Conception Formelle : Module Modelisation et Vérification

Cours

Alain Griffault

Master 1 Informatique
Université Bordeaux 1
2007-2008

Outline

The bank of New-York (1985)

Facts

On November, 21, an integer (coding the different government securities issues involved in transactions) increments from 32768 to 0.

Consequences

- More than 2 days without services.
- The bank had to borrow 24 billion USD from the New York Fed.
- The Bank of New York was out of pocket about 5 million USD, to pay interest on the money it had to borrow that Thursday.

The ATT crash (1990)

Facts

A patch on January, 15

```
switch(i) {
  case 1 : function1(); break;
  case 2 : function2(); // a forgotten break;
  default: defaultfunction();
}
```

for better performances, was used during 9 hours.

Consequences

- 75 million phone calls across the US went unanswered.
- American Airlines estimated this error cost it 200,000 reservations.
- The reputation of AT&T was damaged.

The Pentium's Bug (1994)

Facts

- June : Thomas Nicely, an American professor of mathematics, discovered that the calculus 1/824633702441.0 is erroneous.
- October 30 : He concluded that there is a bug in the floating point unit after he had eliminated all other likely sources of error. He send a mail to Intel.
- November 7 : An error in a lookup table was discovered.

Consequences

- Intel acknowledged the bug but claimed that it was not serious and would not affect most users. Intel offered to replace processors to users who could prove that they were affected.
- On December 20, 1994 Intel offered to replace all flawed Pentium processors, in response to mounting public pressure.
- Intel and AMD now publish reports showing theirs arithmetic algorithms and their proofs.

The Ariane 5 rocket (1996)

Facts

- 39 seconds after launch, the rocket self destructs.
- 36.7 seconds after launch, a conversion from a 64-bit format to a 16-bit format produces an overflow error.
- Normally, conversions are protected by extra lines of code.
- Sometime, due to physical knowledge, no protection are done.
- This code was exactly the same than the one in Ariane 4.
- Unluckily, Ariane 5 was a faster rocket than Ariane 4...
- Two computers, but the same code, and so the same bug...
- This function is not required for Ariane 5, but was maintained for commodity...

Consequences

- Cluster mission : 500 million USD.
- Ariane 502 : 1 year and 4 months later.

Therac-25 accidents (1982-1987)

Facts

- 1982 : The Therac-25 delivers photons or electrons at various energy levels. EACL said that Therac-25 is "easy to use".
- June 1985 : for a treatment to the clavicle area, the patient said "You burned me."...engineers said "Impossible".
- July 1985, new accident. A suspected transient failure in a micro-switch is corrected. The patient died on November 1985.
- December 1985 : new accident. AECL writes : "...this damage could not have been produced by the Therac-25."
- March 1986 : new accident. AECL engineers suggested an electrical problem. The patient died five months after.
- April 1986 : new accident. A race condition bug is reproduced. The soft is modify. The patient died three weeks after.
- January 1987 : new accident. New bug due to shared

Therac-25 accidents (1982-1987)

Consequences

- February 1987 : The machine was recalled for extensive design changes, including hardware safeguards against software errors.
- The first software responsible for the death of four or more persons.

Limits of traditional design methods

- Cost of testing.
- Different interpretations of usual words → UML.
- Ambiguity if semi-formal methods (# semantics for UML).
- Design patterns for POO are good, but event or reactive programming are more difficult and there are no "pattern".
- UML (RdP, ST, MSC), but how composition of a RdP and a MSC is done ?
- Adding functionalities without regression is a difficult task.

New trends and recommendations

- Common Criteria for Information Technology Security Evaluation.
- Common Criteria Assurance Levels has seven levels.
  1. Functionally Tested.
  2. Structurally Tested.
  3. Methodically Tested and Checked.
  4. Methodically Designed, Tested and Reviewed.
  5. Semi-formally Designed and Tested.
  6. Semi-formally Verified Design and Tested.
  7. Formally Verified Design and Tested.
- Today, only one product at level 5 !
- Mandatory for big company which require more and more to their subcontractors, causing a snowball effect for 3 years.

More examples

Some URLs

```
http://catless.ncl.ac.uk/Risks
http://www.zdnet.co.uk/toolkits/disasterrecovery/

ATT : http://catless.ncl.ac.uk/Risks/1.31.html#subj4
Pentium : http://www.trnicely.net/#PENT
Ariane : http://sunnyday.mit.edu/accidents/Ariane5accident:
Therac 25 : http://courses.cs.vt.edu/~cs3604/lib/Therac_25,

ITSEC and EAL : http://www.cesg.gov.uk/site/iacs/
```

Outline

Formal method concepts

Goal and approach

- Being able to reason about software and systems to determine their behavior and control.
- Systems are mathematical objects.

Process

- Getting a formal model of the software or system.
- Analyze the model with an adequate formal techniques.
- Translating the results of the models to real systems.

Problems

- Is the model realistic and correct ? *validation*.
- Can we verify all models ? *decidability*.
- Can we always translate results ? *abstraction*

Critical systems

Formal methods : For who ?

For economic reasons, the extra work due by formal techniques implies that only designers and/or developers of complex and/or critical systems are looking for formal techniques.

Formal methods : For what ?

A system is said critical when either :

- The lives of people is tied to its effective functioning. This is the case of embedded software for the transport (air, train, car, bus ...), controllers systems (nuclear power plants, medical equipment ...).
- The economic cost of a failure is catastrophic. This is the case of software made in silicon to be produced in very large amounts (appliances, phones ...).

Survey of formal methods

Theorem prover

- Automated Theorem Proving (ATP) deals with proofs as in mathematics.
- ATP tools are interactive, and help the user to construct a certified proof.
- Hardware verification is the largest industrial application of ATP.
- Not well suited for reactive systems.

Some tools

- Coq (INRIA France)
- Isabelle (Larry Paulson) Cambridge.
- ACL2 (Matt Kaufmann and J Strother Moore) Austin.
- ...

Survey of formal methods

Abstract interpretation

- Transforming a concrete problem in a more simple abstract model, in which the *proofs* will be easier to make.
- "counting modulo 9" is an example of abstract interpretation.
- $(M_C \models \phi) \Rightarrow (M_A \models \phi)$.

Some tools

- PolySpace, Astree (P. Cousot) France
- BLAST : Berkeley Lazy Abstraction Software Verification Tool (Henzinger)
- SLAM : The Software, Languages, Analysis and Model checking (Microsoft).
- ...

Survey of formal methods

Domain Specific Languages

- A language with primitives dedicated to application type.
- A compiler to a classical language. (C ; ADA).

Some tools

- XML, HTML, LaTeX to produce texts
- YACC, LEX to produce parsers and compilers.
- Languages for protocols, drivers.
- Esterel for real time software. (G. Berry) France
- ...

Survey of formal methods

Model checking

- Tools that can automatically decide if a model M satisfies a logical property ϕ.
- "Validation" is necessary.
- State explosion problem.

Some tools

- CESAR, MEC, EMC (1985-1995)
- SMV, NuSMV, SPIN, StateMate
- Uppaal, Kronos
- Tina, PIPN,
- ...

Survey of formal methods

Refinement and implementation

- Allows, in a number of steps more or less important to transform a specification to an implementation by preserving at each stage the essential properties of the system.
- Each step has to be prove, refinement often use a theorem prover.

Some tools

- Z (J. R. Abrial) Oxford, GB
- B, Event- B (J. R. Abrial) Lausanne
- ...

Survey of formal methods

Tests generation

- Generating test sequences from a model of the specification.
- Tests are relatives to specific goals.
- Conformity tests are used to validate the real system.
- Interoperability tests are used to validate the real system in an open context.

Some tools

- TGV (T. Jéron) Rennes, France
- ...

Survey of formal methods

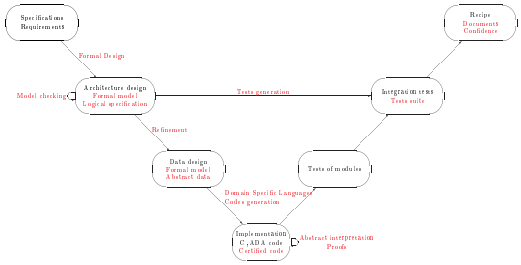
Stochastic systems

- Markov chains and stochastic processes add probability to events.
- To compute the probability that an event occurs.
- To compute witch elementary failures are involved in such an event.
- Reliability and safety analysis domain.

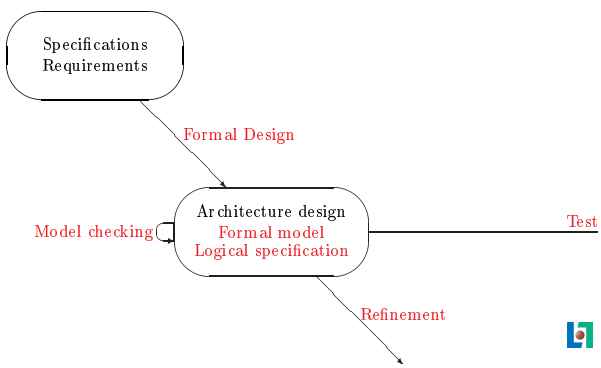
Some tools

- Aralia (A. Rauzy) France.
- ...

Life cycle and formal methods



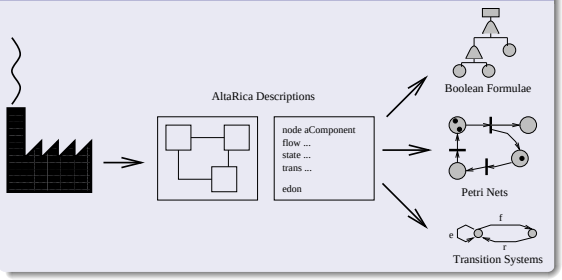
The course : formal design



Outline

The AltaRica project

Goals



The AltaRica project

A brief story of the project

- 97/99 : definition of the language and first prototype.
  - Academics LaBRI and LADS.
  - Companies Dassault, Renault, Total, Schneider, IPSN.
- Today few tools and numerous users.
  - OCAS AltaRica Dassault (simulation and Fault Tree)
  - SIMFIA V2 Airbus (simulation and Fault Tree)
  - AltaRica DF A. Rauzy (stochastic and Fault Tree)
  - arc, mec 5 LaBRI (semantic and verification).
  - T_AltaRica (IRCCyN).
  - AltaRica↔Lustre (European projects with CERT ONERA).
  - AltaRica + Abstract Data Types (ACI Projet Persee).

The AltaRica model of calculus

Constraint automata

- a finite set of state variables $\vec{s}$,
- a finite set of flow variables $\vec{f}$,
- a finite set of events E,
- a set of transitions :

$$G(\vec{s},\vec{f}) \xrightarrow{e} \vec{s} := \sigma(\vec{s},\vec{f})$$

  $G(\vec{s},\vec{f})$ is a guard (ie a boolean formula) and $e \in E$,
- an assertion (invariant) $A(\vec{s},\vec{f})$,
- a partial order on E to define priorities.

Fundamental concepts of AltaRica

Modular and compositional

- Hierarchy (a tree).
- Visibility of components.
- Leaves :
  - Difference between flow and state variables.
  - Guarded transitions with post-condition.
  - An assertion binding flow and state variables.
- Interaction between components :
  - An assertion binding all visible flow and state variables.
  - Generalized synchronization vectors.
  - Priorities between events.
- Bisimulation.

Main result

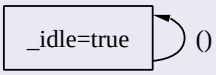
The hierarchy preserves the bisimulation relation.

The minimal node and the ϵ event

Syntax

```
node Minimal
edon
```

The semantic



The ϵ transition

In all AltaRica components, the transition
`True |- ϵ -> ;`
is implicit.

Guarded transition with post condition

A FIFO

```
node FIFO_V1
  state nbMessages : [0,2]; init nbMessages:=0;
  event put, get;
  trans
    nbMessages<2 |- put -> nbMessages:=nbMessages+1;
    nbMessages>0 |- get -> nbMessages:=nbMessages-1;
edon
```

The semantic

Guarded transition with post condition

Another equivalent FIFO

```
node FIFO_V2
  state nbMessages : [0,2]; init nbMessages:=0;
  event put, get;
  trans
    true |- put -> nbMessages:=nbMessages+1;
    true |- get -> nbMessages:=nbMessages-1;
edon
```

The semantic

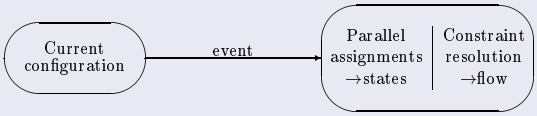
Flow variables and assertion

A switch

```
node Switch
  state  on : bool : public;
  init   on := true;
  flow   f1, f2 : [0,1];
  event  push;
  trans  true  |- push   -> on := ~on;
  assert on => (f1=f2);
edon
```

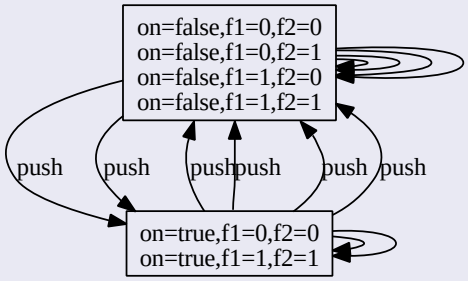
Flow variables and assertion

Computing the semantic



Flow variables and assertion

The semantic



Exercise : Flow variables and assertion

A Generator

```
node Generator
  flow   plus, minus : [0,1];
  state  on : bool : public;
  init   on := true;
  event  failure, repair;
  trans  on |- failure -> on := false;
         ~on |- repair -> on := true;
  assert plus = 1;
         on = (minus = 0);
edon
```

Exercise : a lamplight

A Lamplight

```
node Lamplight
  flow  f1, f2 : [0,1];
  state on, ok : bool;
  init  ok := true;
  event reaction;
  trans
    ok & (f1=1&f2=1)  |- reaction -> ok:=false,
                                     on:=false;
    ok & (on=(f1=f2)) |- reaction -> on:=~on;
edon
```

Hierarchy and asynchrony

A first electrical circuit

A circuit links a switch, a generator and a lamplight.

```
node CircuitV1
  sub    G : Generator;
         S : Switch;
         L : Lamplight
  assert S.f1 = G.plus;
         L.f1 = S.f2;
         L.f2 = G.minus;
edon
```

Hierarchy and asynchrony

The semantic : state and flow variables

```
node CircuitV1
 flow
  G.plus : [0,1];
  G.minus : [0,1];
  S.f1 : [0,1];
  S.f2 : [0,1];
  L.f1 : [0,1];
  L.f2 : [0,1];
 state
  G.on : bool;
  S.on : bool;
  L.on : bool;
  L.ok : bool;
 init G.on := true, S.on := true, L.ok := true;
```

Hierarchy and asynchrony

The semantic : assertion

```
 assert
  S.f1 = G.plus;
  L.f1 = S.f2;
  L.f2 = G.minus;
  G.plus = 1;
  G.on = G.minus = 0;
  not S.on or S.f1 = S.f2;
```

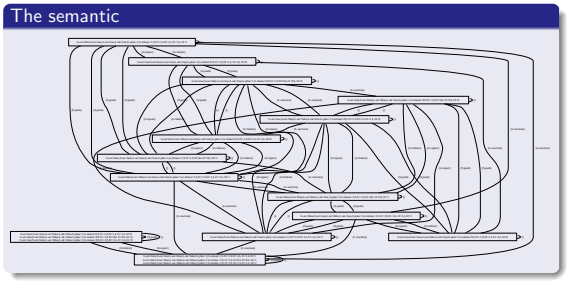
Hierarchy and asynchrony

The semantic : events and transitions

```
event
 /* 1 */ '<$,G.repair,S.$,L.$>',
 /* 2 */ '<$,G.$,S.$,L.$>',
 /* 3 */ '<$,G.$,S.$,L.reaction>',
 /* 4 */ '<$,G.failure,S.$,L.$>',
 /* 5 */ '<$,G.$,S.push,L.$>';
trans
 true     |-'<$,G.$,S.$,L.$>'       -> ;
 G.on     |-'<$,G.failure,S.$,L.$>'-> G.on:=false;
 not G.on|-'<$,G.repair,S.$,L.$>' -> G.on:=true;
 true     |-'<$,G.$,S.push,L.$>'    -> S.on:=not S.on;
 L.ok andL.f1 = 1 and L.f2 = 1
  |-'<$,G.$,S.$,L.reaction>'->L.on:=false,L.ok:=false;
 L.ok and L.on = L.f1 = L.f2
  |-'<$,G.$,S.$,L.reaction>'->L.on:=not L.on;
```

Hierarchy and asynchrony

The semantic



Hierarchy and synchronization

A second electrical circuit

```
node CircuitV2
  sub    G : Generator;
         S : Switch;
         L : Lamplight
  state  safe : bool;   init safe := false;
  event  begin, repair, end;
  trans  ~safe & ~G.on |- begin -> safe := true;
         safe          |- repair ->;
         safe & G.on   |- end -> safe := false;
  sync   <repair, G.repair>;
  assert ~safe => (S.f1 = G.plus);
         safe => (S.f1 = 0);
         L.f1 = S.f2;
         L.f2 = G.minus;
edon
```

Hierarchy and synchronization

The semantic : state and flow variables

```
node CircuitV2
 flow  G.plus : [0,1];
       G.minus : [0,1];
       S.f1 : [0,1];
       S.f2 : [0,1];
       L.f1 : [0,1];
       L.f2 : [0,1];
 state safe : bool;
       G.on : bool;
       S.on : bool;
       L.on : bool;
       L.ok : bool;
 init  safe:=false, G.on:=true, S.on:=true, L.ok:=true;
```

Hierarchy and synchronization

The semantic : assertion and events

```
assert safe or S.f1 = G.plus;
       not safe or S.f1 = 0;
       L.f1 = S.f2;
       L.f2 = G.minus;
       G.plus = 1;
       G.on = G.minus = 0;
       not S.on or S.f1 = S.f2;
event  /* 1 */ '<end,G.$,S.$,L.$>',
       /* 2 */ '<repair,G.repair,S.$,L.$>',
       /* 3 */ '<$,G.$,S.$,L.$>',
       /* 4 */ '<begin,G.$,S.$,L.$>',
       /* 5 */ '<$,G.$,S.$,L.reaction>',
       /* 6 */ '<$,G.failure,S.$,L.$>',
       /* 7 */ '<$,G.$,S.push,L.$>';
```

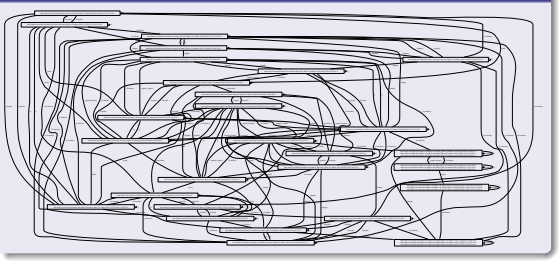
Hierarchy and synchronization

The semantic : transitions

```
trans
  true |- '<$,G.$,S.$,L.$>' ->;
  G.on |- '<$,G.failure,S.$,L.$>' -> G.on:=false;
  true |- '<$,G.$,S.push,L.$>' -> S.on:=not S.on;
  safe and G.on|-'<end,G.$,S.$,L.$>'-> safe:=false;
  not G.on and safe
   |-'<repair,G.repair,S.$,L.$>'-> G.on:=true;
  not safe and not G.on
   |-'<begin,G.$,S.$,L.$>'-> safe:=true;
  L.ok and L.f1 = 1 and L.f2 = 1
   |-'<$,G.$,S.$,L.reaction>'-> L.on:=false,L.ok:=false;
  L.ok and L.on = L.f1 = L.f2
   |-'<$,G.$,S.$,L.reaction>'->  L.on:=not L.on;
```

Hierarchy and synchronization

The semantic



Exercise : Are these models valid ?

Exercise : Are these models valid ?

A correction

```
node CircuitV1_OK
  sub    G : Generator;
         S : Switch;
         L : Lamplight
  assert S.f1 = G.plus;
         L.f1 = S.f2;
         L.f2 = G.minus;
// the switch must be oriented
(S.f2=1) => S.on;
edon
```

Priority

A random scheduler

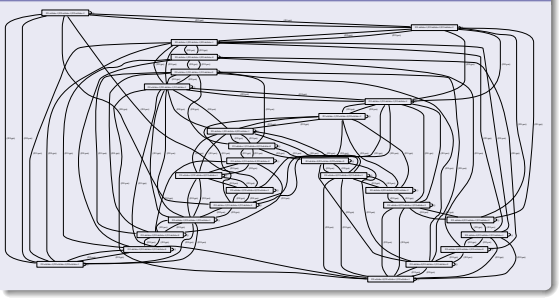
Consider a scheduler which takes randomly jobs in three different FIFO representing pool of jobs.

```
node PoolOfJobs
  state nbJobs : [0,2] : public;
  init  nbJobs := 0;
  event put, get;
  trans
    true |- put -> nbJobs := nbJobs + 1;
    true |- get -> nbJobs := nbJobs - 1;
edon

node SchedulerRandom
  sub PJ1, PJ2, PJ3 : PoolOfJobs;
edon
```

Priority

Semantic of the random scheduler



Priority

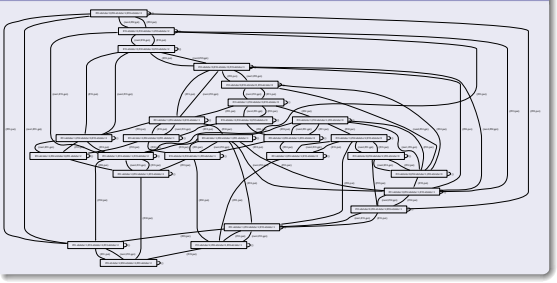
A scheduler with priorities

Jobs in PJ1 must be done before jobs in PJ2 and so on...

```
node SchedulerPriority
  sub    PJ1, PJ2, PJ3 : PoolOfJobs;
  event  run1, run2, run3;
  trans  true |- run1 -> ;
         PJ1.nbJobs=0 |- run2 -> ;
         PJ1.nbJobs=0 & PJ2.nbJobs=0 |- run3 -> ;
  sync   <run1, PJ1.get>;
         <run2, PJ2.get>;
         <run3, PJ3.get>;
edon
```

Priority

Semantic of the priority scheduler



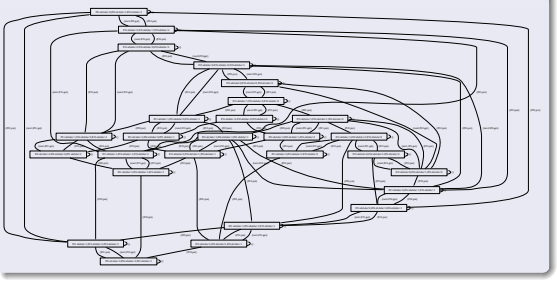
Priority

A scheduler using the priority concept

```
node Scheduler
  sub    PJ1, PJ2, PJ3 : PoolOfJobs;
  event  run1 > run2;
         run2 > run3;
  trans  true |- run1 -> ;
         true |- run2 -> ;
         true |- run3 -> ;
  sync   <run1, PJ1.get>;
         <run2, PJ2.get>;
         <run3, PJ3.get>;
edon
```

Priority

Semantic of the scheduler with priority



Broadcast

A Teacher

A teacher speaks and gives papers to students during a course.

```
node Teacher
  state present : bool;
  init  present := false;
  event in_out, speaks, paper;
  trans true |- in_out -> present:=~present;
        present |- speaks, paper -> ;
edon
```

Exercise : semantic of this teacher.

Broadcast

A Student

A student some times comes without its pencil.

```
node Student
  state present, havePencil : bool;
  init  present:=false, havePencil:=true;
  event in_out, listen, write;
  trans present  |- in_out -> present:=false,
                              havePencil:=true;
        ~present |- in_out -> present:=true,
                              havePencil:=true;
        ~present |- in_out -> present:=true,
                              havePencil:=false;
        present  |- listen -> ;
        present & havePencil |- write -> ;
edon
```

Broadcast

A very difficult course

3 students and the teacher brings only 2 papers.

```
node CoursesDifficult
 sub  T : Teacher;
      S1, S2, S3 : Student;
 sync
  <T.speaks,S1.listen,S2.listen,S3.listen>;
  <T.paper,S1.write,S2.write>;
  <T.paper,S1.write,S3.write>;
  <T.paper,S2.write,S3.write>;
edon
```

Exercise : semantic of this difficult course.

Broadcast

A course with priority (1)

```
node CoursesPriority
 sub  T : Teacher;
      S1, S2, S3 : Student;
 event pr123 > {pr12,pr13,pr23};
       pr12 > {pr1, pr2};
       pr13 > {pr1, pr3};
       pr23 > {pr2, pr3};
       pr_0 < {pr1, pr2, pr3};
       st12 > {st1, st2};
       st13 > {st1, st3};
       st23 > {st2, st3};
       st_0 < {st1, st2, st3};
 trans
  true |- pr123,pr12,pr13,pr23,pr1,pr2,pr3,pr_0 -> ;
  true |-       st12,st13,st23,st1,st2,st3,st_0 -> ;
```

Broadcast

A course with priority (2)

```
 sync <pr123,T.speaks,S1.listen,S2.listen,S3.listen>;
      <pr12, T.speaks,S1.listen,S2.listen>;
      <pr13, T.speaks,S1.listen,S3.listen>;
      <pr23, T.speaks,S2.listen,S3.listen>;
      <pr1,  T.speaks,S1.listen>;
      <pr2,  T.speaks,S2.listen>;
      <pr3,  T.speaks,S3.listen>;
      <pr_0, T.speaks>;
      <st12, T.paper,S1.write,S2.write>;
      <st13, T.paper,S1.write,S3.write>;
      <st23, T.paper,S2.write,S3.write>;
      <st1,  T.paper,S1.write>;
      <st2,  T.paper,S2.write>;
      <st3,  T.paper,S3.write>;
      <st_0, T.paper>;
```

Broadcast

A course using the concept of broadcast

```
node Courses
 sub  T : Teacher;
      S1, S2, S3 : Student;
 sync
  <T.speaks,S1.listen?,S2.listen?,S3.listen?>;
  <T.paper,S1.write?,S2.write?,S3.write?> <=2;
edon
```

Visibility

Default value in bold.

State visibility

Keyword	Me	Brother	Father	(Grand-father)*
private	Guard Assignment Assertion			
parent	Guard Assignment Assertion		Guard Assertion	
public	Guard Assignment Assertion		Guard Assertion	Guard Assertion

Visibility

Default value in bold.

Flow visibility

Keyword	Me	Brother	Father	(Grand-father)*
private	Guard Assertion			
parent	Guard Assertion		Guard Assertion	
public	Guard Assertion		Guard Assertion	Guard Assertion

Visibility

Default value in bold.

Event visibility

Keyword	Me	Brother	Father	(Grand-father)*
parent	Transition Synchro		Synchro	
public	Transition Synchro		Synchro	Synchro *if not previously* *in one synchro*

Declaration

Constant and Domain

```
Const  LastLevel = 3;
Domain Levels = [0,LastLevel];
```

Comment

```
// A comment that stops at the end of the line

/* A 5 lines comment.
 * No imbrication
 *
 *
*/
```

Declaration

Initial values of states

```
node A
  state a : bool;
edon
node B
  sub A1 : A;
  state b : bool;
  init b:= true, A1.a := false;
edon
node C
  sub A2 : A;
      B1 : B;
  state c : bool;
  init A2.a := true, B1.b := false, B1.A1.a := true;
edon
```

Outline

Informal definitions

Validation

A model is valid if it looks like the system. The model is a good candidate to be the *real* system.

How to decide if a model is valid

- The model must satisfy all properties that define a good candidate...
  but there is no such list of properties.
- Minimal list of properties depends on the application type.
- Simulation is considered as a good way to validate a system.

Validation ensures that "you built the right thing".

Informal definitions

Verification

A validate model is verify if it satisfies all requirements.

How to decide if a model is verify

- The model must satisfy all requirements describe in the specification document.
- Simulation is **not** considered as a good way to verify a critical system.
- Model checking is considered as a good way to verify a critical system.

Verification ensures that "you built it right".

Model checking

Principle

- The system is represented by a model M.
- Requirements are described as a logical property ϕ.
- A model-checker is a tool that takes as input M and ϕ, and automatically gives as output either $M \models \phi$ or $M \not\models \phi$. In that second case, it can give an explanation (a counter example).

Model checking

Theoretical limits

$M \models \phi$ must be decidable so (M, ϕ) must be in a certain class of models and formula.

- Finite systems : all formula in behavior logics are decidable.
  → numerous academic tools.
- Infinite systems : very few formula in behavior logics are decidable. A lot of results for specific infinite systems (automata with counters, with clocks, with FIFO...).
  → numerous scientific papers and few tools.

Model checking

Two main classes of property.

Safety (reachability) property

- Something is impossible. (a configuration is not reachable)
- Ex : After an event, an other event is possible.
- If a safety property is not satisfy, the counter example is a finite word (a finite execution of the system).

Safety properties usually state that something bad never happens.

Liveness property

- Something is always possible.
- Ex : An event is always followed in a finite time by an other event.
- If a liveness property is not satisfy, the counter example is an infinite word (an infinite execution of the system).

Liveness properties state that something good keeps happening

Model checking

Practical limits

- Depends on computer memory when you use an explicit representation of the reachability graph.
- Depends on computer velocity when you use some implicit representation of the reachability graph (BDD, DBM, ...).
- The difficulty of the modelization task.
- The difficulty to be sure that the model is valid.
- The difficulty to write logical property.

→ numerous results in algorithms, data structures and in design methods.

Formalisms

Formalisms to describe systems

- Petri nets (numerous variations).
- State-charts.
- Message Sequence Charts (MSC).
- Data flow diagrams.
- Process algebra (CCS, CSP, LOTOS ...)
- Domain Specific Language (Lustre, Esterel, Promela, SMV, AltaRica, ...)
- ...

Formalisms

Behavior logics

- CTL (Computational Tree Logics) and the extension CTL*.
- LTL (Linear Time Logics) : The most popular, a subset of CTL*.
- HML (Hennessy-Milner Logics).
- $\mu-$calculus : a very expressive logics.
- Dicky's logics : a logics based on sets of states and sets of transitions.
- ...

Choices for this courses.

AltaRica finite automata

All variables must have finite domains. No integer, no real, ...

AltaRica Checker : ARC and acheck

The Dicky's logics with non alternating $\mu-$calculus.

Another tool :

AltaRica Checker : Mec V

The Park $\mu-$calculus.

Outline

Dicky's logics : part 1

Concepts

- A logical property of an automaton can be seen as the set of all entities that satisfy the formula.
- This property can be checked by putting some marks during a depth-first-search algorithm on the reachability graph.
- Conjunction (resp. disjunction) of logical properties corresponds to intersection (resp. union) of sets.
- Two kinds of properties for a graph $G(V, E)$: state properties ($S \subseteq V$) and transition properties ($T \subseteq V \times E \times V$).

Dicky's logics : part 1

Constants

For each automaton :

- The empty set : $\emptyset$
- The set of all states : `any_s`
- The set of all transitions : `any_t`
- The set of all initial states : `initial`

Dicky's logics : part 1

Set operators

F_1, F_2 are two formula of same type (state or transition)

- $[\![F_1 \text{ and } F_2]\!] = [\![F_1 \text{ \& } F_2]\!] = [\![F_1]\!] \cap [\![F_2]\!]$
- $[\![F_1 \text{ or } F_2]\!] = [\![F_1 \text{ | } F_2]\!] = [\![F_1]\!] \cup [\![F_2]\!]$
- $[\![F_1 \text{ - } F_2]\!] = [\![F_1]\!] \setminus [\![F_2]\!]$

S is a state formula and T a transition formula.

- $[\![\text{not } S]\!] = [\![\text{any\_s}]\!] \setminus [\![S]\!]$
- $[\![\text{not } T]\!] = [\![\text{any\_t}]\!] \setminus [\![T]\!]$

Dicky's logics : part 1

src, tgt, rsrc and rtgt operators

S is a state formula and T a transition formula.

- $\texttt{src}(T)$ is a state formula and $[\![\texttt{src}(T)]\!] = \{s | \exists (s, e, t) \in T\}$ (states that are source of a transition of T)
- $\texttt{tgt}(T)$ is a state formula and $[\![\texttt{tgt}(T)]\!] = \{t | \exists (s, e, t) \in T\}$ (states that are target of a transition of T)
- $\texttt{rsrc}(S)$ is a transition formula and $[\![\texttt{rsrc}(S)]\!] = \{(s, e, t) | s \in S, \}$ (transitions having their source in S)
- $\texttt{rtgt}(S)$ is a transition formula and $[\![\texttt{rtgt}(S)]\!] = \{(s, e, t) | t \in S, \}$ (transitions having their target in S)

Dicky's logics : Exercise 1

What is the meaning of these formula ?

- $\texttt{any\_s} - \texttt{src}(\texttt{any\_t})$
- $\texttt{src}(\texttt{rtgt}(S) \cap\ T)$
- $\texttt{tgt}(\texttt{rsrc}(S) \cap\ T)$

Compare these sets

- $\texttt{src}(\texttt{rsrc}(S))$ with S — $\texttt{rsrc}(\texttt{src}(T))$ with T
- $\texttt{tgt}(\texttt{rtgt}(S))$ with S — $\texttt{rtgt}(\texttt{tgt}(T))$ with T

Which properties are compute ?

- $A - B = \emptyset$ — $A \cap\ (\texttt{any\_s} - B) = \emptyset$
- $\texttt{tgt}(T_1) \cap\ (\texttt{any\_s} - \texttt{src}(T_2)) = \emptyset$
- $\texttt{rsrc}(\texttt{tgt}(T_1)) \cap\ (T_2) \neq \emptyset$

Dicky's logics : part 2

reach, coreach and loop operators

S is a state formula and T, T_1, T_2 are transition formula.

- $\texttt{reach}(S, T)$ is a state formula that computes all reachable states starting from S and using only transitions in T.
- $\texttt{coreach}(S, T)$ is a state formula that computes all states from them, it is possible to reach S by using only transitions in T.
- $\texttt{loop}(T_1, T_2)$ is a transition formula that compute the strongly connected components (SCC) T_3 defined by $T_3 \subseteq T_2$ and $T_3 \cap T_1 \neq \emptyset$.

Dicky's logics : Exercise 2

What is the meaning of these properties ?

- $\texttt{reach}(S_1, \texttt{ any\_t - } T) \cap S_2 = \emptyset$ — $\neq \emptyset$
- $\texttt{any\_s - coreach}(\texttt{any\_s} - \texttt{src}(\texttt{any\_t}), \texttt{ any\_t}) = \emptyset$ — $\neq \emptyset$
- $\texttt{loop}(T, T) = \emptyset$ — $\neq \emptyset$
- $\texttt{any\_t - loop}(\texttt{rsrc}(\texttt{initial}), \texttt{ any\_t}) = \emptyset$ — $\neq \emptyset$
- $\texttt{loop}(T_3, \texttt{ reach}(\texttt{tgt}(T_1), \texttt{ any\_t - } T_2)) = \emptyset$ — $\neq \emptyset$

Acheck

Principle

- acheck is a *batch* AltaRica model checker.
- acheck takes as input two files : first for the AltaRica model and second for the list of properties to check.
- User must map properties to AltaRica node.
- All output commands can be redirected to files.

Remark

Due to state and flow variables, set of states in Dicky'logics are replaced by set of configurations.

Acheck

Predefine properties

In addition to Dicky's constants $\emptyset$, `any_s`, `any_t` and `initial`, acheck computes these properties for each nodes.

- `epsilon` : set of transitions where each component does ϵ.
- `self` : set of transitions where target configuration is equal to source configuration.
- `self_epsilon` : define by `epsilon` $\cap$ `self`.
- `not_deterministic` : set of non-determinitic transitions, more formely define as the set $\{(s, e, t_1) \in E \mid \exists t_2 \in V, (s, e, t_2) \in E\}$.

Acheck

User's define properties

```
with Switch, CircuitV1, Scheduler do
 deadlock      := any_s - src(any_t - self_epsilon);
 notSCC        := any_t - loop(any_t, any_t);
done
with CircuitV1, CircuitV2, CircuitV1_OK do
 bug           := [L.on & ~L.ok];
 notControl    := (label G.failure | label L.reaction
                   | epsilon) - self_epsilon;
 // IR means infinite reactions
 IR := loop(notControl, notControl);
done
with SchedulerRandom, SchedulerPriority, Scheduler do
 bug := rsrc(tgt(label PJ2.put)) & label PJ3.get;
done
```

Acheck

acheck operators

In addition to Dicky's operators, acheck implements :

- `trace`(S_1, T, S_2) is a set of transitions (not a logical formula) representing one of the shortest path from S_1 to S_2 using only T transitions.
- `project`$(S, T,$ `'aNewNodeName', booleanValue`$)$ or `project`$(S, T,$ `'aNewNodeName', booleanValue, aNode`$)$ builts, with all transitions in T having their origin in S, a new AltaRica node that respect the `aNode` declaration

`trace` is very usefull to understand counter examples and `project` to built controller of systems.

Acheck

acheck output commands

```
with Switch, CircuitV1, Scheduler do
 show(all)          >  '$NODENAME.prop';
 test(deadlock,0)   >  '$NODENAME.res';
 test(notSCC,0)     >> '$NODENAME.res';
done
with CircuitV1, CircuitV2, CircuitV1_OK do
 quot()             >  '$NODENAME.dot';
 tr_IR  := trace(initial, any_t, src(IR));
 ce_IR  := reach(src(tr_IR), tr_IR | IR);
 dot(ce_IR, tr_IR | IR) > '$NODENAME-IR.dot';
 show(tr_IR, ce_IR)     >> '$NODENAME.prop';
done
```

Outline

The AltaRica model of calculus

AltaRica constraint automata

- $\vec{s}$: a finite set of state variables,
- $\vec{f}$: a finite set of flow variables,
- E : a finite set of events,
- T : a set of transitions

$$G_i(\vec{s},\vec{f}) \xrightarrow{e_i} \vec{s} := \sigma_i(\vec{s},\vec{f})$$

  $G_i(\vec{s},\vec{f})$ is a guard (ie a boolean formula) and $e_i \in E$,
- $A(\vec{s},\vec{f})$: an assertion (invariant),
- $\prec_E$: a partial order on E to define priorities.

Notation : $\vec{v}$ represents all `parent` and `public` variables of AC.

Semantic of guarded transitions

Post condition can be removed

$AC^1 =< \vec{s},\vec{f},E,T^1,A(\vec{s},\vec{f}),\prec_E >$
is equivalent to
$AC^2 =< \vec{s},\vec{f},E,T^2,A(\vec{s},\vec{f}),\prec_E >$
where
for each $t_i^1(G_i,e_i,\sigma_i) \in T^1,\ t_i^2(G_i^2,e_i,\sigma_i) \in T^2$
with
$G_i^2(\vec{s},\vec{f}) = G_i^1(\vec{s},\vec{f})\ \&\ \exists \vec{f}' A(\sigma_i(\vec{s},\vec{f}),\vec{f}')$

Semantic of guarded transitions

An example

```
node FIFO_V2
  state nbMessages : [0,2]; init nbMessages:=0;
  event put, get;
  trans
    true |- put -> nbMessages:=nbMessages+1;
    true |- get -> nbMessages:=nbMessages-1;
edon
```

is equivalent to

```
node FIFO_V1
  state nbMessages : [0,2]; init nbMessages:=0;
  event put, get;
  trans
    nbMessages<2 |- put -> nbMessages:=nbMessages+1;
```

Semantic of the partial order on events

The partial order can be removed

$AC^2 =< \vec{s}, \vec{f}, E, T^2, A(\vec{s}, \vec{f}), \prec_E >$
is equivalent to
$AC^3 =< \vec{s}, \vec{f}, E, T^3, A(\vec{s}, \vec{f}), \prec_E >$
where
for each $t_i^2(G_i, e_i, \sigma_i) \in T^2,\ t_i^3(G_i^3, e_i, \sigma_i) \in T^3$
with
$G_i^3(\vec{s}, \vec{f}) = G_i^2(\vec{s}, \vec{f})$ & $(\bigwedge_{e_i < e_j} \neg G_j^2(\vec{s}, \vec{f}))$

Semantic of the partial order on events

An example

```
node SchedulerSimple
  state  nb1, nb2 : [0,2];
  event  put1, put2; get1 > get2;
  trans  true |- put1, put2, get1, get2 -> ;
edon
```

is equivalent to

```
node SchedulerPrioritySimple
  state  nb1, nb2 : [0,2];
  event  put1, put2, get1, get2;
  trans  nb1 < 2 |- put1 -> ;
         nb2 < 2 |- put2 -> ;
         nb1 > 0 |- get1 -> ;
         nb2 > 0 & ~(nb1 > 0) |- get2 -> ;
edon
```

Semantic of a hierarchy $AC = (AC0, (AC1^1, AC2^1))$

Semantic of leaves

$AC =< AC1^1, AC2^1, \vec{s0}, \vec{f0}, E0, T0, A(\vec{s0}, \vec{f0}, \vec{v1}, \vec{v2}), \prec_{E0}, Sync >$
is equivalent to
$AC =< AC1^3, AC2^3, \vec{s0}, \vec{f0}, E0, T0, A(\vec{s0}, \vec{f0}, \vec{v1}, \vec{v2}), \prec_{E0}, Sync >$
by computing the semantic of each leaves.

Semantic of a hierarchy $AC = (AC0, (AC1^1, AC2^1))$

Variables and Assertion

$$\begin{aligned}
\vec{s} =& \vec{s0} \cup \vec{s1} \cup \vec{s2} \\
\vec{f} =& \vec{f0} \cup \vec{f1} \cup \vec{f2} \\
\vec{v} =& \vec{s0}, \vec{f0} : \{parent, public\} \\
& \cup\ \vec{s1}, \vec{f1} : \{public\} \\
& \cup\ \vec{s2}, \vec{f2} : \{public\} \\
A(\vec{s}, \vec{f}) =& A(\vec{s0}, \vec{f0}, \vec{v1}, \vec{v2}) \\
& \&\ A(\vec{s1}, \vec{f1}) \\
& \&\ A(\vec{s2}, \vec{f2})
\end{aligned}$$

Semantic of a hierarchy $AC = (AC0, (AC1^1, AC2^1))$

Semantic of broadcast

$Sync \subseteq (E0 \cup \{\epsilon\} \cup E0?) \times (E1 \cup \{\epsilon\} \cup E1?) \times (E2 \cup \{\epsilon\} \cup E2?)$
is equivalent to
$Sync^1 \subseteq (E0 \cup \{\epsilon\}) \times (E1 \cup \{\epsilon\}) \times (E2 \cup \{\epsilon\})$ and $\prec_{E^{sync}}$

Semantic of a hierarchy $AC = (AC0, (AC1^1, AC2^1))$

Events and partial order

$$E0^{sync} = E0 - \pi(Sync^1, 1)$$
$$E1^{sync} = E1 - \pi(Sync^1, 2)$$
$$E2^{sync} = E2 - \pi(Sync^1, 3)$$

$$\begin{aligned} E = & Sync^1 \\ & \cup (E0^{sync} \times \{AC1.\epsilon\} \times \{AC2.\epsilon\}) \\ & \cup (\{\epsilon\} \times E1^{sync} : parent \times \{AC2.\epsilon\}) \\ & \cup (E1^{sync} : public \times E1^{sync} : public \times \{AC2.\epsilon\}) \\ & \cup (\{\epsilon\} \times \{AC1.\epsilon\} \times E2^{sync} : parent) \\ & \cup (E2^{sync} : public \times \{AC1.\epsilon\} \times E2^{sync} : public) \end{aligned}$$

Semantic of a hierarchy $AC = (AC0, (AC1^1, AC2^1))$

Transitions

T is the set of

$$AC0.G_i \ \& \ AC1.G_j^3 \ \& \ AC2.G_k^3 \xrightarrow{<AC0.e_i, <AC1.e_j, AC2.e_k>>} \vec{s} := (\sigma_i(\vec{s0}, \vec{f0}), \sigma_j(\vec{s1}, \vec{f1}), \sigma_k(\vec{s2}, \vec{f2}))$$

for all $< AC0.e_i, < AC1.e_j, AC2.e_k >> \in E$

Semantic of a hierarchy $AC = (AC0, (AC1^1, AC2^1))$

Flatten semantic

$AC =< AC1^1, AC2^1, \vec{s0}, \vec{f0}, E0, T0, A(\vec{s0}, \vec{f0}, \vec{v1}, \vec{v2}), \prec_{E0}, Sync >$
After computing all these steps
$AC =< \vec{s}, \vec{f}, E, T, A(\vec{s}, \vec{f}), \prec_E >$
Recursively from bottom to top of the hierarchy.

Outline

The process

The first model

1. Identification of all basic components by a top-down analysis.
2. Choice between functionnal or architectural design.
3. Choice between open or close system.
4. Built of the hierarchy from bottom to top.

The process

Validation of the first model

1. Topology of the reachability graph (deadlock, SCC, ...)
2. Properties depending of the application (No infinite reaction, ...).
3. All events are usefull.
4. Visualisation of small components.
5. Simulation.

Step 1 and 2 must be repeat as long the model is not a valid one.

The process

Verification of the model

1. Specification has to be write as a list of logical properties.
2. For each property which is not satisfy, a counter example as small as possible must be compute.

Step 1, 2 and 3 must be repeat as long the model doesn't satisfy all its requirements.