Modeling and Analyzing Concurrent Systems using Model Checking

Robert B. France
What is Model Checking?

• “Model checking is an automated technique that, given a **finite-state model** of a system and a **logical property**, systematically checks whether this property holds for (a given initial state in) that model.” [Clarke & Emerson 1981]:

• Model checking tools automatically verify whether $M \models \varphi$, holds, where $M$ is a **(finite-state) model of a system and property** $\varphi$ (phi) characterizes a set of allowed behaviors.
  
  – M has behavior that is allowed by $\varphi$
  
  – Check that M is a model of $\varphi$
Model Checking process

1. Construct a model of the system (M)
2. Formalize the properties that will be evaluated in the model (P)
3. Use a model checker to determine if M satisfies P. Three results are possible:
   1. The model M satisfies the property P, i.e. M |= P
   2. M does not satisfy P; in this case a counterexample is produced
   3. No conclusive result is produced by the model checker (model checker ran out of space or time)
The eagle’s view

• What is a transition system?
  – Description of system behavior

• What is a linear time property?
  – Set of behaviors that satisfy the property

• How do we check the satisfaction property algorithmically?
  – Convert temporal properties to automatons
  – Compose automatons with transition system descriptions of behavior
Transition System (TS): Formal Definition

A transition system $TS$ is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- $S$ is a set of states,
- $Act$ is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation (the first element in the triplet is the source state, the second element is an action and the third element is the target state of the transition),
- $I \subseteq S$ is a set of initial states,
- $AP$ is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labeling function ($2^{AP}$ is the power set of $AP$)

$TS$ is called *finite* if $S$, $Act$, and $AP$ are finite.

$(s, act, s')$ in $\rightarrow$ is written as $s \xrightarrow{a} s'$

$L(s)$ are the atomic propositions in $AP$ that are satisfied in state $s$. Given a formula, $f$, a state $s$ satisfies $f$ (i.e., is a model of $f$) if and only if $f$ can be derived from the atomic propositions associated with state $s$ via the labeling function $L$, that is:

$s \models f$ iff $L(s) \models f$
Toy Example

The atomic propositions in a transition system are chosen based on the properties the modeler wants to check.

Example property to verify: The vending machine only delivers a drink after the user pays (inserts a coin).

Relevant atomic propositions: AP = \{paid, delivered\}

Appropriate Labeling function:

L(pay) = empty set
L(soda)=L(beer)=\{paid, delivered\}
L(select)=\{paid\}
Some TS Operators

- **Post(s)** consists of all the target states associated with s via transitions from s.
- The **state graph** of a TS = (S, Act, ->, I, AP, L), G(TS) is the digraph (V, E) with vertices V = S and edges E = {(s,s’) ∈ S x S | s’ ∈ Post(s)}
  - G(TS) is obtained by omitting all atomic propositions in states, and all action labels.
  - Initial states are not distinguished in a state graph.
  - Multiple transitions between two states are represented by one edge in a state graph.
- **Post*(s)**: the set of states that are reachable from s in a state graph.
- If C is a set of states then Post*(C) = U s ∈ C Post*(s)
Modeling concurrent systems that manipulate data

- In software the transition from one state to another often depends on conditions expressed in terms of data
  - **Conditional transitions** are higher-level constructs used to describe actions that are performed only under certain conditions

- Models with conditional transitions are called **program graphs**
  - Program graphs are “higher-level” in that they can be transformed into TSs (Note: TSs do not have conditional transitions) via a process called **unfolding**
Program Graph (PG): Formal Definition

A program graph $PG$ over set $Var$ of typed variables is a tuple $(Loc, Act, Effect, \rightarrow, Loc_0, g_0)$ where

- $Loc$ is a set of locations and $Act$ is a set of actions,
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ is the effect function,
  - $Eval(Var)$ is the set of assignments of values to variables in $Var$, e.g., \{<nbeer:= 10, nsoda:=20>, <nbeer:= 1, nsoda:=20>, <nbeer:=0, nsoda:=4>, ...\} is the set of assignments when $Var = \{nbeer, nsoda\}$
- $\rightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$ is the conditional transition relation,
  - $Cond(Var)$ is the set of all Boolean conditions (propositions) over $Var$
- $Loc_0 \subseteq Loc$ is a set of initial locations,
- $g_0 \in Cond(Var)$ is the initial condition.
select and start are called locations
nsoda, and nbeer are variables
coin, refill, sget, bget, ret_coin are actions
A simple text representation of the vending machine PG

start:
  coin; go to select
  refill{nsoda := max; nbeer := max}; go to start
select:
  nsoda > 0:: sget{nsoda := nsoda -1}; go to start
  nbeer > 0:: bget{nbeer := nbeer-1}; go to start
  nsoda = 0 and nbeer = 0:: ret_coin; go to start
Unfolding the vending machine PG
TS semantics of program graphs

• The TS is produced by unfolding the program graph
  – You can think of unfolding as a representation of the execution of a program described by a PG

• A state consists of a location (a point in the program) and an assignment of values to variables: \(<l, \eta>\)

• An initial state consists of an initial location and an assignment that satisfies the condition \(g_0\) defined in the PG
  – \(<l_0, \eta>\) is an initial state if \(l_0\) is an initial location and \(\eta \models g_0\)

• The propositions consists of the locations together with Cond(Var)
  – The proposition \(\text{loc}\) is true in any state of the form \(<\text{loc}, \eta>\), and false otherwise
The transition system $TS(PG)$ of program graph

$$PG = (Loc, Act, Effect, \rightarrow, Loc_0, g_0)$$

over set $\text{Var}$ of variables is the tuple $(S, Act, \rightarrow, I, AP, L)$ where

- $S = Loc \times \text{Eval}(\text{Var})$
- $\rightarrow \subseteq S \times Act \times S$ is defined by the following rule (see remark below):

$$\frac{\ell \xrightarrow{g;\alpha} \ell' \quad \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle}$$

- $I = \{\langle \ell, \eta \rangle \mid \ell \in Loc_0, \eta \models g_0\}$
- $AP = Loc \cup \text{Cond}(\text{Var})$
- $L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{g \in \text{Cond}(\text{Var}) \mid \eta \models g\}$. 
Types of parallel composition operators

• **Interleaving**
  – Actions of concurrent processes are interleaved in a non-deterministic manner
  – Used to model processes whose behaviors are completely independent (*asynchronous* system of processes)

• **Communication via shared variables**
  – A process can influence the behavior of another process by changing the value of a variable that is shared with the process

• **Handshaking**
  – Two processes that want to interact must synchronize their actions such that they take part in the interaction at the same time

• **Channel systems**
  – In a channel system processes interact by reading from and writing to channels connecting them
Behavior: executions, paths, traces

- A **finite/infinite execution fragment** of a TS is a finite/infinite sequence of state transitions.
  - $s_0$-act1-$\rightarrow s_1$, $s_1$-act2-$\rightarrow s_3$ is written as an alternating finite execution that ends in a state, $s_0$,act1,$s_1$,act2,$s_3$

- A path fragment is a path $s_0$, $s_1$, $s_2$, ... where $s_1$ in Post($s_0$), $s_2$ in Post($s_1$) etc.
  - **Path($s$)** is the set of maximal path fragments in which the first element is $s$

- The execution $s_0$,act0,$s_1$,act1,$s_2$,act2,$s_3$, ... can be represented as a **trace**, $L(s_0),L(s_1),L(s_2),L(s_3),...$ in a state view of a transition system
  - A trace is thus a **word** over the power set of AP in a transition system $2^{AP}$
Trace operators

• trace(Π) is the set of traces obtained from the paths in the set of paths, Π
  
  \[ \text{trace}(Π) = \{ \text{trace}(\pi) \mid \pi \in Π \} \]

• Traces(s) is the set of traces of s
  
  \[ \text{Traces}(s) = \text{traces}(\text{Paths}(s)) \]

• Traces(TS) is the set of all traces for all initial states of TS
  
  \[ \text{Traces}(\text{TS}) = \bigcup_{s \in \text{in} \text{TS}} \text{Traces}(s) \]
LT property

• A linear temporal property over a set of atomic propositions, AP is a subset of the set of all infinite words formed using only elements in AP (denoted $(2^{AP})^\omega$)

Definition 3.11. Satisfaction Relation for LT Properties

Let $P$ be an LT property over AP and $TS = (S, Act, \rightarrow, I, AP, L)$ a transition system without terminal states.

$TS = (S, Act, \rightarrow, I, AP, L)$ satisfies $P$, denoted $TS \models P$, iff $\text{Traces}(TS) \subseteq P$. State $s \in S$ satisfies $P$, notation $s \models P$, whenever $\text{Traces}(s) \subseteq P$. 

Starvation Freedom Example

- A process that wants to enter its critical section will eventually do so ($AP = \{ \text{wait1, crit1, wait2, crit2} \}$)
  - $P_{\text{finwait}} = \text{set of infinite words } A_0 A_1 A_2 \ldots \text{ such that } \forall j. \text{wait}_i \in A_j \Rightarrow \exists k \geq j. \text{crit}_i \in A_k \text{ for each } i \in \{1, 2\}$

- A process that waits often enters its critical section often
  - $P_{\text{nostarve}} = \text{set of infinite words } A_0 A_1 A_2 \ldots \text{ such that: } (\forall k \geq 0. \exists j \geq k. \text{wait}_i \in A_j ) \Rightarrow (\forall k \geq 0. \exists j \geq k. \text{crit}_i \in A_j)$ for each $i \in \{1, 2\}$
    - In abbreviated form we write: $\exists^\infty j. \text{wait}_i \in A_j \Rightarrow \exists^\infty j. \text{crit}_i \in A_j$ for each $i \in \{1, 2\}$, where $\exists^\infty$ stands for “there are infinitely many”.
Trace inclusion and equivalence

- Trace inclusion: TS is a correct implementation of TS’ if \( \text{Traces}(TS) \) is a subset of \( \text{Traces}(TS’) \).
- Equivalent statement: For any LT property \( P \): \( TS’ \models P \) implies \( TS \models P \).
- Transition systems \( TS \) and \( TS’ \) are trace-equivalent with respect to the set of propositions \( \text{AP} \) if \( \text{Traces}_{\text{AP}}(TS) = \text{Traces}_{\text{AP}}(TS’) \).
- \( \text{Traces}(TS) = \text{Traces}(TS’) \) iff \( TS \) and \( TS’ \) satisfy the same LT properties.
Equivalent TS example

For AP = \{pay, soda, beer\} the two TSs are trace equivalent

There does not exist an LT property that distinguishes between the two vending machine models

Figure 3.8: Two beverage vending machines.
Types of Linear Time Properties

- Important to distinguish between different types of properties because the approach to checking the properties will vary with the type.
- Safety properties: A property that is finitely refutable
  - Informally: Nothing bad ever happens
- Liveness properties: A property that is not finitely refutable
  - Informally: Something good eventually happens
Formal definition of a safety property

An LT property $P_{\text{safe}}$ over $AP$ is called a safety property if for all words $\sigma \in (2^{AP})^\omega \setminus P_{\text{safe}}$ there exists a finite prefix $\sigma^\wedge$ of $\sigma$ such that $P_{\text{safe}} \cap \{\sigma' \in (2^{AP})^\omega \mid \sigma^\wedge \text{ is a finite prefix of } \sigma\} = \emptyset$

- $\sigma^\wedge$ is called a bad prefix for $P_{\text{safe}}$
- A bad prefix is minimal if there is no smaller prefix that is bad
- $\text{BadPref}(P_{\text{safe}})$ denotes set of all bad prefixes for $P_{\text{safe}}$
Satisfying safety properties

$TS \models P_{safe}$ if and only if $\text{Traces}_{fin}(TS) \cap \text{BadPref}(P_{safe}) = \emptyset$

Alternative: $P_{safe}$ is a safety property iff

$\text{closure}(P_{safe}) = P_{safe}$

i.e., $P_{safe}$ contains all the infinite traces whose finite prefixes are also prefixes of $P_{safe}$

$\text{closure}(P) = \{\sigma \in (2^AP)^\omega \mid \text{pref}(\sigma) \subseteq \text{pref}(P)\}$

where $\text{pref}(\sigma)$ is the set of finite prefixes of the word $\sigma$
Trace inclusion and safety properties

• Let TS and TS’ be transition systems without terminal states and with the same set of propositions AP. Then the following statements are equivalent:
  – \( \text{Traces}_{\text{fin}}(TS) \subseteq \text{Traces}_{\text{fin}}(TS') \)
  – For any safety property \( P_{\text{safe}} : TS' \models P_{\text{safe}} \) implies \( TS \models P_{\text{safe}} \)

• Note that even if \( \text{Traces}(TS) \) is not a subset of \( \text{Traces}(TS) \), but the finite traces are (a weaker condition), then safety properties of TS’ also holds for TS
Finite vs. infinite systems

- Traces(TS) is not a subset of Traces (TS) but Traces_{fin}(TS) is a subset of Traces_{fin}(TS').
- Property: eventually b holds.
Liveness definition

• A property $P$ over $AP$ is a liveness property when $\text{pref}(P) = (2^{AP})^*$

• Each finite word can be extended to an infinite word that satisfies $P$

• Stated differently, $P$ is a liveness property iff for all finite words $w \in (2^{AP})^*$ there exists an infinite word $\sigma \in (2^{AP})^\omega$ satisfying $w\sigma \in P$
Examples

• Each process will eventually enter its critical section
  \((\exists j \geq 0. \text{crit}_1 \in A_j) \land (\exists j \geq 0. \text{crit}_2 \in A_j)\)

• Each process will enter its critical section infinitely often
  \((\forall k \geq 0. \exists j \geq k. \text{crit}_1 \in A_j) \land (\forall k \geq 0. \exists j \geq k. \text{crit}_2 \in A_j)\)

• Each waiting process will eventually enter its critical section
  \(\forall j \geq 0. (\text{wait}_1 \in A_j \Rightarrow (\exists k > j. \text{crit}_1 \in A_k)) \land \forall j \geq 0. (\text{wait}_2 \in A_j \Rightarrow (\exists k > j. \text{crit}_2 \in A_k))\)
Safety and liveness properties

• Are safety and liveness properties disjoint? Yes, if you exclude the set of all traces
• Are all linear properties either a safety or liveness property? No

• Theorem 3.37. Decomposition Theorem
  For any LT property $P$ over $AP$ there exists a safety property $P_{safe}$ and a liveness property $P_{live}$ (both over $AP$) such that $P = P_{safe} \cap P_{live}$.
Fairness constraints

- Fairness constraints are used to rule out “unrealistic” behaviors from a transition system semantics of a concurrent system
  - E.g., refine model to resolve non-deterministic behaviors
- Different types of fairness constraints
  - **Unconditional fairness** (impartiality): e.g., a process can execute infinitely often
  - **Strong fairness** (compassion): e.g., a process that is enabled infinitely often gets it turns to execute infinitely often
  - **Weak fairness** (justice): e.g., a process that is continuously enabled after a certain time, gets its turn to execute infinitely often
Expressing fairness constraints
(action view)

For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ without
terminal states, $A \subseteq Act$, and infinite execution fragment

$\rho = s_0 \rightarrow s_1 \rightarrow s_2 - \ldots$ of $TS$:

• $\rho$ is unconditionally $A$-fair whenever $\exists^\infty j. \alpha_j \in A$

• $\rho$ is strongly $A$-fair whenever

  $(\exists^\infty j. Act(s_j) \cap A \neq \emptyset) \Rightarrow (\exists^\infty j. \alpha_j \in A)$

• $\rho$ is weakly $A$-fair whenever

  $(\forall^\infty j. Act(s_j) \cap A \neq \emptyset) \Rightarrow (\exists^\infty j. \alpha_j \in A)$
A = \{\text{enter}_2\}: The dashed line execution is not unconditionally A-fair, but is strongly A-fair (vacuously)
The execution shown in dotted lines is not strongly A-fair, but is weakly A-fair.
A **fairness assumption** for Act is a triple

$$\mathcal{F} = (\mathcal{F}_{u\text{cond}}, \mathcal{F}_{\text{strong}}, \mathcal{F}_{\text{weak}})$$

with $\mathcal{F}_{u\text{cond}}, \mathcal{F}_{\text{strong}}, \mathcal{F}_{\text{weak}} \subseteq 2^{\text{Act}}$. Execution $\rho$ is $\mathcal{F}$-fair if

- it is unconditionally $A$-fair for all $A \in \mathcal{F}_{u\text{cond}},$
- it is strongly $A$-fair for all $A \in \mathcal{F}_{\text{strong}},$ and
- it is weakly $A$-fair for all $A \in \mathcal{F}_{\text{weak}}.$

If the set $\mathcal{F}$ is clear from the context, we use the term fair instead of $\mathcal{F}$-fair.
Satisfaction and Fairness

Definition 3.48. Fair Satisfaction Relation for LT Properties

Let $P$ be an LT property over $AP$ and $F$ a fairness assumption over $Act$. Transition system $TS = (S, Act, \rightarrow, I, AP, L)$ fairly satisfies $P$, notation $TS \models_F P$, if and only if $\text{FairTraces}_F(TS) \subseteq P$.

Note that the following may occur: $TS \models_F P$ whereas $TS \not\models_F P$.

Unconditional fairness rules out more behaviors than strong fairness, and strong fairness excludes more behaviors than weak fairness.

$$TS \models_{F_{\text{weak}}} P \Rightarrow TS \models_{F_{\text{strong}}} P \Rightarrow TS \models_{F_{\text{ucond}}} P.$$
Fairness and safety properties

- Fairness may be necessary to verify liveness properties, but they are not needed for proving safety properties when a suitable scheduling strategy is used.

**Definition 3.54. Realizable Fairness Assumption**

Let $TS$ be a transition system with the set of actions $Act$ and $F$ a fairness assumption for $Act$. $F$ is called *realizable* for $TS$ if for every reachable state $s$: $\text{FairPaths}_F(s) \neq \emptyset$.

**Theorem 3.55. Realizable Fairness is Irrelevant for Safety Properties**

Let $TS$ be a transition system with set of propositions $AP$, $F$ a realizable fairness assumption for $TS$, and $P_{safe}$ a safety property property over $AP$. Then:

$$TS \models P_{safe} \iff TS \models F \ P_{safe}.$$
Checking Linear Temporal Safety Properties

• Regular safety properties: a safety property whose bad prefixes form a regular language
  – Recall that a trace is a word (sequence of sets of atomic propositions)
  – Satisfaction checking reduced to invariant-checking on a transition system produced by forming the product of the system TS and the automaton characterizing bad prefixes

• $\omega$-regular properties: Generalization of above approach that is also applicable to checking some liveness properties
  – Buchi automata accept infinite words
  – Satisfaction checking reduced to persistence checking (checking for “eventually forever the property holds”)
