DOTA2024:3
Defense of the Ancients
Third topic - Complex systems

Hugh Anderson

National University of Singapore
School of Computing

July, 2024

A warning...

Outline

1. **Overview - the scale of the problem**
   - Motivation
   - Application architectures
2. **Fighting back**
   - Design principles and standards
   - Security models - confinement, BLP, BIBA, Chinese wall
   - Formal Methods

Outline

1. **Overview - the scale of the problem**
   - Motivation
   - Application architectures

2. **Fighting back**
   - Design principles and standards
   - Security models - confinement, BLP, BIBA, Chinese wall
   - Formal Methods

Outline

1. **Overview - the scale of the problem**
   - Motivation
   - Application architectures

2. **Fighting back**
   - Design principles and standards
   - Security models - confinement, BLP, BIBA, Chinese wall
   - Formal Methods

Do we really need any motivation???

Software is considered less reliable. Two warranties:

- PC Manufacturer warrants that (a) the SOFTWARE will perform substantially in accordance with the accompanying written materials for a period of ninety (90) days from the date of receipt, and (b) any Microsoft hardware accompanying the SOFTWARE will be free from defects in materials and workmanship under normal use and service for a period of one (1) year from the date of receipt.
- ACCTON warrants to the original owner that the product delivered in this package will be free from defects in material and workmanship for the lifetime of the product.

from RISKS...

- The LA counties pension fund lost US$1,200,000,000 through programming error.
- A Mercedes 500SE with graceful no-skid brake computers left 200m skid marks. A passenger was killed.
- A computer controlled elevator door in Ottawa killed two people.
- An automated toilet seat in Paris killed a child.
- The Amsterdam air-freight computer software crashed, killing giraffes.

Do we really need any motivation???

Abstraction and software engineering...

Consider these two approaches to checking system behaviour:

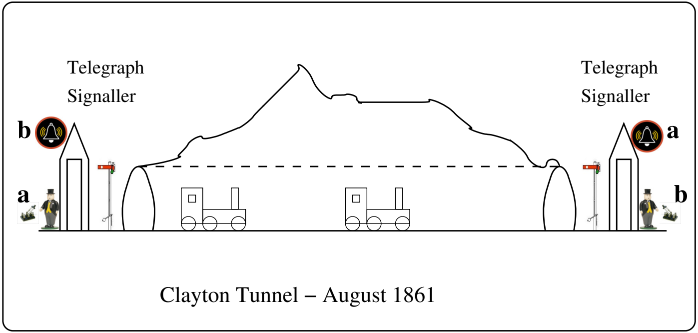
1. *"model it using a small C program"*, *"Run it a few times and see what happens"*, or perhaps *"Start with a file with one record of each type, then try a bigger file until a pattern emerges"*.
2. Turn to mathematics for help.

When software engineers meet a problem that is too large or difficult to understand, they sometimes have a poor attitude, choosing (1) above instead of more serious engineering techniques.

A central issue with IT security is the complexity of modern systems, and our inability to correctly reason about, or even enumerate, the behaviour of modern software systems. When we build a bridge, in general, using more bricks makes the bridge more stable. The same cannot be said for software systems. ..

Multiple concurrent activities...

Concurrency can be hard:

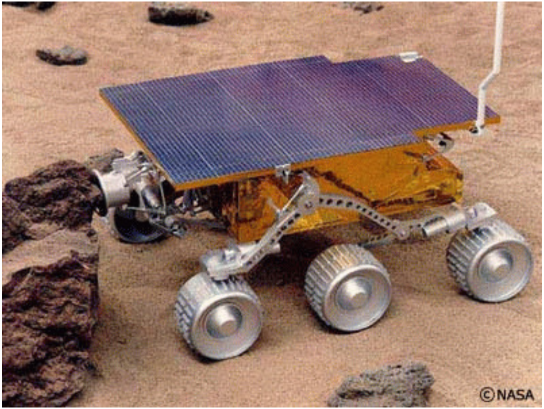


Signallers, flags, bells, train drivers, trains, ... and a tunnel.

The Clayton tunnel disaster. 21 years of faultless operation of a bad protocol.

Mars pathfinder mission in 1997

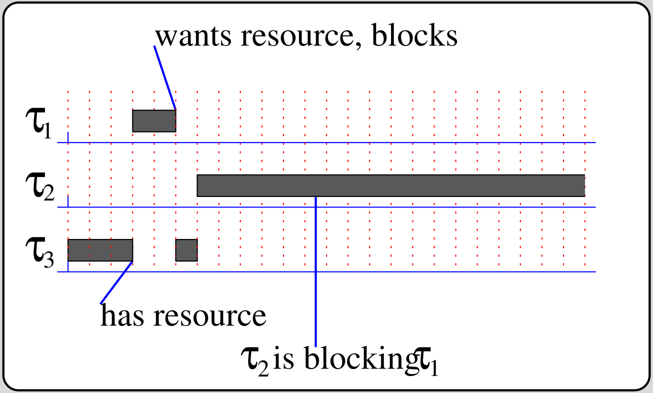
Ran into serious problems:



The spacecraft began experiencing total system resets with loss of data each time dues to priority inversion.

Priority inversion scenario

Three prioritized tasks



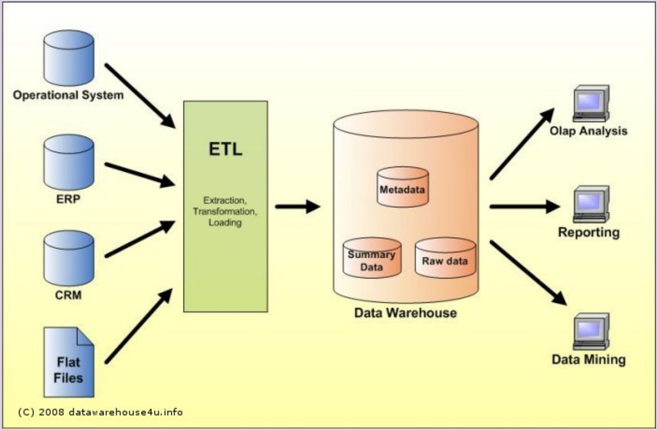
Higher priority task τ_1 blocked by the much lower priority task that is holding a shared resource.
The lower priority task τ_3 has acquired this resource and then been preempted by the medium priority task τ_2. In summary, τ_2 is blocking τ_1.

Outline

1. **Overview - the scale of the problem**
   - Motivation
   - Application architectures
2. **Fighting back**
   - Design principles and standards
   - Security models - confinement, BLP, BIBA, Chinese wall
   - Formal Methods

The data warehouse...

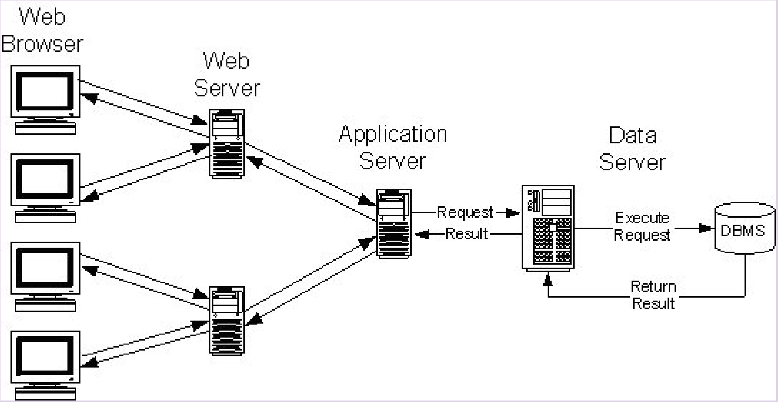
Repository of important data



Central repositories storing current and historical data, used for creating important reports for an organization.

The web application server...

Browsers and remote databases



The application server sends queries to the organization's database.

Outline

Saltzer and Schroeder's design principles

8 key points from paper summarized below:

`http://web.mit.edu/Saltzer/www/publications/protection/index.html`

- **Economy of mechanism:** Keep design as simple and small as possible.
- **Fail-safe defaults:** Base access decisions on permission rather than exclusion. The default is no access.
- **Complete mediation:** Every access to every object must be checked for authority.
- **Open design:** The design should not be secret.
- **Separation of privilege:** Two keys are better than one. No single event can compromise the system. Dual controls.
- **Least privilege:** Every program and every user of the system should operate using the least set of privileges necessary to complete the job.
- **Least common mechanism:** Minimize the amount of mechanism common to more than one user and depended on by all users.
- **Psychological acceptability:** Human interfaces should be easy to use.

Design principles for complexity

Economy of mechanism

KISS - keep it simple (stupid)

Why? Fewer errors, and checking correctness is easier.
Complex mechanisms make more assumptions, and it is hard to test for all these assumptions.

Economy of mechanism failures

IPSEC: Can do almost everything to secure TCP/IP but it is complex, and implementations vary in behaviour, and sometimes are incompatible with other implementations.
Perhaps this is because it was designed by committee?

Economy of mechanism successes

People switch to SSL VPNs which are much simpler, proven, compatible, robust...

Design principles for complexity

Least common mechanism

Clients (subjects/processes) should minimize the amount of mechanism common to more than one user and depended on by all users.

Why? A common mechanism may provide a path of information leaks (Confinement/Covert storage channels). Common mechanisms must be trustworthy - what if a user found a way to corrupt or damage the shared mechanism, and as a result all users were affected? By default, clients should not share anything.

Least common mechanism failures

Microsoft NT architecture: FTP and Web services on the same computer shared a common thread pool.
Exhausting the FTP thread pool will cause failed connection requests for the Web service.

Least common mechanism successes

- libc

Sample design rules

Possible rule arising from the principles

- Use a standard design pattern - Is your system architecture a well understood pattern?
- Minimize subsytems - Is each component of a composite system actually necessary? Can we remove a sub component entirely? This sort of optimization may be done at the design phase.
- Minimize the interfaces - Between each component are interfaces (perhaps communication or just calls). We should minimize the interfaces, only leaving those that are absolutely necessary.
- Make explicit the interfaces - We should also make such interfaces explicit. It is a very bad idea to have a component that relies on something in another component, with no explicit annotation that tells you of this reliance.
- Isolate components - Is each component stand-alone? Does it always do its job, even if all the components it commuicates with are lying to it?

Security standards: the Rainbow documents

For evaluating security of machines

The NSA created various documents describing the criteria for evaluating the security behaviour of machines. These criteria were published in a series of documents with brightly coloured covers, and hence the name Rainbow Documents.

TCSEC document

DOD 5200.28-STD - "Department of Defense Trusted Computer System Evaluation Criteria": to provide a standard to manufacturers (for security features related to confidentiality), to provide DoD components with a metric with which to evaluate the degree of trust, and to provide a basis for specifying security requirements in acquisition specifications.

Some of the Rainbow series have been superceded by the Common Criteria Evaluation and Validation Scheme (CCEVS). For background and further information, see the CCEVS web site here:
http://www.niap-ccevs.org/cc-scheme/

Security standards: Peculiar language...

Extracted from the document (TCSEC)...

- *The TCB[a] shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate.*
- *Furthermore, the TCB shall use a protected mechanism (e.g., passwords) to authenticate the user's identity.*

[a] Trusted Computing Base.

How useful is C2?

Windows systems have completed C2 testing, but only certified if using the same hardware, and installed software, and no network connection.
Many UNIX systems have also got C2 certification, and come configured this way from the manufacturer.
There are numerous examples of hacked Windows and UNIX systems.
C2 certification is probably not a good guide as to the security of your system.

Security standards: formal evaluation - TCSEC

TCSEC (The Orange book) - first rating system for security

- **C1** - For **same-level** security access. Not currently used.
- **C2** - **Controlled access protection** - users are individually accountable for their actions.
- **B1** - **Mandatory BLP policies** - for more secure systems handling classified data.
- **B2** - **structured protection** - mandatory access control for all objects in the system. Formal models.
- **B3** - **security domains** - more controls, minimal complexity, provable consistency of model.
- **A1** - **Verified design** - consistency proofs between model and specification.

Security standards: formal evaluation - ITSEC

ITSEC derives from...

National security evaluation **criteria** from multiple countries.

A “sponsor” determines operational requirements, threats and security objectives. ITSEC specifies the **interactions** and documents between the sponsor and the evaluator.

Levels as in TCSEC

There are various levels of evaluation: **E0..E6**, with **E6** giving the highest level of assurance - it requires two independant formal verifications.

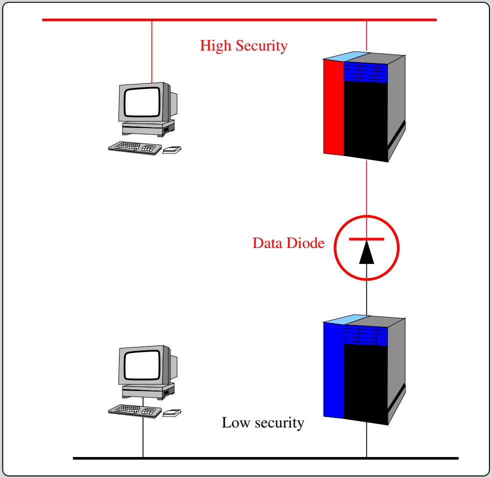
The first **E6** certification of a smart-card system was in 1998, for smart-cards used as electronic purses - that is they carry value, and forgery must be impossible.

The certification encompassed the communication with the card, as well as the software within the card, and at the bank.

Example: Data Pump/Diode E6, BLP

https://www.commoncriteriaportal.org/files/epfiles/st_vid9513-st.pdf

An example



Outline

Preliminaries - formal security models

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. [J. von Neumann]

Definition: *a range of formal policies/methods for specifying the security of a system in terms of a (mathematical) model.*

A three step approach

1. Have or develop some sort of formal model
2. Determine and formalize some interesting/required properties
3. Check/verify the properties hold for the model, and then verify implementations.

Confinement and covert channels

Secret channels for leaking information

The confinement problem is one of preventing a system from leaking (possibly partial) information.

Sometimes a system can have an unexpected path of transmission of data, termed a covert channel, and through the use of this covert channel information may be leaked either by a malicious program, or by accident.

Classification of covert channels

We categorize covert channels into two:

1. **Storage channels:** using the presence or absence of objects
2. **Timing channels:** the speed of events

We can attempt to identify covert channels by building a shared resource matrix, determining which processes can read and write which resources.

Confinement and covert channels

An unscrupulous program could modify access permissions on a file to transmit a low data-rate message to another program.

Specifying properties formally

By tabulating the types of data in a system, and the properties of the operations (read, write, execute, transitive), it may be possible to specify that the system cannot leak information or be used to transfer information.

NRL Pump: example of a one-way network

For confidentiality it is OK for data to go from low to high security levels.

However, communication protocols (TCP/IP etc) include ACK messages (from high to low) to acknowledge reception of data. A malicious participant at the high level could have a covert channel by altering the timing of the ACKs.

To prevent this, the NRL network pump is the router between the high and low levels, and buffers the packets, sending ACKs back to the low level. The ACKS vary in time randomly (although related to a moving average of previous overall activity).

Bell-LaPadula, confidentiality

BLP from the names of the two authors of [BL75]

Military style model to assure *confidentiality* services.

Security levels are in a (total) ordering formalizing a policy which restricts information flow from a higher security level to a lower security level.

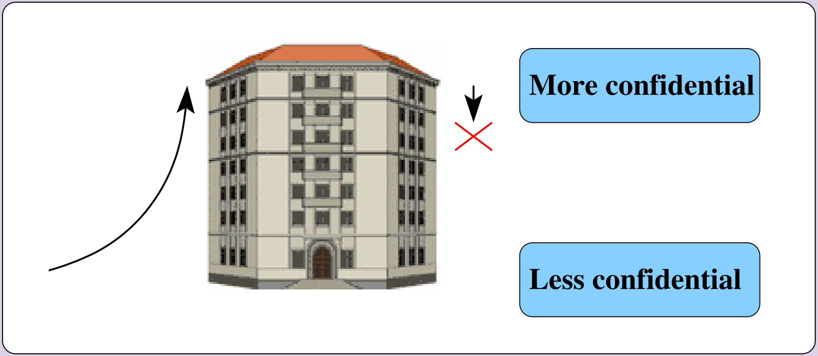
BLP has four levels of security:

1. Top secret (T)
2. Secret (S)
3. Confidential (C)
4. Unclassified (U)

where $T > S > C > U$. Access operations visualized using an access control matrix, and are drawn from **`{read, write}`**.

Import of the properties

We can view them as the activities in a secure building

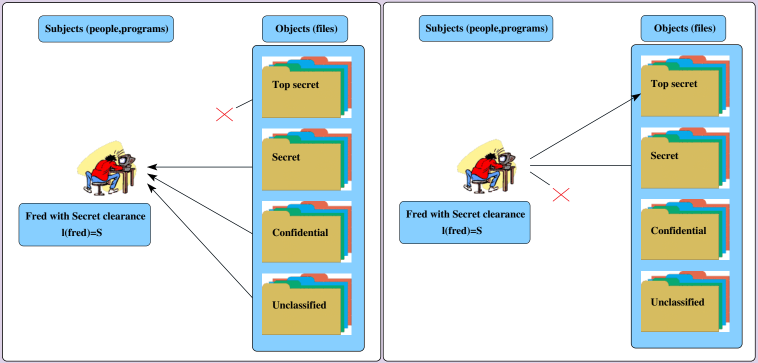


Our security policy for confidentiality is that we do not want confidential items to be leaked (downwards).

No read-up-1: s can **`read`** o if and only if $l_o \leq l_s$.
No write-down-1: s can **`write`** o if and only if $l_s \leq l_o$.

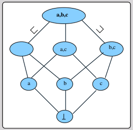
BLP - no read up, no write down

Levels for no-read-up and no-write-down:



BLP extended includes categories

Like sales, marketing, invasion plans...



A security category $c \in \mathscr{C}$ is used to classify objects in the model, with any object belonging to a set of categories. Each pair $(l \times c)$ is termed a *security level*, and forms a lattice.

We define a relation between security levels:

- A security level (l, c) dominates (l', c') (written (l, c) **dom** (l', c')) if $l' \leq l$, and $c' \subseteq c$.

Properties for the new extended model

The new properties are:

- **No read-up:** s can **`read`** o if and only if s **dom** o.
- **No write-down:** s can **`write`** o if and only if o **dom** s.
- **Discretionary:** s can **`read/write`** o if and only if no-read-up, no-write-down, and access permitted by discretionary policy.

BLP security

The security theorem

A system is considered *secure* in the current state if all the current accesses are permitted by the properties.

A transition from one state to the next is considered secure if it goes from one secure state to another secure state.

The basic security theorem states that if the initial state of a system is secure, and if all state transitions are secure, then the system will always be secure.

Note the limitations of this system

BLP is a static model, not providing techniques for changing access rights or security levels[a].
However the model does demonstrate initial ideas into how to model, and how to build security systems that are provably secure.

[a] You might want to explore the Harrison-Ruzo-Ullman model for this capability.

Biba model, integrity

A different kind of assurance

Biba model is concerned with the Trustworthiness of data and programs - assurance for *integrity* services.

It uses levels like **`clean`** or **`dirty`** (in reference, say, to database entries).

Biba model is a kind of *dual* for Bell-LaPadula. *integrity* vs *confidentiality*.

Approach like BLP, only integrity instead of confidentiality:

- The integrity levels $\mathscr{I}$ are ordered as for the security levels
- Function $i : \mathscr{O} \rightarrow \mathscr{I}$ ($i : \mathscr{S} \rightarrow \mathscr{I}$) which returns the integrity level of an object (subject).

Biba properties

Strict integrity policy rules

- **No read-down:** s can `read` o iff $i(s) \leq i(o)$.
- **No write-up:** s can `write` o iff $i(o) \leq i(s)$.
- **No invoke-up:** s_1 can `execute` s_2 iff $i(s_2) \leq i(s_1)$.

Low-watermark policy rules

Biba models can have dynamic integrity levels, where the level of a subject reduces if it accesses an object at a lower level (i.e. it *got dirty*).

- **No write-up:** s can `write` o iff $i(o) \leq i(s)$.
- **Subject lowers:** if s `reads` o then $i'(s) = \min(i(s), i(o))$.
- **No invoke-up:** s_1 can `execute` s_2 iff $i(s_2) \leq i(s_1)$.

Direct modification only (ring) policy rules

- **All read:** s can `read` o regardless.
- **No write-up:** s can `write` o if and only if $i(o) \leq i(s)$.
- **No invoke-up:** s_1 can `execute` s_2 if and only if $i(s_2) \leq i(s_1)$.

The Chinese wall model

Separation of duty

An underlying idea is that subjects cannot work for their client's competitors. We can write this in a similar fashion to the BLP model, using the notation $y(c)$ for c's company, and $x(c)$ for c's competitors.

- **SimpleProperty:** s can **access** c if and only if for all c' that s can read, either $y(c) \notin x(c')$ or $y(c) = y(c')$.
- ***-Property:** s can **write** c only if s cannot read any c' with $x(c') \neq \emptyset$ and $y(c) \neq y(c')$.

Outline

1. Overview - the scale of the problem
   - Motivation
   - Application architectures
2. Fighting back
   - Design principles and standards
   - Security models - confinement, BLP, BIBA, Chinese wall
   - Formal Methods

How formal is formal?

What are formal methods?

Formal methods involve the use of mathematically based techniques for the specification, development and verification of software and hardware systems[a]. Formal methods typically use some assortment of “computer science” fundamentals - process calculi, automata theory...

Formal specifications precisely describe a system to be developed and it’s properties.

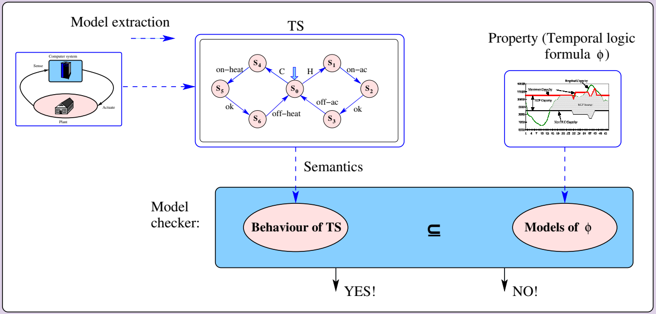
The verification of a system involves proving or disproving the correctness of a system with respect to the formal specification or property.

Model checking is one well established approach to verification.

[a] Well, according to Wikipedia :)

Model checking in a slide...

Properties and behaviour:



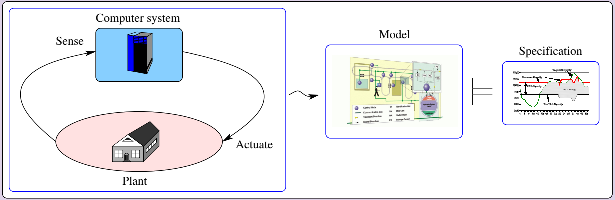
TS represents the behaviour of the system, expressed as the allowable set of traces (or computations) of the system.

A model-checker checks if this *behaviour* of the system is a subset of the set of traces induced by an arbitrary property ϕ, returning YES or NO.

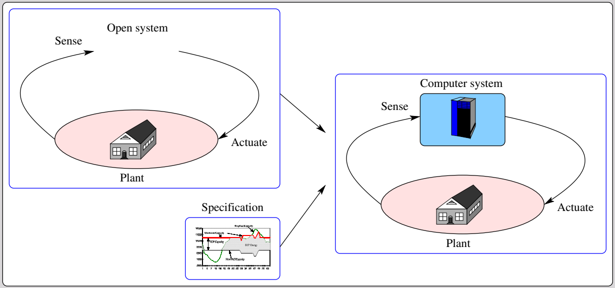
When the model checker returns NO, it provides a counter-example - a trace leading to the error.

Steps towards assurance...

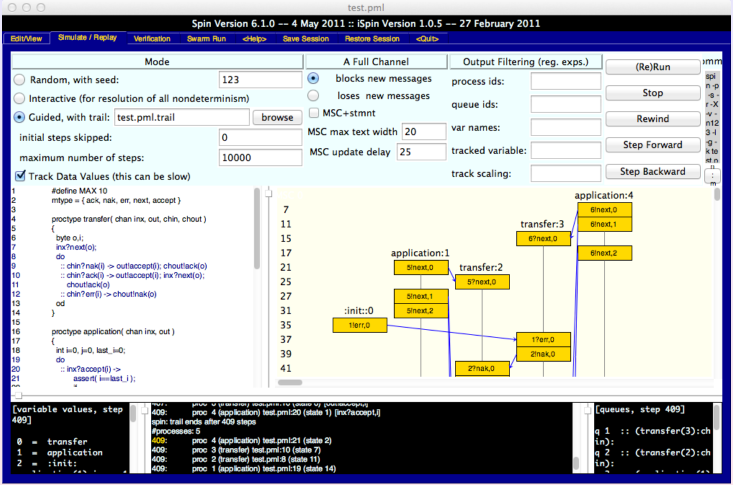
Modelling a system



Synthesizing a system



Example: Promela and spin



Spin is the **checker** for Promela models. It allows you to make assertions about the model: **assert(some_boolean_condition);**

Promela and spin

What is Promela?

The **language** Promela is 'C' like, with an initialization procedure. It can model asynchronous or synchronous, deterministic or non-deterministic systems.

It supports model checking with both safety and liveness assertions. What this means, is that in addition to boolean assertions scattered throughout the model, we can make time/temporal based assertions/claims.

Examples of these extended claims?

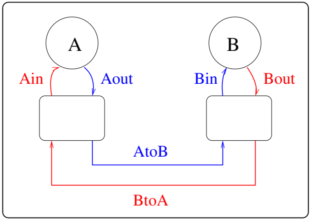
We got here again without making any progress!

The support for temporal claims takes the form of:

- **Endstate** labels - for determining valid endstates
- **Progress** labels - claim no non-progress cycles
- **Never** claims - impossible temporal assertions

Promela example

4 processes, 6 channels...



The "mainline"

```
mtype = {ack,nak,err,next,accept}
init
{
 chan AtoB = [1] of { mtype,byte };
 chan BtoA = [1] of { mtype,byte };
 chan Ain  = [2] of { mtype,byte };
 chan Bin  = [2] of { mtype,byte };
 chan Aout = [2] of { mtype,byte };
 chan Bout = [2] of { mtype,byte };
 atomic {
  run application( Ain,Aout );
  run xfer( Aout,Ain,BtoA,AtoB );
  run xfer( Bout,Bin,AtoB,BtoA );
  run application( Bin,Bout )
 };
 AtoB!err(0)
}
```

This is Lynch's protocol - with two applications sending data continuously to each other. Lynch's protocol was described in detail, used for many years, but had a flaw. It could get into a state where it would no longer send data one way.

Promela example

Transfer/protocol rules

```
proctype xfer(chan in,out,chin,chout)
{
 byte o,i;
 in?next(o);
 do
  ::chin?nak(i) -> out!accept(i);
                   chout!ack(o)
  ::chin?ack(i) -> out!accept(i);
                   in?next(o);
                   chout!ack(o)
  ::chin?err(i) -> chout!nak(o)
 od
}
```

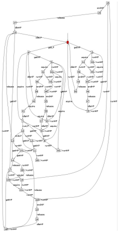
An application for testing

```
#define MAX 10
proctype application( chan in, out )
{
  int i=0, j=0, lasti=0;
  do
    ::in?accept(i) ->
      assert( i==lasti );
      if
       ::(lasti!=MAX) -> lasti=lasti+1
       ::(lasti==MAX)
      fi
    ::out!next(j) ->
      if
       ::(j!=MAX) -> j=j+1
       ::(j==MAX)
      fi
  od
}
```

The assertion tests if the last message had a correct number, and is always OK. But one of the applications can make no progress. Formal methods catch these hard-to-find errors.

A (CSP) model for the pathfinder software

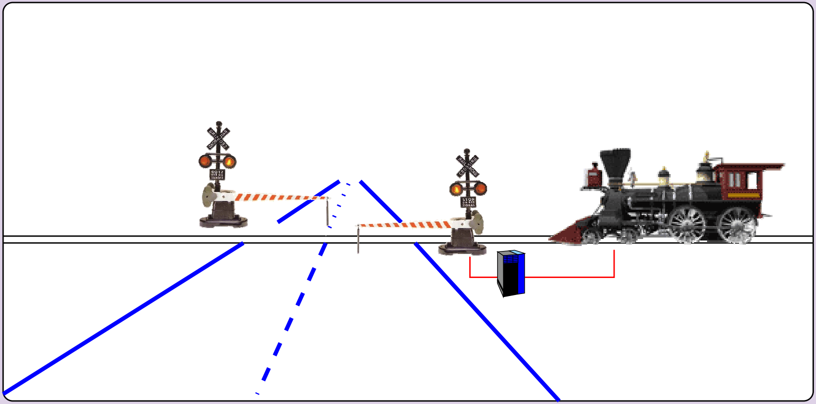
Three tasks, High, Med and low - initially idle:



```
#define idle 0;
#define wait 1;
#define run 2;
var L=idle;
var H=idle;
var M=idle;
var mutex=true;

GetMutex()  = [mutex]aquire{mutex=false;} -> Skip();
FreeMutex() = [!mutex]release{mutex=true;} -> Skip();

HiPri()  = getHP{H=wait;M=idle;} -> GetMutex();
           runHP -> DoHigh();
DoHigh() = endHP{H=run;} -> FreeMutex();
           idleHP{H=idle;} -> HiPri();

MedPri() = [H!=run]runMP{M=run;} -> MedPri();

LowPri() = [H==idle&&M==idle]getLP{L=wait;} -> GetMutex();
           [H==idle&&M==idle]runLP -> DoLow();
DoLow()  = [H==idle&&M==idle]endLP{L=run;} -> FreeMutex();
           [H==idle&&M==idle]idleLP{L=idle;} -> LowPri();

AllTasks() = HiPri() ||| MedPri() ||| LowPri();

#assert AllTasks() deadlockfree;
#assert AllTasks() |= [](getHP -> <>endHP);
```

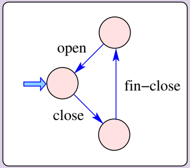
Another example system...

Train, gate, controller

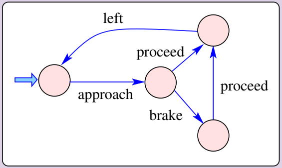


Modelling the system...

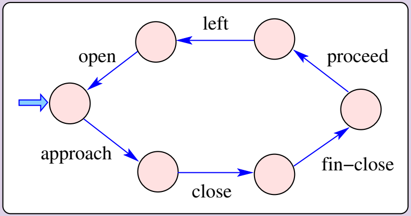
Three simple transition systems



Gate



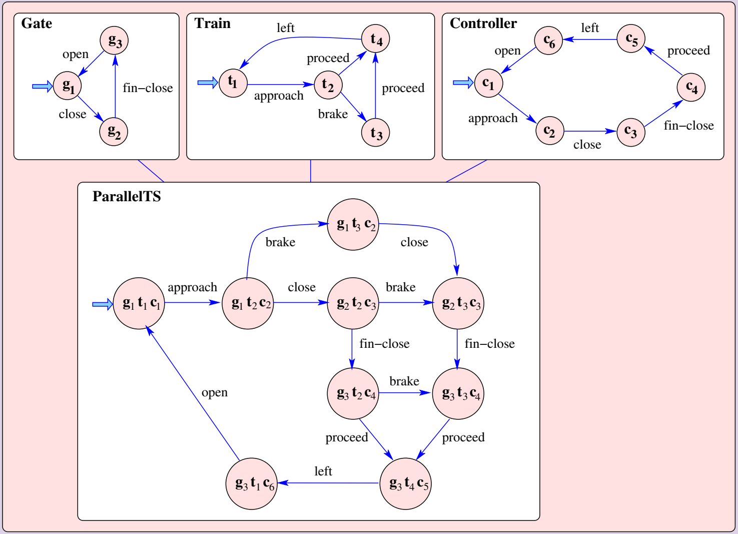
Train



Gate Controller

Modelling the system...

Construct a parallel composition ...



Summary

Today, we have seen...

Examples - Complex systems are everywhere, and a lot of software is produced without much thought for complexity. Programmers and software designers should adopt better engineering approaches.

Architectures - There are standard desiign patterns in software that should be adopted. Dont re-invent the wheel.

Standards and formal methods - Two important tools for developing better software.