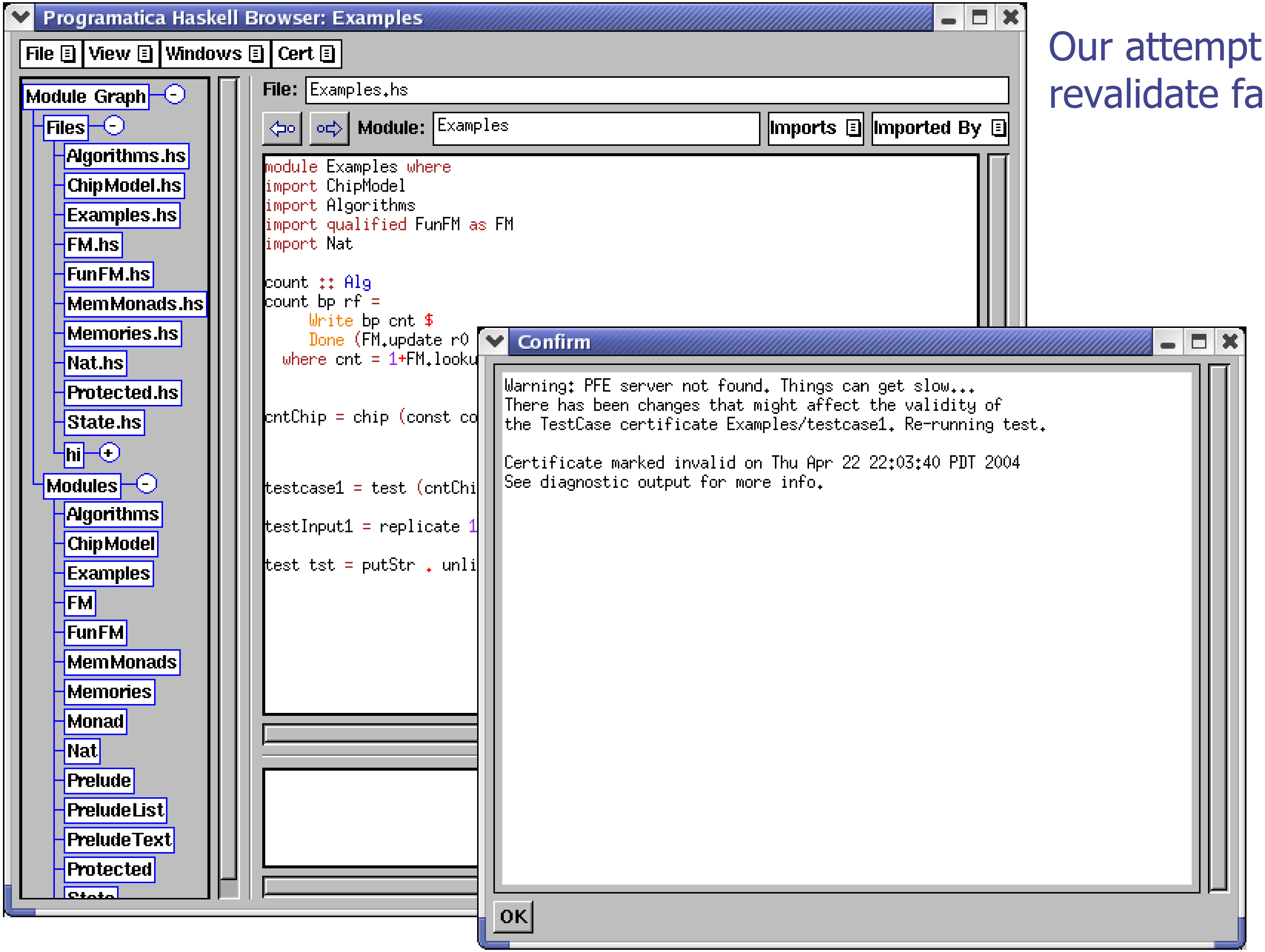
File: Examples.hs

Module: Examples

```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
    Write bp cnt $
    Done (FM.update r0
  where cnt = 1+FM.looku


cntChip = chip (const co



testcase1 = test (cntChi

testInput1 = replicate 1

test tst = putStr . unli
```

```
This is the first time this test has been run.
Using the following output as reference to test agains in future runs.


__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2003
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: November 2003  _________________________________________

Hugs mode: Restart with command line option +98 for Haskell 98 mode

Type :? for help
Examples> (0,[1,100,500,1000])
(0,[2,100,500,1000])
(0,[3,100,500,1000])
(0,[4,100,500,1000])
(0,[5,100,500,1000])
(0,[6,100,500,1000])
(0,[7,100,500,1000])
(0,[8,100,500,1000])
(0,[9,100,500,1000])
(0,[10,100,500,1000])

Examples> [Leaving Hugs]
```

# Our attempt to revalidate fails!

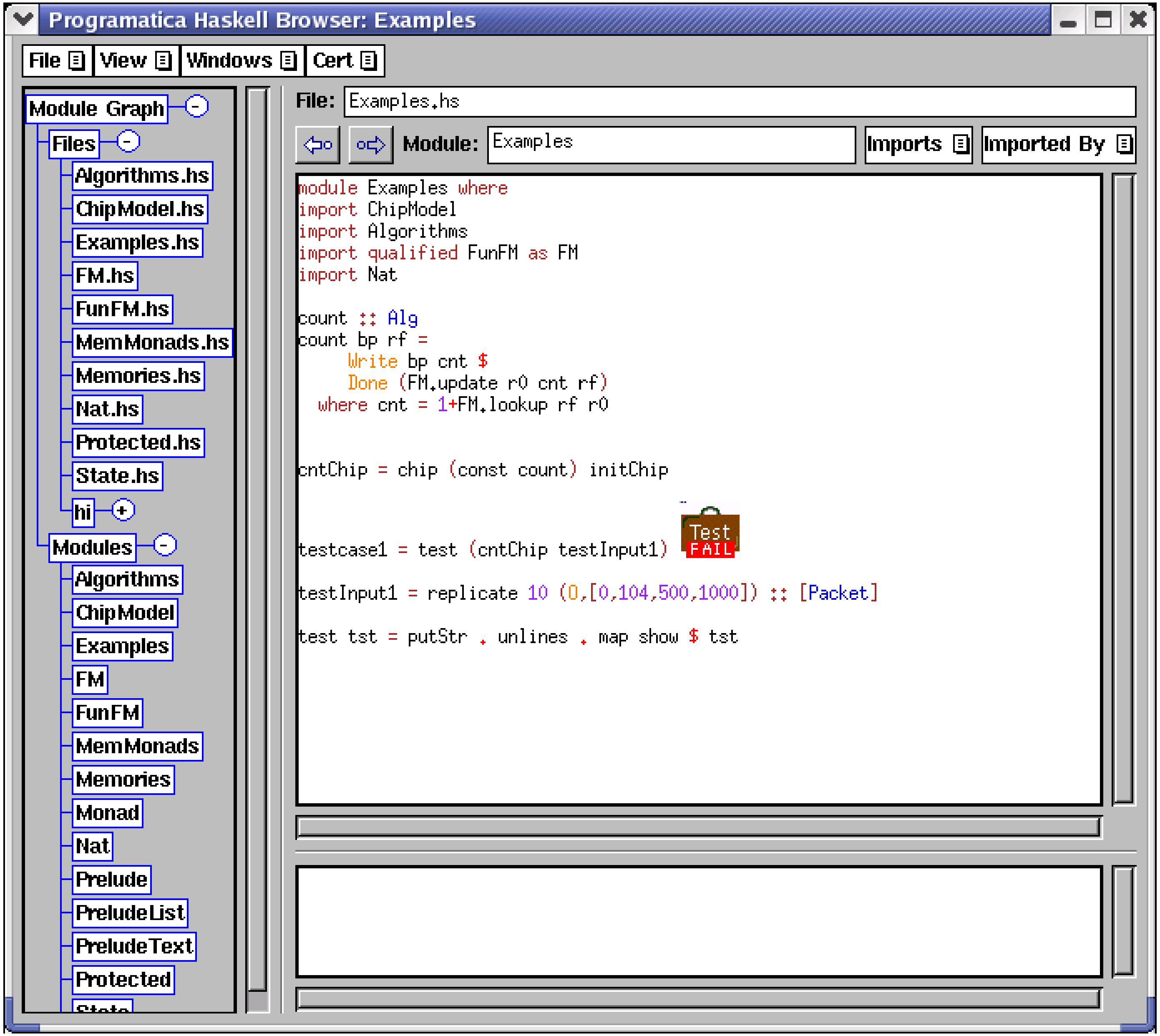


File: Examples.hs

Module: Examples

```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
     Write bp cnt $
     Done (FM.update r0
  where cnt = 1+FM.looku


cntChip = chip (const co


testcase1 = test (cntChi

testInput1 = replicate 1

test tst = putStr . unli
```

```
Warning: PFE server not found. Things can get slow...
There has been changes that might affect the validity of
the TestCase certificate Examples/testcase1. Re-running test.

Certificate marked invalid on Thu Apr 22 22:03:40 PDT 2004
See diagnostic output for more info.
```

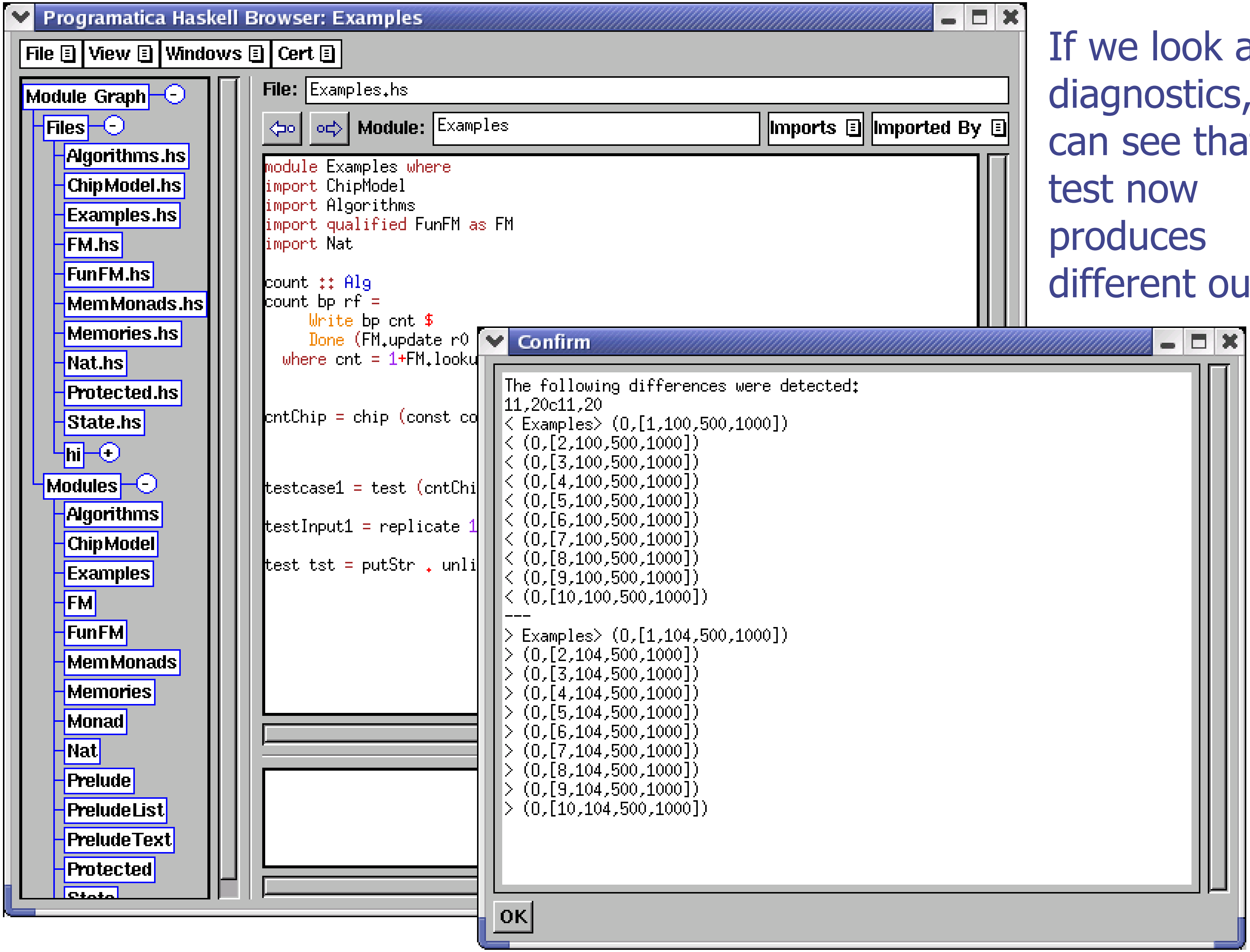


File: Examples.hs

Module: Examples

```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
     Write bp cnt $
     Done (FM.update r0 cnt rf)
  where cnt = 1+FM.lookup rf r0


cntChip = chip (const count) initChip



testcase1 = test (cntChip testInput1)

testInput1 = replicate 10 (0,[0,104,500,1000]) :: [Packet]

test tst = putStr . unlines . map show $ tst
```

And the certificate icon changes again to reflect the problem ...



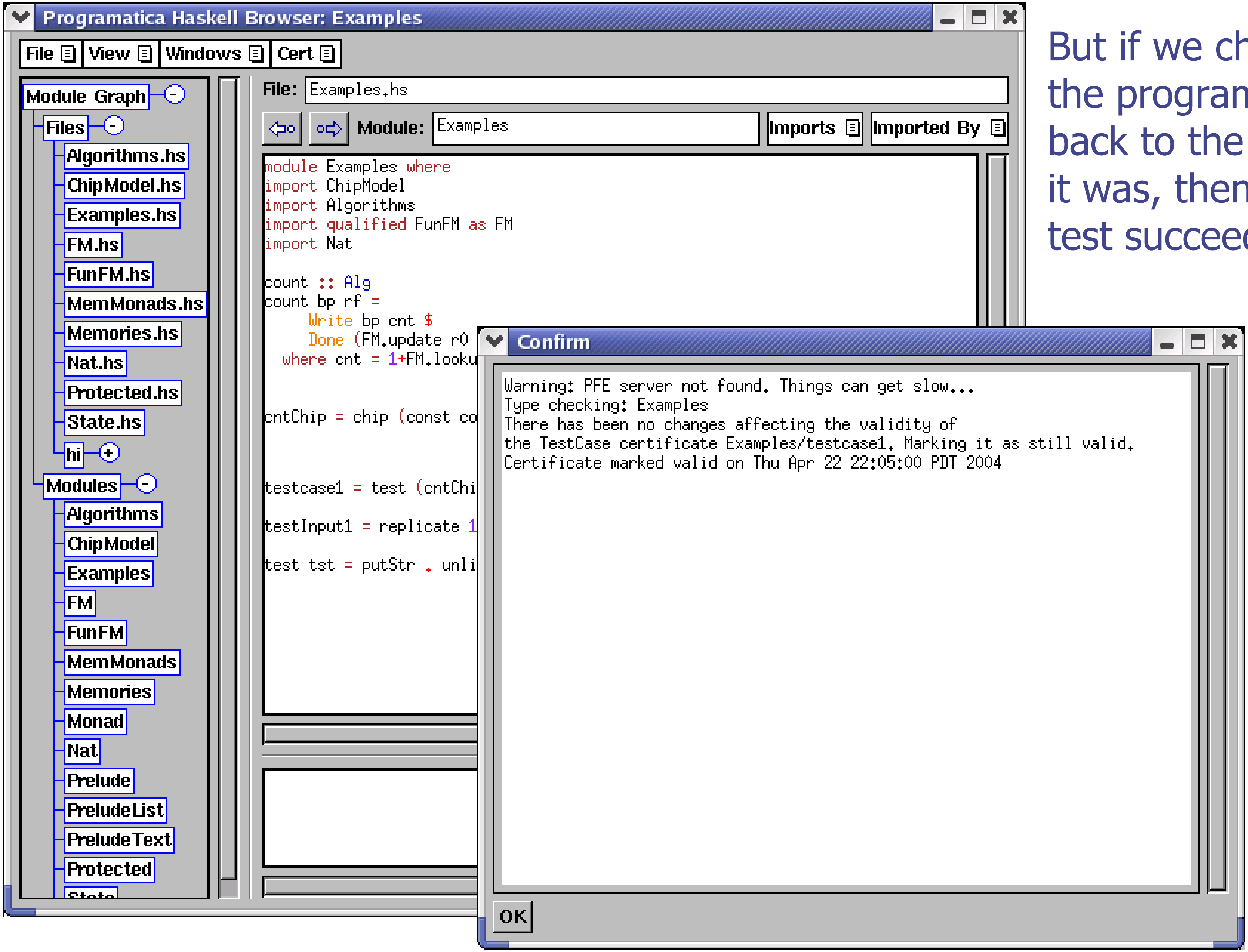
File: Examples.hs

Module: Examples

```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
     Write bp cnt $
     Done (FM.update r0
  where cnt = 1+FM.looku


cntChip = chip (const co


testcase1 = test (cntChi

testInput1 = replicate 1

test tst = putStr . unli
```

```
The following differences were detected:
11,20c11,20
< Examples> (0,[1,100,500,1000])
< (0,[2,100,500,1000])
< (0,[3,100,500,1000])
< (0,[4,100,500,1000])
< (0,[5,100,500,1000])
< (0,[6,100,500,1000])
< (0,[7,100,500,1000])
< (0,[8,100,500,1000])
< (0,[9,100,500,1000])
< (0,[10,100,500,1000])
---
> Examples> (0,[1,104,500,1000])
> (0,[2,104,500,1000])
> (0,[3,104,500,1000])
> (0,[4,104,500,1000])
> (0,[5,104,500,1000])
> (0,[6,104,500,1000])
> (0,[7,104,500,1000])
> (0,[8,104,500,1000])
> (0,[9,104,500,1000])
> (0,[10,104,500,1000])
```

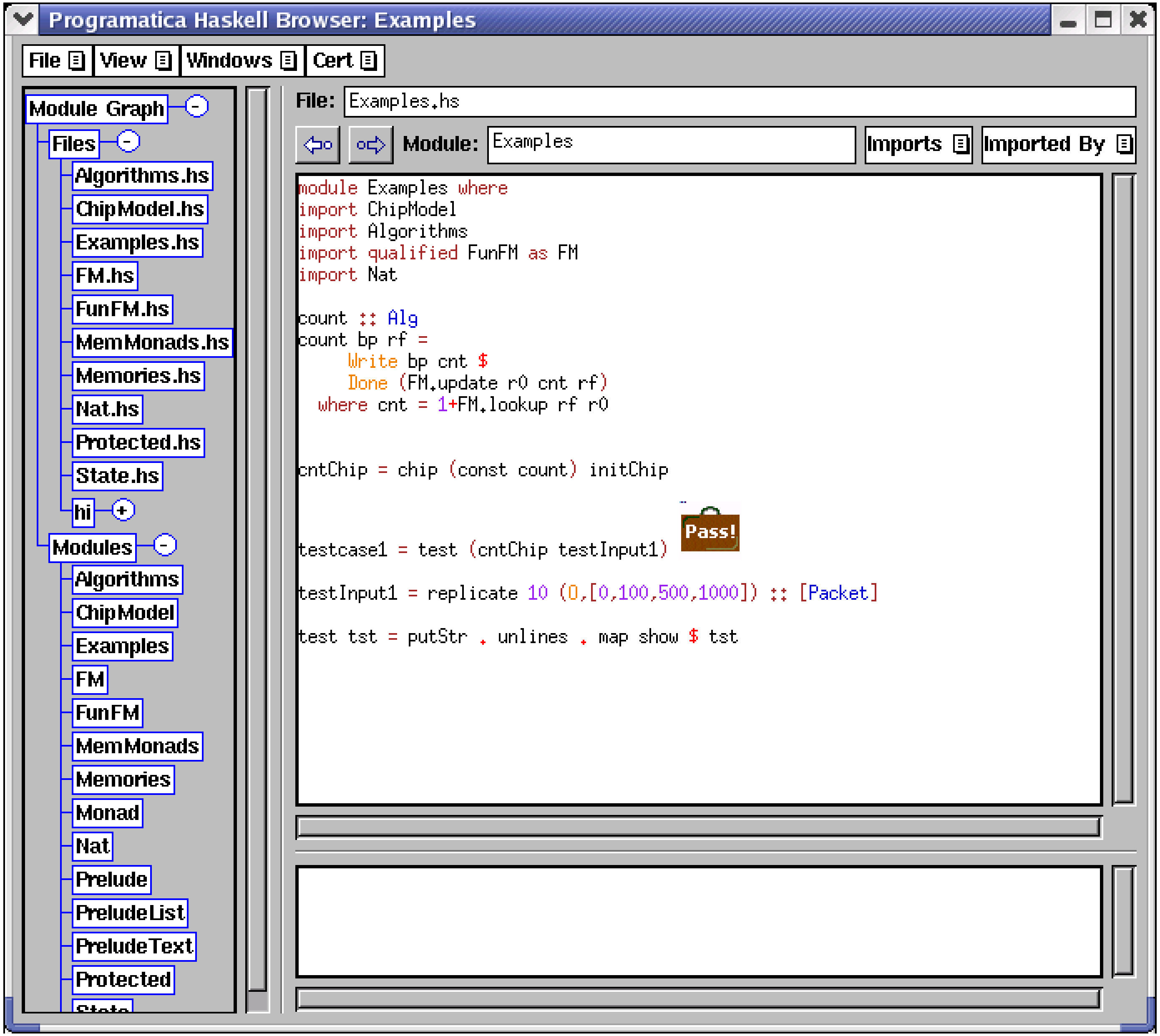
If we look at the diagnostics, we can see that the test now produces different output!



```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
     Write bp cnt $
     Done (FM.update r0
  where cnt = 1+FM.looku


cntChip = chip (const co



testcase1 = test (cntChi

testInput1 = replicate 1

test tst = putStr . unli
```

```
Warning: PFE server not found. Things can get slow...
Type checking: Examples
There has been no changes affecting the validity of
the TestCase certificate Examples/testcase1. Marking it as still valid.
Certificate marked valid on Thu Apr 22 22:05:00 PDT 2004
```

But if we change the program back to the way it was, then the test succeeds ...

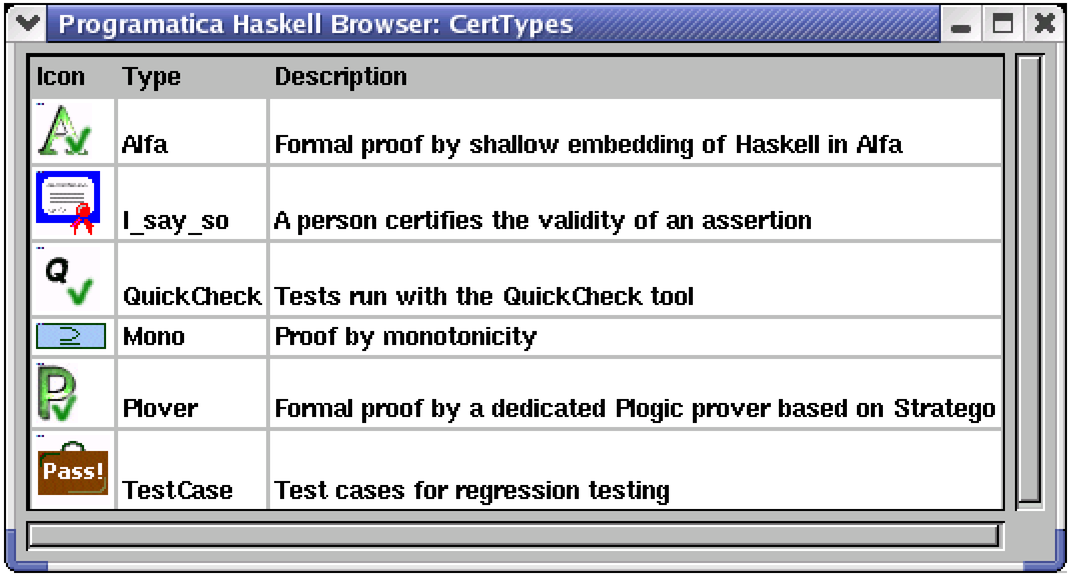


File: Examples.hs

Module: Examples

```
module Examples where
import ChipModel
import Algorithms
import qualified FunFM as FM
import Nat

count :: Alg
count bp rf =
     Write bp cnt $
     Done (FM.update r0 cnt rf)
  where cnt = 1+FM.lookup rf r0


cntChip = chip (const count) initChip



testcase1 = test (cntChip testInput1)

testInput1 = replicate 10 (0,[0,100,500,1000]) :: [Packet]

test tst = putStr . unlines . map show $ tst
```

And the certificate is valid once again!

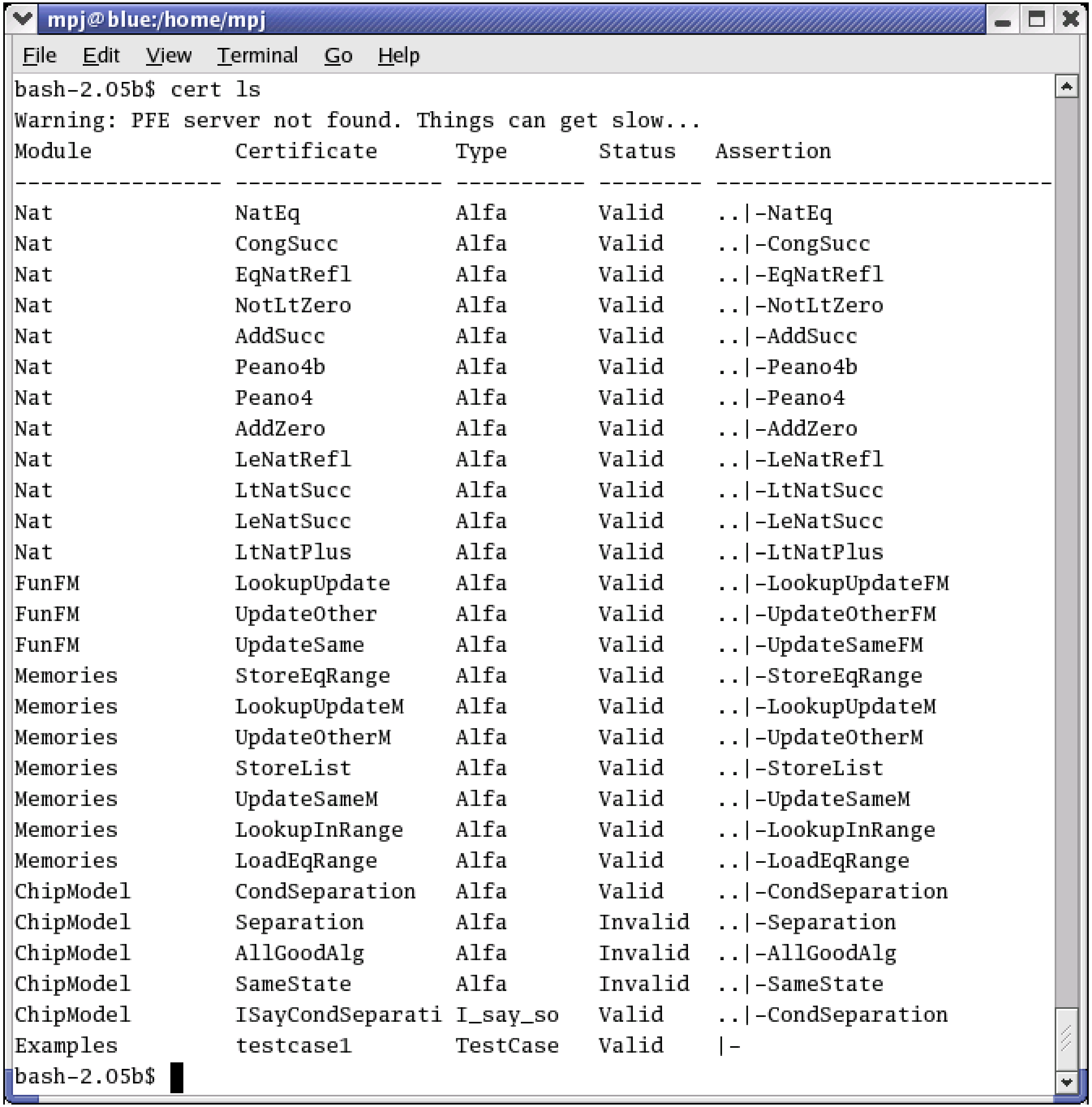


Programatica Haskell Browser: CertTypes

| Icon | Type | Description |
|---|---|---|
| A✓ | Alfa | Formal proof by shallow embedding of Haskell in Alfa |
|  | I_say_so | A person certifies the validity of an assertion |
| Q✓ | QuickCheck | Tests run with the QuickCheck tool |
| ⊒ | Mono | Proof by monotonicity |
| P✓ | Plover | Formal proof by a dedicated Plogic prover based on Stratego |
| Pass! | TestCase | Test cases for regression testing |

What we've seen here looks a lot like the kind of functionality provided by the unit/regression testing tools that are used in extreme programming

Programatica generalizes these ideas so that they can be used with other types of evidence too, including testing, informal assertions, and formal methods ...



```
bash-2.05b$ cert ls
Warning: PFE server not found. Things can get slow...
Module           Certificate      Type       Status   Assertion
---------------- ---------------- ---------- -------- --------------------------
Nat              NatEq            Alfa       Valid    ..|-NatEq
Nat              CongSucc         Alfa       Valid    ..|-CongSucc
Nat              EqNatRefl        Alfa       Valid    ..|-EqNatRefl
Nat              NotLtZero        Alfa       Valid    ..|-NotLtZero
Nat              AddSucc          Alfa       Valid    ..|-AddSucc
Nat              Peano4b          Alfa       Valid    ..|-Peano4b
Nat              Peano4           Alfa       Valid    ..|-Peano4
Nat              AddZero          Alfa       Valid    ..|-AddZero
Nat              LeNatRefl        Alfa       Valid    ..|-LeNatRefl
Nat              LtNatSucc        Alfa       Valid    ..|-LtNatSucc
Nat              LeNatSucc        Alfa       Valid    ..|-LeNatSucc
Nat              LtNatPlus        Alfa       Valid    ..|-LtNatPlus
FunFM            LookupUpdate     Alfa       Valid    ..|-LookupUpdateFM
FunFM            UpdateOther      Alfa       Valid    ..|-UpdateOtherFM
FunFM            UpdateSame       Alfa       Valid    ..|-UpdateSameFM
Memories         StoreEqRange     Alfa       Valid    ..|-StoreEqRange
Memories         LookupUpdateM    Alfa       Valid    ..|-LookupUpdateM
Memories         UpdateOtherM     Alfa       Valid    ..|-UpdateOtherM
Memories         StoreList        Alfa       Valid    ..|-StoreList
Memories         UpdateSameM      Alfa       Valid    ..|-UpdateSameM
Memories         LookupInRange    Alfa       Valid    ..|-LookupInRange
Memories         LoadEqRange      Alfa       Valid    ..|-LoadEqRange
ChipModel        CondSeparation   Alfa       Valid    ..|-CondSeparation
ChipModel        Separation       Alfa       Invalid  ..|-Separation
ChipModel        AllGoodAlg       Alfa       Invalid  ..|-AllGoodAlg
ChipModel        SameState        Alfa       Invalid  ..|-SameState
ChipModel        ISayCondSeparati I_say_so   Valid    ..|-CondSeparation
Examples         testcase1        TestCase   Valid    |-
bash-2.05b$
```

... and Programatica also provides tools to help manage the corresponding collection of evidence throughout the project's lifetime

Ok. So how does this work?

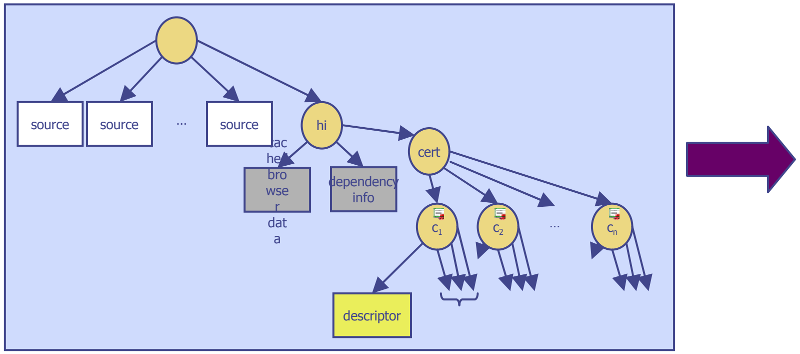
Back to the talk to explain ...

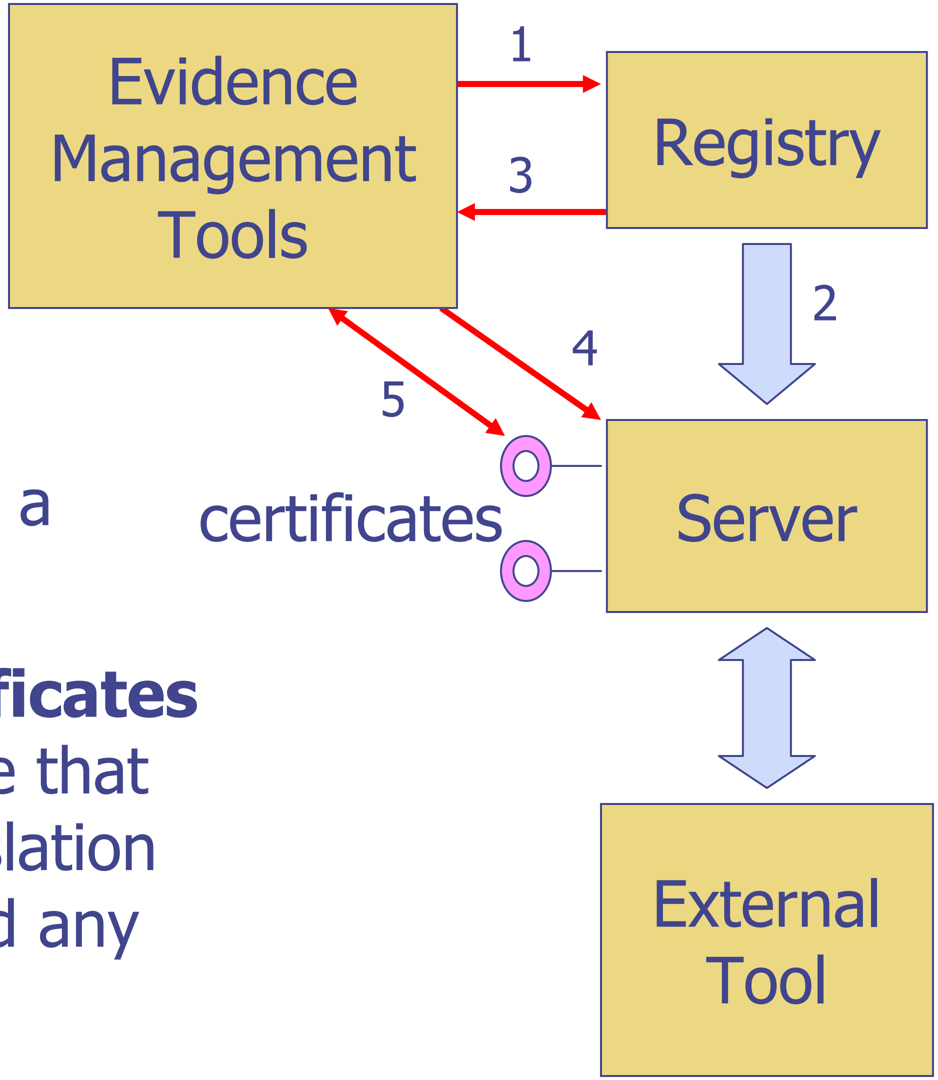
# Programatica Servers:

# Programatica Servers:

- A **server** is a Programatica plugin that knows how to interpret the data in a particular type of certificate
- Key to the extensible architecture described earlier
- Servers present a uniform API for evidence management that is independent of certificate type

# Servers and Certificates:



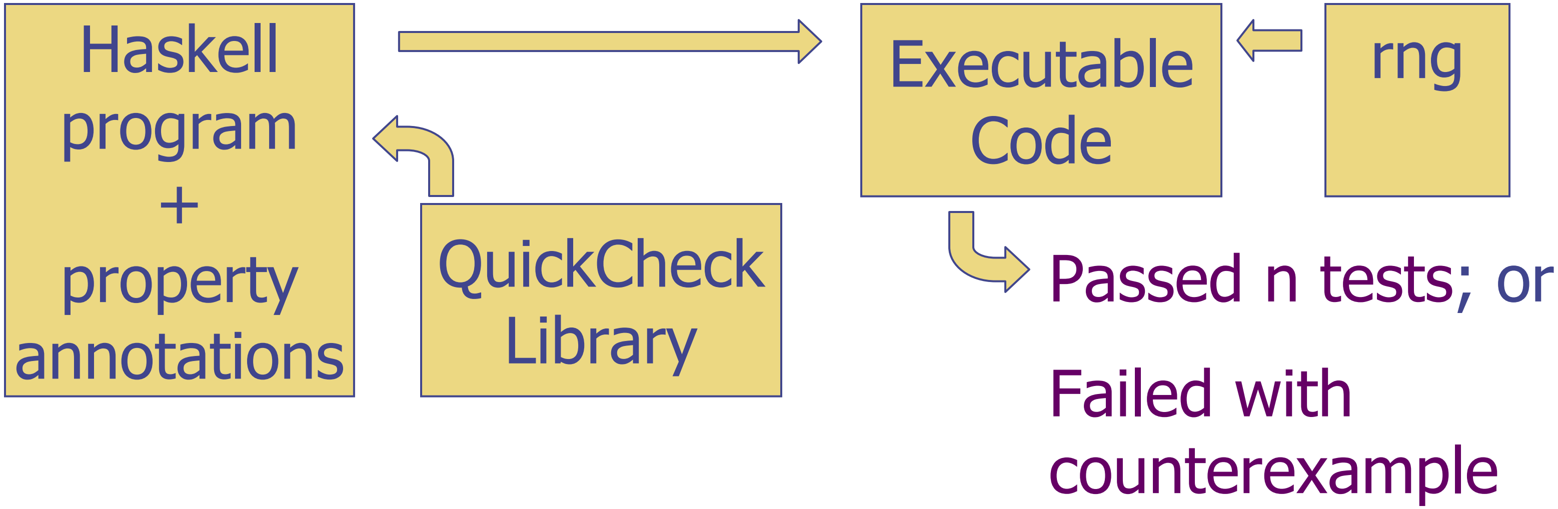


- Use of a **registry** enables a flexible, extensible system
- Use of **servers** and **certificates** permits a generic interface that automates/hides the translation between Programatica and any external tools

# Using QuickCheck:

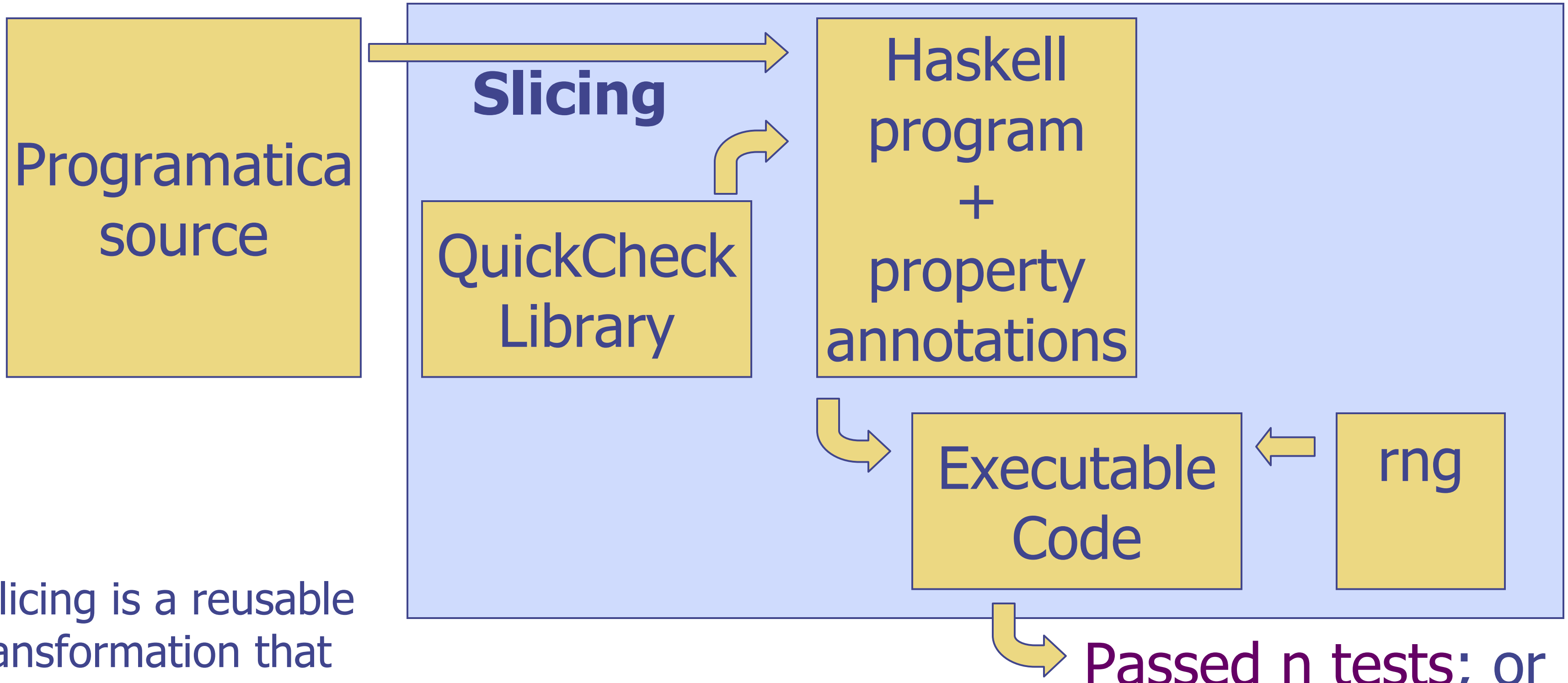
- QuickCheck is an independently developed random testing tool (Hughes and Claessen, Chalmers University, Sweden)

- Haskell developer's perspective:


Haskell program + property annotations
QuickCheck Library
Executable Code
rng
Passed n tests; or
Failed with counterexample

# Using QuickCheck with pfe:

- Programatica implementer's perspective:



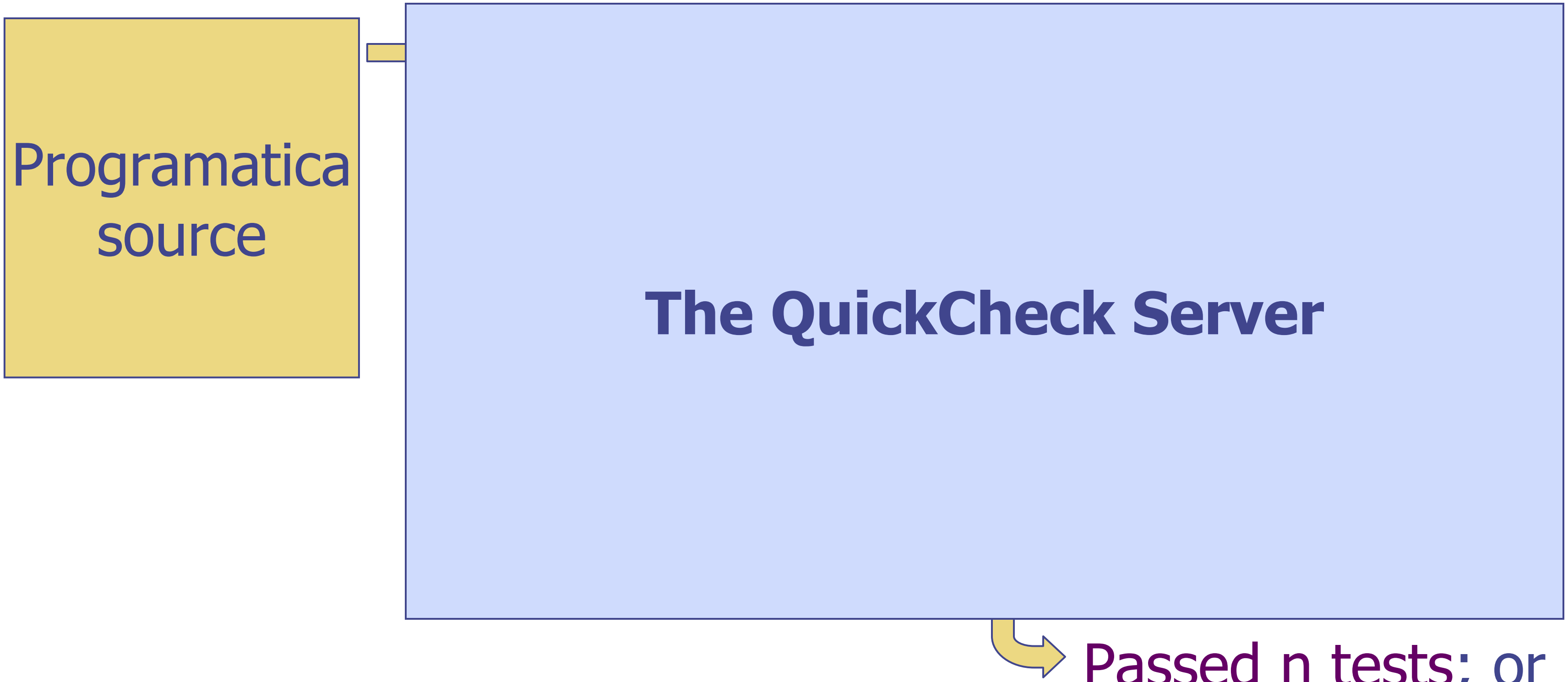
Passed n tests; or

Failed with counterexample

(Slicing is a reusable transformation that reduces the size of the code that is passed to QuickCheck, and eliminates spurious dependencies)

# Using QuickCheck with pfe:

- Programatica user's perspective:



Programatica source

The QuickCheck Server

Passed n tests; or

Failed with counterexample

# Integrating Multiple Servers:

- PFE currently includes servers for:
  - supported assertions ("I say so")
  - individual test cases
  - random testing (QuickCheck)
  - automated theorem proving (Plover)
  - interactive proof editing (Alfa)

- Others planned/in progress include:
  - Isabelle/HOL
  - Internal servers for certificate combination

# Dealing with Change:

- Our model, our specification, or both must be revised to complete the task in hand
- Whatever happens, some of the evidence we have collected may no longer be valid.
- Some evidence can be reconstructed automatically, but some will be quite expensive to reconstruct
- In software development, change is the norm, not the exception, so we need to handle change as efficiently as possible.

# Hashing to Detect Change:

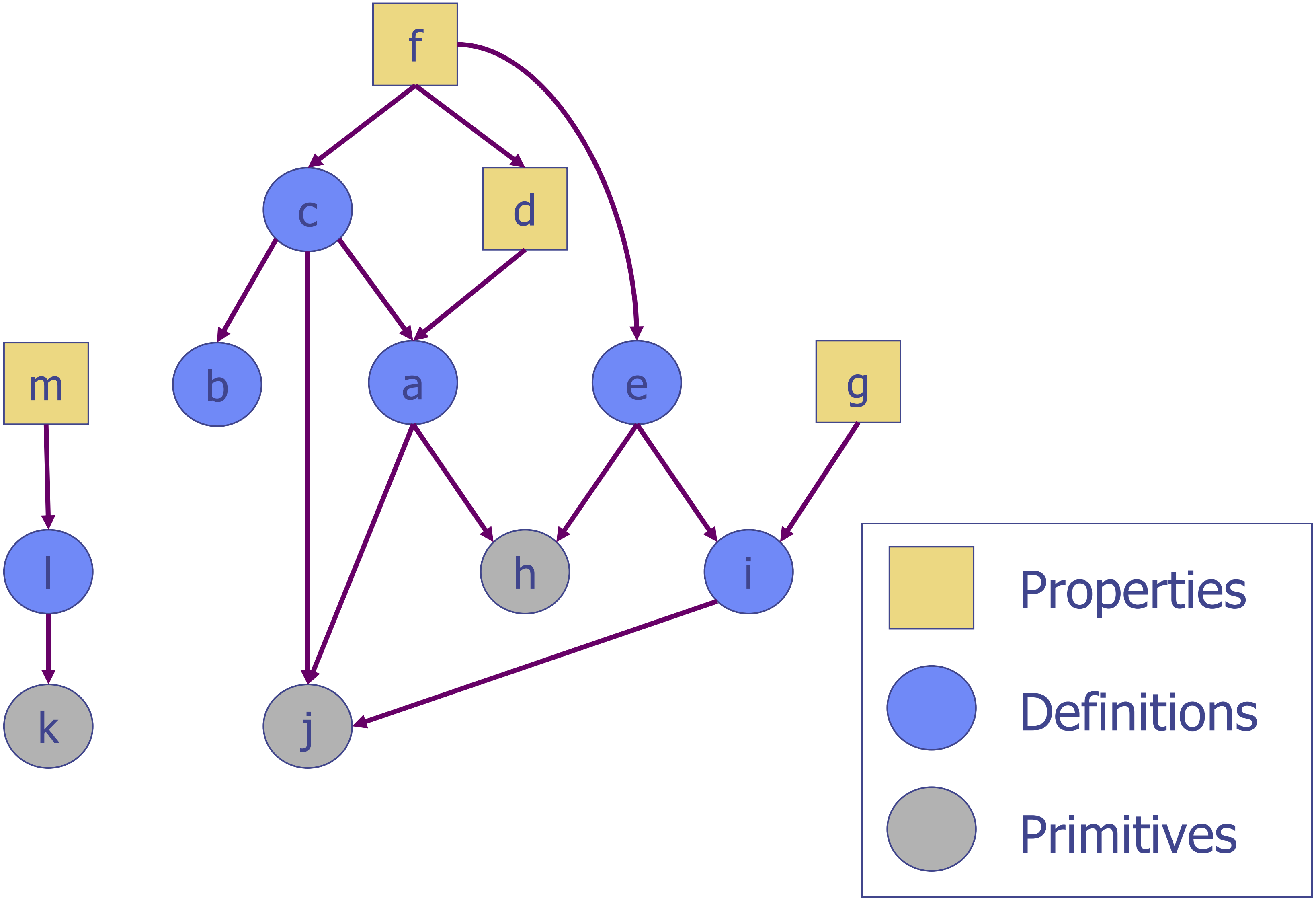
- When we parse a source file, we calculate a cryptographically robust hash over the abstract syntax of each definition

- These hashes are cached within each project:

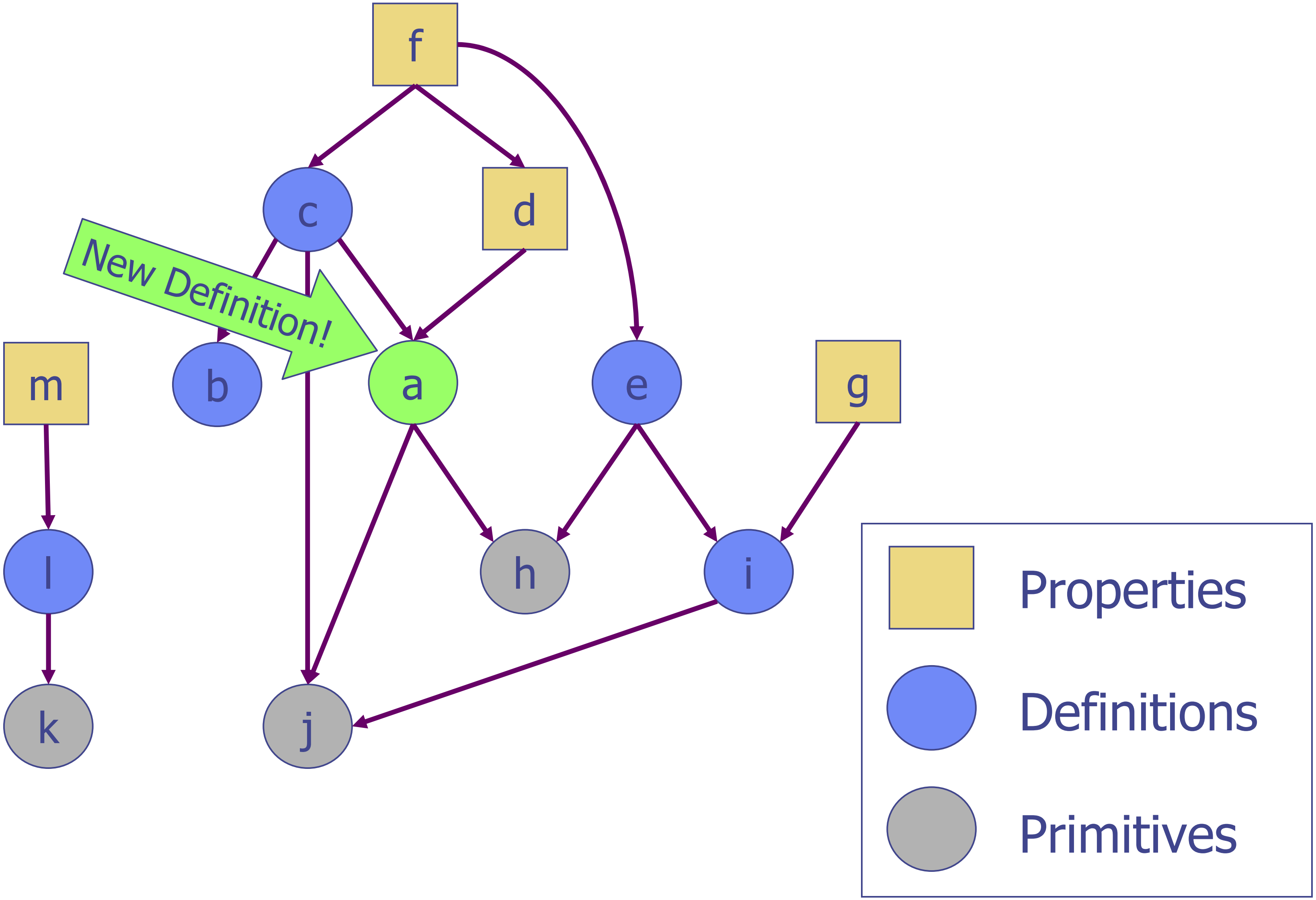
```
0cc175b9c0f1b6a831c399e269772661
92eb5ffee6ae2fec3ad71c777531578f
81a5fe3d544359af13848e6192ece475
445a4ca24e10824e03ef42e2e1d755d9
987dd8f5f1293857dc7932c14c7f3d80
8b3ee2a3933b9c01878bcddc298ff9e2
bb53046df3ef7793ee7c37aec0d090d0
ad797e6f29cf558f7aeb8200563ecd3a
8959f36e873441e58dcc9222777b6d47
84de7ff93b201e8c5b4cf0e006dfe848
7a5acfc765e1875a49daffd8561ae025
```

- If we find a definition whose hash is not listed, then it must be new/modified.

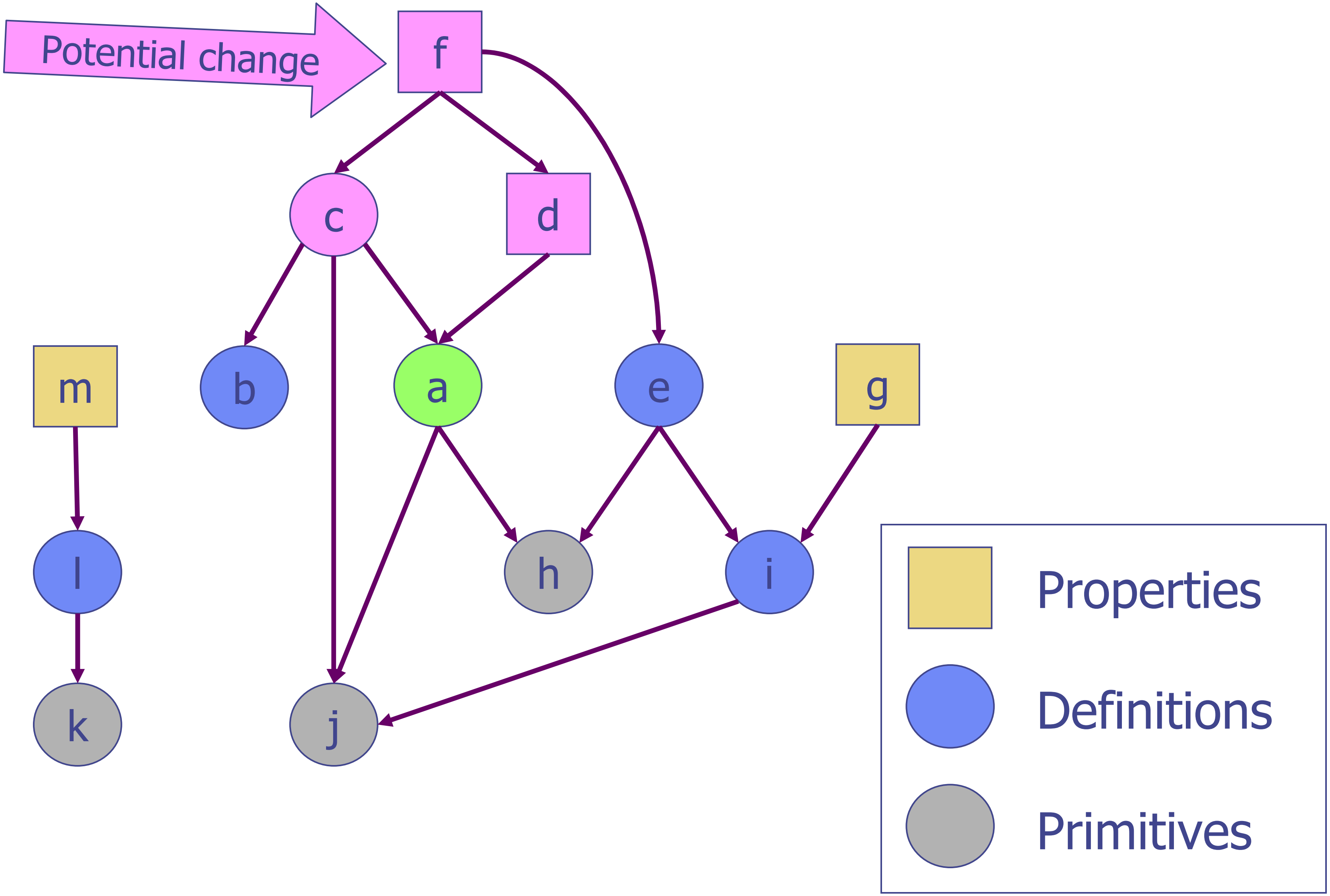
# Using a Dependency Graph:



# Using a Dependency Graph:



# Using a Dependency Graph:



# Benefits of Hashing:

- Fine-grained dependency analysis reduces the cost of reconstructing evidence after the program has been modified

- By hashing over abstract syntax, we do not flag any changes if the source text is reformatted, if comments are changed, etc...

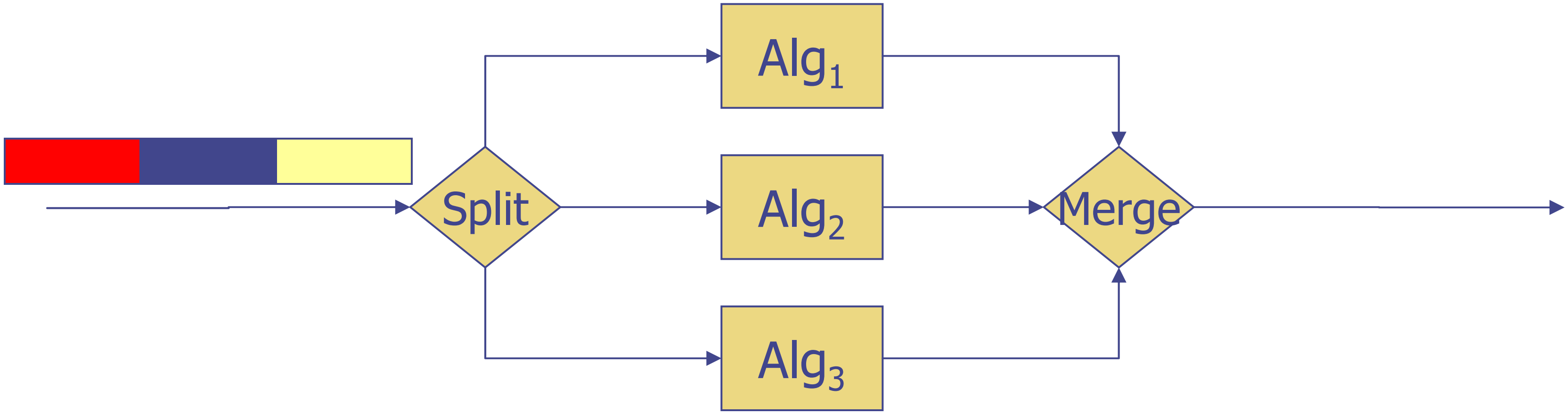
# Re-establishing Validity:

- How do we revalidate an invalid certificate?
- It depends on the type of certificate
- Typical process:
  - Gather relevant data using sequent, dependencies, and abstract syntax
  - Translate to form suitable for external tool
  - Save artifacts in certificate directory
  - Invoke external tool
  - Capture Potentially useful feedback
- This could be a lot more expensive ...
- ... but we hope it will be a lot less frequent

# Separation:

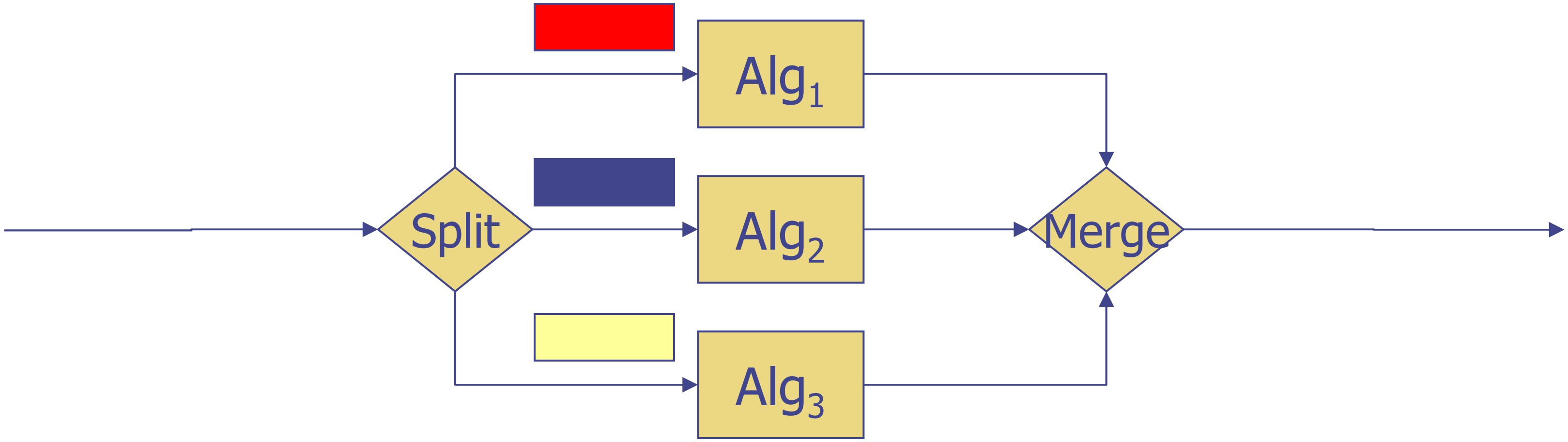
# Separation:

- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



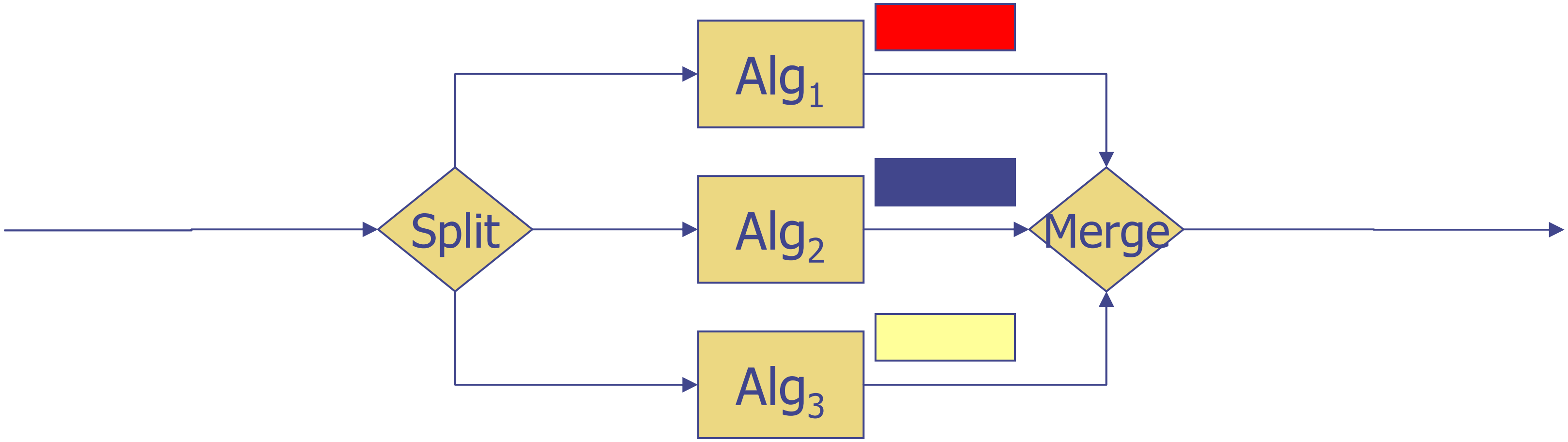
# Separation:

- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



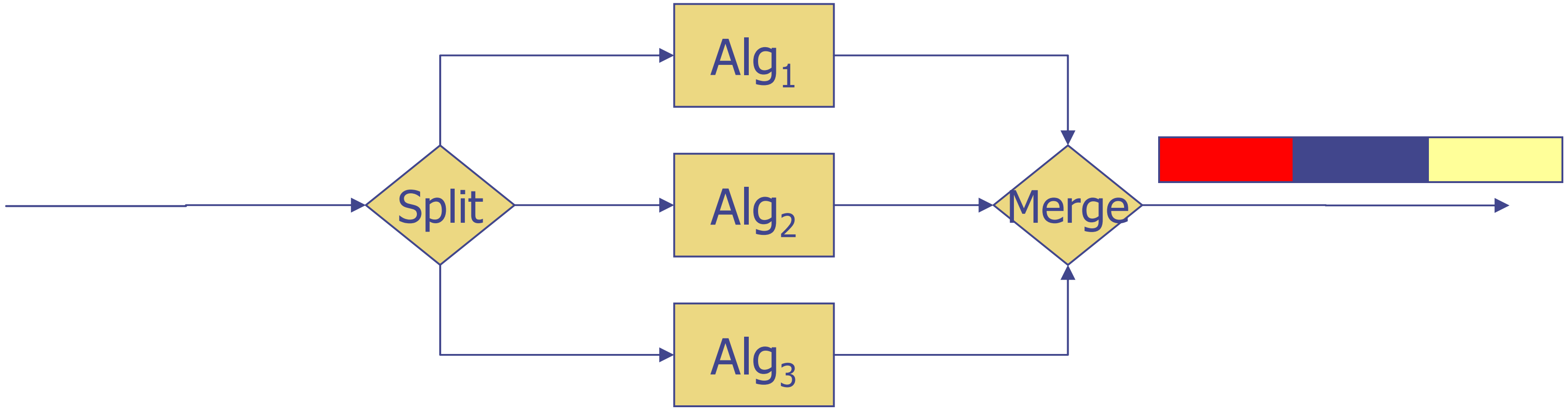
# Separation:

- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



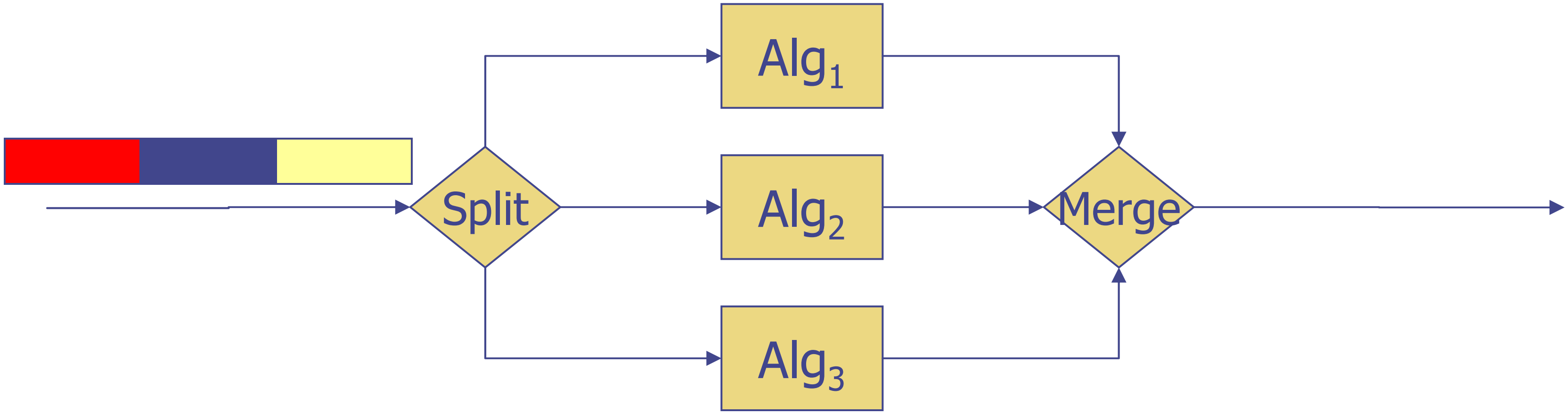
# Separation:

- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



# Separation:

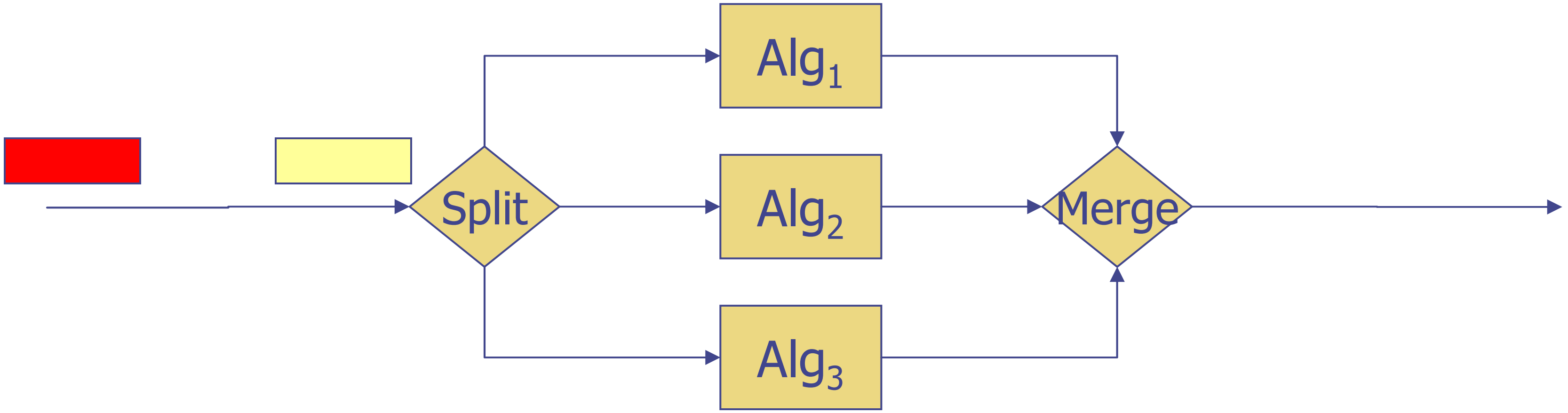
- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



- If we filter out blue packets before they reach the chip …

# Separation:

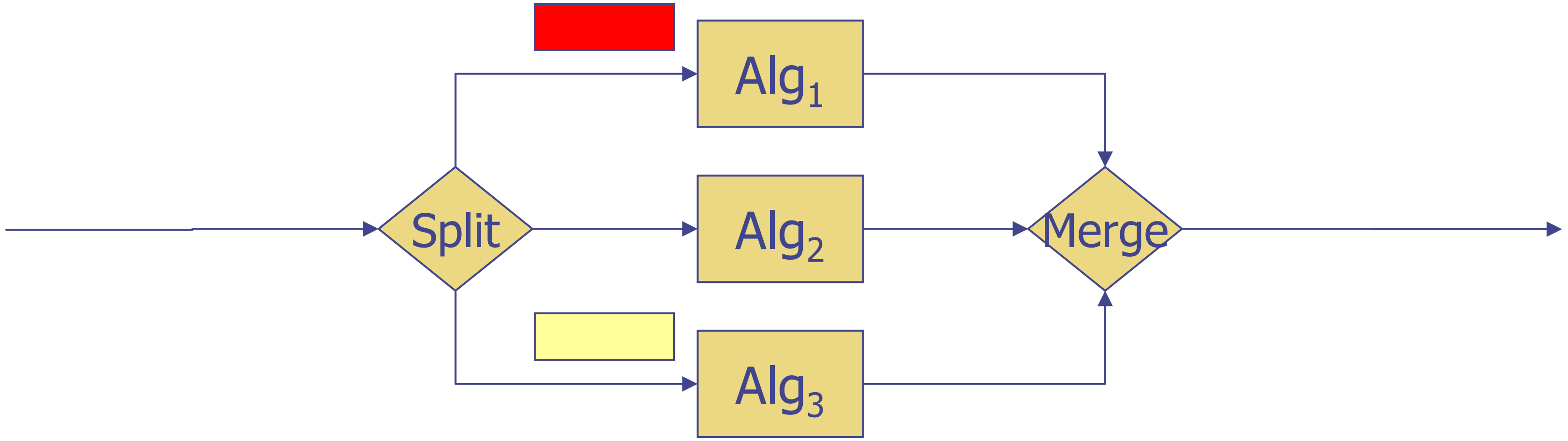
- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



- If we filter out blue packets before they reach the chip …

# Separation:

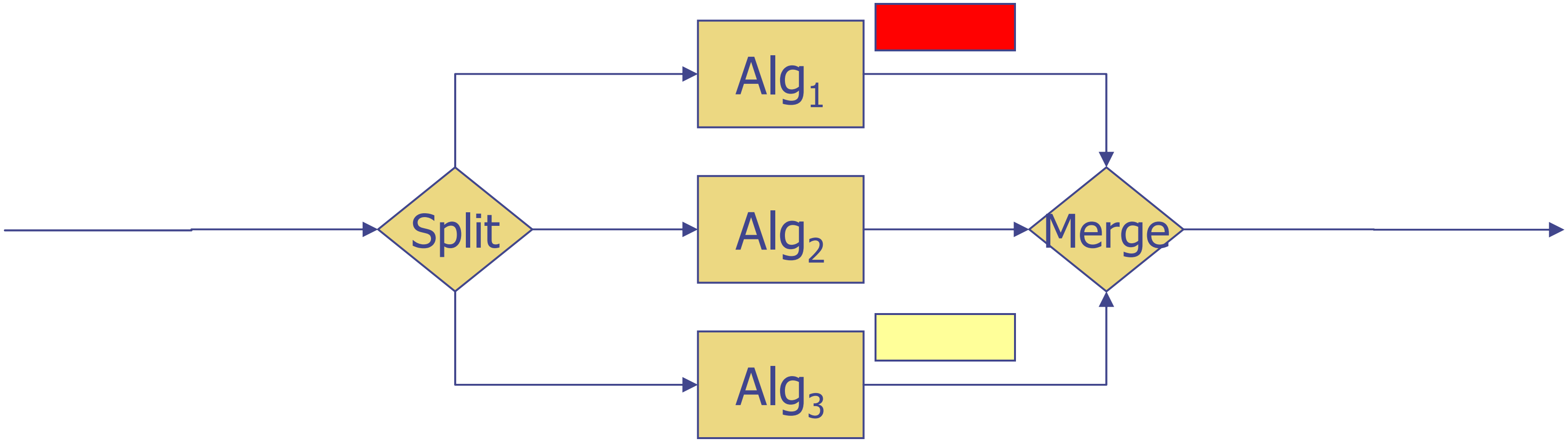
- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



- ... the remaining packets should flow through as before and produce the same outputs ...

# Separation:

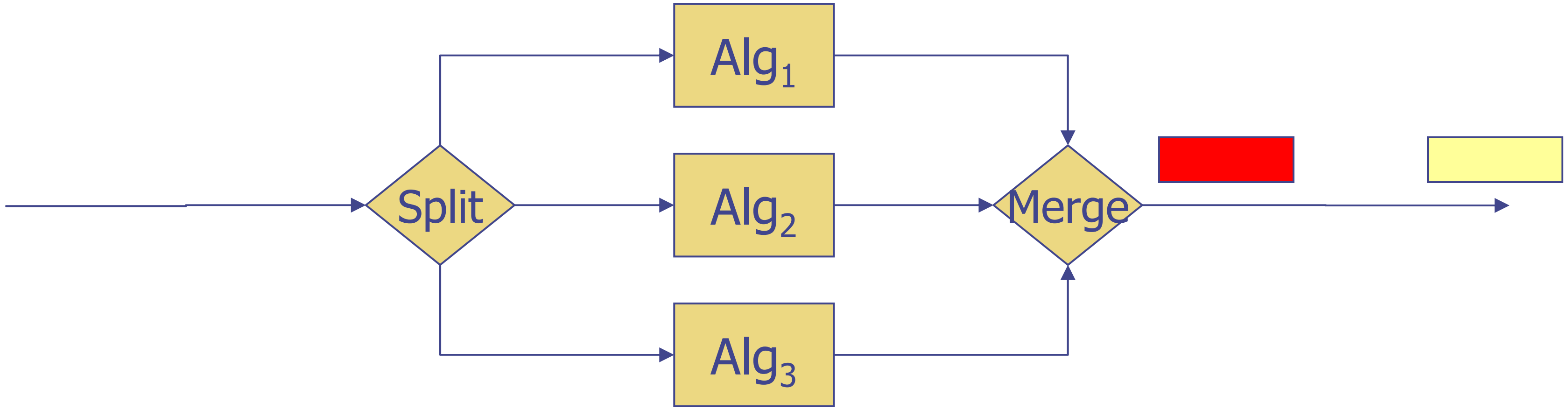
- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



- ... the remaining packets should flow through as before and produce the same outputs ...

# Separation:

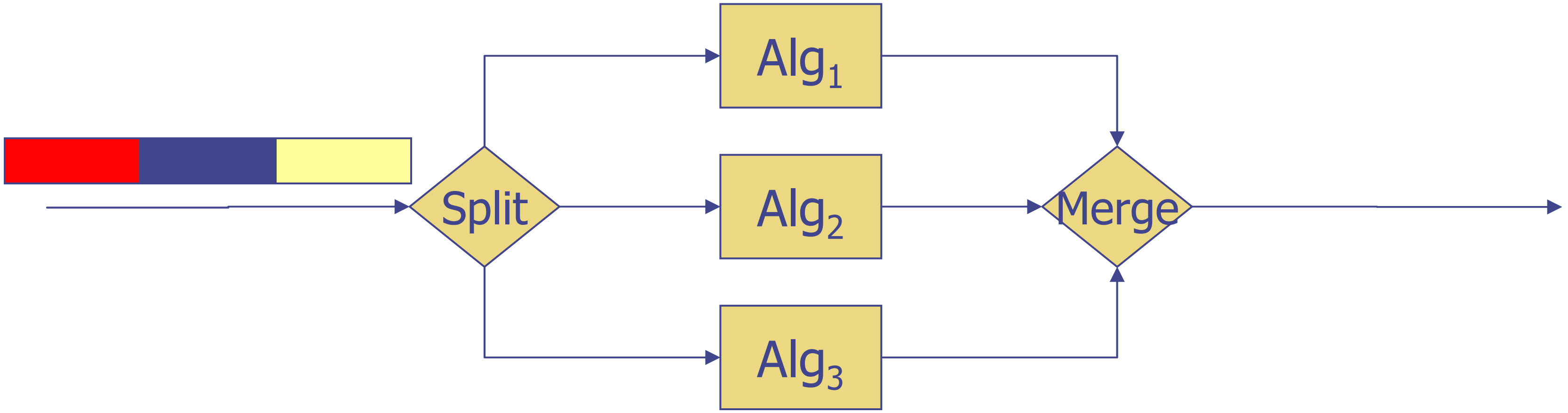
- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



- ... the remaining packets should flow through as before and produce the same outputs ...

# Separation:

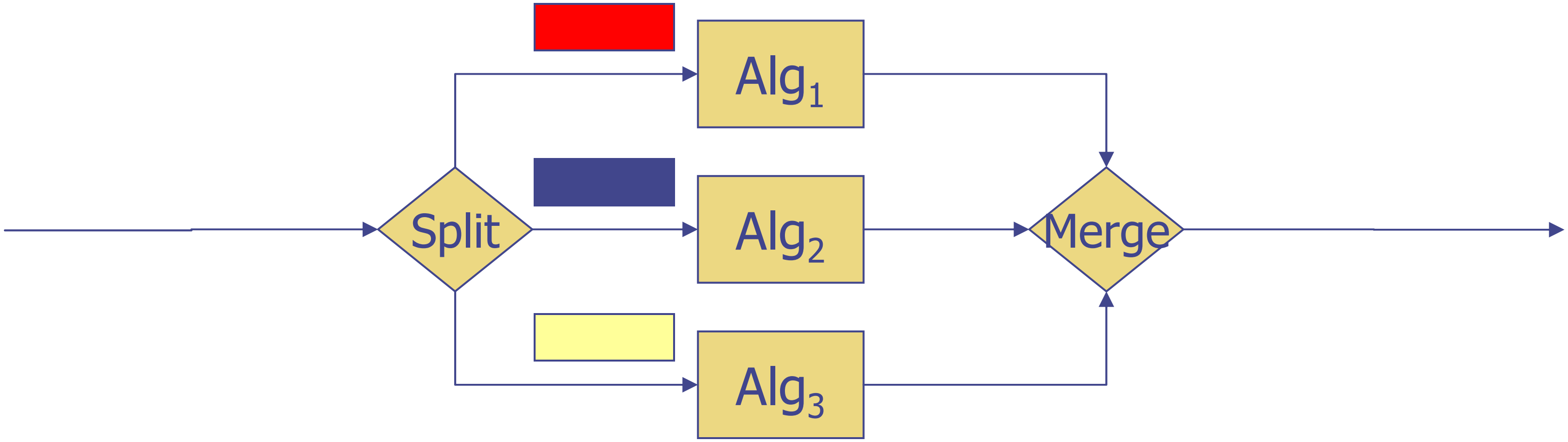
- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



- Or we could let all of the packets through the chip …

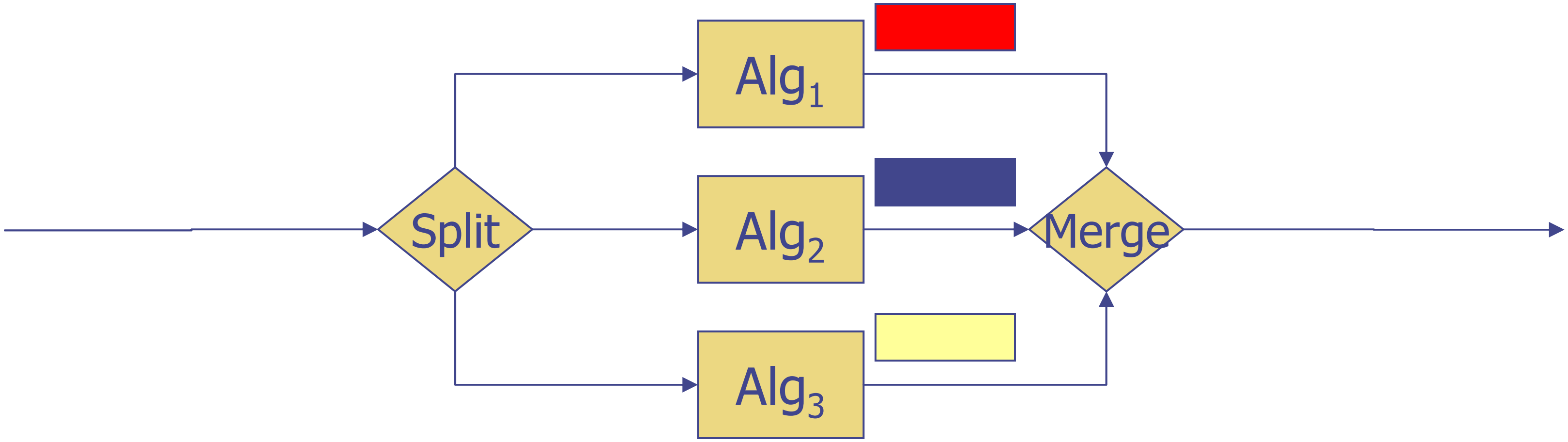
# Separation:

- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



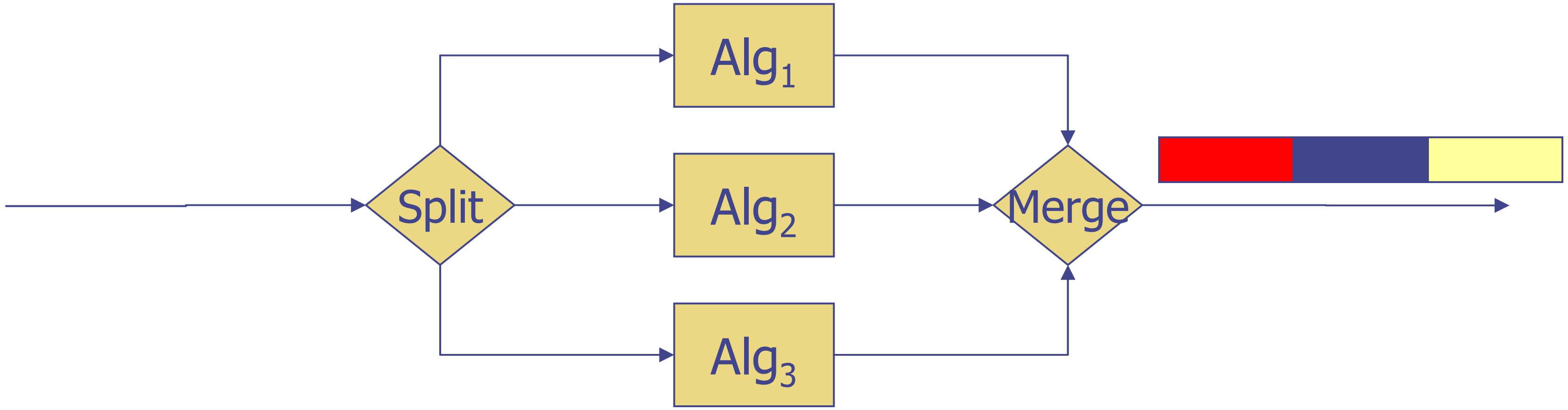
# Separation:

- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



# Separation:

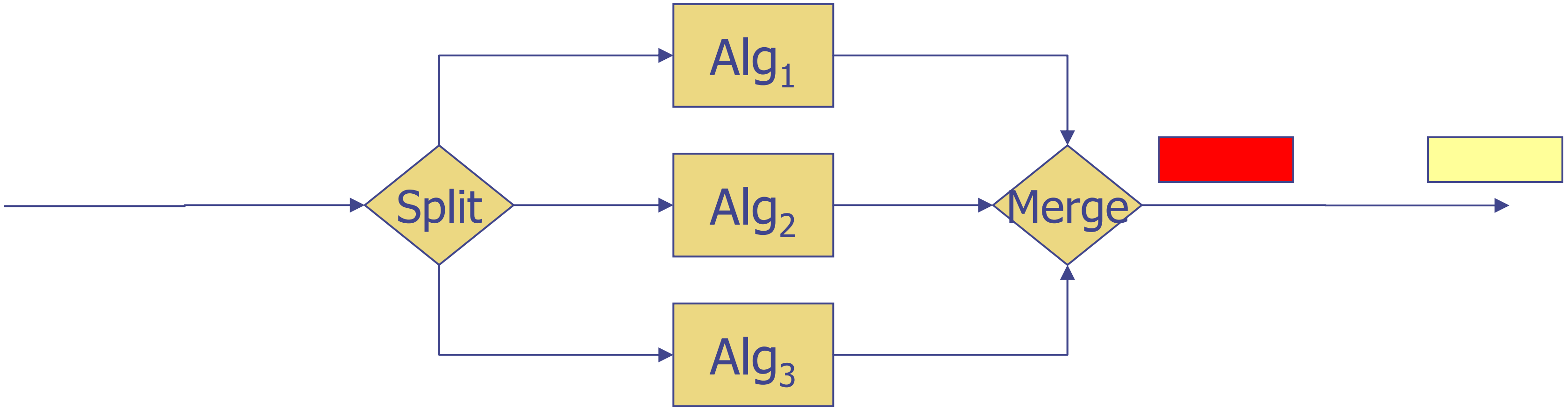
- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



- ... and only then discard the blue packets ...

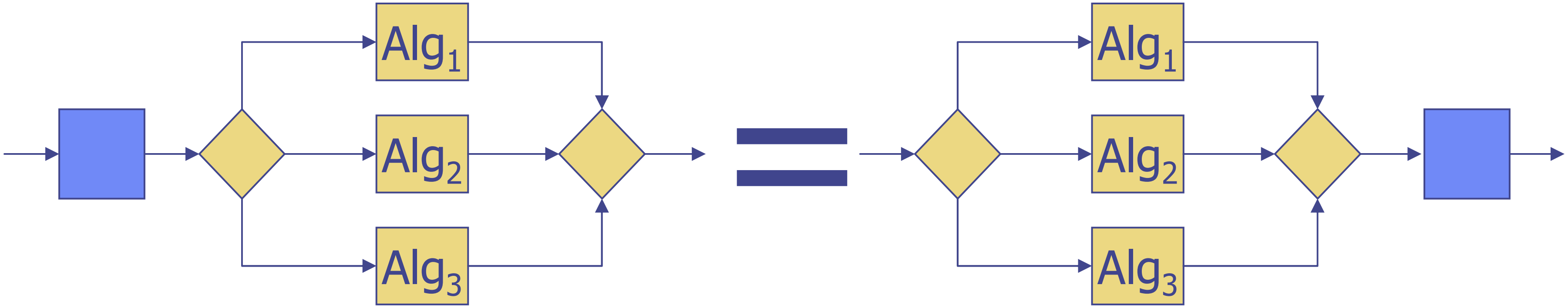
# Separation:

- Packets are labeled for different channels
- The behavior on one channel should not affect the behavior on any other channel



- The final result should be the same: yellow and red are independent of blue

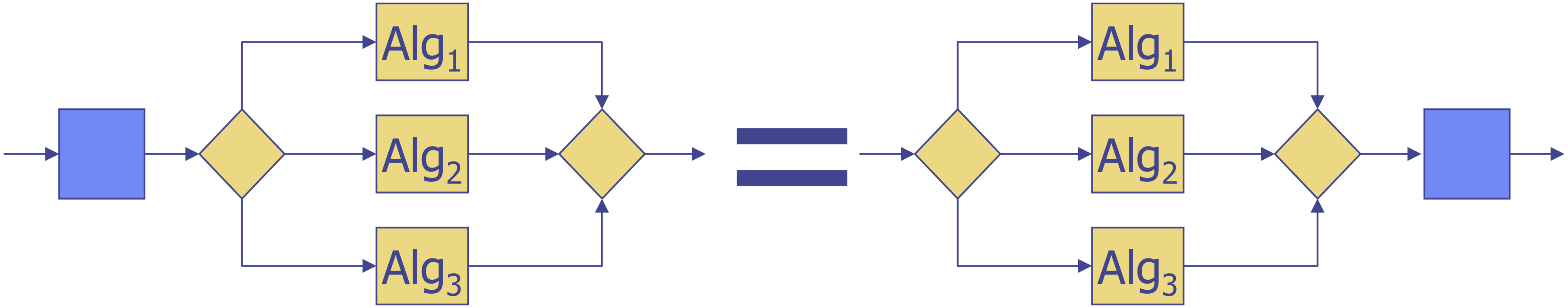
# The Separation Property:



This law guarantees that:

- Outputs do not depend on inputs to other channels.
- Channels do not generate spurious outputs.

# The Separation Property:



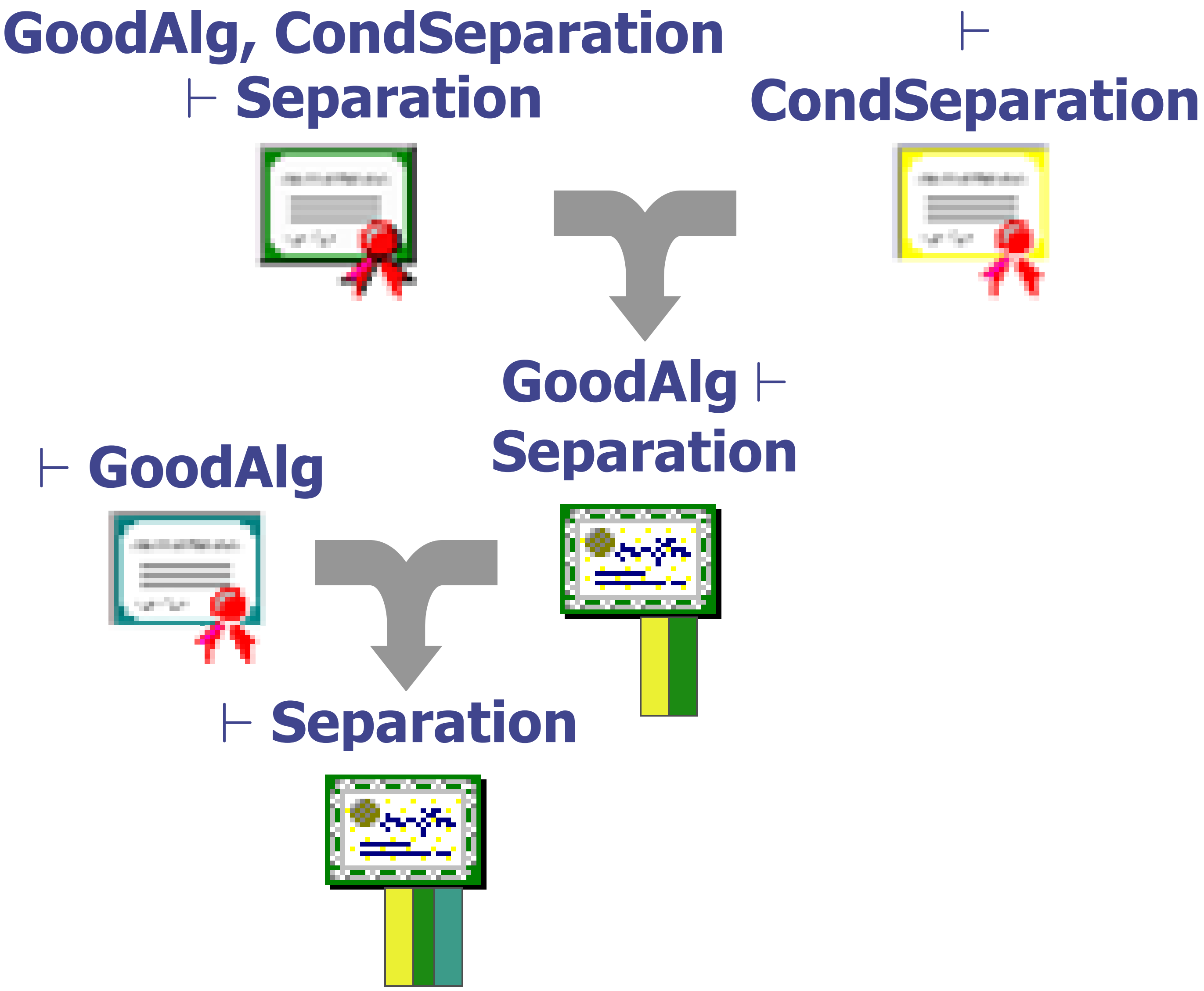
```
assert Separation =
 All algs :: Algs.
  All select :: (ChannelId → Bool).
    { filter (select . fst) . chip algs }
         ===
    { chip algs . filter (select . fst) }
```

# Validation and Combination:

We want to validate and combine evidence from different sources:

- Certificates carry **sequents** "Assume $\vdash$ Conclude" that act as an interface/contract between Programatica and any external tools.
- Servers for external tools are used to test **validity** (i.e., to check that a certificate's sequent is consistent with its evidence)
- Built-in servers use sequents of existing certificates to guide the construction of new, composite certificates.

# Combining Evidence:



GoodAlg, CondSeparation ⊢ Separation

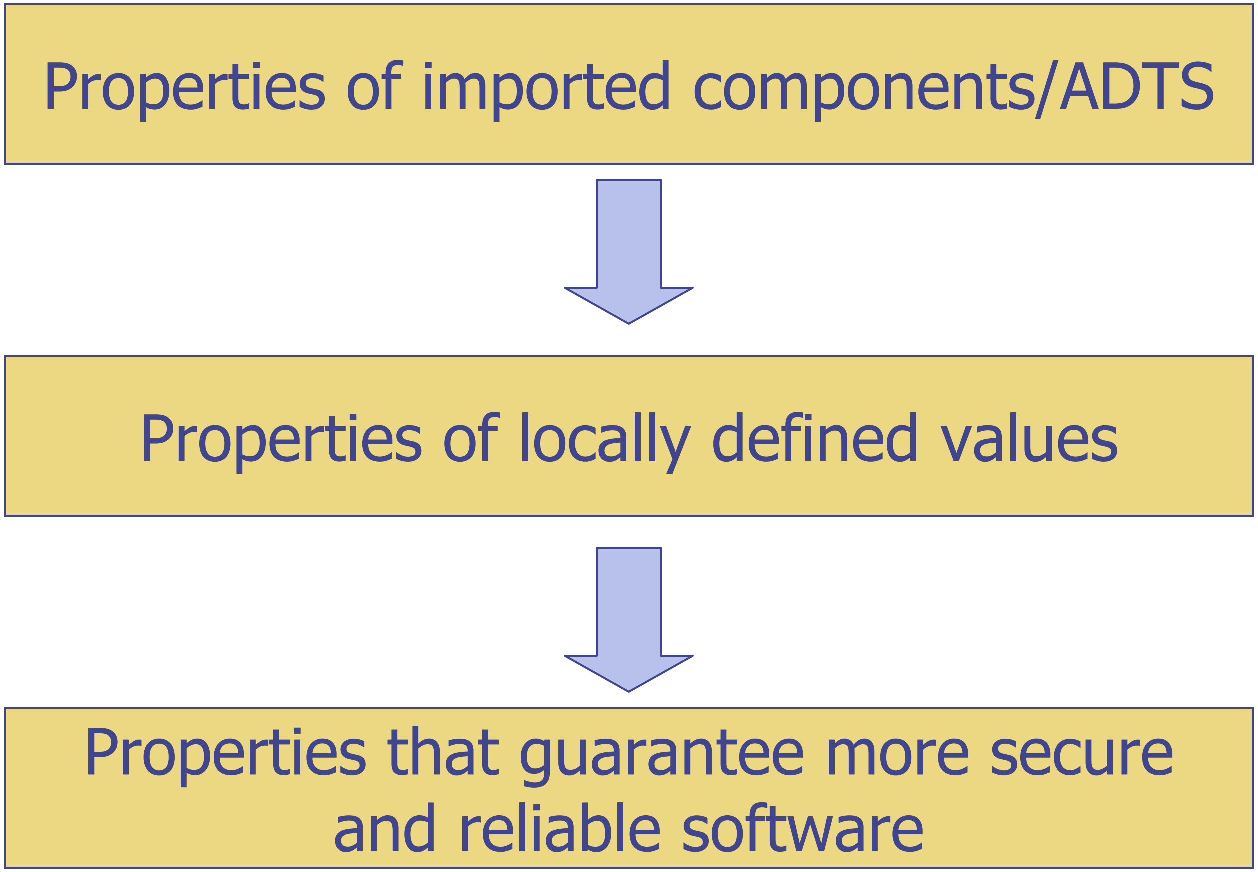
⊢ CondSeparation

GoodAlg ⊢ Separation

⊢ GoodAlg

⊢ Separation

# Property propagation:



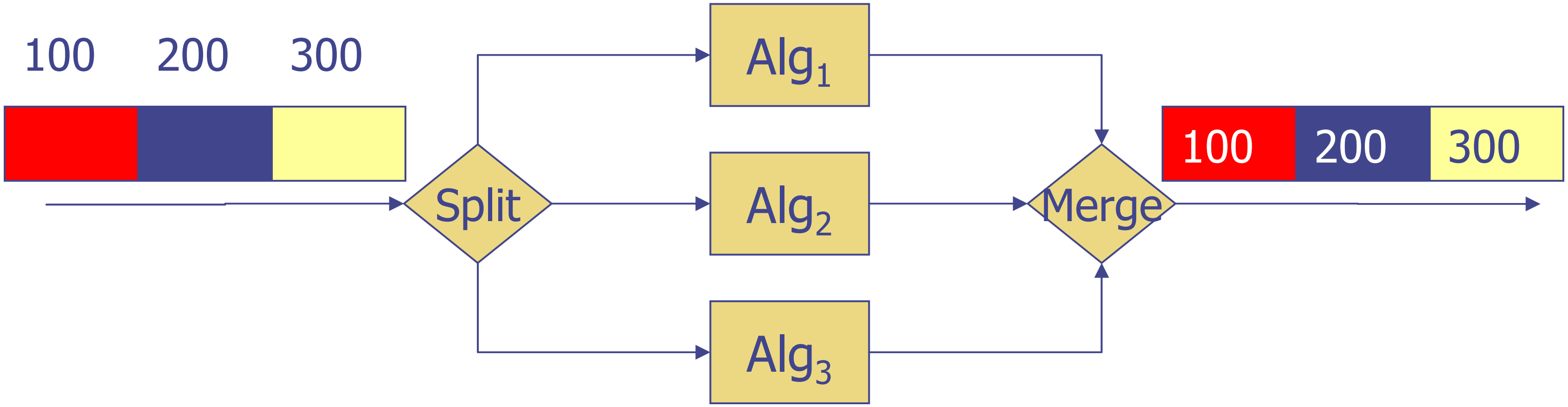
Properties of imported components/ADTS

Properties of locally defined values

Properties that guarantee more secure and reliable software

# Separation Fails:

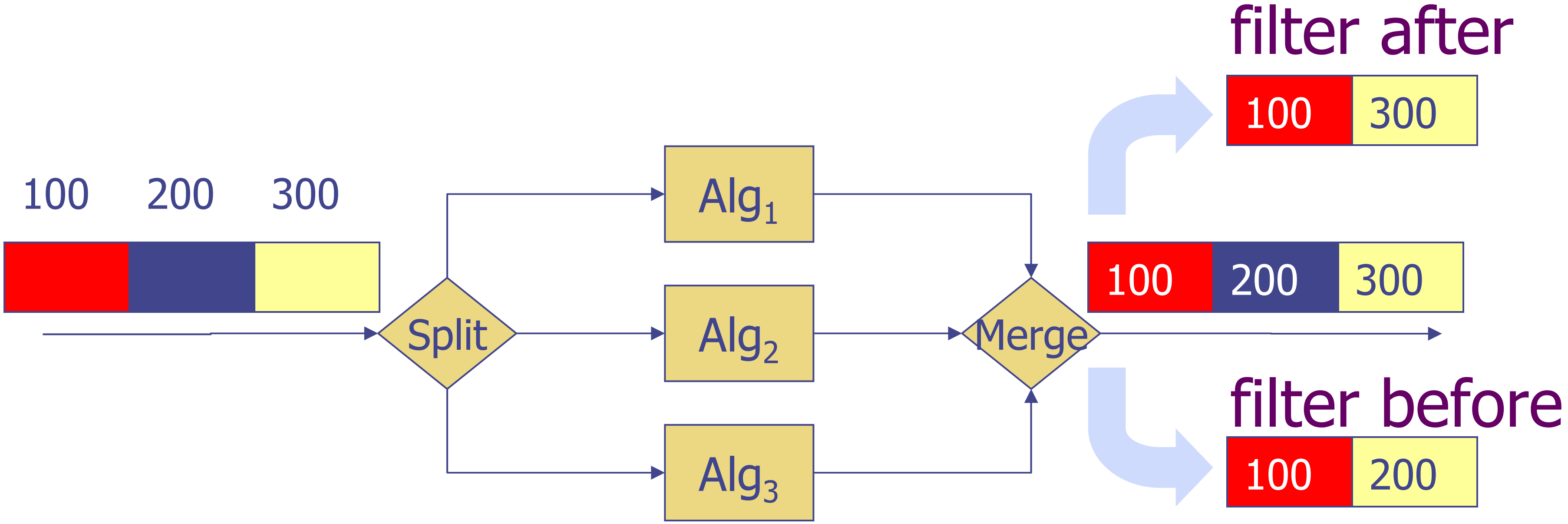
- Packets are written into shared memory
- Absolute addresses of packets are passed to lower engine algorithms …



- … what if an algorithm writes the absolute address into its output?

# Separation Fails:

- Packets are written into shared memory
- Absolute addresses of packets are passed to lower engine algorithms ...



- ... what if an algorithm writes the absolute address into its output?

# Separation Restored!

This is a violation of the separation property!

Our analysis leads us to raise several questions:

- Is it a bug in the code or the specification?
- Is it a security problem (a covert channel)?
- How can it be fixed?
    - Fixing packet start addressing
    - Relative addressing
    - Fixed address
    - ...

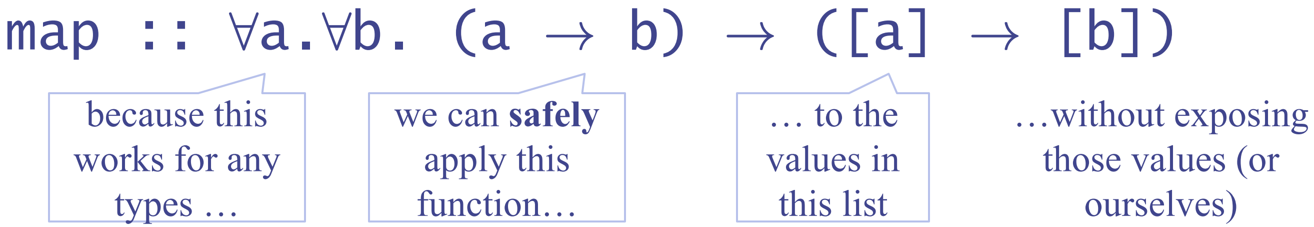
The method provides important feedback for the designer/developer to discuss and then address ...

# Why Haskell?

# Why Haskell?

- **Purity**: the result of a function, depends only on the argument value (i.e., no hidden dependencies)

- **Polymorphic Types**: powerful and expressive; parametricity provides "theorems for free":



```
map :: ∀a.∀b. (a → b) → ([a] → [b])
```

because this works for any types …

we can **safely** apply this function…

… to the values in this list

…without exposing those values (or ourselves)

- **Formal semantics**: a foundation for meaningful assurance guarantees

# Why Haskell?  The Big Win:

**Monads**

Modular, scalable encapsulation and reasoning about effects

# What are Effects?

- Standard examples: State, I/O, Exceptions, ...
- Why are they a concern?
  - Interactions between effects can lead to unexpected behavior, nasty bugs, and compromised security
- How do programmers tackle these challenges? How do programming languages help them?
  - some specific examples
  - generalized by monads

# Exceptions in Java:

```
void method(int x) {
    ...
    throw Exception("File not found");
    ...
}
```

- a method must declare any exceptions that it throws

# Exceptions in Java:

```
void method(int x) throws Exception {
    ...
    throw Exception("File not found");
    ...
}
```

✓

- the platform (compiler, verifier, VM) ensures that programmers follow this particular discipline.

# Hidden State in Java:

```
class SecureProcess {
    private byte[] key;
    ...
}
```

- modifiers control access to portions of state
- the platform enforces these restriction

# Exposing Hidden State in Java:

```
class SecureProcess {
    private byte[] pubkey;
    ...
    public byte[] getPubkey() {
      return pubkey;
    }
}
```

provides both read **and** write access!

- ... but a careless programmer might open the gates
- and nothing in the platform will prevent this ...

# Abstract Datatypes (ADTs):

```
interface Stack {
    void push(int value);
    int  pop();
    ...
}
```

- interface constrains allowed operations
- compiler enforces correct use
- reuse + managed cost of certification

- In these examples:
  - the platform checks/guarantees some properties
  - others are assured only by careful, insightful programming

- Summary:
  - ad-hoc mechanisms
  - patchy coverage
  - limited extensibility
  - ultimate reliance on disciplined programming

# Monads: ADTs for computations

- monads provide a uniform and general way to  encapsulate and control the scope of effects
- the type system tracks & enforces correct usage
- the platform guarantees safety
- a general & extensible framework:
  - handles state, exceptions, I/O, concurrency, ...
  - new, user definable monads
  - modular construction and separation using monad transformers

# "Mostly Types, a Little Theorem Proving"

- The chip model (and separation proof) abstracts away from specifics of any instruction set
  - Algorithms described at a high-level in terms of their use of memory
- Specific instruction sets can be modeled on top of this framework
  - Separation follows "for free" by type checking

# "Mostly Types, a Little Theorem Proving"

- Example: We have built a simple instruction set model in 146 lines of Haskell code that allows us to write packet processing algorithms like the following:

```
sumPacket   = loadI 0 r1    -- read size value into r1
            $ loadC 1 r0    -- set pointer to start of data
            $ loadC 0 r2    -- initialize running total
            $ jmp loop
loop        = jzero r1 done
            $ load r0 r3    -- read value from packet
            $ add r3 r2 r2  -- add to running total
            $ incr r0       -- move to next packet location
            $ decr r1
            $ jmp loop
done        = storeI r2 0   -- save result at start of packet
            $ ret
```

# Separating Separation

Based on our experience with Osker:

- Separation can be achieved for complex APIs
  - Mostly through types

- Separation can be separated from the API
  - Assurance of separation independent from the API

- Separation can be encapsulated using monads and monad transformers

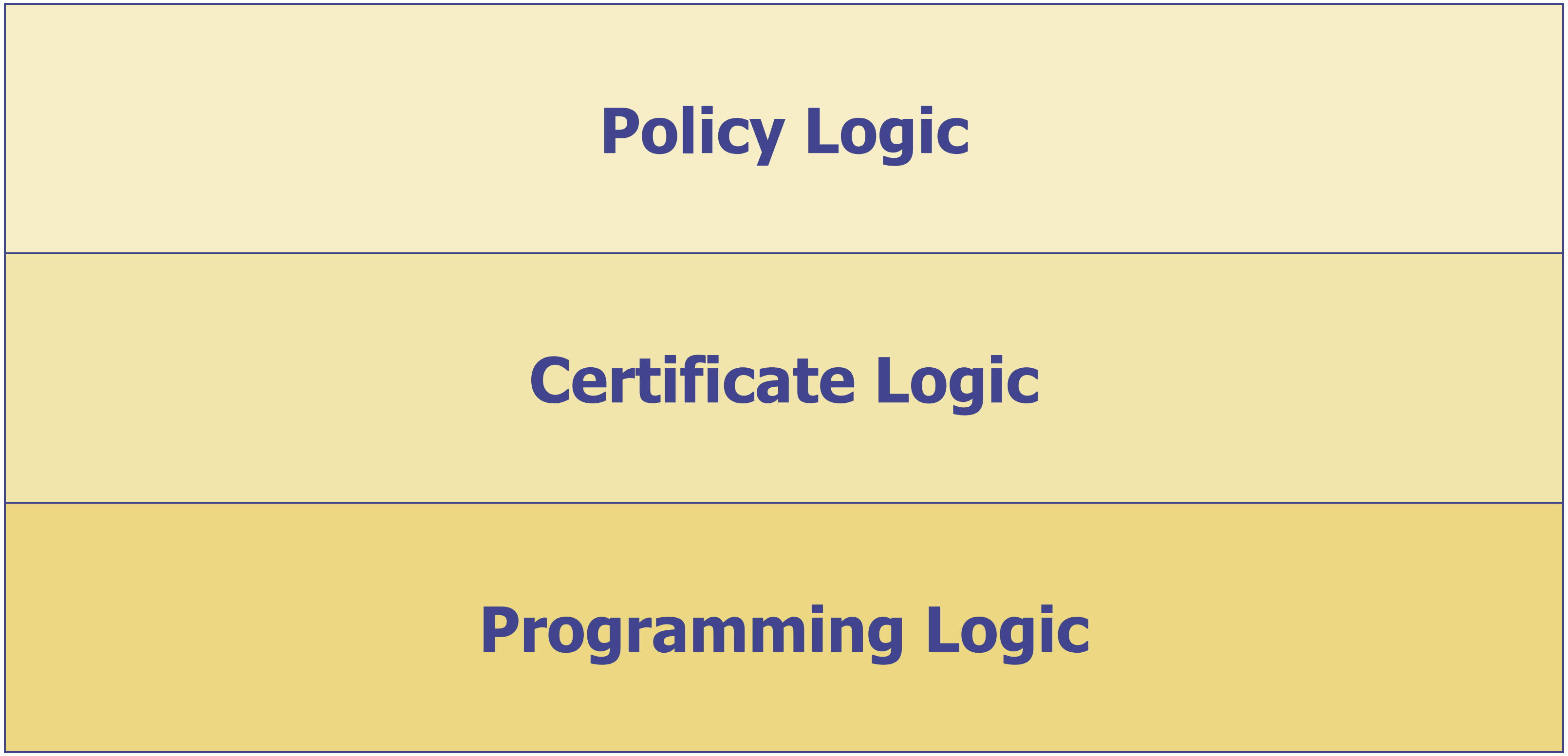
# Alternatives to Haskell?

- Purity, polymorphic type system, and support for monads play critical roles in our current use of Programatica
  - "Mostly types, a little theorem proving"
  - "Separating separation"
- They are not necessarily unique to Haskell

# Alternatives to Haskell?

- The Programatica certificate abstraction and our architecture for evidence management seem to be language independent
  - More precisely, languages and logics can be seen as parameters.
  - Our current implementation does not yet reflect this.
- Programatica for Domain-Specific languages?
- Programatica for general purpose languages?

# Multiple Logics:



Policy Logic

Certificate Logic

Programming Logic

# Key points:

- Building on powerful rapid prototyping platform that has been used for problems of engineering significance

- Logic directly connected to programming language

- Certificate management:
  - tracks dependencies and validity
  - integrates evidence from many external sources

- Formal methods and high-assurance within the context/chaos of standard software development processes

## For more information:

http://www.cse.ogi.edu/pacsoft/projects/programatica/