Swansea University
Department of Computer Science

U Berger
December 2022

CS_385/CS_M85 Modelling and Verification Techniques

Revision Guide

These notes contain are a brief summary of the main topics relevant for the exam. They are a guide for revision, not a complete revision material. The material for revision consists of the course notes as well as courseworks and lab tasks (including their solutions) which can be found on Canvas.

Topics.

1 Labelled transition systems (LTS), traces, bisimulation

2 Modelling processes in CSP

3 Verification of processes using model checking of trace refinement

4 Modeling and verifying security protocols

5 Program extraction from proofs

1 Labelled transition systems (LTS), traces, bisimulation

Revision material: Handout "Labelled transition systems, traces, bisimulation", available on Canvas. See also the handout "Pointers to the literature" on Canvas for more material.

The set of traces, $\mathrm{Traces}(s)$, of a process (or state) s is defined as follows:

(i) $\langle\rangle \in \mathrm{Traces}(s)$ holds for every process $s \in S$.

(ii) $a\,w \in \mathrm{Traces}(s)$ holds iff there exists $s' \in S$ such that $s \xrightarrow{a} s'$ and $w \in \mathrm{Traces}(s')$.

Two states $s, t \in S$ are called *trace equivalent*, written $s =_{\mathrm{T}} t$, if $\mathrm{Traces}(s) = \mathrm{Traces}(t)$.

A *bisimulation* is a relation $R \subseteq S \times S$ satisfying: For all $s, t \in S$, if $s\,R\,t$, then

(1) $\forall a \in A \forall s' \in S(s \xrightarrow{a} s' \Longrightarrow \exists t' \in S(t \xrightarrow{a} t' \wedge s'\,R\,t'))$

(2) $\forall a \in A \forall t' \in S(t \xrightarrow{a} t' \Longrightarrow \exists s' \in S(s \xrightarrow{a} s' \wedge s'\,R\,t'))$

Two states $s, t \in S$ are called *bisimilar*, written $s \sim t$, if there exists a bisimulation R such that $s\,R\,t$.

In order to prove that s and t are bisimilar one has to find a bisimulation R such that $s\,R\,t$. This proof principle is commonly known as *coinduction*.

Bisimilarity can also be explained via the *bisimulation game*:

- The *configurations* of the game are pairs of states (s, t).
- There are two players:
  - The *attacker*, who wants to prove that s and t are *not* bisimilar.
  - The *defender*, who wants to prove that s and t *are* bisimilar.
- A *move* at a configuration (s, t) has 2 phases:
  - Phase 1: The attacker chooses a transition starting with one of the two given states s or t, say $s \xrightarrow{a} s'$.

  - Phase 2: The defender responds with a transition of the other state but the same action a, say $t \xrightarrow{a} t'$.

  The new configuration after the move is (s', t').

  - The attacker wins if the game ends because the defender cannot carry out Phase 2 of a move.
  - The defender wins if either the game runs infinitely or the game ends because the attacker is unable to carry out Phase 1 of a move.

Two states s, t are bisimilar precisely if, starting with configuration (s, t), the defender has a winning strategy.

- If two states are bisimilar, then they are trace equivalent.
- On the other hand, trace equivalence does *not* imply bisimilarity.
- For *deterministic* LTS, trace equivalence does imply bisimilarity.

Example. In the following LTS, which states are trace equivalent, which are bisimilar?

2 Modelling processes in CSP

Revision material: Lecture notes `verifiction-csp.pdf` and the accompanying csp-file `csp-intro.csp`. See "Pointers to the literature" for more material. The solutions to the lab assigments 1 and 2, as well as the lab on the dining philosopers provide further useful examples of CSP machine code.

You should be familiar with the most common CSP-constructs and their semantics, such as

Equations	$P = \ldots$ or $P(x) = \ldots$ (possibly recursive)
Prefixing	$a \to P$
Input	$c?x \to P$ (a.k.a. prefix choice)
Output	$c!x \to P$ or $c.x \to P$
External choice	$P \,\Box\, Q$ (machine syntax `P [] Q`)
Internal choice	$P \sqcap Q$ (machine syntax `P |~| Q`)
Interleaving	$P \,\|\|\|\, Q$
Generalised Parallel	$P \,[\|X\|]\, Q$ (X a set of events or channels)
If-then-else	**if** t **then** P **else** Q
Composition	$P \,;\, Q$
Hiding	$P \setminus X$ (X a set of events or channels)
Stop, Skip	

You should be able to model simple processes and be able to understand what a given process definition means. Examples are variants of vending machines and the robot in the coursework.

One distinguishes between *observable* events and the *unobservable* or *silent* event τ. The silent event τ plays an important role in the firing rules for equations, internal choice, if-then-else and hiding.

Revision material on Canvas:

CSP Syntax and Semantics with examples

CSP Syntax

3 Verification of processes using model checking

Revision material: The lecture notes `verification-csp.pdf` give an introduction to modeling processes in CSP and model checking through deadlock- and trace-refinement-checking. Further examples can be found in the model solutions to the labs.

Deadlock

In the first lab we modelled various processes and checked whether they contain deadlocks. A *deadlock* is a state where no action is possible.

Example: In lab1, the given process `DinPhils` describing the dining philosophers, has deadlocks. This can be found out by evaluating in FDR the assertion

```
assert DinPhils :[deadlock free]
```

By modifying the behaviour of the philosophers slightly, we could avoid the deadlock.

Trace refinement

In order to show that a process P (the implementation) has a certain "allowed" behaviour, one can define another process S (the specification) whose traces are exactly the allowed behaviours, and then show, by model checking, that $\text{Traces}(P) \subseteq \text{Traces}(S)$, that is $S \sqsubseteq_T P$ (P trace-refines S).

Example: In the verification of NSP/NSPL (see Sect. 4) we defined a specification processes SPEC that specifies all nonces the intruder is allowed to know:

```
SPEC = know ? n : noncesI -> SPEC
```

Then we checked that the system adheres to this specification, hiding the channels send and receive:

$$\text{SPEC} \sqsubseteq_T \text{SYSTEM} \setminus \{\text{send}, \text{receive}\}$$

4 Modeling and verifying security protocols

Revision material: Lecture notes `verification-nsp.pdf`, solutions to the lab tasks on NSP, original articles about the Needham-Schroeder Public-Key Protocol, available on Canvas.

You should be able to explain the common short notation for protocols, for example, the Needham-Schroeder protocol:

$$\begin{aligned} &(1) \quad A \to B : \{N_A, A\}_{pk_B} \\ &(2) \quad B \to A : \{N_A, N_B\}_{pk_A} \\ &(3) \quad A \to B : \{N_B\}_{pk_B} \end{aligned}$$

- A, B are user names
- $A \to B : m$ means that A sends message m to B.
- N_A, N_B are nonces (random numbers used as challenges).
- $\{\ldots\}_{pk_B}$ is public-key encryption.

The purpose of the protocol above is authentication.

You should be able to explain the attack and the fix of the protocol by Lowe.

You should be able to explain the strategy to to verify the correctness of the amended protocol: Every event of the form know.n must be such that the nonce n is in the set noncesI of nonces the intruder is allowed to know. See Sect. 3 for details of how to achieve this through model checking a trace refinement statement.

5 Program extraction from proofs

Revision material: Handout on program extraction, Logic lecture notes, available on Canvas.

The basis of program extraction from proofs is the *Curry-Howard correspondence* between constructive logic and programming: according to which *formulas* correspond to *data types* and *proofs* correspond to *programs*.

Formula		Data Type	
conjunction	$A \wedge B$	$A \times B$	cartesian product
implication	$A \to B$	$A \to B$	function type
disjunction	$A \vee B$	$A + B$	disjoint sum
for all	$\forall x\, A(x)$	$D \to A$	function type
exists	$\exists x\, A(x)$	$D \times A$	cartesian product
equations	$s = t$	$\{*\}$	a singleton set
falsity	$\bot$	$\{\}$	the empty set

Proof rules correspond to *programming constructs*.

For example, *induction* corresponds to *recursion* or *loops*.

The following table shows the complete correspondence between proof rules and programming constructs:

assumption	variable	$u : A$
$\wedge^+$	pairing	$\dfrac{d : A \qquad e : B}{\langle d, e\rangle : A \wedge B}\,\wedge^+$
$\wedge^-$	projections	$\dfrac{d : A \wedge B}{\pi_{\mathrm{l}}(d) : A}\,\wedge^-_{\mathrm{l}} \qquad \dfrac{d : A \wedge B}{\pi_{\mathrm{r}}(d) : B}\,\wedge^-_{\mathrm{r}}$
$\rightarrow^+$	abstraction	$\dfrac{d : B}{\lambda u : A.d : A \rightarrow B}\,\rightarrow^+$
$\rightarrow^-$	procedure call	$\dfrac{d : A \rightarrow B \qquad e : A}{(de) : B}\,\rightarrow^-$
$\vee^+$	injections	$\dfrac{d : A}{\mathrm{inl}_B(d) : A \vee B}\,\vee^+_{\mathrm{l}} \qquad \dfrac{d : B}{\mathrm{inr}_A(d) : A \vee B}\,\vee^+_{\mathrm{r}}$
$\vee^-$	case analysis	$\dfrac{d : A \vee B \qquad e_1 : A \rightarrow C \qquad e_2 : B \rightarrow C}{\mathrm{cases}[d, e_1, e_2] : C}\,\vee^-$
$\forall^+$	abstraction	$\dfrac{d : A(x)}{\lambda x.d : \forall x\, A(x)}\,\forall^+ \quad (*)$
$\forall^-$	procedure call	$\dfrac{d : \forall x\, A(x)}{(dt) : A(t)}\,\forall^-$
$\exists^+$	pairing	$\dfrac{d : A(t)}{\langle t, d\rangle : \exists x\, A(x)}\,\exists^+$
$\exists^-$	matching	$\dfrac{d : \exists x\, A(x) \qquad e : \forall x\,(A(x) \rightarrow B)}{\mathrm{match}[d, e] : B}\,\exists^- \quad (**)$
induction	recursion	$\dfrac{d : A(0) \qquad e : \forall x\,(A(x) \rightarrow A(x+1))}{\mathrm{ind}[d, e] : \forall x\, A(x)}\,\mathrm{ind}$

Figure 1: Correspondence between proof rules and programming constructs

Program Extraction Theorem.

From every constructive, that is, intuitionistic proof of a formula

$$\forall x\, \exists y\, A(x, y)$$

one can extract a program p such that

$$\forall x\, A(x, p(x))$$

is provable, that is, p is provably correct.

A proof is *constructive* (or *intuitionistic*) if it does *not* use the principle of proof by contradiction, that is, the rule

$$\frac{\neg\neg A}{A}\,\mathrm{raa}$$

Proofs that do use this rule are called *classical*.

Example: Quotient and remainder

$$(+) \qquad \forall b\, (b > 0 \to \forall a\, \exists q\, \exists r\, (a = b * q + r \wedge r < b))$$

where the variables range over natural numbers. This formula says that division with remainder by a positive number b is possible for all b. The numbers q and r whose existence is claimed are the quotient and the remainder of this division.

According to the theorem above a constructive proof of $(+)$ yields a program that for inputs b and a, where $b > 0$, computes numbers q and r such that

$$a = b * q + r \quad \text{and} \quad r < b.$$

The proof uses induction, therefore, the extracted program is recursive.