Software Engineering

Lydie du Bousquet

Frédéric Lang

Software Engineering – Verification using formal methods

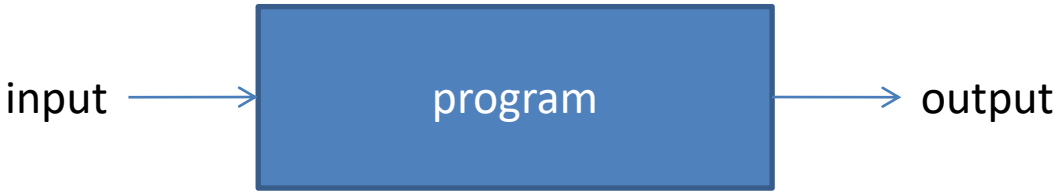
Part II: Reactive/concurrent programs

Summary of last week

- Transformational programs can be proven correct
- This requires to:
  - Specify the program using a language with mathematically-defined semantics
  - Specify the properties that the program should satisfy, e.g., using **assertions**
  - Possibly derive implications that must be proven, called **proof obligations**
  - Prove logical prperties using pencil-paper or (semi-) automated tools
- Tool support exists: e.g., B method, Spec#, …

Transformational program

Last week, we considered **transformational programs**

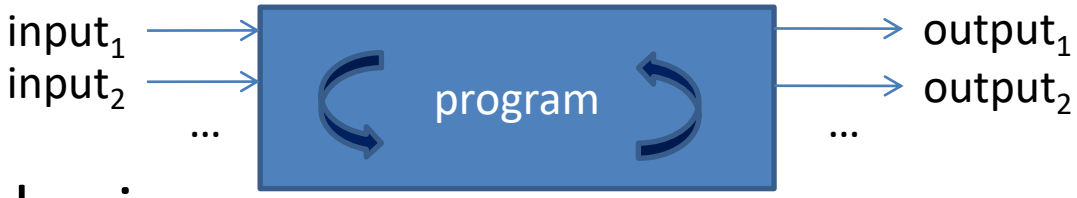


- Sequential behaviour
- Non-termination is an error
- Compute an output in function of an input

  output = f (input)

Reactive program

This week, we consider **reactive programs**



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

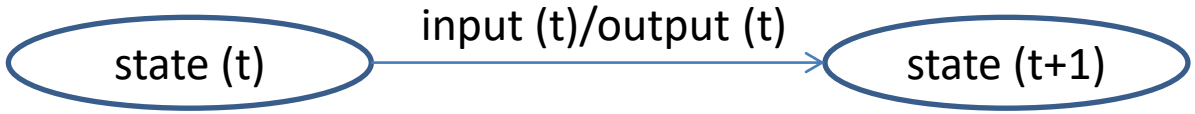
Examples

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

Characteristics of reactive programs

- The same input may produce different outputs if read at different instants
  Example: double click in a 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

$\Rightarrow$automaton

Principles of reactive programs

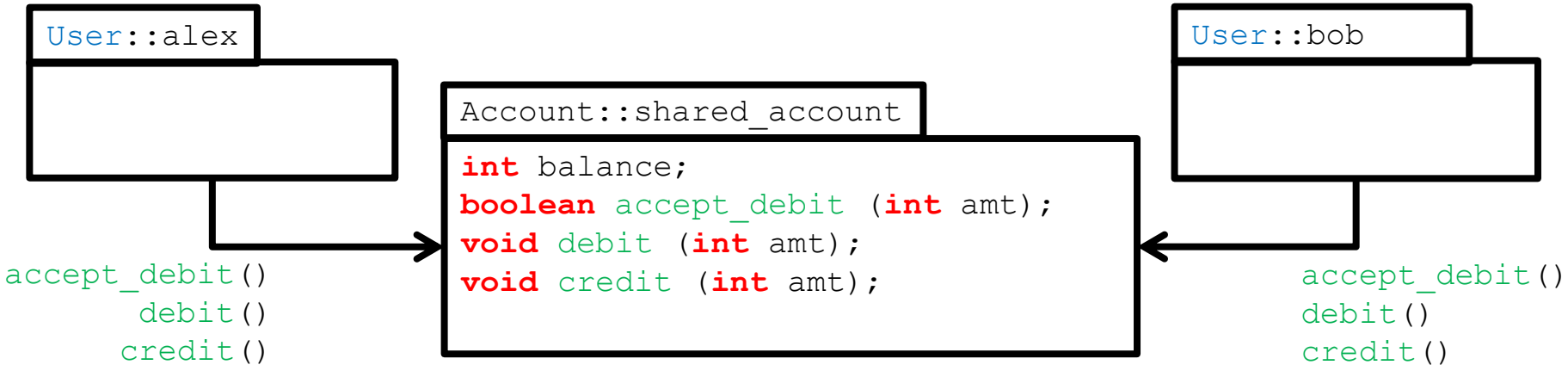
Modular decomposition of transformational program is mostly based on function composition

Reactive programs require a decomposition into parallel components (*processes* or *tasks*), involving:

- **Concurrency**: Simultaneous execution of the processes in competition to access common resources
- **Communication**: Data exchange between processes (message sending, shared variables, …)
- **Synchronisation** (rendezvous, preemption, …)
- **Cooperation**: Collaboration between processes to achieve a common goal

Example: shared bank account

- Design a bank account shared between 2 users, Alex and Bob
- **Requirement**: Balance of account should never get negative
- Example of implementation using Java: thread interactions using remote methods accept_debit(), debit() and credit()



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; }

}
```

Requirement: a debit leading to a negative balance should be rejected

User class

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

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

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

  private void deposit_cash (int amt) {
    acc.credit (amt);
  }
}
```

Note: protocol ensures that balance is high enough to get cash => should entail the account's requirement

Is this program correct?

- Expectedly yes: we check that the account has enough money before taking the money

```
boolean b = acc.accept_debit (amt); // to avoid negative balance
if (b) acc.debit (amt);
```

- Test: After some time, balance gets negative

- Why ? Uneasy to debug due to concurrency

Verifying reactive programs

- In this example, the error was revealed by test...
- ... but in general, testing suffers the same limitations as for transformational programs
- Such errors may be hard to detect and debug
  Could we find this error before testing?
  Are there tools to help us debug them?
- Proof techniques are not well-adapted to handle **concurrency**
- Alternative techniques are needed, namely **automata-based verification**

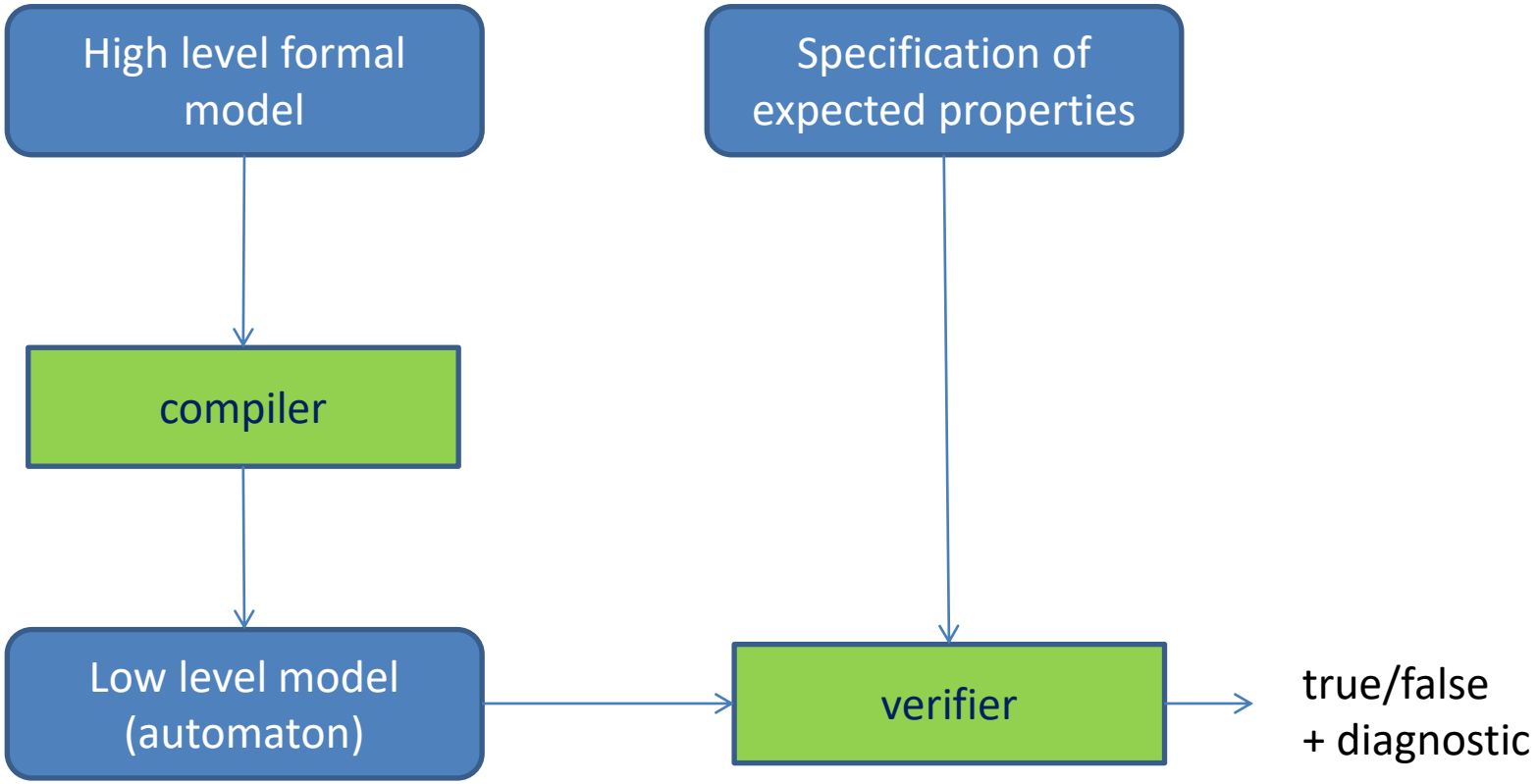
Principles of automata-based verification

- Write an abstract **formal model** that describes the **behaviour** (**temporal evolution**) of the program under design
- Write a specification of the **expected properties**
  Example: negative balance cannot be reached
- **Check the properties** on the model using automated software tools
- Only once all errors are corrected, implement the concrete program (possibly through refinement)

Automata-based verification

- The model is an **automaton** (e.g., *Labelled Transition System*) or more precisely 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**

Automata-based verification



Checking programs vs. models

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

- The programming language must have **formal semantics**
- As for transformational programs, **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, …

3.1. MODELING CONCURRENT PROGRAMS AS AUTOMATA

LTS (Labelled Transition System)

- An LTS is a simple kind of (low-level) automaton used in this lecture to illustrate the principles of automata-based verification
- Formally: LTS is a 4-tuple $(S, A, \rightarrow, s_0)$ 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
  - $s_0 \in S$ is a particular state called the **initial state**

Graphical representation of an LTS

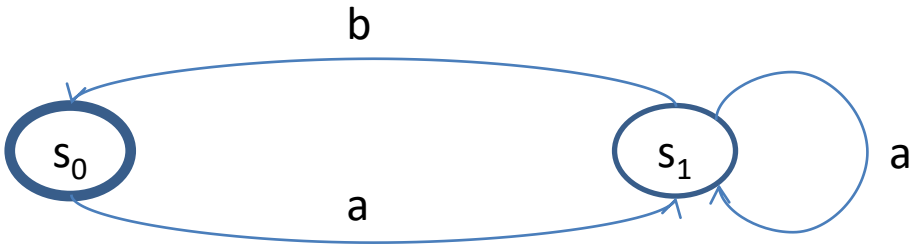
- In general, LTSs will be represented **graphically**
- **Example**: $(S, A, \rightarrow, s_0)$

  where $S = \{ s_0, s_1 \}$

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

  $\rightarrow = \{ (s_0, a, s_1), (s_1, b, s_0), (s_1, a, s_1) \}$

  will be represented graphically as follows:



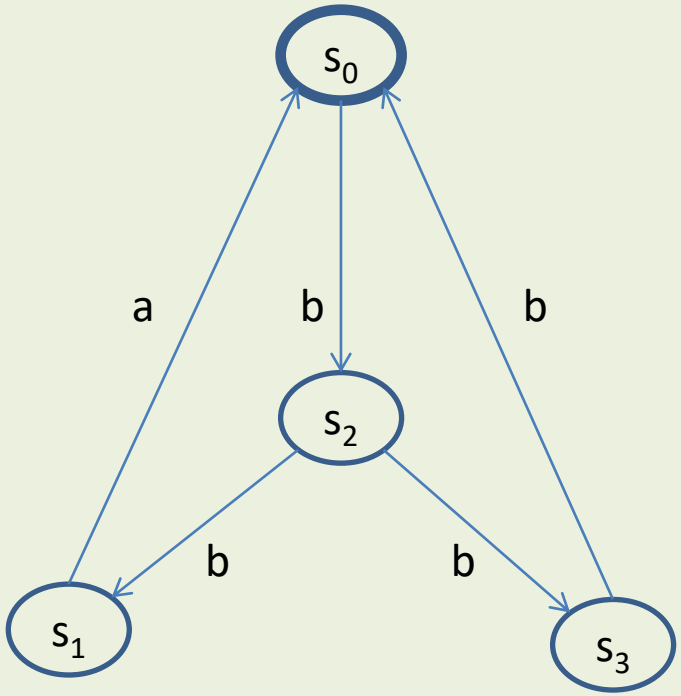
Exercise

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

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

Exercise

- Describe the following LTS as a 4-tuple

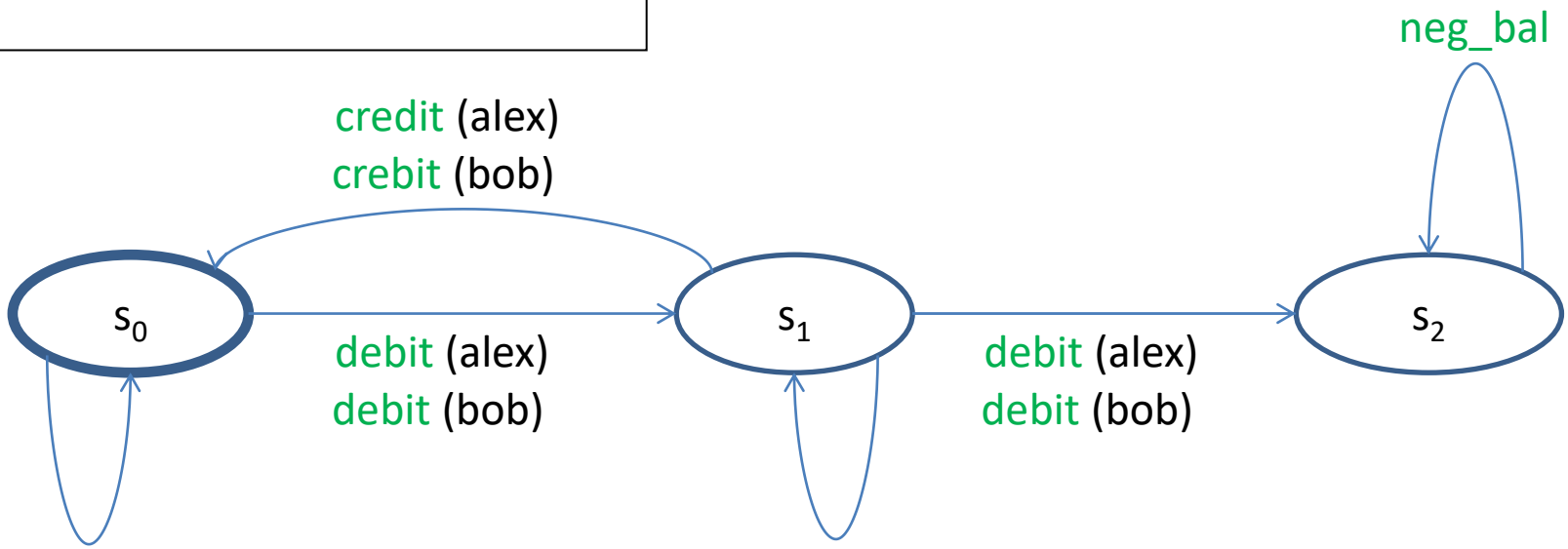


Bank example: LTSs

- Users and account can be modeled by LTS
- To avoid state space explosion:
  - amount of each debit/credit is 1
  - account balance stays in [-1, 1] (initially 1)
- Interactions between LTS are modeled by atomic « *synchronized* » actions:
  - credit (*u*): user *u* credits account
  - debit (u): user *u* debits account
  - accept_debit (*u*, *b*): user *u* asks for debit
    b = true iff account balance is strictly positive
- Additional action neg_bal: balance is negative

```
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; }

}
```

Bank example: LTS of account



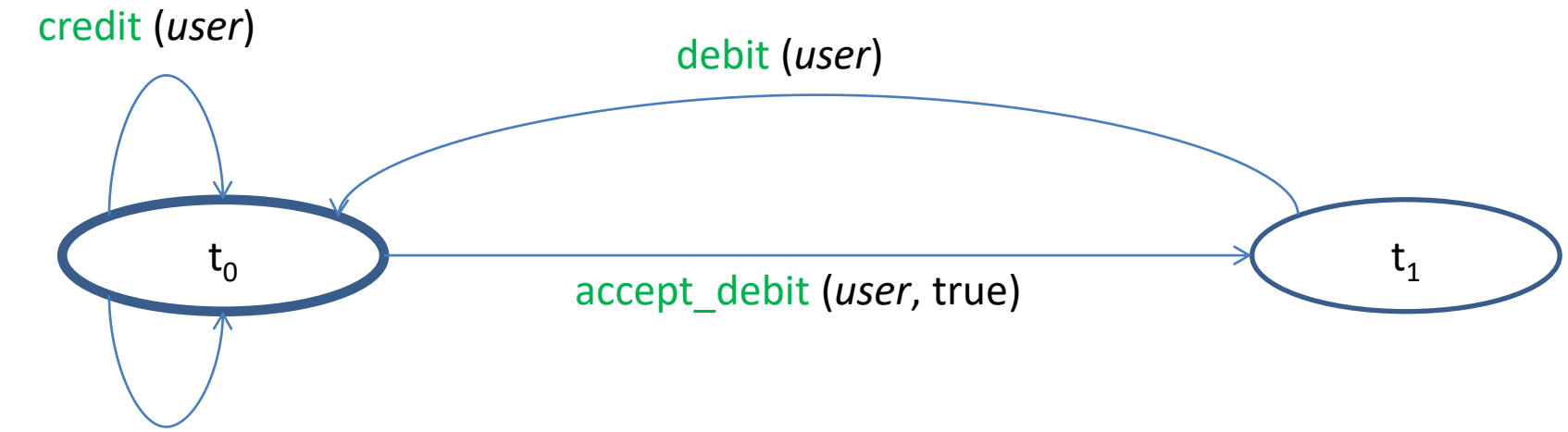
accept_debit (alex, true)
accept_debit (bob, true)

accept_debit (alex, false)
accept_debit (bob, false)

In s_0, balance = 1 (maximum balance : account cannot be credited)
In s_1, balance = 0
In s_2, balance = -1 (error state)

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

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

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

  private void deposit_cash (int amt) {
    acc.credit (amt);
  }
}
```

Bank example: LTSs of Alex and Bob



user ∈ {alex, 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**)
- Product is computed automatically

Formal definition of product of LTSs

Given:

- two LTSs $P_1 = (S_1, A_1, \rightarrow_1, s_{0,1})$ and $P_2 = (S_2, A_2, \rightarrow_2, s_{0,2})$
- 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

$$(S_1 \times S_2, A_1 \cup A_2, \rightarrow, (s_{0,1}, s_{0,2}))$$

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

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

Bank example

The bank example is modelled as the product:

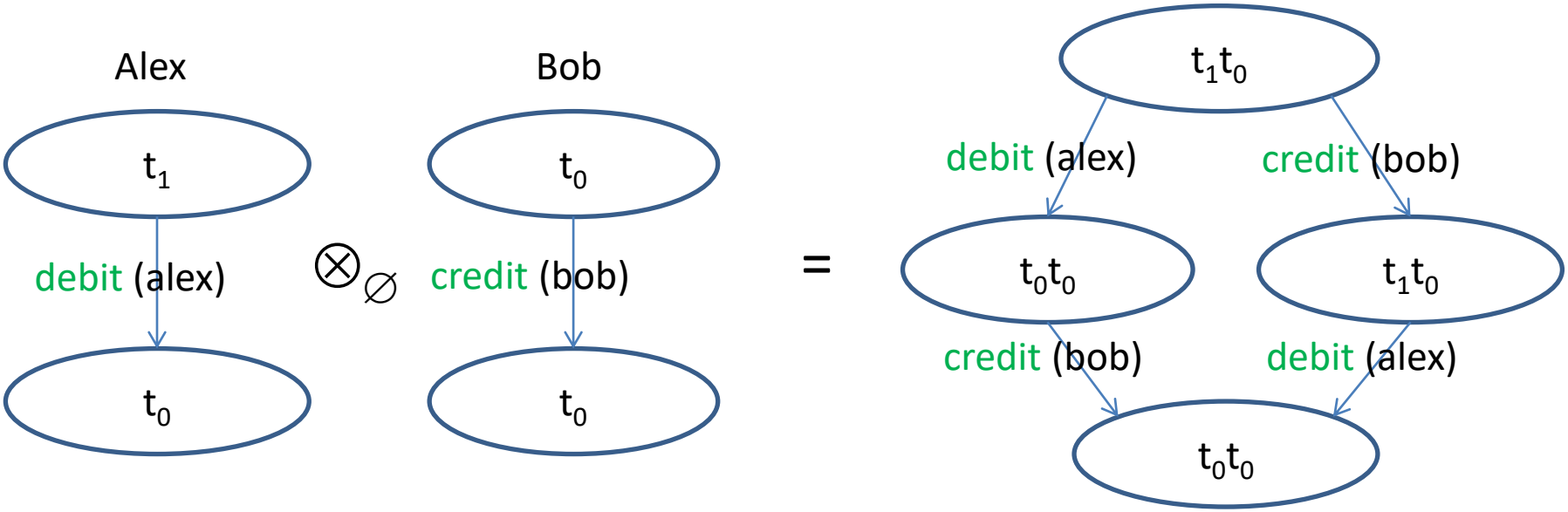
$$\text{Account} \otimes_{\mathbf{A} \cup \mathbf{B}} (\text{Alex} \otimes_{\varnothing} \text{Bob})$$

Indeed:

- Alex and Bob should not synchronise
- Alex and Account should synchronise on **A** = {accept_debit(u, b), debit(u), credit(u) | u=alex, $b \in$ bool}
- Bob and Account should synchronise on **B** = {accept_debit(u, b), debit(u), credit(u) | u=bob, $b \in$ bool}

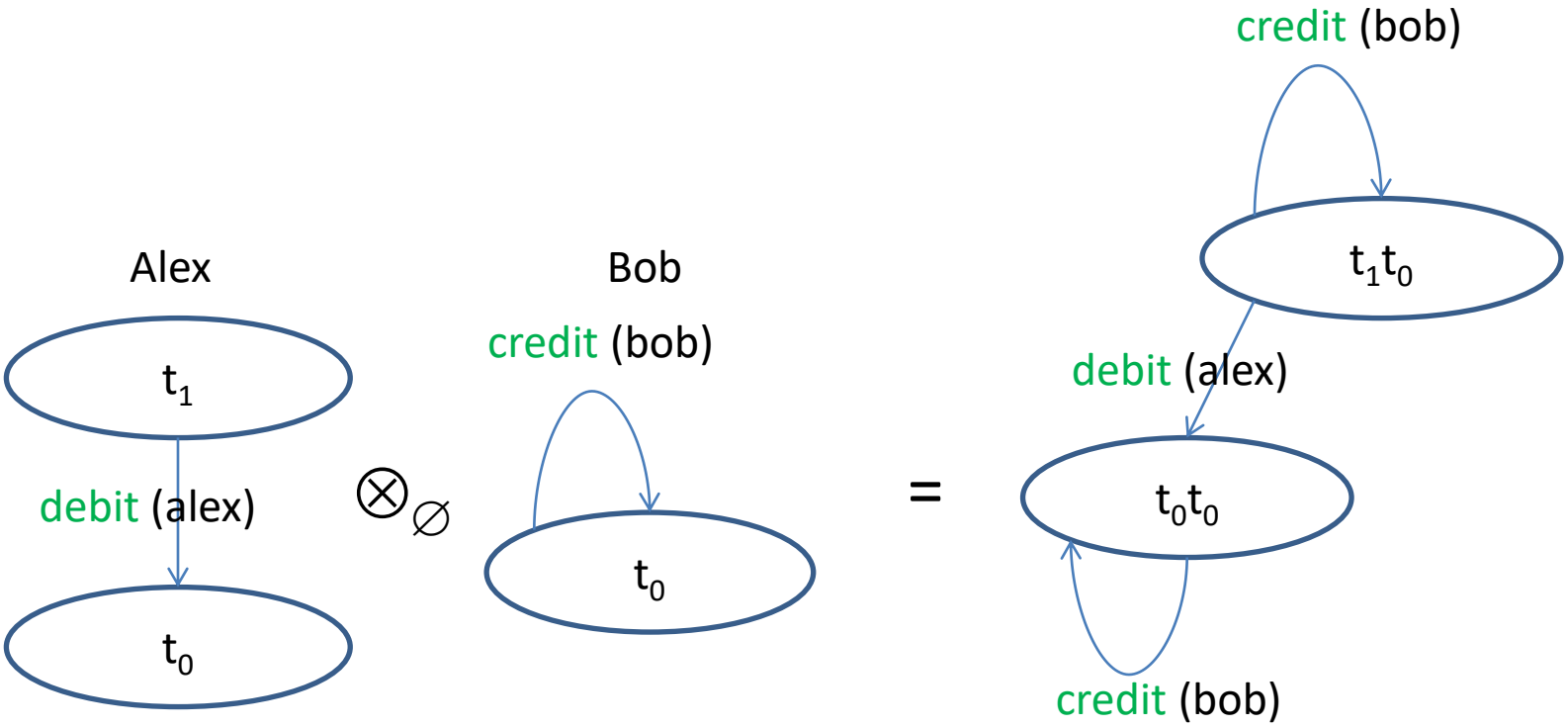
Non-synchronising actions

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



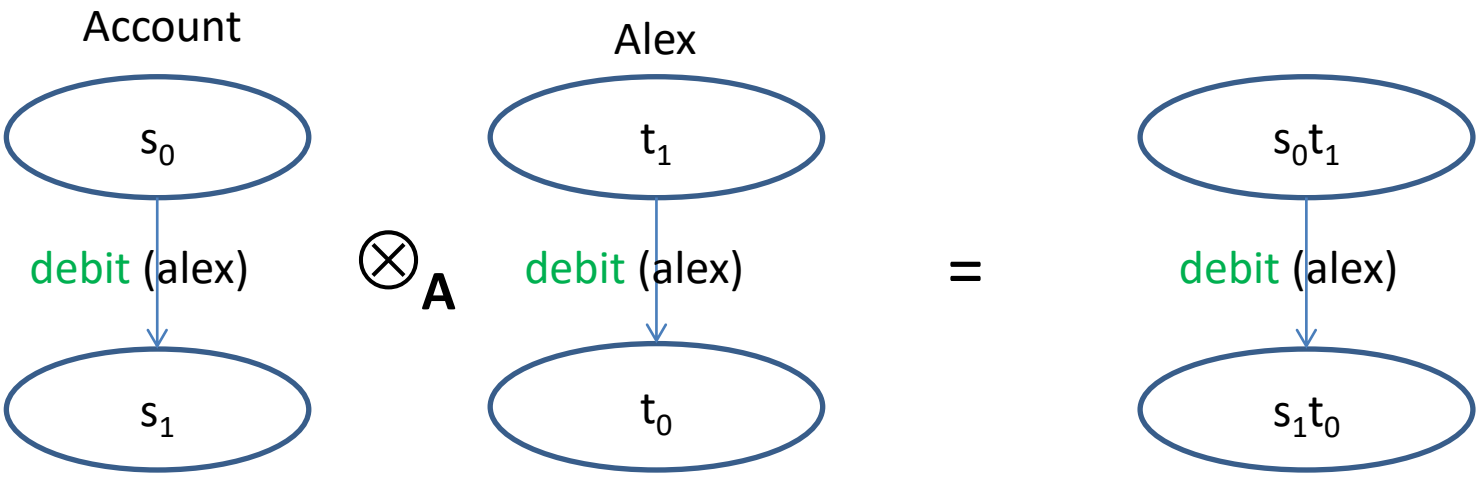
Non-synchronising actions

- Second example



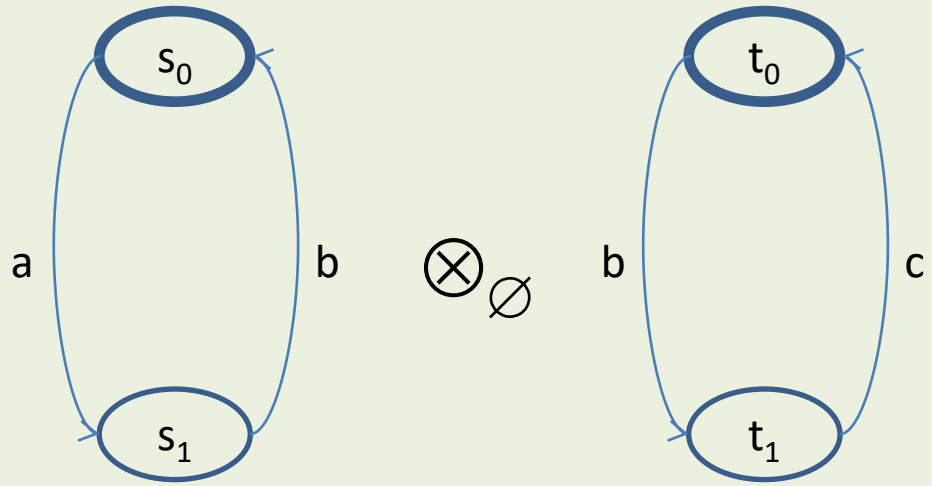
Synchronising actions

- Synchronising actions execute together at once
- Example:



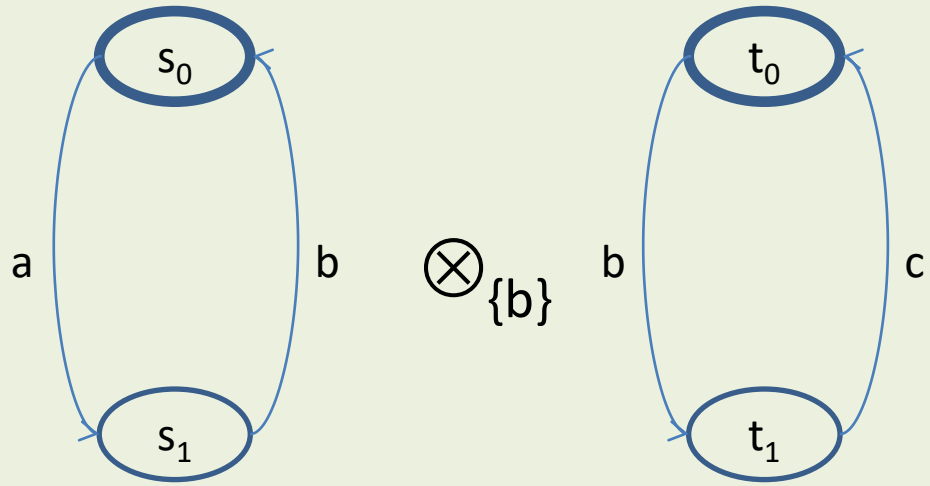
Exercise (1/2)

Draw the result of the following product of LTSs



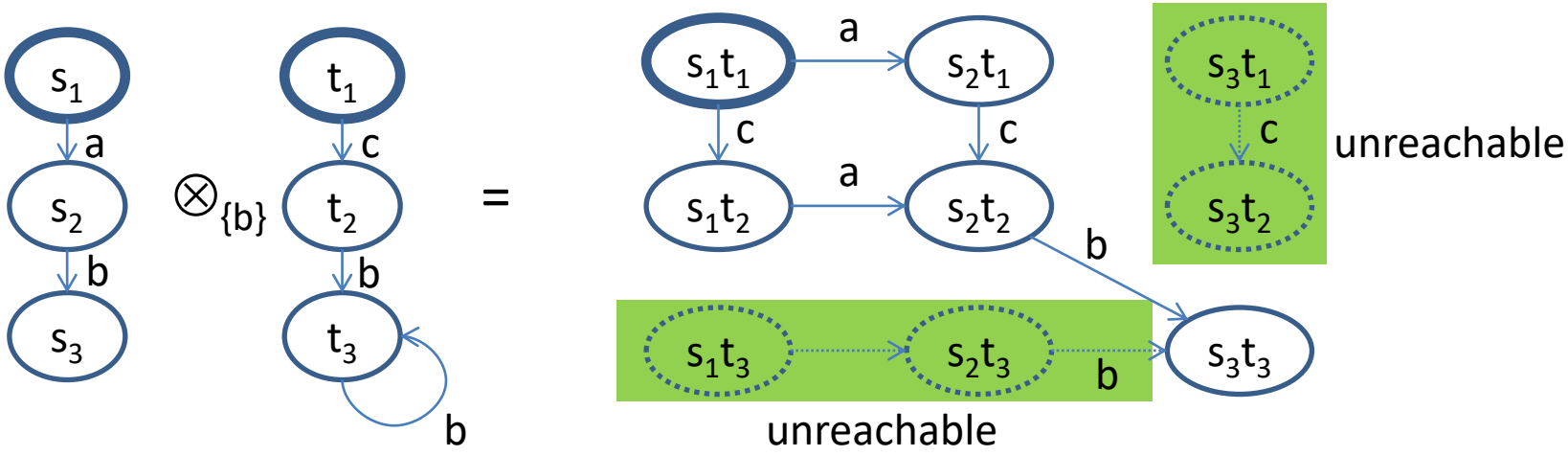
Exercise (2/2)

Draw the result of the following product of LTSs



Reachable product of LTSs

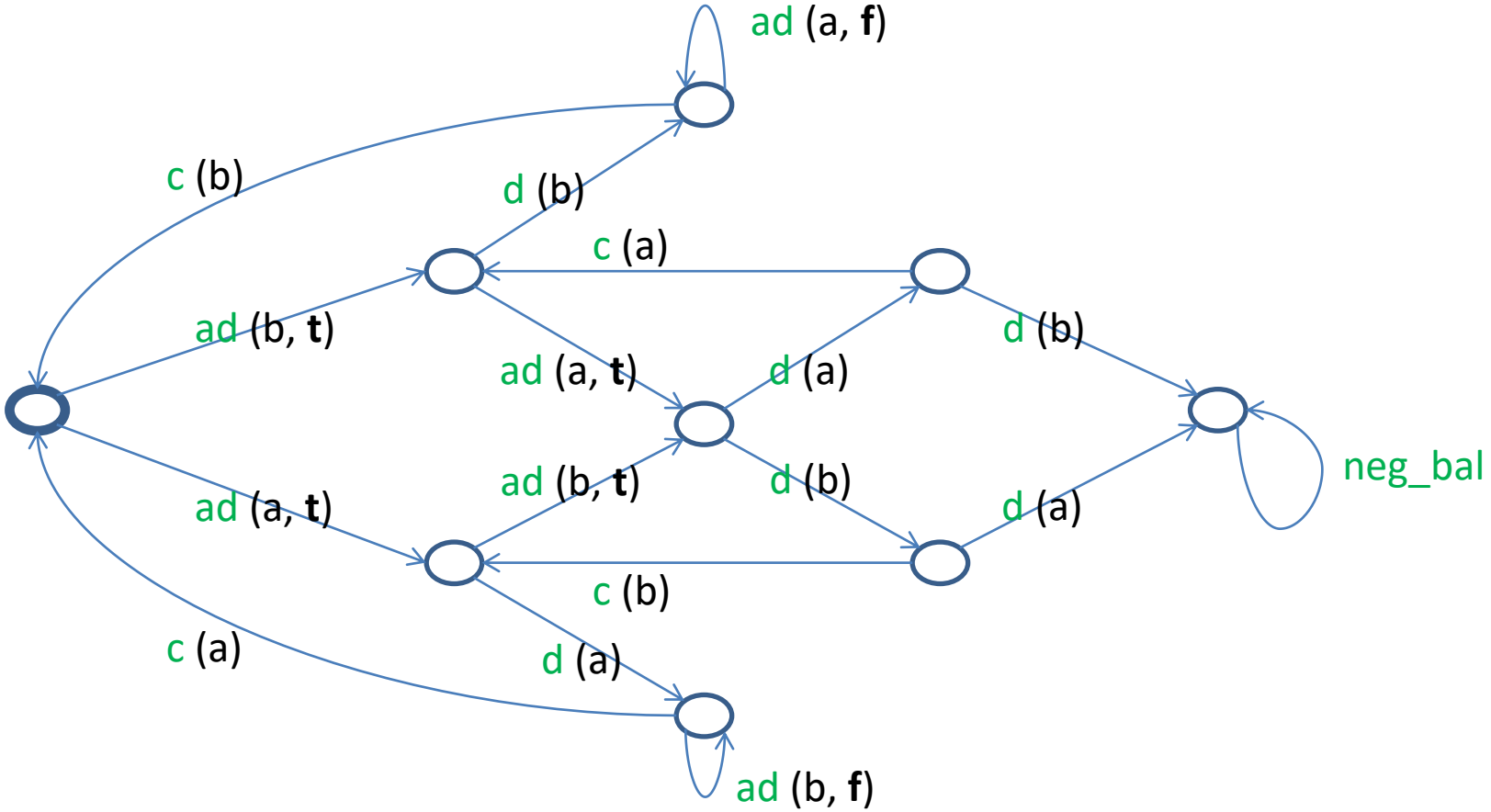
- Definition of product includes states that are **not reachable** from the initial state
- In general, we **restrict the product to the reachable part**, i.e., unreachable states are ignored
- **Example**:



Bank example: product of LTSs

Account $\otimes_{A \cup B}$ (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



3.2. MODELLING AND 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**

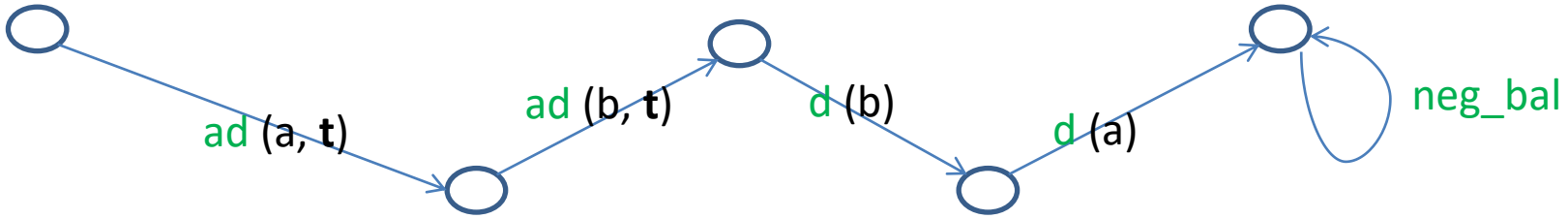
Reachability analysis

Is action a reachable in $(S, A, \rightarrow, s_0)$?

```
function reachability (a : A) : bool is
    visited := {s0}; explored := {}
    while visited ≠ {} loop
        choose s ∈ visited; visited := visited \ {s}
        explored := explored ∪ {s}
        if (∃s') s –a-> s' then return true end if
        foreach s' such that (∃b) s –b-> s' loop
            if s' ∉ explored then
                visited := visited ∪ {s'}
            end if
        end loop
    end loop
    return false
```

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

Properties specified as regular expressions

- Checking existence or absence of a **finite path matching a regular expression** on actions
- **Example**: Is there a path in which Alex debits twice without an accept debit in between?

  true* . ‘d (a)’ . **not** ‘ad (a, **t**)’* . ‘d (a)’

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

Checking regular expressions

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

Properties expressed in temporal logic

- Regular expressions are not sufficient to model all properties of interest
- **Example**: is any accept debit **necessarily** followed by a debit?
- **Temporal logics** introduce **modalities**, to deal with the notions of **possibility** and **necessity**

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:

φ ::= **true**
| **false**
| $\varphi_1 \wedge \varphi_2$
| $\varphi_1 \vee \varphi_2$
| $\neg\varphi_0$
| $< \beta > \varphi_0$ *possibility*
| $[\ \beta\ ]\ \varphi_0$ *necessity*

where β is a regular expression on actions

PDL modalities

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

[**true*** . ‘ad (a, **t**)’] < ‘d (a)’ > **true**

and

[**true*** . ‘ad (b, **t**)’] < ‘d (b)’ > **true**

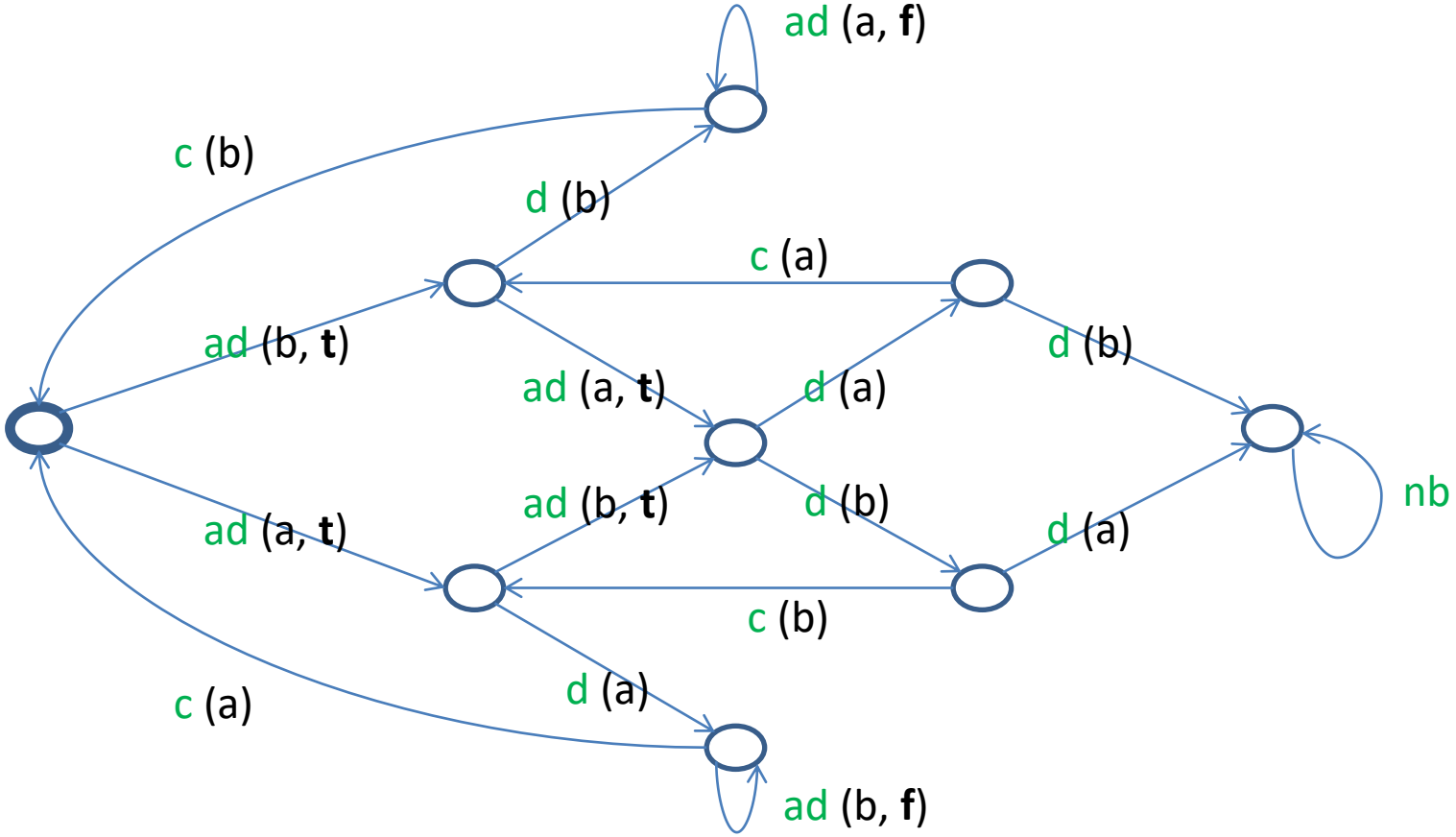
Example: check the 4 PDL formulas

(1) < **true*** . nb > **true**

(2) [**true***] < **true** > **true**

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

(4) < **true*** > (< 'd (a)' > **true** ∧ < 'd (b)' > **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**
  - **Absence of 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)

3.3. 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) to be checked

Example: The language LNT

- Developed by Inria/Convecs team
- Language structured in two parts: **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
type Status is pos, 0, neg end type

function status (bal : int) : Status is
    if bal > 0 then
        return pos
    elsif bal == 0 then
        return 0
    else
        return neg
    end if
end function
```

Bank example in LNT: Account process

```
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; }

}
```

```
process Account [accept_debit, debit, credit, neg_bal : any] is
    var bal : int, res : bool in bal := 1;
        loop alt
            accept_debit (?any User, status (bal) == pos)
        []
            debit (?any User); bal := bal-1;
            if status (bal) == neg then loop neg_bal end loop
        []
            only if bal < 1 then credit(?any User); bal := bal+1 end if
        end alt end loop
    end var
end process
```

Bank example in LNT: User process

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

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

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

  private void deposit_cash (int amt) {
    acc.credit (amt);
  }
}
```

```
process User [accept_debit, debit, credit : any] (name : User) is
   var res : bool in
      loop alt
         accept_debit (name, ?res);
         if res then debit (name) end if
      []
         credit (name)
      end alt 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

Automata-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

Conclusion

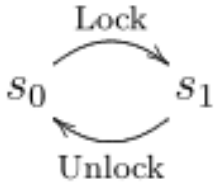
- **Formal methods** can **improve** usual practice in the specification and design steps (mostly based on natural languages and diagrams)
- They require **advanced skills** and concern prioritarily **critical systems** (avionics, nuclear, transport, circuit design, security, …)
- The **cost** of their 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**: transformational, reactive, concurrent, real time, etc.

Competence and Knowledge which will be evaluated

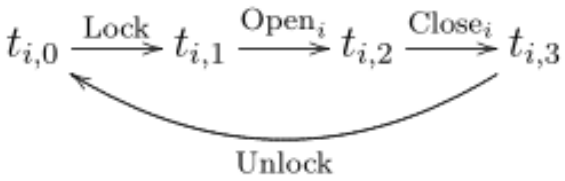
- be able to
  - Compute the **synchronised product** of two LTSs
  - Check for the **existence of a path matching a regular expression**
- know
  - The difficulty of **engineering concurrent reactive programs**
  - The principles and benefits of **model based verification** using **formal methods**

Exercise 1 (automata)

P :



Q_i $(i \in \{1, 2\})$:



Calculate $P \otimes_{\{\text{Lock},\text{Unlock}\}} (Q_1 \otimes_{\emptyset} Q_2)$

Exercice 2 (algebraic specifications)

$$\mathbf{true} : \rightarrow \mathbf{Bool} \qquad \mathbf{nil} : \rightarrow \mathbf{Bool\_List}$$
$$\mathbf{false} : \rightarrow \mathbf{Bool} \qquad \mathbf{cons} : \mathbf{Bool} \times \mathbf{Bool\_List} \rightarrow \mathbf{Bool\_List}$$

$$\mathbf{and} : \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}$$
$$(\forall b \in \mathbf{Bool})\ \mathbf{true\ and}\ b = b \quad (a1)$$
$$(\forall b \in \mathbf{Bool})\ \mathbf{false\ and}\ b = \mathbf{false} \quad (a2)$$
$$\mathbf{cat} : \mathbf{Bool\_List} \times \mathbf{Bool\_List} \rightarrow \mathbf{Bool\_List}$$
$$(\forall l_0 \in \mathbf{Bool\_List})\ \mathbf{cat}\ (\mathbf{nil}, l_0) = l_0 \quad (c1)$$
$$(\forall b \in \mathbf{Bool}, l_1, l_2 \in \mathbf{Bool\_List})\ \mathbf{cat}\ (\mathbf{cons}\ (b, l_1), l_2) = \mathbf{cons}\ (b, \mathbf{cat}\ (l_1, l_2)) \quad (c2)$$
$$\mathbf{and\_list} : \mathbf{Bool\_List} \rightarrow \mathbf{Bool}$$
$$\mathbf{and\_list}\ (\mathbf{nil}) = \mathbf{true} \quad (l1)$$
$$(\forall b \in \mathbf{Bool}, l_0 \in \mathbf{Bool\_List})\ \mathbf{and\_list}\ (\mathbf{cons}\ (b, l)) = b\ \mathbf{and}\ \mathbf{and\_list}\ (l) \quad (l2)$$

Show

$$(\forall l_1, l_2 \in \mathbf{Bool\_List})\ \mathbf{and\_list}\ (\mathbf{cat}\ (l_1, l_2)) = \mathbf{and\_list}\ (l_1)\ \mathbf{and}\ \mathbf{and\_list}\ (l_2) \quad (eqn)$$

Exercice 2 (algebraic specification)

$\mathbf{true} : \rightarrow \mathbf{Bool}$
$\mathbf{false} : \rightarrow \mathbf{Bool}$

$\mathbf{nil} : \rightarrow \mathbf{Bool\_List}$
$\mathbf{cons} : \mathbf{Bool} \times \mathbf{Bool\_List} \rightarrow \mathbf{Bool\_List}$

$\mathbf{and} : \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}$

$(\forall b \in \mathbf{Bool})\ \mathbf{true\ and}\ b = b$ (a1)

$(\forall b \in \mathbf{Bool})\ \mathbf{false\ and}\ b = \mathbf{false}$ (a2)

$\mathbf{cat} : \mathbf{Bool\_List} \times \mathbf{Bool\_List} \rightarrow \mathbf{Bool\_List}$

$(\forall l_0 \in \mathbf{Bool\_List})\ \mathbf{cat}\ (\mathbf{nil}, l_0) = l_0$ (c1)

$(\forall b \in \mathbf{Bool}, l_1, l_2 \in \mathbf{Bool\_List})\ \mathbf{cat}\ (\mathbf{cons}\ (b, l_1), l_2) = \mathbf{cons}\ (b, \mathbf{cat}\ (l_1, l_2))$ (c2)

$\mathbf{and\_list} : \mathbf{Bool\_List} \rightarrow \mathbf{Bool}$

$\mathbf{and\_list}\ (\mathbf{nil}) = \mathbf{true}$ (l1)

$(\forall b \in \mathbf{Bool}, l_0 \in \mathbf{Bool\_List})\ \mathbf{and\_list}\ (\mathbf{cons}\ (b, l)) = b\ \mathbf{and}\ \mathbf{and\_list}\ (l)$ (l2)

1. (0.5 point) Consider the case where $l_1 = \mathbf{nil}$ and l_2 is an arbitrary list, and complete the following lines (please rewrite these lines on the copy of your exam, do not respond directly on the subject for this question):

$\mathbf{and\_list}\ (\mathbf{cat}\ (\mathbf{nil}, l_2)) = \ldots$ by the axiom (c1)

$\mathbf{and\_list}\ (\mathbf{nil})\ \mathbf{and}\ \mathbf{and\_list}\ (l_2) = \ldots$ by the axiom (l1)

$= \ldots$ by the axiom (a1)

Exercice 2 (algebraic specification)

$$\textbf{true} : \rightarrow \textbf{Bool} \qquad \textbf{nil} : \rightarrow \textbf{Bool\_List}$$
$$\textbf{false} : \rightarrow \textbf{Bool} \qquad \textbf{cons} : \textbf{Bool} \times \textbf{Bool\_List} \rightarrow \textbf{Bool\_List}$$

$$\textbf{and} : \textbf{Bool} \times \textbf{Bool} \rightarrow \textbf{Bool}$$
$$(\forall b \in \textbf{Bool})\ \textbf{true and}\ b = b \quad \text{(a1)}$$
$$(\forall b \in \textbf{Bool})\ \textbf{false and}\ b = \textbf{false} \quad \text{(a2)}$$
$$\textbf{cat} : \textbf{Bool\_List} \times \textbf{Bool\_List} \rightarrow \textbf{Bool\_List}$$
$$(\forall l_0 \in \textbf{Bool\_List})\ \textbf{cat}\ (\textbf{nil}, l_0) = l_0 \quad \text{(c1)}$$
$$(\forall b \in \textbf{Bool}, l_1, l_2 \in \textbf{Bool\_List})\ \textbf{cat}\ (\textbf{cons}\ (b, l_1), l_2) = \textbf{cons}\ (b, \textbf{cat}\ (l_1, l_2)) \quad \text{(c2)}$$
$$\textbf{and\_list} : \textbf{Bool\_List} \rightarrow \textbf{Bool}$$
$$\textbf{and\_list}\ (\textbf{nil}) = \textbf{true} \quad \text{(l1)}$$
$$(\forall b \in \textbf{Bool}, l_0 \in \textbf{Bool\_List})\ \textbf{and\_list}\ (\textbf{cons}\ (b, l)) = b\ \textbf{and}\ \textbf{and\_list}\ (l) \quad \text{(l2)}$$

2. (1 point) We now assume that there exists at least one list $l_3 \in \textbf{Bool\_List}$ such that equation (eqn) holds, i.e.,

$$(\forall l_2 \in \textbf{Bool\_List})\ \textbf{and\_list}\ (\textbf{cat}\ (l_3, l_2)) = \textbf{and\_list}\ (l_3)\ \textbf{and}\ \textbf{and\_list}\ (l_2) \quad \text{(ih)}$$

We consider the list $l_4 = \textbf{cons}\ (b, l_3)$ where b is an arbitrary Boolean, and we then show the following using (ih):

$$(\forall l_2 \in \textbf{Bool\_List})\ \textbf{and\_list}\ (\textbf{cat}\ (l_4, l_2)) = \textbf{and\_list}\ (l_4)\ \textbf{and}\ \textbf{and\_list}\ (l_2)$$

Exercise 3 (B method)

MACHINE File_Processing

SETS FILES

VARIABLES received, valid

INVARIANT $received \subseteq FILES \wedge valid \subseteq received$

INITIALISATION $received := \emptyset;\ valid := \emptyset$

OPERATIONS

receive (f) =
  PRE $f \in FILES$
  THEN $received := received \cup \{f\}$
  END;

validate (f) =
  PRE $f \in received$
  THEN $valid := valid \cup \{f\}$
  END;

discard (f) =
  PRE $f \in received$
  THEN $received := received \setminus \{f\}$
  END;

What are the proof obligations? Do they hold?