Formal Modelling and Verification

Design and Integration of Embedded Systems

István Majzik
The role of formal verification
Example software lifecycle (V-model)

Is the design correct w.r.t. the specification?
Answer: Formal (mathematically precise) modelling + verification of properties
Techniques and measures in standards


- Example: Software architecture design

<table>
<thead>
<tr>
<th>Technique/Measure*</th>
<th>Ref</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Fault detection and diagnosis</td>
<td>C.3.1</td>
<td>---</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>2 Error detecting and correcting codes</td>
<td>C.3.2</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>3a Failure assertion programming</td>
<td>C.3.3</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3b Safety bag techniques</td>
<td>C.3.4</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3c Diverse programming</td>
<td>C.3.5</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>3d Recovery block</td>
<td>C.3.6</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3e Backward recovery</td>
<td>C.3.7</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3f Forward recovery</td>
<td>C.3.8</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3g Re-try fault recovery mechanisms</td>
<td>C.3.9</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>3h Memorising executed cases</td>
<td>C.3.10</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>4 Graceful degradation</td>
<td>C.3.11</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>5 Artificial intelligence - fault correction</td>
<td>C.3.12</td>
<td>---</td>
<td>NR</td>
<td>NR</td>
<td>NR</td>
</tr>
<tr>
<td>6 Dynamic reconfiguration</td>
<td>C.3.13</td>
<td>---</td>
<td>NR</td>
<td>NR</td>
<td>NR</td>
</tr>
<tr>
<td>7a Structured methods including for example, JSD, MASCOT, SADT and Yourdon.</td>
<td>C.2.1</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>7b Semi-formal methods</td>
<td>Table B.7</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>7c Formal methods including for example, CCS, CSP, HCL, LOTOS, OBJ, temporal logic, VDM and Z</td>
<td>C.2.4</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
</tbody>
</table>

Table A.2 – Software design and development: software architecture design (see 7.4.3)

NOTE – The measures in this table concerning fault tolerance (control of failures) should be considered with the requirements for architecture and control of failures for the hardware of the programmable electronics in IEC 61508-2.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.
Goals of formal modeling and verification

- System model
- Formalized properties

Automated model checker tool

Correct: OK
Faulty: Counter-example
Modeling with timed automata
Goals of formal modeling and verification

- Modeling with timed automata
- Timed automata can also be derived from higher-level models (e.g., from UML state machines)
Automata and variables

- **Goal:** Modeling event driven, state based behavior

- **Basic formalism:** *Finite state machine* (FSM)
  - Control locations (with names), as part of the state of the FSM
  - Transitions among control locations

- **Extension:** *Using integer variables*
  - Modelling computations with integer arithmetic
  - Types and ranges of potential values can be specified
  - Constants can be defined

- **Using integer variables on transitions**
  - **Guard:** Conditions on variables
    (guard shall be true in order to enable the transition)
  - **Action:** Assignments to the variables
Extensions using clock variables

- **Goal:** Modelling time dependent behavior
  - Time passes in given states of the component
  - Relative time measurement by resetting and reading timers; behavior depends on timer value (e.g., timeout)

- **Model extension:** Clock variables
  - Represent timers
  - Automatically measure time elapse by a uniform constant rate

- **Using clock variables on transitions:**
  - **Guard:** Condition over clock variables and constants
  - **Action:** Resetting selected clock variables (independently)

- **Use of clock variables in control locations:**
  - **State invariant:** Condition over clock variables, being in a location is valid until its invariant holds
Example: Revolving door

- **Control location name**: idle, activated = true, wait, closed, opening, closing, open
- **Guard**: $x \geq 5$, $x = 0$
- **Invariant**: $x \leq 5$, $x = 6$, $x = 0$, activated = false
- **Action**: $x \geq 4$, $x = 0$, $x \leq 6$, $x \leq 8$
The value of clock $x$ is in the range $[4, 8]$ when leaving the location open.
Extensions for modeling distributed systems

- **Goal:** Modeling *networks of interacting timed automata*
  - Interaction: Simultaneous execution of transitions in different automata
  - Represents synchronous communication (rendezvous)
    - Sending and receiving of a message occurs at the same time
    - This primitive can also be used to model asynchronous communication

- **Model extension:** *Synchronized actions*
  - Channels for message exchange (synchronous channels)
  - Message sending action: `!` operator on the channel
    Message receiving action: `?` operator on the channel
  - E.g., on the channel `a` the actions are `a!` and `a?`

- **Parameterization**
  - Arrays of channels (indexed)
    - E.g., `a[id]` is a channel indexed by the value of variable `id`
  - Typically used in instances of automaton templates
    - E.g., `Door(bool &id)` with `id` as a parameter
Example: Modeling an interaction (pushing a button)

Declarations:

clock t, u;
chan press;

Switch:

User:

“Receiving a message”
(interaction)

“Sending a message”
(interaction)
Further extensions

- **Broadcast channel**
  - Single sender (able to send without receiver)
  - Several receivers (all synchronized that are ready for synchronization)

- **Urgent channel**: prohibit time delay
  - The synchronization is executed without delay
    (instant transitions are possible before it)

- **Urgent state**: prohibit time delay
  - Time is not allowed to progress in the state

- **Committed state**: Atomic state transitions
  - Before executing the outgoing transition, execution of a transition of another automaton is not allowed:
    the incoming and the outgoing transitions are executed in an atomic operation

![Diagram of broadcast and urgent channels]
Example: Modeling the transfer of messages

Message sequence:

Sender

Start

Connecting

Connecting

Connected

Receiver

Start

Connecting

Connecting

Connected

REQUEST

ACCEPT

ACKNOWLEDGE

DATA

DATA

DATA

DATA

...

Structure of the model:

Sender process

SenderMessage

SenderToReceiver

Receiver process

ReceiverMessage

ReceiverToSender

int SenderMessage;
chan SenderToReceiver;

int ReceiverMessage;
chan ReceiverToSender;

const int REQUEST = 1;
const int ACCEPT = 2;
const int ACKNOWLEDGE = 3;
const int DATA = 4;
Example: Automata models

Sender:

- Committed state between setting message content and synchronization
- Sending is prepared and executed from an urgent state

Receiver:
The UPPAAL tool set

- Development (1999-):
  - Uppsala University, Sweden
  - Aalborg University, Denmark

- Web page (information, downloading, examples):

- Related tools:
  - UPPAAL CoVer: Test generation
  - UPPAAL TRON: On-line testing
  - UPPAAL PORT: Designing component based systems
  - ...

- Commercial version:
Automaton model
Formalizing requirements with temporal logics
Goals of formal modeling and verification

- Precise formalization of properties (requirements) to support automated checking
What are the formalized properties?

An example to illustrate the properties to be formalized:

- **The operating modes** of an air-conditioner:
  - Switched-off, switched-on, faulty, light cooling, strong cooling, heating, ventilating

- **Requirements** for the air-conditioner:
  - After switched-on, it shall start ventilating
  - Strong cooling is allowed only after light cooling
  - Heating shall be followed by ventilating
  - The faulty air-conditioner shall not perform heating
  - ...

© BME-MIT
State based properties

- **Local**: Properties to be evaluated in a given state
  - Evaluation is possible using the current values of the state variables (and clock variables)
  - Example: „In the initial state ventilating shall be provided”

- **Reachability**: Properties to be evaluated on a sequence (trace) of states
  - Evaluation is possible on the state space of the system
    - Example: „Heating shall be followed by ventilating”
  - Typical categories of reachability properties:
    - „Safety” of the system
    - „Liveness” of the system
Safety and liveness properties

- **Safety properties**: Specify that each state shall be safe, i.e., “something bad shall never happen”
  - ”In each state the pressure shall be lower than the critical value.”
  - ”In each operating state the door shall be closed.”
  - “There is no deadlock in the protocol.”
- **Invariant properties** (i.e., for each state)
- **Liveness properties**: Specify that a desired state is reachable, ”something good will happen”
  - “After switch-on, the press shall eventually produce the plate.”
  - “After sending a request the reply shall be received”
  - “The process shall compute the required result”
- **Existential properties** (i.e., for the desired state)
Language to formalize reachability properties

- Reachable states are considered in **logic time**:
  - The present: The current state
  - The next time point: The subsequent state(s)

- **Temporal** operators (referring to logic time) are defined to express the reachability properties
  - Typical temporal operators: „always”, „eventually”, „before”, „until”, „after”, ...
  - **Temporal logic**: Formal language to express propositions qualified in terms of logic time
Temporal logics

- **Linear time:**
  The subsequent states form a linear sequence: each state has only one successor
  → logic time forms a linear timeline

- **Branching time:**
  The subsequent states form a tree structure: each state may have multiple successors
  → logic time forms branching timelines
The computational tree

Automaton (FSM) with labelled states

Computational tree: Structure of the potential successor states
Quantifying paths and characterizing states

- Operators that quantify the paths starting from a given state:
  - A: for all paths from the given state
  - E: for at least one (existing) path from the given state

- Operators that characterize states along a given path:
  - F: for a state eventually along the path ("future")
  - G: for all states along the path ("globally")
  - X: for the next state of the path ("next")
  - U: for states until reaching a specified state ("until")
    - E.g., Yellow U Red means that states shall be labeled with Yellow until reaching a state labeled with Red
The Computational Tree Logic (CTL)

- Composite operators are formed
  - First quantifying paths using operators $A$, $E$; then characterizing states along the path by operators $F$, $G$, $X$, $U$
  - Composite operators:
    - For all paths: $AF$, $AG$, $AX$, $A(\ . \ U \ .)$
    - For at least one path: $EF$, $EG$, $EX$, $E(\ . \ U \ .)$
  - Examples:
    - $EF \text{ Red}$: There shall exist a path where a state with Red is reached
    - $AG \text{ Green}$: For all paths, all states shall be labeled with Green
    - $E(Yellow \ U \ Red)$: For at least one path, states shall be labeled with Yellow until a state with label Red is reached

- UPPAAL: Restricted version of CTL is used
  - $AF$, $AG$, $EF$, $EG$ operators (at the beginning of the formula)
### Summary of temporal operators in UPPAAL

<table>
<thead>
<tr>
<th>Operator</th>
<th>Informal semantics</th>
<th>UPPAAL notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG $\varphi$</td>
<td>For all paths, for all states $\varphi$</td>
<td>$A[] \varphi$</td>
</tr>
<tr>
<td>AF $\varphi$</td>
<td>For all paths, for a state eventually $\varphi$</td>
<td>$A&lt;&gt; \varphi$</td>
</tr>
<tr>
<td>EG $\varphi$</td>
<td>For at least one path, for all states $\varphi$</td>
<td>$E[] \varphi$</td>
</tr>
<tr>
<td>EF $\varphi$</td>
<td>For at least one path, for a state eventually $\varphi$</td>
<td>$E&lt;&gt; \varphi$</td>
</tr>
<tr>
<td>$AG(\varphi =&gt; AF \psi)$</td>
<td>After $\varphi$ always $\psi$</td>
<td>$\varphi --&gt; \psi$</td>
</tr>
<tr>
<td></td>
<td>There is no deadlock</td>
<td>AG not deadlock</td>
</tr>
</tbody>
</table>

$\varphi$ and $\psi$ are Boolean expressions on clocks, variables and location names.
Composite operators for all paths

**AG ϕ**: For all paths, for all states ϕ is true

**AF ϕ**: For all paths, for a state eventually ϕ becomes true
Composite operators for at least one path

- $\text{EG } \varphi$: There is at least one path, where for all states $\varphi$ is true
- $\text{EF } \varphi$: There is at least one path, where eventually $\varphi$ becomes true

- Is there a relation between $\text{AG}$ and $\text{EF}$?
- Is there a relation between $\text{AF}$ and $\text{EG}$?
Conditional reachability

- $\text{AG}(\varphi \Rightarrow \text{AF} \psi) \equiv \varphi \Rightarrow \psi$
  For all paths, for all states: if $\varphi$ is true then it implies that on all paths eventually a state occurs in which $\psi$ becomes true

- Reachability with a timing condition: $\varphi \Rightarrow (\psi \text{ and } x \leq t)$
  where $x$ is a clock variable that is reset when $\varphi$ becomes true
Examples: formalizing properties using temporal logic

Let us consider an air-conditioner

- States are characterized using the following local properties:
  \{Switched-off, Switched-on, Faulty, Cooling, Heating, Ventilating\}

To formalize requirements:

- The local properties can be used in the requirements
- In a state several local properties may hold
- The reachability properties are defined considering behaviour from the initial state of the system
- The behaviour of the air-conditioner may not be known when the properties are formalized
Examples: formalizing properties using temporal logic

States of the air-conditioner are characterized using propositions:
{Switched-off, Switched-on, Faulty, Cooling, Heating, Ventilating}

Examples for formalized properties:

- The air-conditioner shall not perform cooling and heating at the same time:
  \[ \text{AG} \left( \neg (\text{Cooling} \land \text{Heating}) \right) \]

- The ventilating mode shall eventually be turned on:
  \[ \text{AF} (\text{Ventilating}) \]

- The air-conditioner can be operated (being switched on) in such a way that it does not perform cooling:
  \[ \text{EG} (\text{Switched-on} \land \neg \text{Cooling}) \]

- If the air-conditioner is faulty then it shall eventually be switched off:
  \[ \text{AG}(\text{Faulty} => \text{AF} (\text{Switched-off})) \text{ or } \text{Faulty} --> \text{Switched-off} \]
Model checking

Timed automata model

System model

Formalized properties

Automated model checker

Temporal logic properties

Correct

Faulty

OK

Counter-example
The UPPAAL model checker

- Properties can be formalized using temporal logic
  - Verification of the properties is automated
- Verification is performed by an exhaustive exploration of the state space of the model
  - Breadth-first, or depth-first search can be configured
- Diagnostic trace can be generated
  - Counter-example (for safety properties) or witness (for liveness properties)
  - Shortest, fastest, or some (any) diagnostic trace can be configured
  - The diagnostic trace can be loaded into the simulator to investigate and debug the behaviour
The UPPAAL model checker

Overview

E<> Gate.Occ
E<> Train(0).Cross
E<> Train(1).Cross
E<> Train(0).Cross and Train(1).Stop
E<> Train(0).Cross and (forall (i : id_t) i != 0 imply Train(i).Stop)

Query

E<> Train(0).Cross

Comment

Train 0 can reach crossing.

Status

Established direct connection to local server.
(Academic) UPPAAL version 4.0.7 (rev. 4140), November 2008 -- server.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.7 (rev. 4140), November 2008 -- server.
E<> Train(0).Cross
Property is satisfied.
Counter-example in the simulator
A case study
A solution for the mutual exclusion problem

- 2 processes, 3 shared variables (H. Hyman, 1966)
  - **blocked0**: The first process \((P0)\) wants to enter the critical section
  - **blocked1**: The second process \((P1)\) wants to enter the critical section
  - **turn**: Which process will enter \((P0\) in case of 0, \(P1\) in case of 1)

```java
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section here
    blocked0 = false;
    // Do other things
}

while (true) {
    blocked1 = true;
    while (turn!=1) {
        while (blocked0==true) {
            skip;
        }
        turn=1;
    }
    // Critical section here
    blocked1 = false;
    // Do other things
}
```

Is this algorithm correct?
Properties to be verified

- Mutual exclusion:
  - At most one process can be in the critical section (it shall never happen that two processes are there)

- It is possible to enter the critical section:
  - P0 is able to enter the critical section
  - P1 is able to enter the critical section

- There is no starvation:
  - P0 will eventually enter the critical section on all paths
  - P1 will eventually enter the critical section on all paths

- Freedom from deadlock:
  - The two processes shall not stop executing
The model in UPPAAL (first version)

Declarations:
bool blocked0;
bool blocked1;
int[0,1] turn=0;
system P0, P1;

The P0 automata:

while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}

Modeling techniques used:
• Global declaration of shared variables
• Limiting the range of variables
The model in UPPAAL (second version)

Declarations:

```plaintext
int[0,1] blocked[2];
int[0,1] turn;
P0 = P(0);
P1 = P(1);
```

System:

```
P0, P1;
```

The P template with pid parameter:

```plaintext
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

Modeling techniques used:
- Global declaration of shared variables
- Limiting the range of variables
- The processes are instantiated using the same template
- Instantiation with parameters (here: pid)
- Using arrays for variables (here: blocked)
Formalizing properties in UPPAAL

- **Mutual exclusion:**
  - Only one process may enter the critical section at the same time: $A[] \neg (P0.cs \text{ and } P1.cs)$

- **Freedom from deadlock:**
  - The two processes shall not stop executing: $A[] \neg \text{deadlock}$

- **It is possible to enter the critical section:**
  - P0 is able to enter the critical section: $E<> (P0.cs)$
  - P1 is able to enter the critical section: $E<> (P1.cs)$

- **There is no starvation:**
  - P0 will eventually enter the critical section on all paths: $A<> (P0.cs)$
  - P0 will eventually enter the critical section on all paths: $A<> (P1.cs)$
Verifying the properties in UPPAAL

- There is no deadlock
- It is possible to enter the critical section
  - Each process is able to enter the critical section
- The mutual exclusion property is not satisfied!
  - The model checker produces a diagnostic trace (counter-example): There is a specific interleaved behavior in which both processes are in the critical section at the same time
  - The counter-example can be investigated in the simulator
- Starvation cannot be checked without modelling time-dependent behavior
  - Trivial counter-examples may include “waiting forever” in any state
  - Modifying the model: Urgent states (if valid)
  - Here: there is still a cyclic behavior that results in starvation
Correction of the mutual exclusion

New algorithm by Peterson

- For process P0
  (for P1 it is similar):

**Hyman:**

```java
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

**Peterson:**

```java
while (true) {
    blocked0 = true;
    turn=1;
    while (blocked1==true &&
           turn!=0) {
        skip;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```
Summary: Properties of model checking

- **Advantages:**
  - It offers a complete exploration of the state space of the model
  - It is possible to check **huge state spaces** (using compact representation)
    - $10^{20}$, or even $10^{100}$ states can be checked automatically (in specific cases)
  - There are fully **automated tools**, there is no need to perform manual adjustment, mathematical operations, or heuristics
  - Diagnostic trace is generated, which supports debugging and correction

- **Problems:**
  - **Scalability** is limited (state space must fit into memory)
  - Effective for **control-oriented models**
    - Complex data structures result in huge state space
  - It is not easy to generalize the results
    - If a protocol is correct for 2 processes, is it correct for N processes as well?
  - The formalization of properties is difficult
    - There are different „temporal logic languages“