Hoare logic and Model checking

Part II: Model checking

Lecture 7: Introduction to model checking

Christopher Pulte cp526
University of Cambridge

CST Part II – 2022/23

Acknowledgements

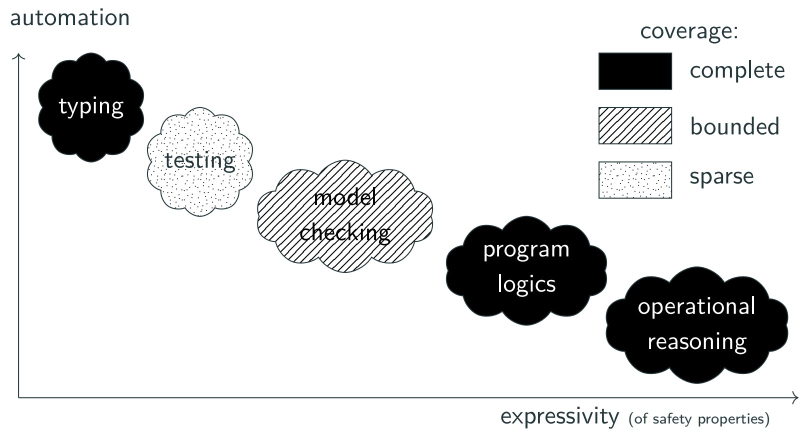
These slides are heavily based on (1) Jean Pichon-Pharabod's slides from the CST Part II – 2020/2021 course, which are in turn heavily based on previous versions by Mike Gordon, Dominic Mulligan, Alan Mycroft, and Conrad Watt, and (2, again) on Dominic Mulligan's slides for LTL and CTL.

They are also inspired by slides and lecture notes by John Gallagher, Gourinath Banda, and Pierre Ganty, by Paul Gastin, by Orna Grumberg, by Arie Gurfinkel, by Daniel Kroening, by Antoine Miné, by Julien Schmaltz, by David A. Schmidt, and by Carsten Sinz and Tomáš Balyo.

Thanks to Neel Krishnaswami, David Kaloper Meršinjak, Jack Parkinson, Peter Rugg, Ben Simner, and two anonymous students, for remarks and reporting mistakes.

Background

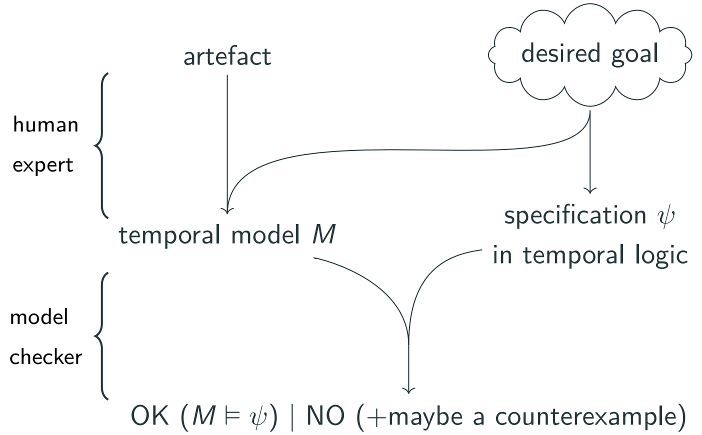
There are many verification & validation techniques of varying coverage, expressivity, level of automation, ..., for example:



Model checking application examples

- software, e.g. programs with complex control flow
- distributed systems
- protocols
- asynchronous circuits and hardware
- ...

Model checking



This diagram gives a very static, top-down picture; in practice these tasks feed back into each other.

Toy example

Suppose we are given an algorithm that is supposed to transfer, from one bank of the Cam to the other, using only a punt with seat for one, a wolf, a goat, and a cabbage[1].

The success criteria are

- safety: the cabbage and the goat, and the wolf and the goat, cannot be left alone on a bank;
- liveness: all three items are moved to the other bank.

[1] it is a large cabbage, so it takes up the whole seat

Toy example

How to model the problem?

- Option 1:

$$\mathcal{L} = -\frac{1}{4}F_{\mu\nu}F^{\mu\nu} + i\bar{\psi}\not{D}\psi + \text{h.c.} + \bar{\psi}_i y_{ij}\psi_j\phi + \text{h.c.} + |D_\mu\phi|^2 - V(\phi) + ???$$

- Option 2: (G. Doré, anonymous (Wellcome coll.), G. Waddington)



- Option 3: (apologies to the Phaistos cat)

Side ::= Left | Right Item ::= 

State $\stackrel{def}{=}$ Item $\to$ Side

...

About finding good models

"All models are wrong, some are useful" applies. The designer must ensure the model captures the significant aspects of the real system. Achieving it is a special skill, the acquisition of which requires thoughtful practice

— How Amazon Web Services Uses Formal Methods

Temporal models

Temporal models

A **temporal model** over **atomic propositions** AP is a left-total transition system where states are labelled with some of AP, and where some states are distinguished as initial:

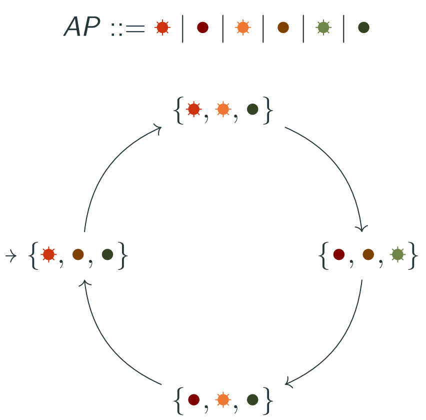
$$
\begin{array}{ll}
M, \ldots \in \mathrm{TModel} \overset{\textit{def}}{=} & \\
\quad (S \in \mathrm{Set}) \times & \text{states} \\
\quad\quad (S_0 \in \mathrm{sub}\ S) \times & \text{initial states} \\
\quad\quad\quad (① \ T \ ② \in \mathrm{relation}\ S\ S) \times & \text{transition} \\
\quad\quad\quad\quad (\ell \in (S \to \mathrm{sub}\ AP)) & \text{state labelling}
\end{array}
$$

such that T is left-total:

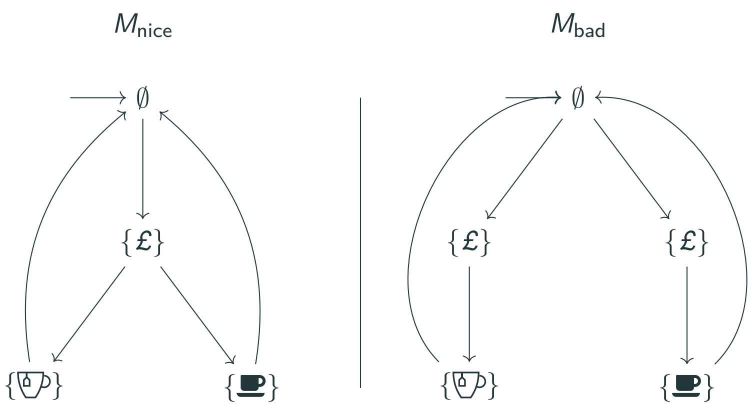
$$\forall s \in S.\, \exists s' \in S.\, s\ T\ s'$$

.

Temporal model of traffic lights



Tea & coffee machines



Corner case: the initial temporal model

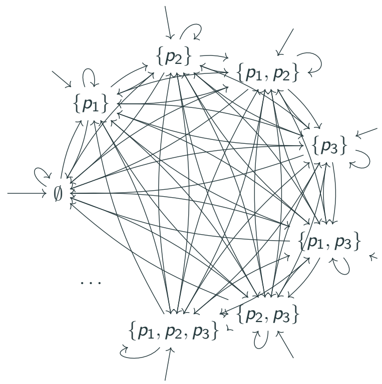
$$\mathbb{0} \in \text{TModel}$$

$$\mathbb{0} \overset{def}{=} \left\langle \begin{array}{l} \mathbb{0}, \\ \emptyset, \\ \emptyset, \\ s \mapsto \emptyset \end{array} \right\rangle \qquad \textit{(it is empty)}$$

Corner case: the terminal temporal model

$$\mathbb{1} \in \text{TModel}$$

$$\mathbb{1} \overset{def}{=} \left\langle \begin{array}{l} AP \to \mathbb{B}, \\ \{s \mid \top\}, \\ \{s_0, s_1 \mid \top\}, \\ s \mapsto \{p \mid s\ p\} \end{array} \right\rangle$$



Temporal model of a terrible punter

A punter with no concern for goat welfare or cabbage welfare:

$$\text{Side} ::= \text{Left} \mid \text{Right} \qquad \text{Item} ::= \ \mid \ \mid \ \mid$$

$$\text{State} \overset{def}{=} \text{Item} \to \text{Side}$$

$$\ldots$$

$$AP = \text{State}$$

$$M = \left\langle \begin{array}{l} \text{State}, \\ \{s \mid \forall i.\ s\ i = \text{Left}\}, \\ \left\{ s, s' \;\middle|\; \left(\begin{array}{l} (s\ ) = \text{flip}\ (s'\ ) \wedge \\ (\text{moveone}\ s\ s' \vee \text{movezero}\ s\ s') \end{array} \right) \right\}, \\ (s \mapsto \{s\}) \end{array} \right\rangle$$

Temporal model of a terrible punter

$$\text{flip Left} \overset{def}{=} \text{Right} \quad \text{flip Right} \overset{def}{=} \text{Left}$$

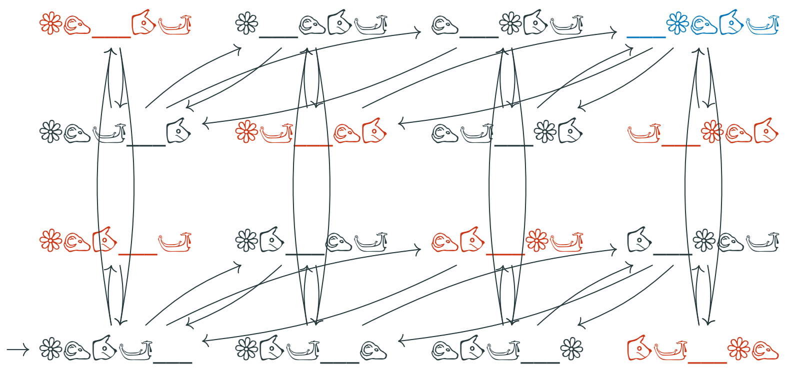
$$\text{moveone}\ s\ s' \overset{def}{=}$$

$$\left(\begin{array}{l} (\text{move}\ s\ s'\ \wedge \text{stay}\ s\ s'\ \wedge \text{stay}\ s\ s'\ ) \vee \\ (\text{move}\ s\ s'\ \wedge \text{stay}\ s\ s'\ \wedge \text{stay}\ s\ s'\ ) \vee \\ (\text{move}\ s\ s'\ \wedge \text{stay}\ s\ s'\ \wedge \text{stay}\ s\ s'\ ) \end{array} \right)$$

$$\text{movezero}\ s\ s' \overset{def}{=} \text{stay}\ s\ s'\ \wedge \text{stay}\ s\ s'\ \wedge \text{stay}\ s\ s'$$

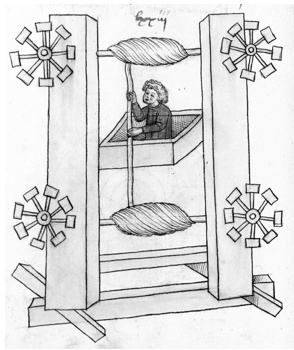
$$\text{move}\ s\ s'\ item \overset{def}{=} (s\ item) = (s\ ) \wedge (s'\ item) = (s'\ )$$
$$\text{stay}\ s\ s'\ item \overset{def}{=} (s'\ item) = (s\ item)$$

Temporal model of a terrible punter



Safety: we never go through a red state.
Liveness: we eventually reach the blue state.
Both are pretty clearly false! :-(

Informal temporal model of an elevator



Let us try to describe how an elevator for a building with 3 levels works:

- it starts at the ground floor, with the door closed, and goes back there when it is not called;
- if going through a level where it is called, it stops there and opens its door;
- . . .

Textual descriptions do not scale very well.

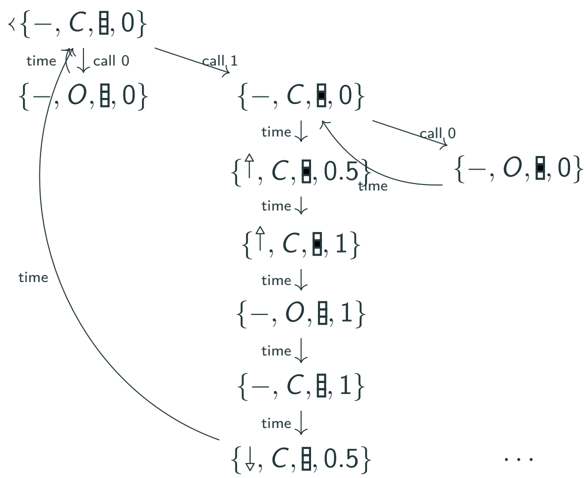
Temporal model of an elevator: statics and specification

Direction ::= stay | up | down
Level ::= 0 | 1 | 2
Location ::= 0 | 0.5 | 1 | 1.5 | 2
Called $\stackrel{def}{=}$ Level $\to \mathbb{B}$
DoorStatus ::= open | closed
ElevatorStatus $\stackrel{def}{=}$ Direction $\times$ Location $\times$ Called $\times$ DoorStatus

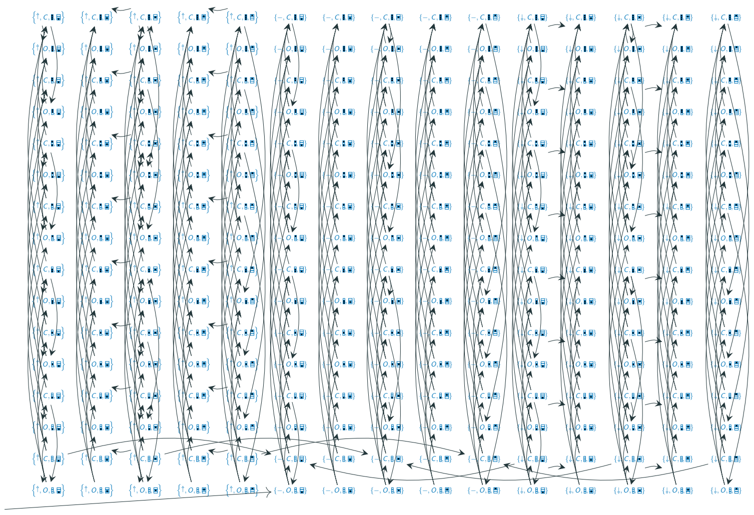
Desired goals:

- the door is not open at half-levels;
- if the elevator is called to a level, then it eventually gets there;
- the elevator does not lock people in;
- the path of the elevator is not entirely idiotic.

Temporal model of an elevator: partial dynamics



Temporal model of an elevator: complete (?) dynamics



How to have any confidence that this is correct? house by Petr Olšák

Definitions

(Infinite) Paths

$$\text{stream} \in \text{Set} \to \text{Set}$$
$$\text{stream } A \overset{\text{def}}{=} \mathbb{N} \to A$$

$$\text{IsPath} \in (M \in \text{TModel}) \to \text{stream } M.S \to \text{Prop}$$
$$\text{IsPath } M\ \pi \overset{\text{def}}{=} \forall n \in \mathbb{N}.\, (\pi\ n)\ M.T\ (\pi\ (n+1))$$

$$\text{Path} \in \text{TModel} \to \text{Set}$$
$$\text{Path } M \overset{\text{def}}{=} \{\pi \in \text{stream } M.S \mid \text{IsPath } M\ \pi\}$$

Reachable states & the tail operation

Because the transition relation is left-total, these infinite paths are "complete", in the sense that they capture reachability:

$$\text{Reachable} \in (M \in \text{TModel}) \to M.S \to \text{Prop}$$
$$\text{Reachable } M\ s \overset{\text{def}}{=} \exists \pi \in \text{stream } M.S, n \in \mathbb{N}.$$
$$\text{IsPath } M\ \pi \wedge M.S_0\ (\pi\ 0) \wedge s = \pi\ n$$

$$\text{tailn} \in (A \in \text{Set}) \to \mathbb{N} \to \text{stream } A \to \text{stream } A$$
$$\text{tailn } A\ n\ \pi \overset{\text{def}}{=} i \mapsto \pi\ (i+n)$$

Stuttering

A temporal model is **stuttering** when all states loop back to themselves:

$$\begin{aligned} &\text{stuttering} \in \text{TModel} \to \text{Prop} \\ &\text{stuttering } M \stackrel{\text{def}}{=} \forall s \in M.S.\ s\ M.T\ s \end{aligned}$$

⚠ If the temporal model is not stuttering, then we can count transitions. This is only sound if they exactly match those of the system being analysed.

See "What good is temporal logic" §2.3, by Leslie Lamport
`https://lamport.azurewebsites.net/pubs/what-good.pdf`

Applications of model checking

Applications of model checking

- Hardware:
  - circuits (with memory) directly translate to temporal models
  - lots of protocols
    - cache protocols
    - bus protocols
    - ...

    their specification involves lots of temporal "liveness" ("eventually something good") properties
- Software: often not finite a priori, but "proper modelling", or bounded model-checking
- Security protocols
- Distributed systems
- ...

The common denominator of many of these is the "killer app" of model checking: concurrency.

Examples

In the rest of this lecture, we will sketch how some of these are approached.

The point is not the details of any individual temporal model, but the overall approach.

Temporal model from operational semantics

Temporal model from operational semantics

An initial configuration for a small-step operational semantics naturally leads to a temporal model: take

- configurations as states,
- the initial configuration as the (only) initial state,
- steps as transitions, and
- some interesting properties as atomic propositions, for example

$$
\begin{array}{lll}
X, Y, Z, \ldots & \in & \mathsf{Var} \\
v & \in & \mathbb{Z} \\
AP & ::= & X \doteq v \mid X \doteq Y \mid X \dot{<} Y \mid \\
 & & X \dot{+} Y \dot{<} Z \mid X \dot{\times} Y \dot{<} Z \mid \\
 & & \ldots
\end{array}
$$

Temporal model from operational semantics

For example, for a language with a concurrent composition with interleaving dynamics (as in lecture 6):

$$
\begin{tikzcd}
 & & \langle C_1'||C_2||C_3', s_e\rangle \\
 & \langle C_1'||C_2||C_3, s_b\rangle \arrow[ur] \arrow[r] \arrow[dr] & \langle C_1''||C_2||C_3, s_f\rangle \\
 & & \langle C_1'||C_2'||C_3, s_g\rangle \\
 & & \langle C_1'||C_2'||C_3, s_h\rangle \\
\langle C_1||C_2||C_3, s_a\rangle \arrow[uuur] \arrow[r] \arrow[dddr] & \langle C_1||C_2'||C_3, s_c\rangle \arrow[ur] \arrow[r] \arrow[dr] & \langle C_1||C_2''||C_3, s_i\rangle & \cdots \\
 & & \langle C_1||C_2'||C_3', s_h\rangle \\
 & & \langle C_1||C_2'||C_3', s_k\rangle \\
 & \langle C_1||C_2||C_3', s_d\rangle \arrow[ur] \arrow[r] \arrow[dr] & \langle C_1||C_2||C_3'', s_l\rangle \\
 & & \langle C_1'||C_2||C_3', s_m\rangle
\end{tikzcd}
$$

Dealing with the size of temporal model from operational semantics

These temporal models are very often infinite or intractably large!

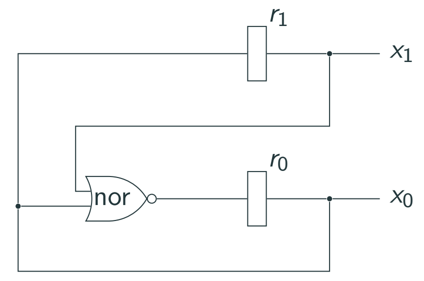
Many approaches:

- bounded model checking:
  - assume (and possibly check whether) loops execute no more than n times
  - consider executions of length smaller than n
  - ...
- use a model checking DSL to write an idealised version of the program
- use abstraction

Temporal model from circuits

Example circuit

Synchronous (the clock is left implicit) counter that goes $0, 1, 2, 0, 1, 2, \dots$ (assuming all registers are initially 0):



Registers make the circuit not be a simple function, which motivates using a temporal model.

Example circuit temporal model

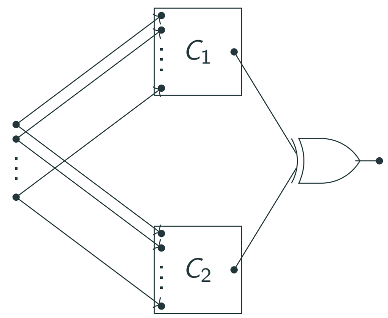
The states of the temporal model are the state of the registers, and the labels are which registers are set to 1:

$$\begin{tikzcd}
\emptyset \arrow[r] & \{r_0\} \arrow[d] \\
\{r_0, r_1\} \arrow[r] & \{r_1\} \arrow[ul]
\end{tikzcd}$$

Safety: The state $\{r_0, r_1\}$ should never be reached.
Liveness: all other states should be visited infinitely often.

Difference circuit

Given two circuits $C_1, C_2 \in \text{SCircuit } i \text{ } 1$, we can define their difference circuit $C_1 \ominus C_2$:



If the answer is always 0, then they are equivalent.
The typical use case is to have a simple, clearly correct C_1, and a complex C_2 to verify.

Temporal models of distributed algorithms

Models of cache algorithms

Temporal models of distributed algorithms

110 *6. DISTRIBUTED CONSENSUS WITH PROCESS FAILURES*

other processes; this can help to make the algorithm descriptions more uniform. These messages are technically not permitted in the model, but there is no harm in allowing them because the fictional transmissions could just be simulated by local computation.

***EIGStop* algorithm:**

For every string x that occurs as a label of a node of T, each process has a variable $val(x)$. Variable $val(x)$ is used to hold the value with which the process decorates the node labelled x. Initially, each process i decorates the root of its tree with its own initial value, that is, it sets its $val(\lambda)$ to its initial value.

Round 1*:* Process i broadcasts $val(\lambda)$ to all processes, including i itself. Then process i records the incoming information:

1. If a message with value $v \in V$ arrives at i from j, then i sets its $val(j)$ to v.
2. If no message with a value in V arrives at i from j, then i sets $val(j)$ to *null*.

Round k, $2 \leq k \leq f+1$*:* Process i broadcasts all pairs (x, v), where x is a level $k-1$ label in T that does not contain index i, $v \in V$, and $v = val(x)$.[1] Then process i records the incoming information:

1. If xj is a level k node label in T, where x is a string of process indices and j is a single index, and a message saying that $val(x) = v \in V$ arrives at i from j, then i sets $val(xj)$ to v.
2. If xj is a level k node label and no message with a value in V for $val(x)$ arrives at i from j, then i sets $val(xj)$ to *null*.

At the end of $f+1$ rounds, process i applies a decision rule. Namely, let W be the set of non-*null* *val*s that decorate nodes of i's tree. If W is a singleton set, then i decides on the unique element of W; otherwise, i decides on v_0.

It should not be hard to see that the trees get decorated with the values we indicated earlier. That is, process i's root gets decorated with i's input value. Also, if process i's node labelled by the string $i_1 \ldots i_k$, $1 \leq k \leq f+1$, is decorated by a value $v \in V$, then it must be that i_k has told i at round k that i_{k-1} has told

[1] In order to fit our formal model, in which only one message can be sent from i to each other process at each round, all the messages with the same destination are packaged together into one large message.

Nodes in distributed algorithms are often specified in terms of interacting automata; the temporal model directly results from their interaction.

See IB Concurrent and Distributed Systems

Distributed Algorithms, by Nancy Lynch.

Models of cache algorithms

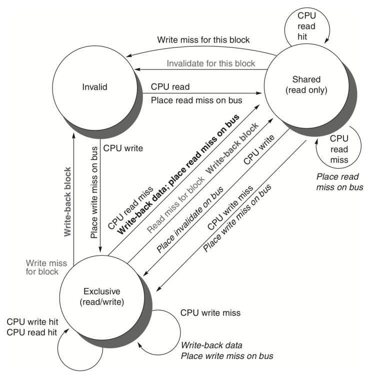


Figure 5.7 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. As in Figure 5.6, the activities on a transition are shown in bold.

Cache algorithms are also often specified in terms of interacting automata (they are distributed algorithms too).

See Section 21.5.2.1 German's Protocol in the Handbook of Model Checking.

Computer Architecture: A Quantitative Approach, by Hennessy & Patterson.

Models of security protocols

Models of security protocols

Given a security protocol, define a temporal model where a state contains:

- the state of each agent
- the set of messages sent
- the set of all the messages that can be deduced from the messages sent; this includes taking messages apart, and reassembling them, including via hashing or encrypting using known keys

and where there is a transition from one state to another when

- an agent sends a message
- an adversary sends a deducible message to an agent

See Chapter 22 Model Checking Security Protocols, in the Handbook of Model Checking.

Remark on examples

As illustrated, interesting programs are big, often too big to work on by hand. This is why we use model checkers.

We cannot easily work with such examples here. Instead, we will mostly look at toy examples like the cabbage-goat-wolf puzzle here.

Summary

Temporal models make it possible to describe systems that evolve in time. Model checking allows checking temporal properties of such models.

Temporal models have to capture the relevant parts of an artefact. They can sometimes be extracted directly, for example from circuits, or are hand-crafted to do that.

In the next lecture, we will see how to use temporal logic(s) to specify the behaviour of temporal models.