T-79.186 Reactive Systems

Spring 2003, Lecture 1

Keijo Heljanko

February 11, 2003

1 Course Arrangements

The course is a period course on the 1st half of the semester (15.1-5.3).

To pass the course you need to do both (a) and (b):

(a) Give your share of seminar talks. You will get a base grade from the seminar talks.

(b) Do home exercises. There will be 5 rounds of 6 points/round graded as follows:
- At least 50% of the points needed to pass.
- At least 80% of the points gives "+1" to your seminar talk grade.

The weekly course schedule is as follows:

- Tue 8:45 Homework deadline
- Tue 8:45 Homework distribution
- Tue 8:45-10:15 Lectures (Keijo Heljanko)
- Tue 10:15-11:00 Tutorial (Timo Latvala) Homework answers + occasionally demos
- Wed 16:15-18:45 Seminar (Keijo Heljanko and Timo Latvala)

Return the home exercises by email (Postscript or PDF, no Word docs!) to the course assistant Timo Latvala (Timo.Latvala@hut.fi).

The homework schedule is as follows:

- Tue 21.1
  - Distribution of 1st home exercise
- Tue 28.1
  - Deadline of 1st home exercise, distribution of 2nd home exercise
- Tue 4.2
  - Deadline of 2nd home exercise, distribution of 3rd home exercise
- Tue 11.2
  - Deadline of 3rd home exercise, distribution of 4th home exercise
- Tue 18.2
  - Deadline of 4th home exercise, distribution of 5th home exercise
- Tue 25.2
  - Deadline of 5th home exercise

2 Reactive Systems

Reactive systems are a class of software and/or hardware systems which have ongoing behavior. (They do not terminate.)

Examples of reactive systems include:

- Traffic lights
- Elevators (lifts)
- Operating systems
- Data communication protocols (Internet,telephone switches)
- Mobile phones

Reactive systems do not fulfill the definition of an algorithm, which says that an algorithm should:

- Terminate, and
- upon termination, provide a return value.

If we want to specify the correctness of an algorithm, we usually specify it as follows:

- The algorithm should terminate on all inputs, and
- on termination, the provided output should correct (with respect to a specification).

How do we specify the correctness of reactive systems?

How do we check whether the system meets its specification?

We will address the specification and verification of reactive systems in this course.

2.1 The Need for Formal Methods

Hardware and software are widely used in applications where failure is unacceptable (safety or business critical systems): ecommerce, communication networks, air traffic control, medical systems, etc.

Two costly system failures experienced:

- Intel: Pentium FDIV bug (1994, estimated $500 million)
- Ariane 5: floating point overflow (1996, $500 million)

Probably our dependence on critical systems (e.g. the Internet, cars, airplanes, ...) is growing instead of diminishing.

2.2 Hardware and Software Verification

The principal methods for the validation of complex systems are

- Testing (using the system itself)
- Simulation (using a model of the system)
- Deductive verification (proof of correctness by e.g. axioms and proof rules, usually including computer aided proof assistants)
- Model Checking ($\approx$ exhaustive testing of a model of a system)

Reactive systems are often concurrent and sometimes also nondeterministic. This limits the applicability of testing based methods.

Deductive verification needs highly advanced personnel and time. (Has been used in highly critical systems where high cost is not an objective.)

3 Model Checking

Model checking is a technique for verifying reactive systems. It has several advantages over traditional approaches (simulation, testing, deductive reasoning) to this problem.

The method has been successfully used to **verify** circuit designs (e.g. microprocessors), and communication protocols.

The main challenge in using the approach is the **state explosion** problem. Tackling this problem is still the main source of research into model checking.

The books discuss how model checking is used to verify complex reactive systems. Also theoretical and algorithmic aspects of model checking are covered.

Model checking limits itself to systems where decidability is guaranteed (e.g. systems with only finite amount of state bits). Given sufficient amount of **time and memory**, a model checking tool is guaranteed to terminate with a YES/NO answer.

Instances of finite state systems handled with model checking include e.g. hardware controllers and communication protocols.

Also some infinite state systems can be handled with model checking by using **abstraction** or **induction** based techniques. Also in some cases bugs can be found from infinite state systems by restricting them to finite state ones. One can for example model message FIFOs of infinite size with bounded size FIFOs, and still find bugs which appear in the (harder) infinite-state case.

Model checking can be performed automatically, which is different from deductive verification. It can thus be used for automatic regression testing.

3.1 The Process of Model Checking

In model checking process the following phase can be identified:

- Modeling - How to model your system in a way acceptable from a model checking perspective. This can be as easy as compilation or may involve deep insight into the system being modeled.
- Specification - What properties should the system satisfy? Most model checkers use **temporal logic** to specify the properties, but there might be some standard properties one would like to check (deadlock freedom).
- Verification - Push the "model check" button. In practice life is not this easy, and analysis of the model checking results is needed. If for example a property does not hold, where does the bug exist? Model checkers produce **counterexamples** which help in locating the bug. The bug might also be in the specification or the abstractions used in the modeling process, and these must be analyzed carefully.

3.2 Temporal Logic and Model Checking

Temporal logics were originally developed by philosophers for reasoning about the way time was used in natural language. Lots of different temporal logics have been suggested. There are two main branches of logics: **linear time** and **branching time** logics. In this books the meaning of a temporal logic formula will be determined with respect to a **Kripke structure**.

Pnueli was the first to use temporal logics for reasoning about concurrent systems. In the early 1980's Clarke and Emerson in USA, and Quielle and Sifakis in France gave model checking algorithms for branching time logics (CTL). The EMC algorithm by Clarke, Emerson and Sistla was the first linear-time algorithm for CTL in 1987.

Sistla and Clarke analyzed the model checking problem for a linear time logic (LTL) and showed it to be PSPACE-complete. Later Lichtenstein and Pnueli made a more careful analysis which showed that the PSPACE-completeness is only in the size of the formula, and not the state space of the system.

Other temporal logics include CTL* and μ-calculus. Also regular expressions and automata on infinite words (ω-automata) have been used for specification.

We will in this course concentrate on LTL model checking using ω-automata.

4 Automata on Finite Words

Automata on finite words are used to model finite state systems. They can also be used to model specifications for systems.

Definition 4.1 A (nondeterministic finite) automaton $\mathcal{A}$ is a tuple $(\Sigma, S, S^0, \rho, F)$, where

- Σ is a finite **alphabet**,
- S is a finite set of **states**,
- $S^0 \subseteq S$ is set of **initial states**,
- $\rho : S \times \Sigma \to 2^S$ is the **transition function**, and
- $F \subseteq S$ is the set of **accepting states**.

An automaton $\mathcal{A}$ is **deterministic** if $|S^0| = 1$ and $|\rho(s, a)| \leq 1$ for all states $s \in S$ and symbols $a \in \Sigma$.

The meaning of the transition function ρ is as follows: $t \in \rho(s, a)$ means that there is a move from state s to state t with symbol a. An alternative (equivalent) definition gives the transition function as a relation $\Delta \subseteq S \times \Sigma \times S$.

A finite automaton accepts a set of words $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ called the **language** accepted by $\mathcal{A}$, defined as follows:

A **run** r of $\mathcal{A}$ on a finite word $a_0, \ldots, a_{n-1} \in \Sigma^*$ is a sequence $s_0, \ldots, s_n$ of $(n+1)$ states in S, such that $s_0 \in S^0$, and $s_{i+1} \in \rho(s_i, a_i)$ for all $0 \leq i < n$.

The run r is **accepting** iff $s_n \in F$. A word $w \in \Sigma^*$ is accepted by $\mathcal{A}$ iff $\mathcal{A}$ has an accepting run on w.

The language of $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ is the set of finite words accepted by $\mathcal{A}$.

A language of automaton $\mathcal{A}$ is said to be **empty** when $\mathcal{L}(\mathcal{A}) = \emptyset$.

We will now start defining the Boolean operators on finite automata:
$\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{A}_1 \cap \mathcal{A}_2$, and $\overline{\mathcal{A}}$.

We start by $\cup$:

Definition 4.2 Let $\mathcal{A}_1 = (\Sigma, S_1, S_1^0, \rho_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, S_2, S_2^0, \rho_2, F_2)$. (We furthermore assume the the automata are disjoint.) We define the **union** automaton to be $\mathcal{A} = (\Sigma, S, S^0, \rho, F)$, where:

- $S = S_1 \cup S_2$,
- $S^0 = S_1^0 \cup S_2^0$,
- $F = F_1 \cup F_2$, and
- $\rho(s, a) = \rho_1(s, a)$ if $s \in S_1$, and $\rho_2(s, a)$ otherwise.

Now for the union automaton $\mathcal{A}$ (also denoted by $\mathcal{A}_1 \cup \mathcal{A}_2$) it holds that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$.

Next we define $\cap$:

Definition 4.3 Let $\mathcal{A}_1 = (\Sigma, S_1, S_1^0, \rho_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, S_2, S_2^0, \rho_2, F_2)$. (We furthermore assume the the automata are disjoint.) We define the **product** automaton to be $\mathcal{A} = (\Sigma, S, S^0, \rho, F)$, where:

- $S = S_1 \times S_2$,
- $S^0 = S_1^0 \times S_2^0$,
- $F = F_1 \times F_2$, and
- $\rho((s,t),a) = \rho_1(s,a) \times \rho_2(t,a)$.

Now for the product automaton $\mathcal{A}$ (also denoted by $\mathcal{A}_1 \times \mathcal{A}_2$) it holds that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

The definition of complementation is slightly more complicated.

We say that an automaton has a **completely specified transition function** $|\rho(s, a)| \geq 1$ for all states $s \in S$ and symbols $a \in \Sigma$.

We first give a definition which **only works for deterministic automata**:

Definition 4.4 Let $\mathcal{A}_1 = (\Sigma, S_1, S_1^0, \rho_1, F_1)$ be a **deterministic** automaton with a completely specified transition function. We define the **deterministic complement** automaton to be $\mathcal{A} = (\Sigma, S, S^0, \rho, F)$, where:

- $S = S_1$,
- $S^0 = S_1^0$,
- $\rho = \rho_1$, and
- $F = S_1 \setminus F_1$.

Now for the automaton $\mathcal{A}$ (also denoted by $\overline{\mathcal{A}_1}$) it holds that $\mathcal{L}(\mathcal{A}) = \Sigma^* \setminus \mathcal{L}(\mathcal{A}_1)$.