T-79.186 Reactive Systems: Introduction and Finite State Automata

Timo Latvala

14.1.2004

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 (hopefully correct) return value.

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

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

Hardware and Software Verification

The principal methods for the validation and verification 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)

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.

The Need for Formal Methods

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.)

What is Needed to Formally Verify Reactive Systems?

We need to address the following issues.

- How do we build a model of a reactive system?
- How do we specify the correctness of reactive systems?
- How do we check whether the system meets its specification?

We will address the issues of modeling, specification, and verification in this course.

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. Reachability analysis can be seen as a very basic model checking approach.

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.

Model checking limits itself to systems where decidability is guaranteed (e.g. systems with only a finite number 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.

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 some of the bugs bugs which appear in the (harder) infinite-state case. Note that if no bugs are found in the finite-state version, that does not mean that the infinite-state version is correct!

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

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. In this course we will mainly use Petri nets as the modeling formalism.
- 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 (e.g. 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 in the system model, so these must be analyzed carefully.

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. 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 mainly concentrate on LTL model checking using ω-automata.

We will start by first introducing finite state automata. They can be seen as a "helper formalism" used inside model checkers to implement model checking algorithms. They can be used to implement model checking of so called safety properties, as well as used to model systems themselves.

Automata on Finite Words

Automata on finite words can also be used to model finite state systems, as well as specifications for systems.

Definition 1 A (nondeterministic finite) automaton $\mathcal{A}$ *is a tuple* $(\Sigma, S, S^0, \Delta, F)$*, where*

- Σ *is a finite* alphabet*,*
- S *is a finite set of* states*,*
- $S^0 \subseteq S$ *is set of* initial states*,*
- $\Delta \subseteq S \times \Sigma \times S$ *is the* transition relation*, and*
- $F \subseteq S$ *is the set of* accepting states*.*

An automaton $\mathcal{A}$ is *deterministic* if $|S^0| = 1$ and for all pairs $s \in S, a \in \Sigma$ it holds that if for some $s' \in S$: $(s, a, s') \in \Delta$ then there is no $s'' \in S$ such that $s'' \neq s'$ and $(s, a, s'') \in \Delta$. (I.e., there is only at most one state which can be reached from s with a.)

The meaning of the transition relation $\Delta \subseteq S \times \Sigma \times S$ is the following: $(s, a, s') \in \Delta$ means that there is a move from state s to state s' with symbol a.

An alternative (equivalent) definition gives the transition relation as a function $\rho : S \times \Sigma \rightarrow 2^S$, where $\rho(s, a)$ gives the set of states to which the automaton can move with a from state s.

Synonyms for the word automaton are: finite state machine (FSM), finite state automaton (FSA), nondeterministic finite automaton (NFA), and finite automaton on finite strings.

Runs and Languages

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 $(s_i, a_i, s_{i+1}) \in \Delta$ 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 $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ is the set of finite words accepted by $\mathcal{A}$.

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

Operations with Automata

To make ourselves more familiar with finite state automata, we will show how simple operations with them can be performed.

We will do this by defining the Boolean operators for finite automata: $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{A} = \mathcal{A}_1 \cap \mathcal{A}_2$, and $\mathcal{A} = \overline{\mathcal{A}_1}$.

These operations will as a result have an automaton $\mathcal{A}$, such that: $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2), \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$, and $\mathcal{L}(\mathcal{A}) = \overline{\mathcal{L}(\mathcal{A}_1)}$, respectively.

In the following we furthermore assume the the automata are disjoint (i.e., they have no states in common), and that they have the same alphabet Σ.

We start by $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$:

Definition 2 *Let $\mathcal{A}_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, S_2, S_2^0, \Delta_2, F_2)$. We define the* union *automaton to be $\mathcal{A} = (\Sigma, S, S^0, \Delta, F)$, where:*

- $S = S_1 \cup S_2$,
- $S^0 = S_1^0 \cup S_2^0$,
- $\Delta = \Delta_1 \cup \Delta_2$, *and*
- $F = F_1 \cup F_2$.

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 $\mathcal{A} = \mathcal{A}_1 \cap \mathcal{A}_2$:

Definition 3 *Let* $\mathcal{A}_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ *and* $\mathcal{A}_2 = (\Sigma, S_2, S_2^0, \Delta_2, F_2)$*. We define the* product *automaton to be* $\mathcal{A} = (\Sigma, S, S^0, \Delta, F)$*, where:*

- $S = S_1 \times S_2$,
- $S^0 = S_1^0 \times S_2^0$,
- *for all* $s, s' \in S_1, t, t' \in S_2, a \in \Sigma$:
  $((s,t), a, (s',t')) \in \Delta$ *iff* $(s, a, s') \in \Delta_1$ *and* $(t, a, t') \in \Delta_2$*; and*
- $F = F_1 \times F_2$.

Now for the product automaton $\mathcal{A}$ (also denoted by $\mathcal{A}_1 \times \mathcal{A}_2$ and $\mathcal{A}_1 \cap \mathcal{A}_2$) 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 relation* if for all states $s \in S$ and symbols $a \in \Sigma$ there exist a state $s' \in S$ such that $(s, a, s') \in \Delta$.

Any automaton which does not have a completely specified transition relation can be turned into one by:

- adding a new *sink state* q_s,
- making q_s a non-accepting state,
- adding for all $a \in \Sigma$ an arc (q_s, a, q_s), and
- for all pairs $s \in S, a \in \Sigma$: if there is no state s' such that $(s, a, s') \in \Delta$, then add an arc (s, a, q_s). (Add all those arcs which are still missing to fulfill the completely specified property.)

Note that this construction does not change the language accepted by the automaton.

We first give a complementation definition which *only works for completely specified deterministic automata*!

Definition 4 *Let* $\mathcal{A}_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ *be a* deterministic *automaton with a completely specified transition relation. We define the* deterministic complement *automaton to be* $\mathcal{A} = (\Sigma, S, S^0, \Delta, F)$*, where:*

- $S = S_1$,
- $S^0 = S_1^0$,
- $\Delta = \Delta_1$, *and*
- $F = S_1 \setminus F_1$.

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