Advanced Topics in Software Engineering (02265)

Ekkart Kindler

DTU Compute
Department of Applied Mathematics and Computer Science
VI. Formalisation and Analysis
1. Motivation

Questions:

- Why do we use models?
  - Understanding problems / solutions
  - Communication of ideas
  - Code generation / execution
  - Analysis and Verification

- How do we define what models mean?
  - MOF can be defined in itself?!
  - In natural language (typically in English)
  - Mathematics (the ultimate resort in every field)

In particular, when it comes to behaviour models, MOF is not (yet?) powerful enough to define it.
Motivation

Questions:

- How do we make sure that the models are correct?
  - Analyse the models (and the state space)
  - "Formal methods": all kinds of clever techniques to analyse and verify models **efficiently** (avoiding exploring all states explicitly, representing sets of states symbolically, ...)

- How can we be sure the generated code is correct?
  - Define the semantics of both the model and the code
  - Verify that the code generator preserves them
Motivation

As long as we cannot express the meaning of models fully in MOF:

- We need to be able to formalize the syntax and the semantics in mathematics

In our case, mainly the behaviour.

A thorough study would be separate courses: “Formal Methods” “Semantics”, “Verification”, “Model checking”

Here, we confine ourselves to a systematic example: Petri nets
2. Formalising (abstract) syntax

Example: Petri nets

Definition 1 (Petri net)

A Petri net $N = (P, T, F)$ consist of two disjoint sets $P$ and $T$ and a relation $F \subseteq (P \times T) \cup (T \times P)$.

The elements of $P$ are called the places of $N$, the elements of $T$ are called the transitions of $N$, and the elements of $F$ are called the arcs of $N$.

The relation $F$ is also called the flow-relation of $N$.

Sometimes, one requires $P$ and $T$ to be finite sets.
Example: Petri nets

Definition 2 (Marking of a Petri net)

Let $N = (P, T, F)$ be a Petri net. A marking of $N$ is a mapping $m: P \rightarrow \mathbb{N}$. 

Standard symbol for the set of natural numbers: $0, 1, 2, 3, \ldots$
Formalising \textbf{(abstract) syntax}

Example: Petri nets
Example: Petri nets

Definition 1 (Petri net)

A Petri net $N = (P, T, F)$ consist of two disjoint sets $P$ and $T$ and a relation $F \subseteq (P \times T) \cup (T \times P)$.

Definition 2 (Marking of a Petri net)

Let $N = (P, T, F)$ be a Petri net. A marking of $N$ is a mapping $m: P \rightarrow \mathbb{N}$.

Definition 3 (Petri net system)

Let $N$ be a Petri net and let $m_0$ be a marking of $N$. Then, we call $\Sigma = (N, m_0)$ a Petri net system.
Formalising (abstract) syntax

Example: Place/Transition system

Definition 1 (Petri net)

A Petri net $N = (P, T, F)$ consist of two disjoint sets $P$ and $T$ and a relation $F \subseteq (P \times T) \cup (T \times P)$.

Definition 2 (Marking of a Petri net)

Let $N = (P, T, F)$ be a Petri net. A marking of $N$ is a mapping $m: P \rightarrow \mathbb{N}$.

Definition 4 (Place/Transition system)

Let $N = (P, T, F)$ be a Petri net, let $m_0$ be a marking of $N$ and $W: F \rightarrow \mathbb{N} \setminus \{0\}$. Then, we call $\sum = (N, W, m_0)$ a Place/Transition-system (P/T-system).
Observations

- Nodes of a formalism represented as sets
  - different sets for different kinds of nodes
  - different kind: disjointness of sets

- Arcs between nodes as a relation
  - Constraints in form of a restriction

- Labels as mappings

- Definitions systematically build on each other (kind of modular)

Conceptually, tokens are labels of places (number of tokens).
3. Formalising semantics

Example

![Diagram of a model involving states and transitions between them, labeled as request₁, critical₁, idle₁, semaphor, critical₂, request₂, idle₂.]
Firing rule

1. request<sub>1</sub> -> critical<sub>1</sub>
2. critical<sub>1</sub> -> request<sub>1</sub>
3. semaphor -> idle<sub>1</sub>
4. idle<sub>1</sub> -> semaphor
5. request<sub>2</sub> -> critical<sub>2</sub>
6. critical<sub>2</sub> -> request<sub>2</sub>
7. semaphor -> idle<sub>2</sub>
8. idle<sub>2</sub> -> semaphor
Reachability graph

Example

\[
\begin{align*}
&[i_1, s, i_2] \\
&[r_1, s, i_2] & [i_1, s, r_2] \\
&[c_1, i_2] \\
&[r_1, s, r_2] & [i_1, c_2] \\
&[c_1, r_2] \\
&[r_1, c_2] \\
\end{align*}
\]
Formalising semantics

Example: Petri nets

Let $N = (P, T, F)$ be a Petri net and $t \in T$ be a transition.

The marking $-t : P \rightarrow \mathbb{N}$ is defined by:

$-t(p) = 1$, if $(p,t) \in F$, and

$-t(p) = 0$, if $(p,t) \notin F$

The marking $t^+ : P \rightarrow \mathbb{N}$ is defined by:

$t^+(p) = 1$, if $(t,p) \in F$, and

$t^+(p) = 0$, if $(t,p) \notin F$

The relations $\geq$, $\leq$, and the operations $+$ and $-$ carry over to markings (pointwise).
Formalising semantics

Example: Petri nets

Let \( N = (P, T, F) \) be a Petri net, \( t \in T \) be a transition, and \( m \) be a marking of \( N \).

A transition \( t \) is enabled in marking \( m \), if \( m \geq -t \).

Then, we write \( m \xrightarrow{t} \).

If the transition \( t \) is enabled in \( m \), the transition can fire, which results in the successor marking \( m' = (m - t^+) + t^+ \).

Then, we write \( m \xrightarrow{t} m' \).
Example: Petri nets

Definition 7 (Reachable markings)

Let $\Sigma = (N, m_0)$ be a Petri net system. The set of reachable markings $R_\Sigma$ of $\Sigma$ is defined as the least set, such that

1. $m_0 \in R_\Sigma$
2. if $m \in R_\Sigma$ and there exists a transition $t$ of $N$ and a marking $m'$ such that $m \xrightarrow{t} m'$, then also $m' \in R_\Sigma$

This inductive definition actually, “defines” an algorithm to calculate all reachable states (if the set is finite).
The way of defining the behaviour very much depends on the formalism, but

- Typically there is some notion of state (markings in our example)
- There is one (or more) initial state
- There is a transition relation $m \xrightarrow{t} m'$
The inductive definition of the reachable states gives an algorithm for computing it (in the finite case):

\[ R := \{ \} \quad \text{\textcopyright set of already found reachable states} \]
\[ U := \{ m_0 \} \quad \text{\textcopyright set of states that are yet undealt with} \]

while \( U \neq \{ \} \) do
  select any \( m \in U \)
  \[ U := U \setminus \{ m \} \]
  \[ R := R \cup \{ m \} \]
  for each \( m' \) with \( m \rightarrow m' \) do
    \[ U := U \cup \{ m' \} \]
result is \( R \)
State space generation

The inductive definition of the reachable states gives an algorithm for computing it (in the finite case):

\[ R := \{ \} \quad \text{// set of already found reachable states} \]
\[ U := \{ m_0 \} \quad \text{// set of states that are yet undealt with} \]

while \( U \neq \{ \} \) do

| select any \( m \in U \)
| \( U := U \setminus \{ m \} \)
| \( R := R \cup \{ m \} \)

for each \( m' \) with \( m \rightarrow m' \) do

| | \( U := U \cup \{ m \} \)

result is \( R \)

---

Warning: This algorithm does not terminate—even when there are only finitely many reachable states!

Why?

As soon as there is some cycle in the reachability graph, this algorithm does not work!
State space generation

The inductive definition of the reachable states gives an algorithm for computing it (in the finite case):

\[ R = \{ \} \]
\[ U = \{ m_0 \} \]

while \( U \neq \{ \} \) do

- select any \( m \in U \)

\[ U := U \setminus \{ m \} \]
\[ R := R \cup \{ m \} \]

for each \( m' \) with \( m \rightarrow m' \) do

- if \( m' \notin R \) then \( U := U \cup \{ m' \} \)

result is \( R \)
State space generation

Where are the bottlenecks?

\[ R = \{ \} \]
\[ U = \{ m_0 \} \]

while \( U \neq \{ \} \) do

1. select any \( m \in U \)
\[ U := U \setminus \{ m \} \]
\[ R := R \cup \{ m \} \]

for each \( m' \) with \( m \rightarrow m' \) do

1. if \( m' \notin R \) then \( U := U \cup \{ m \} \)

result is \( R \)

Sometimes, identifying which transitions are possible requires some effort.

For Petri nets, this can be done much more efficiently if we remember which transitions have been enabled in \( m \) in order to calculate the ones that are enabled in \( m' \).

Similar techniques exist for other formalisms. The details very much depend on the formalism.

\( R \) can be huge! Iterating over it for checking \( \notin \) is a bad idea! Use hash function or another good idea.
Checking properties on the fly

\[ R = \{ \} \]
\[ U = \{ m_0 \} \]

while \( U \neq \{ \} \) do
- select any \( m \in U \)
  \[ U := U \setminus \{ m \} \]
  \[ R := R \cup \{ m \} \]
  for each \( m' \) with \( m \rightarrow m' \) do
    if \( m' \notin R \) then \( U := U \cup \{ m \} \)

result is \( R \)
5. Model checking

Here we consider model checking (or some techniques from model checking) as an example for the systematic analysis of the state space.

Today, model checking is a field of its own and could cover a full 10 ECTS-point course; here, we just give an overview.
5.1. Terminology

- Model Checking
- Validation and Verification
- Reactive System
Model checking is a technology for the fully automatic verification of reactive systems with a finite state space.

Which should typically be quite small. Some advanced techniques can even deal with infinite state spaces.
Terms

- **Technology**
  - principle
  - method
  - concept
  - notation
  - tool

- **System**
  - reactive vs. transformational
  - model

- **Validation**
  - requirements
  - specification
  - simulation
  - test
  - verification
    - deductive
    - model based
Validation

Question: Does the system do what it should do?
Validation

Problems:

- requirements are informal in most cases, imprecise, incomplete, inconsistent, ...
- systems can be very complex
- designing and building systems is very expensive
- the later a flaw is detected the higher the costs to repair it
Validation

requirements

system

validation

formalisation

abstraction

refinement / implementation

(formal)

specification

model

validation

verification

or code generation!
Remarks:

- most requirements are informal
- validation is an inherently informal process
- checking whether a specification captures the requirements is inherently informal

- verification is a formal process (automatic in some cases) that can partially help with validation
Transformational System

- accepts some **input**
- makes some **calculations**
- returns a **result**

In particular:

- terminates always (resp. should terminate)
- no user interaction possible (after the input was accepted)
Reactive System

- reacts permanently to input
- can output results any time (dependent on the input)

In particular:
- is interactive (could even be active or proactive)
- does not terminate (normally)
- reactive systems do not „calculate a function“
Reactive vs. transformational

Information systems are reactive (in most cases)

The classical notions of algorithm and computation are defined from the transformational system’s point of view

Reactive systems have transformational components in most cases
Model checking is tailored to the verification of reactive systems

- special notations for “reactive properties“ (temporal logics)
- abstraction from transformational parts (and often from data)
- appropriate for cyclic behaviour
- but on a high level of abstraction only
Summary

Model checking is a technology for the fully automatic verification of reactive systems with a finite state space.
5.2. Main Concepts and Ideas

- Kripke structures (defining the system/model)
- CTL (specifying the properties)
- algorithms (only basic idea)
- complexity
Systems and Requirements

System meets requirements
Model und Specification

model $M$

Kripke structure

specification $A$

\[ \text{AG ( } a \Rightarrow \text{AF } b \text{ )} \]

Computation Tree Logic (CTL)
A Kripke structure consists of

- a set of **states**,
- with distinguished **initial states**,
- a (total) **transition relation** and
- a **labelling** of states with a set of **atomic propositions**.
The **behaviour** at a state can be represented as a computation tree:

Note that all paths are infinite!

That is a consequence of the totality of the transition relation.
**CTL-Formulas**

\[ a, b, \ldots \]

\[ \land, \lor, \neg, \ldots \]

\[ \text{EX} \], \[ \text{EG} \], \[ \text{E} \lbrack \ . \ U \ . \ rbrack \], \ldots \]

**CTL-formulas** are inductively defined:

- atomic propositions are CTL-formulas
- CTL-formulas combined with a Boolean operator are CTL-formulas
- CTL-formulas combined with temporal operators are CTL-formulas
there exists an (immediate) successor in which $p$ holds true:
**Exists Globally:** \( \text{EG} \ p \)

there exists an infinite path on which \( p \) holds in each state:
\textbf{Exists Until: } \( \exists[p \ U q] \)

there exists a reachable state in which \( b \) holds true, and up to this state \( p \) holds true:
Abbreviations

\( \text{AX} \ p \equiv \neg \ \text{EX} \ \neg \ p \)
for all immediate successors, \( p \) holds true

\( \text{EF} \ p \equiv \ E \ [ \ true \ U \ p \ ] \)
in some reachable state, \( p \) holds true

\( \text{AG} \ p \equiv \neg \ \text{EF} \ \neg \ p \)
in all reachable states, \( p \) holds true

\( \text{AF} \ p \equiv \neg \ \text{EG} \ \neg \ p \)
on each path, there exists a state in which \( p \) holds true
A CTL-formula \textbf{holds} for a Kripke structure if the formula holds in each initial state.
Example

model $M$

specification $p$

$$\text{AG} \ ( \ a \Rightarrow \ AF \ b \ )$$

How do we prove it?
Algorithms

For each sub-formula, we inductively calculate the set of states, in which this sub-formula is true:

- atomic propositions
- Boolean operators
- temporal operators
"Algorithm" for $\text{EX} \, p$

**Given:**
The set of states in which $p$ holds: $S_p$

**Wanted:**
The set of states in which $\text{EX} \, p$ holds: $S_{\text{EX} \, p}$

We also write $\text{EX}(S_p)$ for $S_{\text{EX} \, p}$
Algorithm for \( E[p \cup q] \)

As stated here, this algorithm is quite inefficient. There are more efficient ways to do this.

But, even this inefficient algorithm turns out to be quite efficient when used with the right data structure (ROBDDs, see next lecture).

**Given:** \( S_p \) und \( S_q \)

**Wanted:** \( S_{E[p \cup q]} \)

\[
S_0 = \emptyset \\
S_1 = S_q \cup (S_p \cap EX(S_0)) \\
S_2 = S_q \cup (S_p \cap EX(S_1)) \\
... \\
S_{i+1} = S_q \cup (S_p \cap EX(S_i)) \\
\]

until \( S_{i+1} = S_i = S_{E[p \cup q]} \)
Algorithm for $\text{EG}_p$

Given: $S_p$
Wanted: $S_{\text{EG}_p}$

\[
S_0 = S \\
S_1 = S_p \cap \text{EX}(S_0) \\
S_2 = S_p \cap \text{EX}(S_1) \\
\ldots \\
S_{i+1} = S_p \cap \text{EX}(S_i) \\
\text{until } S_{i+1} = S_i = S_{\text{EG}_p}
\]
Algorithms Summary

CTL model checking \sim \ marking algorithm + iteration

- **EX** $p$

- **E[p U q]**

- **EG** $p$
When implemented in an efficient way, the marking algorithm for each operator is linear in the number of states of the system:

\[ O(|M| \cdot |p|) \]

- size of the model
- size of the formula
When implemented in an efficient way, the marking algorithm for each operator is linear in the number of states of the system:

\[ O( | M | ) \]