Software Engineering

GINF41E7

Olivier Gruber

Lydie du Bousquet

Frédéric Lang

Software Engineering – Week 11
Verification using formal methods

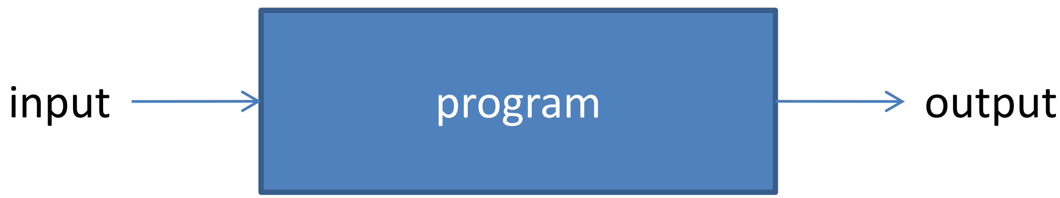
Part II:
Checking concurrent programs

Summary of week 10

- Sequential programs can be proven correct
- This requires to:
  - describe what programs should do using logic properties called **assertions**
  - derive (using Hoare logic) implications that must be proven, called **proof obligations**
  - Prove the proof obligations using pencil-paper or semi-automated tools
- Tool support exist: e.g., Spec#

Transformational program

Last week, we considered **transformational programs**

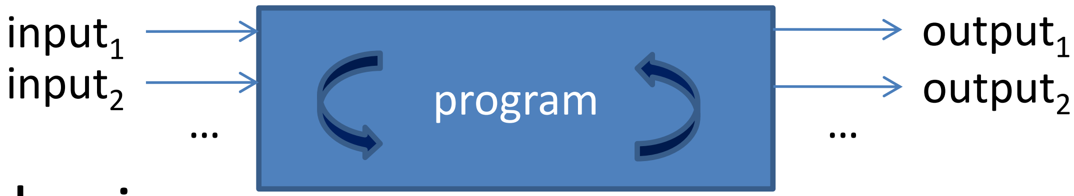


- Sequential behaviour
- Non-termination is an error
- Compute an output in function of an input

  output = f (input)

Reactive program

This week we consider **reactive programs**



- Cyclic behaviour
- Termination is an error
- Read inputs and respond by outputs

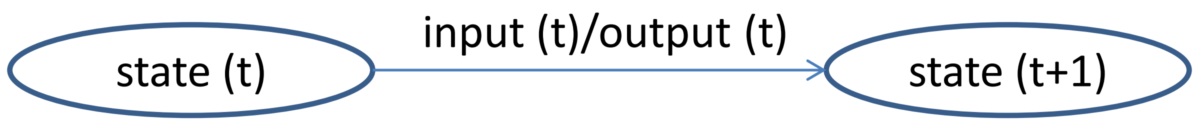
Examples

Graphical user interfaces, Unix daemons, audio/video decoders, device drivers, telecommunication protocols, plant controllers, airplane autopilots, etc.

Characteristics of reactive programs

- A same input may produce different outputs if read at different instants
  Example: double click in GUI
- Inputs at the current instant are not sufficient to compute the output
- One must memorize something about the previous inputs

Characteristics of reactive programs



- Notion of state (memory)

  state (t): state of the program at instant t
  summary of the program history which will be useful for the future

- Outputs and current state

  output (t) = f (input (t), state (t))

  state (t+1) = g (input (t), state (t))

- Notion of transition between states

$\Longrightarrow$ automaton

Principles of reactive programs

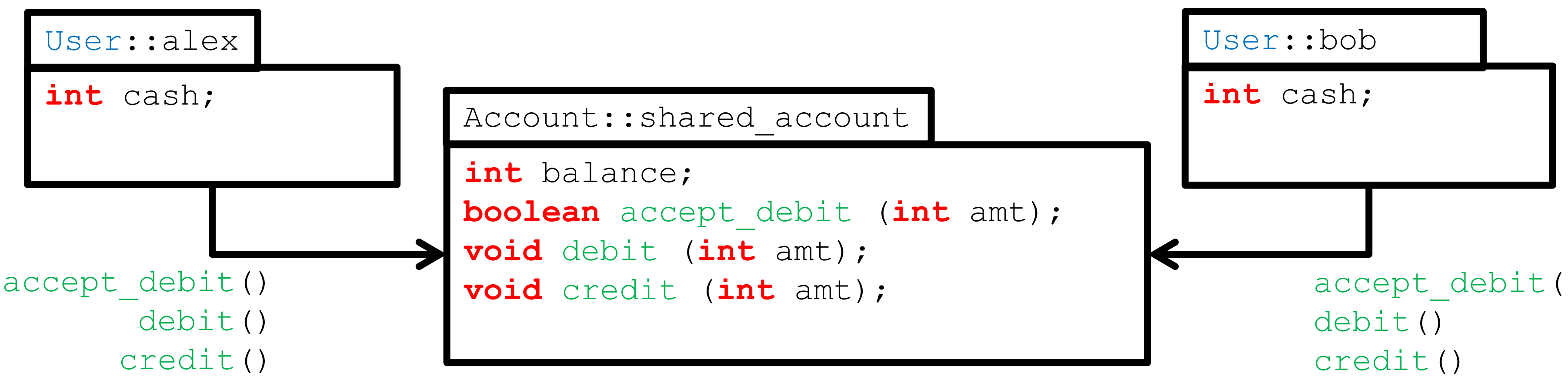
The **modular development of reactive programs** involves the following notions:

- **Concurrency**

  Simultaneous execution of several reactive components (processes) in competition to access common resources
- **Communication**
  Data exchange (message sending or variable sharing) between processes
- **Synchronisation**
  Rendezvous between processes or preemption
- **Cooperation**
  Collaboration between processes to achieve a common goal

Java example: shared bank account

- Multithreaded Java program: Bank account shared between two users, Alex and Bob
- Users interact with account using remote Java methods accept_debit(), debit() and credit()
- **Requirement**: Balance of account should never get negative



The Account class

```
class Account extends Thread {
  int bal;

  Account (int n) { bal = n; }

  public boolean accept_debit (int n) {
    return ((n > 0) && (n <= bal));
  }

  public void debit (int n) { bal = bal - n; }

  public void credit (int n) { bal = bal + n; }

  public void run () {
    while (true) {
      if (bal < 0) {
        System.out.println ("Negative balance");
        System.exit (1);
      }
    }
  }
}
```

The User class

```
class User extends Thread {
  String name; Account acc; int cash;

  User (String n, Account a) { name = n; acc = a; cash = 0; }

  private void get_cash (int amt) {
    boolean b = acc.accept_debit (amt); // to avoid negative balance
    if (b) { acc.debit (amt); cash = cash + amt; }
  }

  private void deposit_cash (int amt) {
    if (cash >= amt) { acc.credit (amt); cash = cash - amt; }
  }

  public void run () {
    Random rand = new Random();
    int amt;
    boolean b;
    while (true) { // simulates arbitrary user behaviour
      amt = (Math.abs (random.nextInt ()) % 5) + 1; // any value in 1..5
      b = random.nextBoolean ();
      if (b) { get_cash (amt); } else { deposit_cash (amt); }
    }
  }
}
```

The Main class

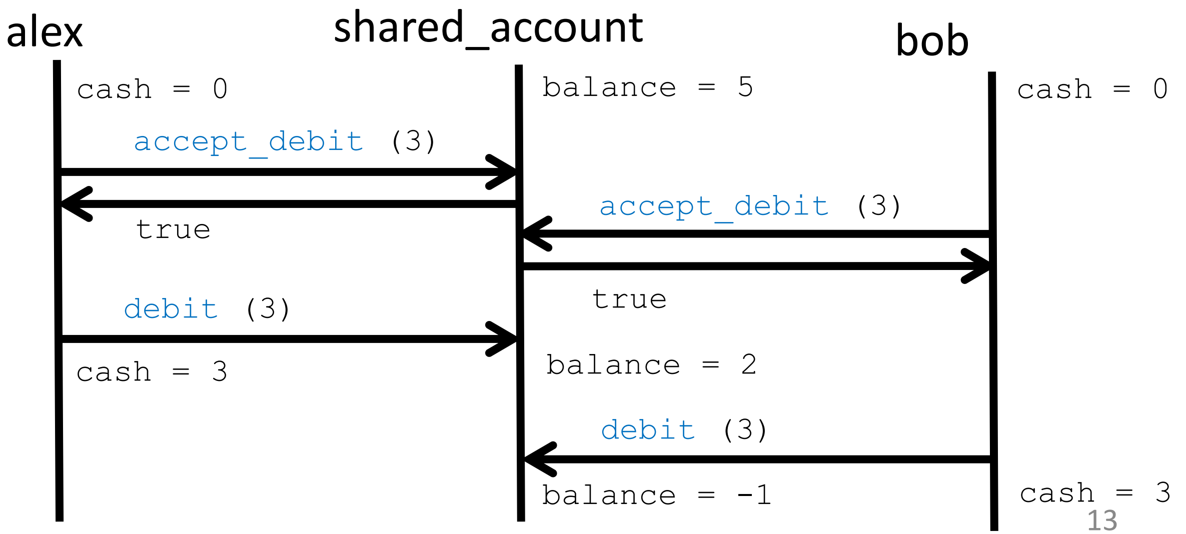
```
public class Main {

  public static void main (String[] args) {
    Account shared_account;
    User alex, bob;
    int n;

    // creating threads
    shared_account = new Account (5);
    alex = new User (« Alex », shared_account);
    bob = new User (« Bob », shared_account);

    // starting threads
    shared_account.start ();
    alex.start ();
    bob.start ();
  }
}
```

Is this program correct?

- Let’s test it!

- Program fastly exits with the message:
  Negative balance

- Why ?



Verifying reactive programs

- Testing techniques suffer the same limitations as for transformational programs
- Such errors are hard to detect!

  Could we have found it before testing?
- Proof techniques are not well-adapted to handle **concurrency**
- Alternative techniques are needed, namely **model-based verification**

Principles of model-based verification

- Write a **formal model** that represents the **temporal evolution** (**behaviour**) of the program
- Write a specification of the **expected properties**
  Example: negative balance cannot be reached
- **Check the properties** on the model using an automated software tools

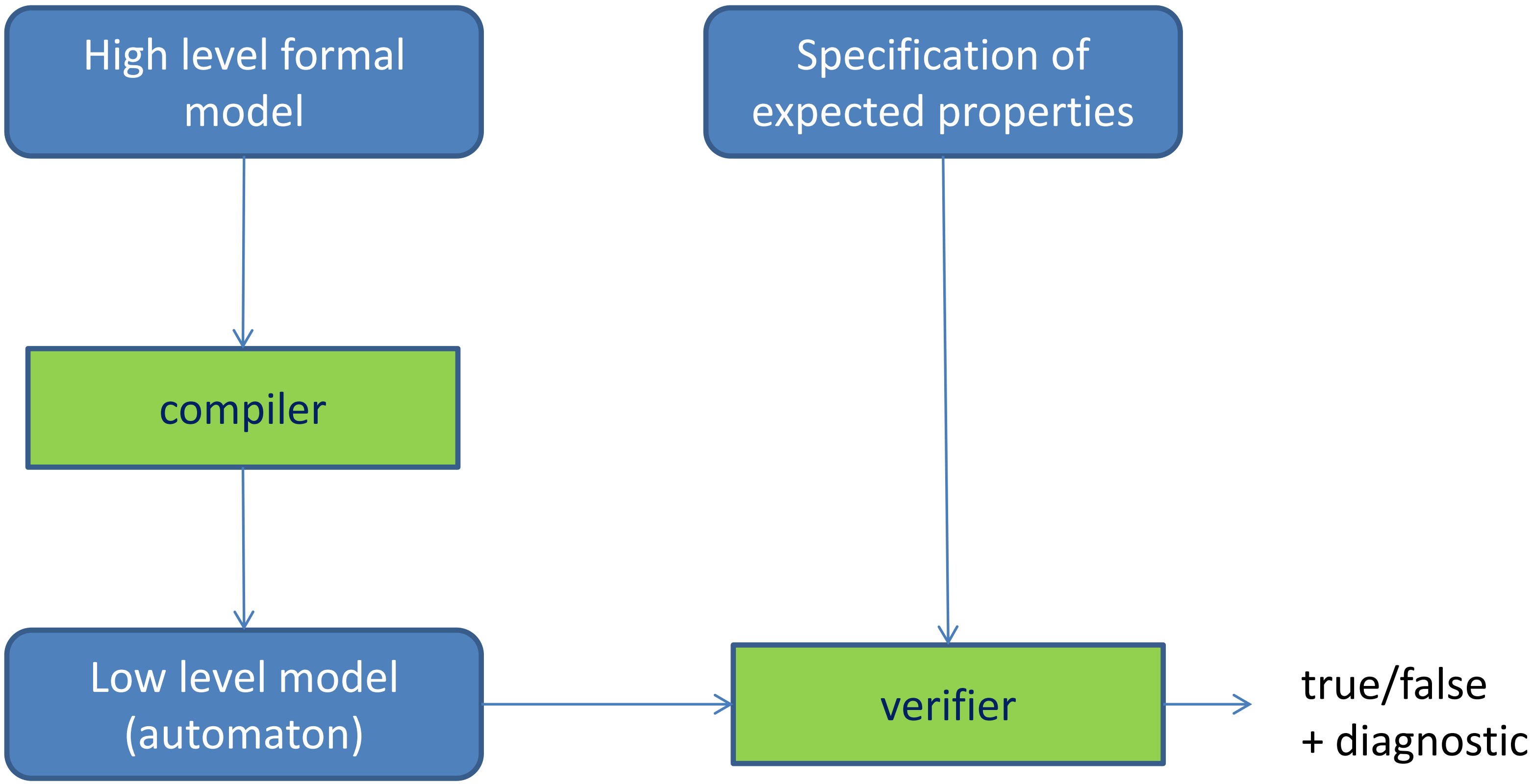
Model based verification

- The model is an **automaton** (e.g., *Labelled Transition System*) or a higher-level description that can be compiled to an automaton according to its **formal semantics**

- Verification consists in **traversing the states and transitions** of this automaton, guided by the property

- Appropriate techniques must be used to fight against **state space explosion**

Model based verification



Checking programs vs. models

Checking programs directly would be great, but no satisfactory general solution exists:

- Concurrent programming languages with **formal semantics** are rare
- **Abstraction** (hiding unnecessary details) is needed to avoid **verification complexity**
- The abstraction depends on the problem, which requires **human expertise** and hinders automation

Advantages of a formal model

Formal models have several **advantages**:

- They are abstract and thus allow a **focus on important design decisions** rather than on unimportant details
- They are formal and thus **nonambiguous**, as compared to diagrams and descriptions in natural language
- Beyond verification, they can find **other usages** all along the software lifecycle:
  - Documentation
  - Prototyping or generation of code skeletons
  - Automated generation of tests, oracle, …

MODELING CONCURRENT PROGRAMS AS AUTOMATA

LTS (Labelled Transition System)

- An LTS is a particular (and simple) type of automaton used in this lecture to illustrate the principles of model based verification
- Formally, an LTS is a 4-tuple (S, A, $\rightarrow$, q) such that:
  - S is a set of **states**
  - A is a set of **actions**
  - $\rightarrow \subseteq S \times A \times S$ is a set of **transitions** between states, labelled by actions
  - $q \in S$ is a particular state called the **initial state**

Graphical representation of an LTS

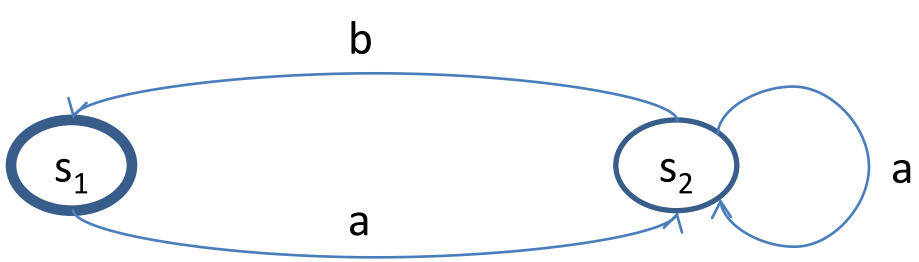
- In general, LTSs will be represented **graphically**
- **Example**: $(S, A, \rightarrow, s_1)$

  where $S = \{ s_1, s_2 \}$

  $A = \{ a, b \}$, and

  $\rightarrow = \{ (s_1, a, s_2), (s_2, b, s_1), (s_2, a, s_2) \}$

  will be represented graphically as follows:



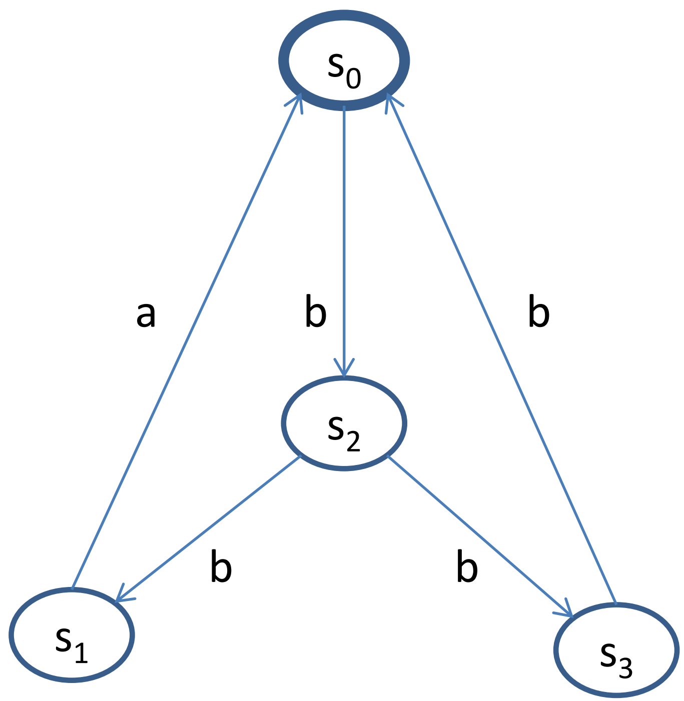
Exercise

Draw the LTS $(S, A, \rightarrow, s_0)$ where:

- $S = \{s_0, \ldots, s_6\}$
- $A = \{a, b, c, d\}$
- $T = \{ (s_0, d, s_2), (s_0, d, s_3), (s_0, d, s_6),$
  $(s_1, b, s_4), (s_1, c, s_5),$
  $(s_2, d, s_2), (s_2, a, s_4),$
  $(s_3, a, s_5),$
  $(s_6, d, s_1), (s_6, d, s_3) \}$

Exercise

- Describe the following LTS as a 4-tuple



Bank example: LTSs

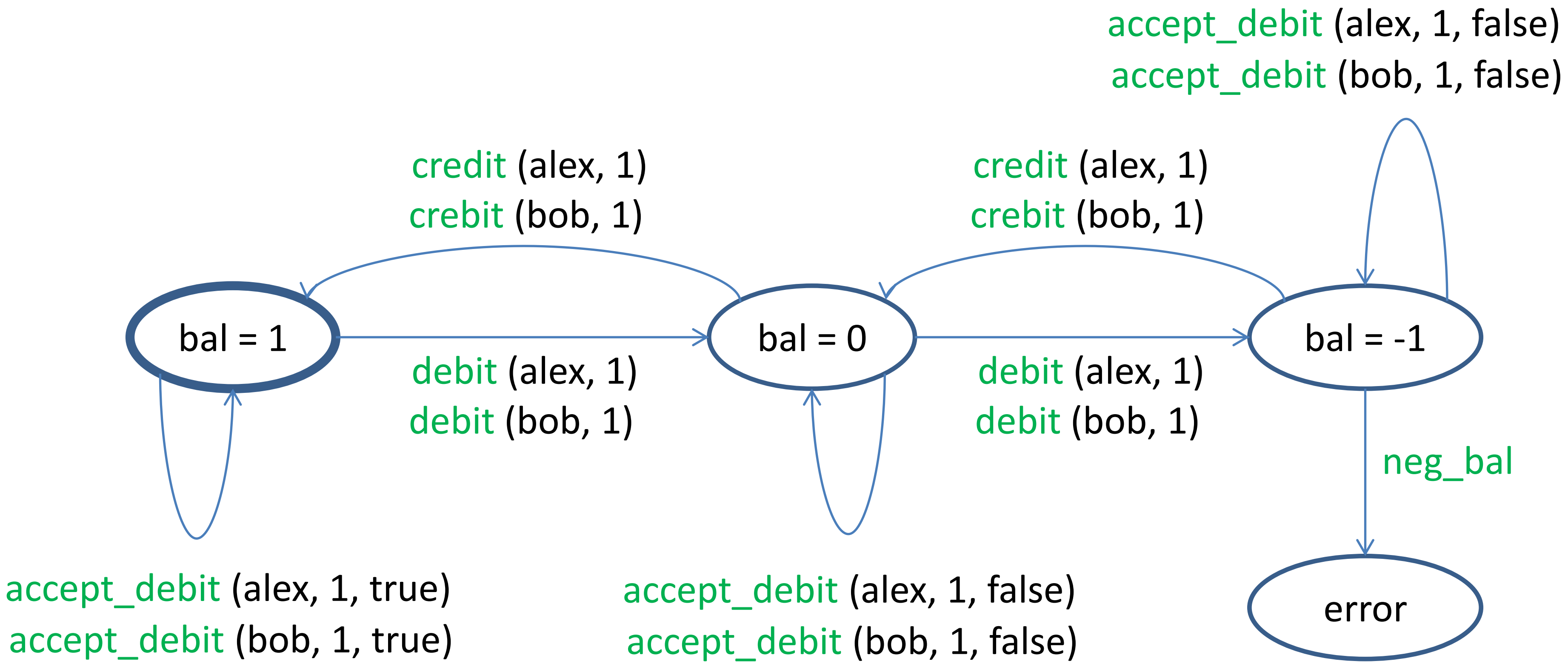
- Each thread of the bank example can be modelled as an LTS
- We choose to model each remote method invocation as an action
  - Each method call acc.m (x) from user u is modelled as:
    - If the method call does not return a result: m (u, x)
    - If the method call returns a result r: m (u, x, r)
    - Remark: acc is not written in actions because it is the only account in the program
  - Negative balance is modelled by an action neg_bal

Bank example: constraints on data

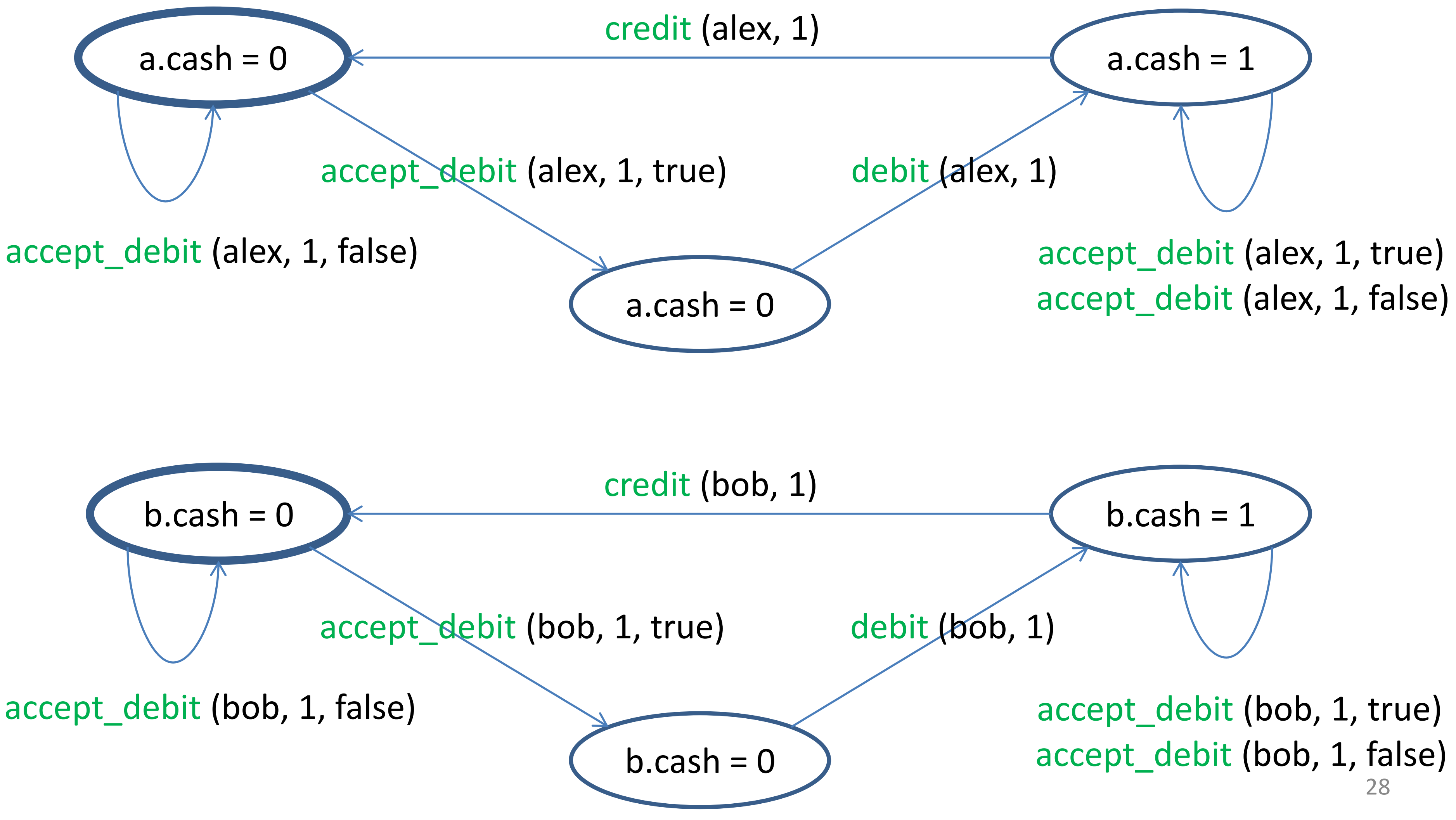
For LTSs to fit in a slide:

- Amount of debit/credit is set to 1
- Account balance is initially set to 1
- User cash is initially set to 0
- Both account balance and user cash are constrained in the interval [-1, 1]

Bank example: LTS of account



Bank example: LTSs of Alex and Bob



Parallel composition of LTSs

- Concurrent behaviours also describe a behaviour
- This is represented by an operation called **product**, which takes two LTSs and returns the LTS of their parallel composition
- In general, concurrent behaviours are not independent and must **interact**
- The interaction primitive between LTSs is **synchronisation on actions** (a.k.a. **rendezvous**)

Formal definition of product of LTSs

Given:

- two LTSs $P_1 = (S_1, A_1, \rightarrow_1, q_1)$ and $P_2 = (S_2, A_2, \rightarrow_2, q_2)$
- a set of actions A

we write $P_1 \otimes_A P_2$ the **product of P_1 and P_2 with synchronisation on A**, defined as the LTS

$$(S_1 \times S_2, A_1 \cup A_2, \rightarrow, (q_1, q_2))$$

where $\rightarrow$ is defined as follows:

$(s_1, s_2) -a\rightarrow (s_1', s_2')$ if and only if either:

- $s_1 -a\rightarrow_1 s_1'$ and $s_2 = s_2'$ and $a \notin A$
- $s_1 = s_1'$ and $s_2 -a\rightarrow_2 s_2'$ and $a \notin A$
- $s_1 -a\rightarrow_1 s_1'$ and $s_2 -a\rightarrow_2 s_2'$ and $a \in A$

Bank example

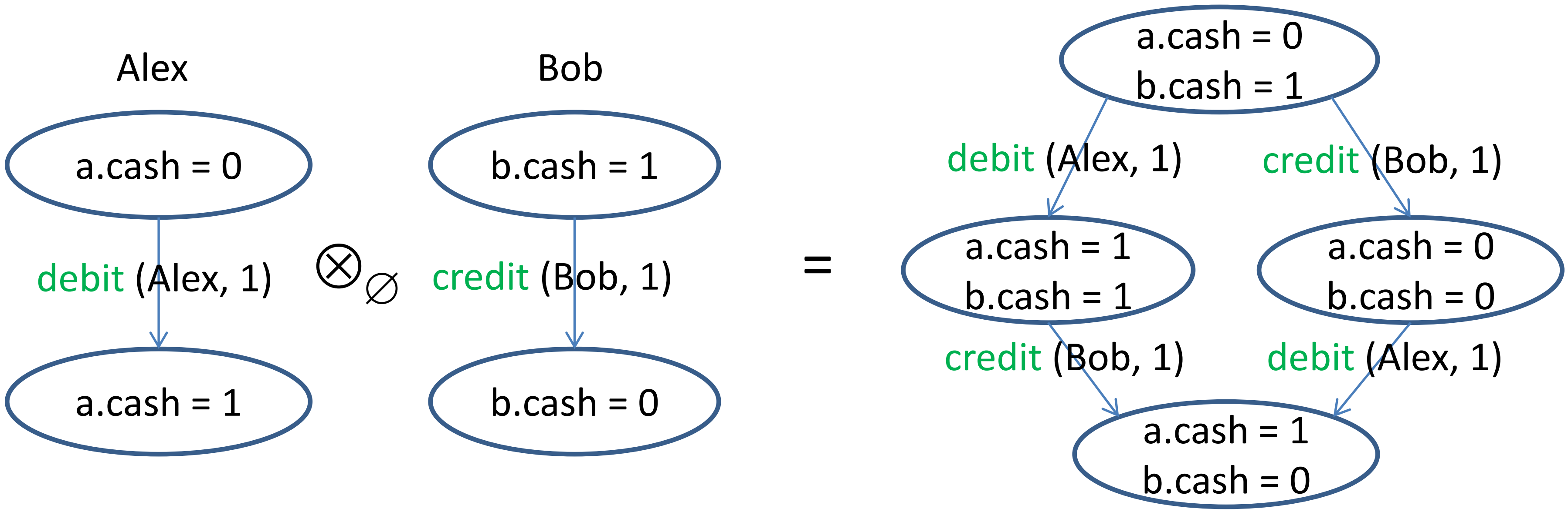
The bank example is modelled as a product of LTSs such that for any method m:

- Account and Alex synchronise on all actions of the form « m (alex, …) » (let **A** be this set)
- Account and Bob synchronise on all actions of the form « m (bob, …) » (let **B** be this set)
- Alex and Bob never synchronise

$$\text{Account} \otimes_{\mathbf{A} \cup \mathbf{B}} (\text{Alex} \otimes_{\varnothing} \text{Bob})$$

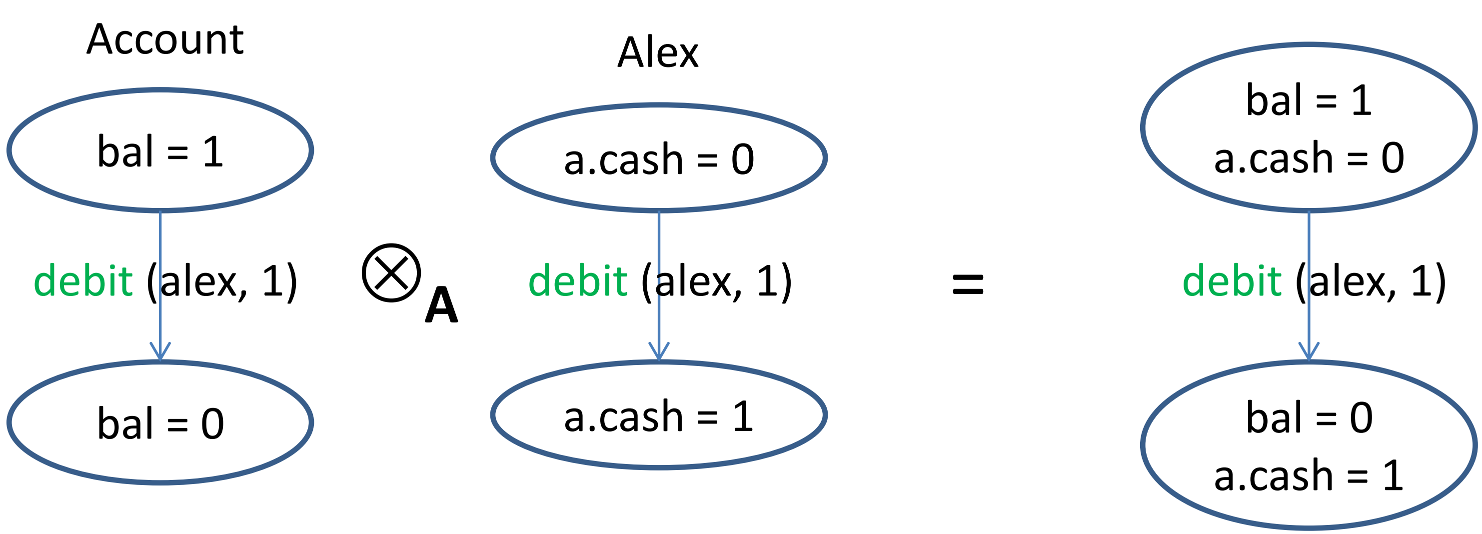
Non synchronising actions

- Non synchronising actions do not execute simultaneously: they **interleave** in the product
- Example with Alex and Bob



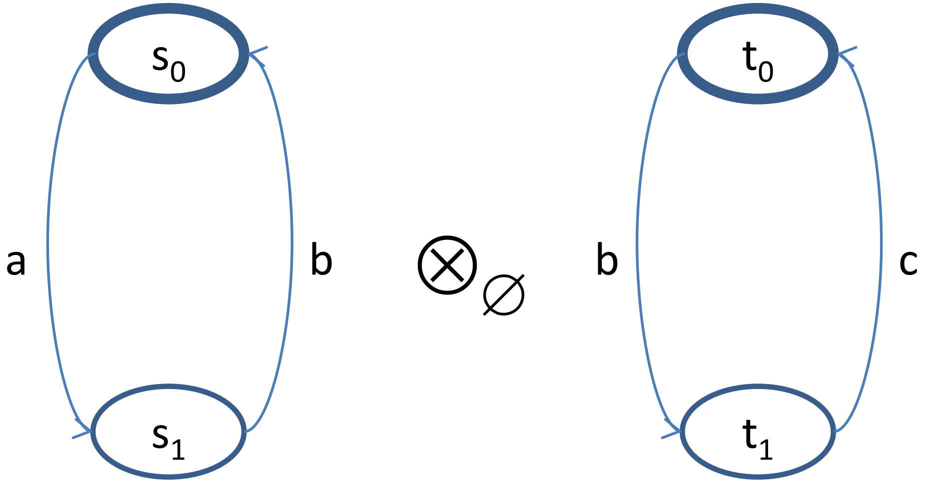
Synchronising actions

- Synchronising actions execute together at once
- Example: remote method invocation



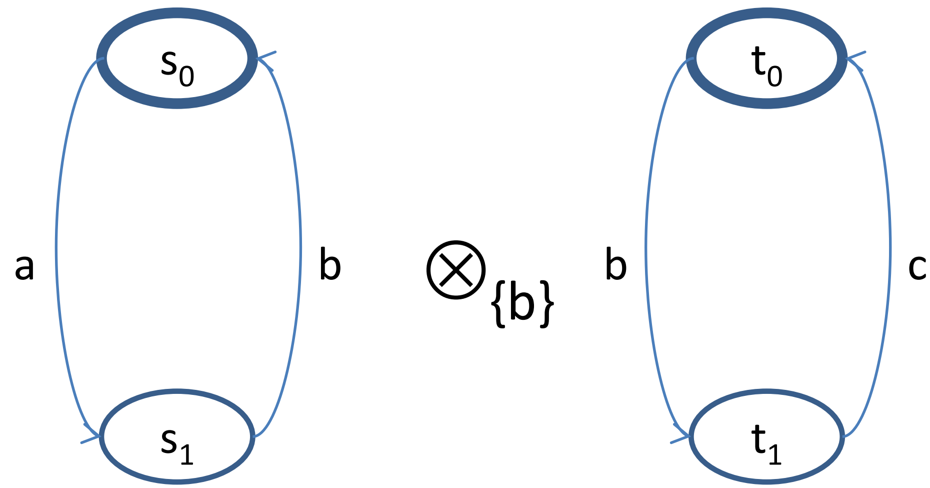
Exercise (1/2)

Draw the result of the following product of LTSs



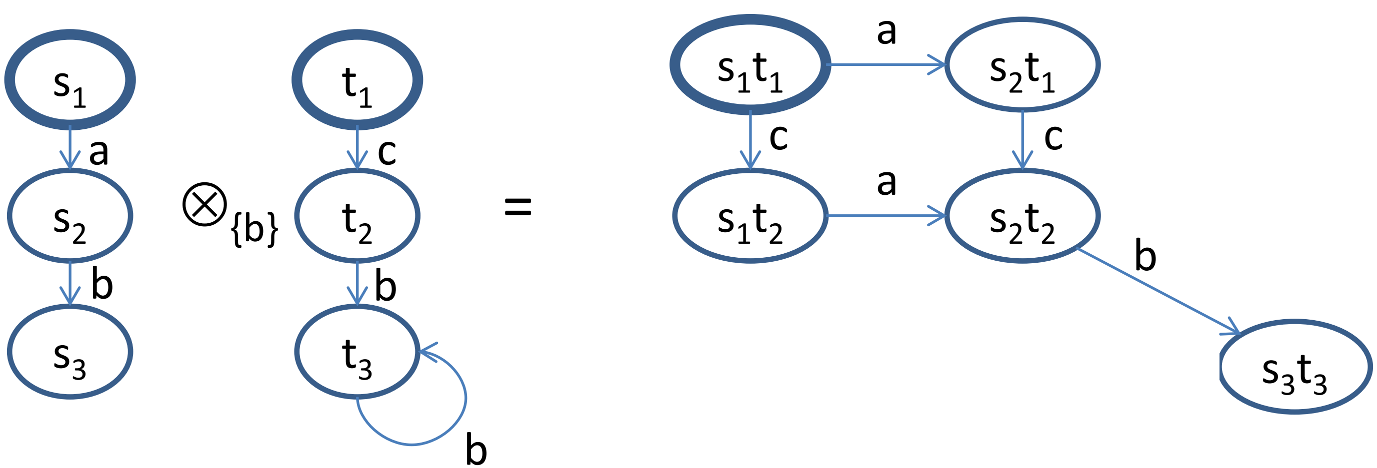
Exercise (2/2)

Draw the result of the following product of LTSs



Reachable product of LTSs

- Definition of product includes states that are **not reachable** from the initial state
- In general, we **restrict the product to the reachable part**, i.e., unreachable states are ignored
- **Example**:



Atomicity of actions

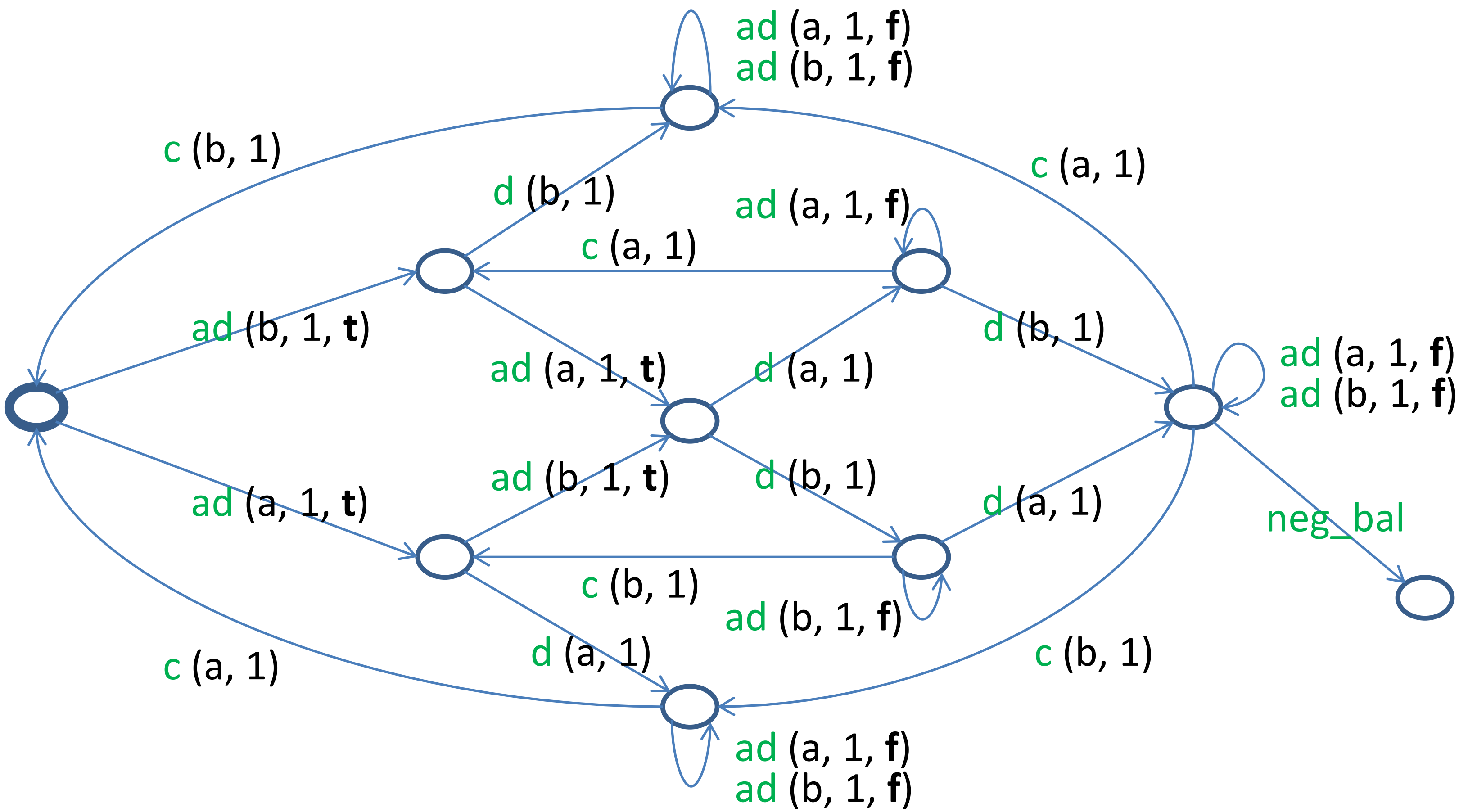
- LTS actions are atomic: they are instantaneous and not simultaneous unless synchronised
- In the bank example, each **remote method invocation** is modelled by a single atomic action
- This is ok in this example because :
  - Remote methods are not invoked from remote methods
  - Remote methods do not share common objects
- Different modelling would have been necessary otherwise (e.g., split into call and return actions)
- **Expertise** is required to model appropriately

Bank example: product of LTSs

Account $\otimes_{\mathbf{A} \cup \mathbf{B}}$ (Alex $\otimes_{\varnothing}$ Bob)

contents of states have been removed and actions are abbreviated as follows:

ad = accept_debit, d = debit, c = credit, a = alex, b = bob, **t** = true, **f** = false



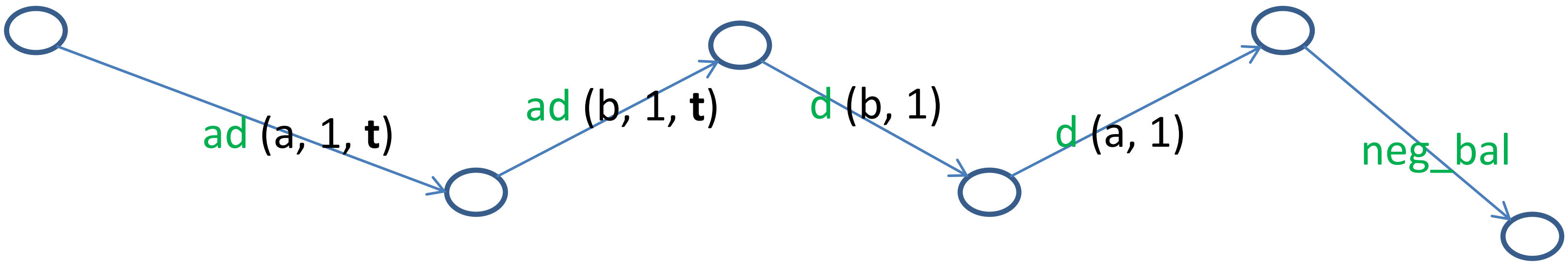
VERIFICATION OF PROPERTIES

Reachability properties

- **Deadlocks**: is there a reachable state from which no action can be executed?
- **Reachability** of an action: is there a reachable state from which some action (e.g., error action) can be executed?
  Example: reachability of neg_bal
- Those properties are easy to check from the product: **reachability analysis**

Reachability analysis for the bank example (1/2)

- Action neg_bal is indeed reachable!
- A sequence to this action (diagnostic) can be extracted automatically from the model:



Reachability analysis for the bank example (2/2)

- This sequence explains the problem:
  - Action ad (a, 1, **t**): Alex asks account whether debit is possible, response is true because bal = 1
  - Action ad (b, 1, **t**): Bob asks account whether debit is possible, response is true because bal = 1
  - Action d (b, 1): Bob debits the account, bal = 0
  - Action d (a, 1): Alex debits the account, bal = -1
- **Correction**: make the contents of the get_cash() method atomic (see courses on distributed systems)

Properties specified as regular expressions

- More elaborate properties: checking existence or absence of a **finite path matching a regular expression** on actions
- **Example**: Is there a path in which Alex debit twice without a debit acceptance in between?

  true* . 'd (a, 1)' . **not** 'ad (a, 1, **t**)'* . 'd (a, 1)'

  (answer is no)
- Reachability of action a is a special case: regular expression of the form **true*** . A

Checking regular expressions

- The problem of checking whether a path matches a given regular expression β can be **reduced to reachability**
  - Turn β to a regular automaton (see language theory) terminated by an action denoting acceptance
  - Compute a product between the regular automaton and the model
  - A path exists if and only if the acceptance action is reachable

Properties expressed using temporal logic

- Regular expressions are not sufficient to model all properties of interest
- **Example**: is any debit acceptance **necessarily** followed by a debit?
- **Temporal logics** introduce **modalities**, which enable to deal with notions such as **necessarily** and **possibly**

Example of temporal logic: PDL

- Extension of Hennessy-Milner logic to regular expressions proposed by Fischer and Ladner in 1979
- PDL formulas satisfy the following syntax:

  $\varphi ::= \textbf{true}$
  $\quad | \ \textbf{false}$
  $\quad | \ \varphi_1 \wedge \varphi_2$
  $\quad | \ \varphi_1 \vee \varphi_2$
  $\quad | \ \neg\varphi_0$
  $\quad | \ < \beta > \varphi_0$ *possibility*
  $\quad | \ [\ \beta\ ]\ \varphi_0$ *necessity*

  where β is a regular expression on actions

PDL modalities

- $< \beta > \varphi_0$ is true if there exists a path matching β that leads to a state satisfying φ_0
- $[\beta] \varphi_0$ is true if all paths matching β lead to states matching φ_0
- **Example**: the property « is any debit acceptance **necessarily** followed by a debit? » can be expressed by the formula

[**true*** . 'ad (a, 1, **t**)'] < 'd (a, 1)' > **true**

and

[**true*** . 'ad (b, 1, **t**)'] < 'd (b, 1)' > **true**

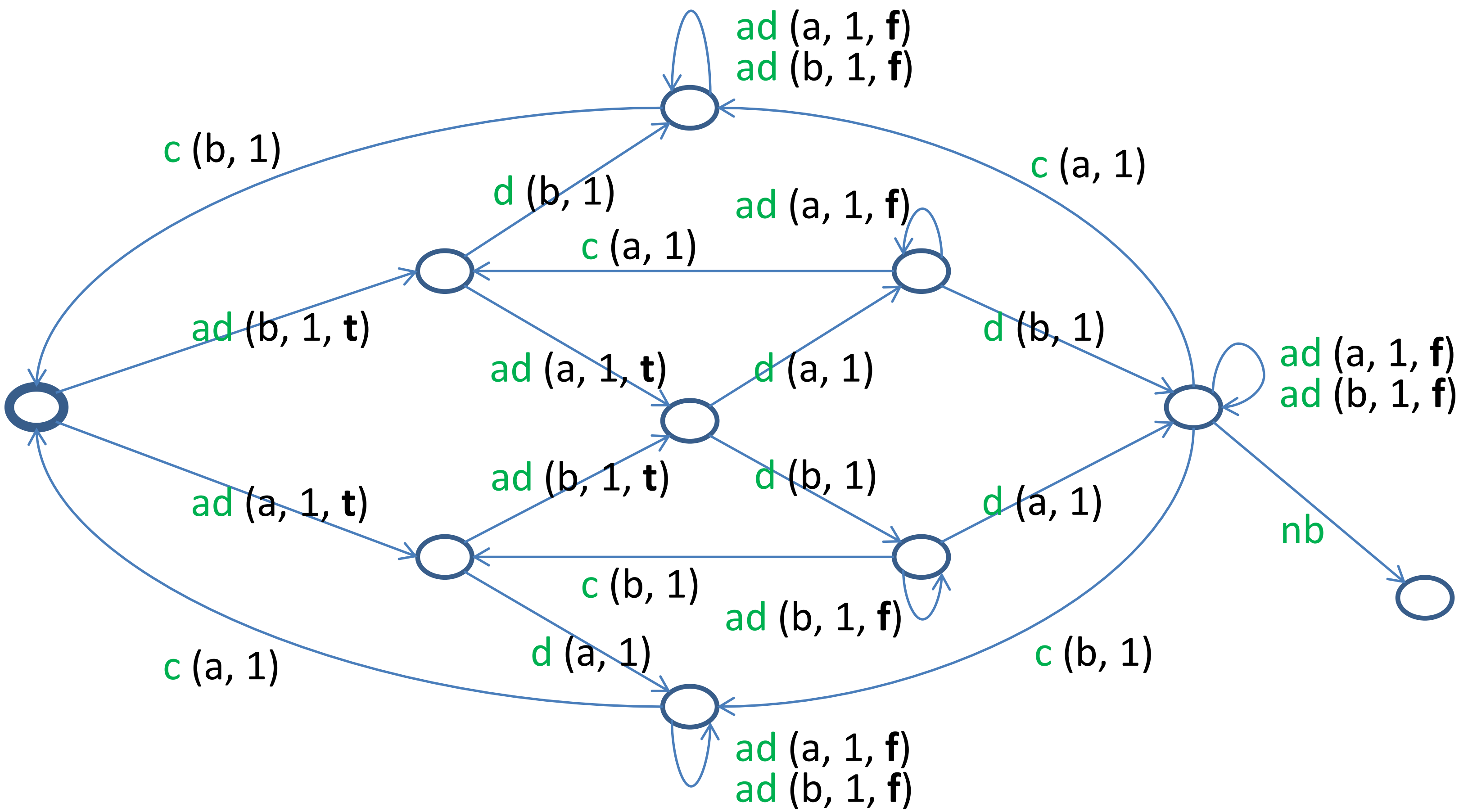
Examples: check the PDL formulas

< **true*** . nb > **true**

[¬(nb)*] < **true** > **true**

[**true*** . 'ad (a, 1, **t**)'] < **true*** . 'd (a, 1)' > **true**

< **true*** > (< 'd (a, 1)' > **true** ∧ < 'd (b, 1)' > **true**)



Remarks about PDL

- Checking PDL formulas is more complex than reachability (e.g., Boolean Equation System)
- Reachability is **a special case of PDL:**
  - **Existence of a path** β: $< \beta >$ **true**
  - **Deadlock:** [**true***] < **true** > **true**
- Other properties cannot be expressed in PDL
  Ex.: Path with unbounded number of debits?
- More expressive temporal logics exist to do so, e.g., the **modal mu-calculus** (Kozen, 1983)

FORMAL MODELLING IN A HIGH-LEVEL LANGUAGE

High-level modeling languages

- It is not convenient to model directly as an LTS
- Textual languages are more convenient:
  - Structured (modules, types, functions, …)
  - Appropriate to describe larger models
- The language must have **formal semantics**, which allows to automatically generate a low-level model (e.g., LTS) which will be checked

Example: The language LNT

- Developped by Inria/Convecs team
- Language structured in three parts: **modules**, **data** (types, functions), and **control** (behaviours)
- **Homogeneous** and **user-friendly** syntax:
  - Control part is a superset of the data part
  - Imperative programming style + features inspired from functional and algebraic programming
- **Formal operational semantics** in terms of **LTS**
- Supported by **verification tools** (CADP)

Bank example in LNT: types and functions

```
type User is Alex, Bob end type

function valid_amt (amt : int) : bool is
  return (amt > 0) and (amt ≤ 5)
end function
```

Bank example in LNT: Account process

```
process Account [accept_debit, debit, credit, neg_bal : any] is
  var bal, amt : int, res : bool in
    bal := 1;
    loop
      select
        accept_debit (?any User, ?amt, ?res)
          where valid_amt (amt) and (res == (amt ≤ bal))
      [] debit (?any User, ?amt) where valid_amt (amt);
         bal := bal – amt
      [] credit (?any User, ?amt) where valid_amt (amt);
         bal := bal + amt
      [] only if bal < 0 then neg_bal; stop end if
      end select
    end loop
  end var
end process
```

Bank example in LNT: User process

```
process User [accept_debit, debit, credit : any] (name : User) is
   var cash, amt, res : int in
      cash := 0;
      loop
         amt := any int where valid_amt (amt);
         select
            accept_debit (name, amt, ?res);
            if res then debit (name, amt); cash := cash + amt end if
         []  only if cash >= amt then
                credit (name, amt); cash := cash – amt
            end if
         end select
      end loop
   end var
end process
```

Bank example in LNT: parallel composition

```
process MAIN [accept_debit, debit, credit, neg_bal : any] is
   par accept_debit, debit, credit in
      Account [accept_debit, debit, credit, neg_bal]
   ||
      par
         User [accept_debit, debit, credit] (Alex)
      ||
         User [accept_debit, debit, credit] (Bob)
      end par
   end par
end process
```

State space explosion

- **Combinatorial blow up** of the state space/LTS (memory exhaustion)
- Factors of explosion:
  - Data
  - Asynchrony between parallel processes
- Guidelines to avoid explosion
  - Model at an **appropriate level of detail**
  - **Abstract/restrict the domains of data** as much as possible
  - **Limit the number of actions** to the minimum necessary
  - Use **state space reduction techniques** provided by the verification tools

Model based verification tools

Tools based on Labelled Transition Systems

- LTSA www.doc.ic.ac.uk/ltsa
- FDR3 www.cs.ox.ac.uk/projects/fdr
- mCRL2 www.mcrl2.org
- CADP cadp.inria.fr

Tools based on different low level models

- SPIN (Kripke structures) spinroot.com
- UPPAAL (timed automata) www.uppaal.com
- Tina (time Petri nets) projects.laas.fr/tina

COSTS AND BENEFITS OF THE USE OF FORMAL METHODS

Initial cost of formal methods

Formal methods have the **drawbacks** of any quality improvement process:

- Formal modeling has an **initial cost** higher than with traditional approaches
- The effort put in formal modelling does not necessarily impact **immediately** the final product delivered to the customer

True, but they also have **advantages**, besides enabling formal verification…

1. Better quality of specifications

- With formal methods, a better attention is put on the initial steps of the project
- Much better specifications are obtained, which will serve as a reference documentation for the project
- Thus, long term maintenance will be favored

2. Simplification of the coding step

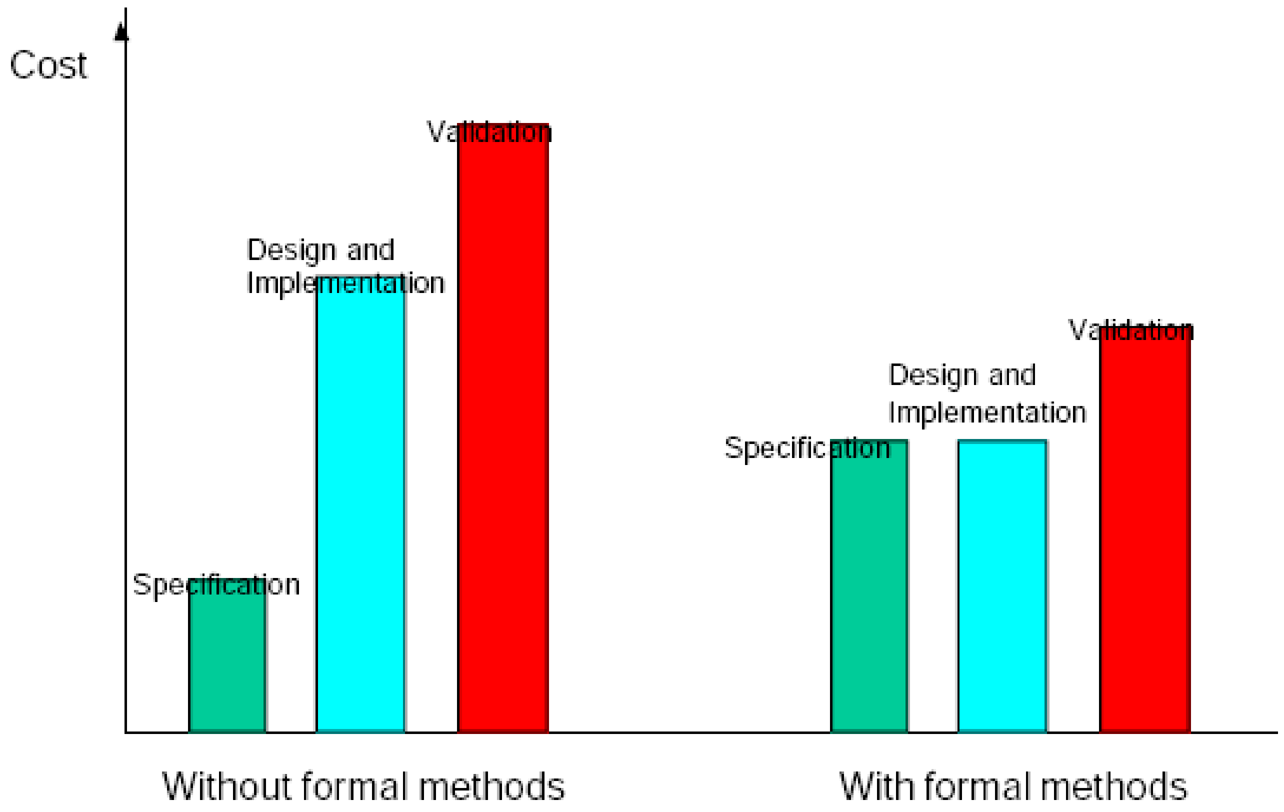
- With formal methods, the coding step is simpler, because the programmer knows precisely what (s)he must implement

- Ambiguities are eliminated: programmers cannot make mistakes caused by misinterpretation of the models they must implement

3. Early error detection

- **Reminder**: the later an error is found in the software lifecycle, the more expensive its correction
- The worst errors are those found once the product has been installed on the customer site…
- With formal methods, errors are found earlier, during the formal modeling step

New deal of efforts and costs



With formal methods, errors are found earlier
=> reduction of testing cost and duration

Other benefits

- **Test generation**: the formal model is used as oracle and as a reference to generate the test sequences
- **Rapid prototyping**: code skeletons are generated automatically from the formal model
- **Cosimulation**: the model is used to pilot the real program
  - The model sends inputs to the program and receives its outputs
  - Observers are put in action to detect any behavioural deviations between the system and the model

IMPACT OF FORMAL METHODS

A slow and difficult dissemination

- Formal methods are not yet massively used in the software industry

- As any (r)evolution, formal methods have been the subject of resistances

- Formal methods are still a young discipline (35 years) as compared to the theoretical and practical complexity of the problems they tackle

- The initial goals have been too ambitious, which has resulted in disappointment and distrust (cf. Bill Gates talking about "*Holy Grail*").

« For things like software verification, this has been the Holy Grail of computer science for many decades.

But now, in some very key areas, for example driver verification, we're building tools that can do actual proofs of the software and how it works in order to guarantee the reliability. »

Bill Gates, 2002

Difficulties inherent to formal methods (1/2)

- They require **competences in logic and mathematics** that not all programmers have

- They require a **learning effort** and more thinking

- They are **not a general method**: they cannot be applied to all aspects of systems, but only to their most complex parts

Difficulties inherent to formal methods (2/2)

- **Other simpler techniques** have enabled software quality improvement by finding more fastly the **easy-to-find errors**, which has reduced the interest for formal methods. But **« difficult » errors remain…**

- In the current economic competition, ***« time to market »*** is often **more important than quality**. Formal methods reduce the number of errors, but it is not clear whether they reduce or augment the time of development

Nevertheless…

- Formal methods are **more and more successful**

- They spread progressively in the most groundbreaking companies

- The concerns on quality (reliability, security, etc.) start to be the object of standards

Successes in hardware

- In hardware design companies, formal methods are now in the habit
- There are verification teams next to the development teams. For complex circuits, 70% of the effort is put on verification and testing
- Example : PSL (*Property Specification Language*) standard of the Accellera consortium
  Web : http://www.accellera.org (-> PSL)
- Designers (Intel, AMD, IBM, etc.) use formal verification tools. The biggest (IBM, Intel) even have their own laboratories, who design formal verification tools

Successes in software

- The B method used to design critical parts of railway systems
- The SPIN model checker (Bell Labs)

  http://spinroot.com/spin/whatispin.html#X
  - Flood control barriers in Rotterdam
  - The Lucent PathStar switch
  - NASA missions: Cassini, Mars, etc.
  - Medical device transmission protocols
- The CADP verification toolbox (INRIA)

  http://www.inrialpes.fr/vasy/cadp/case-studies

  171 case studies in various domains

Formal methods and standardisation

- Recent standards recommend to use formal methods to develop some classes of critical systems

- Example 1: the standard DO-178C (2011)
  Software Considerations in Airborne Systems and Equipment Certification
  http://en.wikipedia.org/wiki/DO-178C
- Example 2: the standard ISO 26262
  functional safety of road vehicles
  http://en.wikipedia.org/wiki/ISO_26262
- Example 3: the standard ISO/CEI 15408
  Common criteria
  http://en.wikipedia.org/wiki/Common_Criteria

Conclusion

- Formal methods can highly improve the usual practice in the specification and design steps (mostly based on natural languages and diagrams)
- They require advanced skills and thus concern prioritarily critical systems (avionics, nuclear, transport, circuit design, security, …)
- The cost of their usage can be compensated by gains on further steps (automated coding, validation, test, etc.). They can thus deeply modify the usual development cycles
- The formal method must be chosen in function of the problem type: sequentiel, concurrent, real time, etc.

Competence and Knowledge which will be evaluated

- be able to
  - Compute the **synchronised product** of two LTSs
  - Check for the **existence of a path matching a regular expression**
- know
  - The difficulty of **engineering concurrent reactive programs**
  - The principles and benefits of **model based verification** using **formal methods**