A crash course on model checking

(slightly revised by Angelo Montanari)

Massimo Franceschet
A short course on model checking for the
Language and Inference Technology Group
Institute for Logic, Language and Computation
University of Amsterdam

Outline - Class I

1. The need for formal methods
2. Hardware and software verification
3. The process of model checking
4. Temporal logic and model checking
5. Symbolic algorithms
6. Partial order reduction

Outline - Class II

1. Modal logic ML
2. Model checking for ML
3. Computation Tree Logic CTL
4. Model checking for CTL
5. The microwave oven example

The need for formal methods

"The Ariane 5 rocket exploded on June 4, 1996, less than forty seconds after it was launched. The committee that investigated the accident found that is was caused by a software error in the computer that was responsible for calculating the rocket's movement. During the launch, an exception occurred when a large 64-bit floating point number was converted to a 16-bit signed integer. This conversion was not protected by code for handling exceptions and caused the computer to fail. The same error also caused the backup computer to fail. As a result incorrect attitude data was transmitted to the on-board computer, which caused the destruction of the rocket."

The need for formal methods

Today, **hardware and software systems** are widely used in applications where failure is unacceptable: electronic commerce, telephone switching networks, highway and air traffic control systems, medical instruments, and more.

Unfortunately, it is no longer feasible to shut down a malfunctioning system in order to restore safety. In fact, in some cases, devices are less safe when they are shut down.

Even when failure is not **life-threatening**, the consequences of having to replace critical code or circuitry can be **economically devastating**. The Intel Pentium bug in the division algorithm is a good example.

Outline - Class I

1. The need for formal methods
2. **Hardware and software verification**
3. The process of model checking
4. Temporal logic and model checking
5. Symbolic algorithms
6. Partial order reduction

Hardware and software verification

The principal validation methods for complex systems are **simulation**, **testing**, **deductive verification**, and **model checking**.

Simulation and testing

Simulation and testing both involve **making experiments** before deploying the system in the field.

While simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. In both cases, these methods typically **inject signals/inputs** at certain points in the system and **observe the resulting signals/outputs** at other points.

These methods can be a cost-efficient way to find many errors. However, checking **all** the possible interactions and potential pitfalls using simulation and testing techniques is rarely possible.

Deductive verification

Deductive verification techniques use **axioms and proof rules** to prove the correctness of systems.

Initially, such proofs were constructed entirely by hand. Eventually, researchers realized that software tools can be apply to suggest various ways to progress from the current stage of the proof.

Deductive verification

An advantage of deductive verification is that it can be used for reasoning about **infinite state systems**. This task can be automated to a limited extent. However, even if the property to be verified is true, no limit can be placed on the amount of time or memory that may be needed in order to find a proof.

Deductive verification is a **time-consuming process** that can be performed only by experts who are educated to logical reasoning. Consequently, it is applied primarily to high sensitive systems such as security protocols.

Model checking

Model checking is a technique for verifying **finite state concurrent systems**. One benefit of this restriction is that verification can be performed **automatically**.

The model checker normally uses an **exhaustive search** of the finite state space of the system to determine if some specification (property of the system) is true or not.

Model checking

When the system fails to satisfy a desired property, the model checker produces a **counterexample** that demonstrates a wrong behavior. This faulty trace provides insight to understand the actual reason for the failure as well as important clues for fixing the problem.

Given sufficient resources, the model checker will always terminate with a yes/no answer. Moreover, it can be implemented by algorithms with **reasonable efficiency**.

The limitations of the computer

It is important to realize that some mathematical tasks cannot be performed by an algorithm. The **theory of computability** provides limitations on what can be decided by an algorithm.

In particular, there cannot be an algorithm that decides whether an arbitrary computer program terminates. This immediately limits what can be verified automatically.

Thus, restrictions on systems as well as on properties to be verified must be taken into account whenever we aim at developing tools for automatic verification.

Outline - Class I

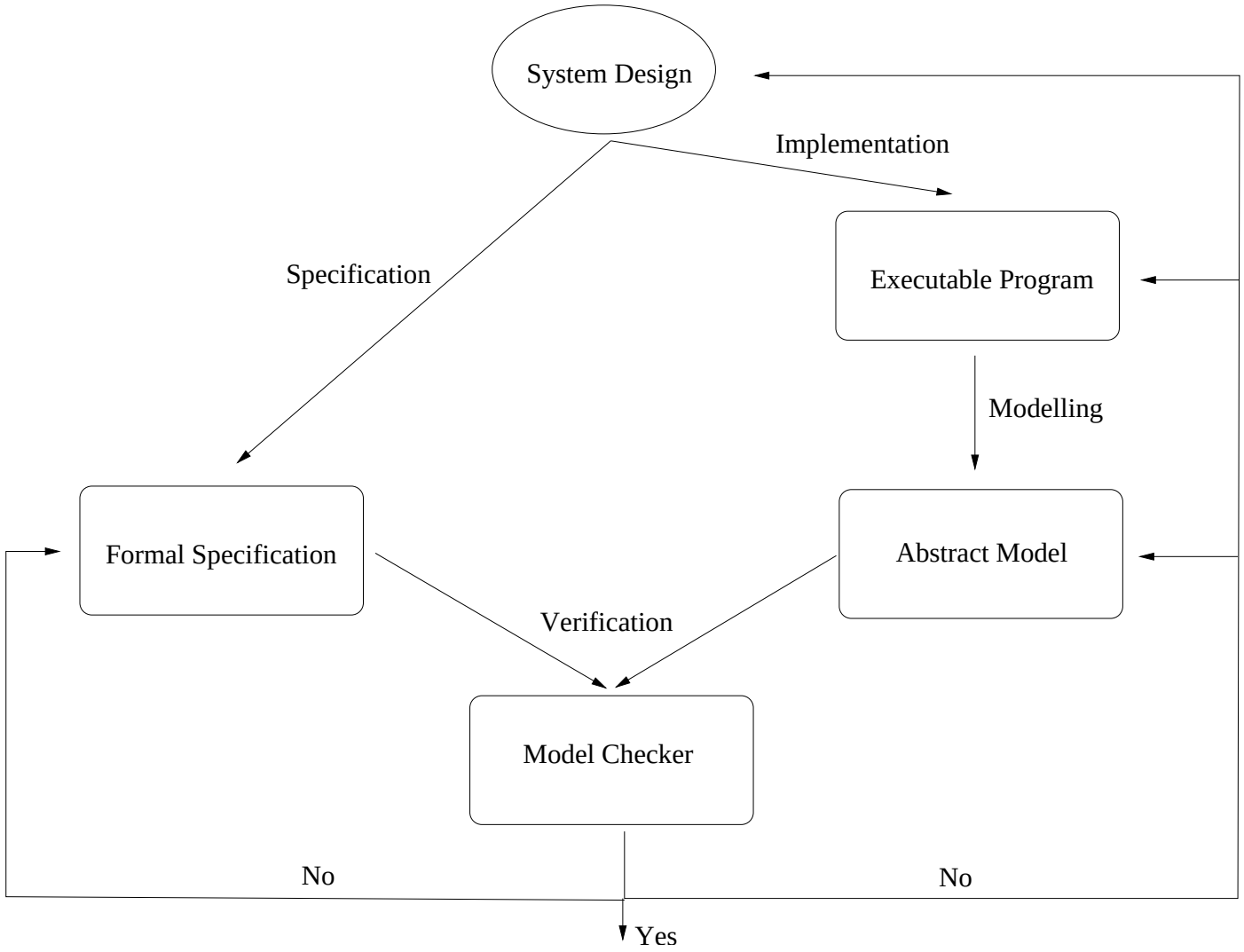
1. The need for formal methods
2. Hardware and software verification
3. **The process of model checking**
4. Temporal logic and model checking
5. Symbolic algorithms
6. Partial order reduction

The process of model checking

Applying model checking to a design consists of several tasks:

1. Modeling
2. Specification
3. Verification

The process of model checking



Modeling

We are interested in **concurrent systems**, that is, nonterminating systems composed of different processes running in parallel.

The first task is to convert the design of the system into an **abstract model** accepted by a model checking tool. The abstract model should eliminate irrelevant details of the design.

In some cases, this conversion can be done automatically (hardware design). In most cases, however, human guidance and assistance in unavoidable.

Modeling

We use a **Kripke structure** to model the behavior of concurrent systems.

A Kripke structure is a node-labeled graph. The nodes of the graph model **system states** and are labeled with information true at that state. The edges of the graph represent **system transitions** as the result of some action of the system.

Specification

Before verifying, it is necessary to state the properties that system design must satisfy.

The specification is usually given in some **logical formalism**. It is common to use **temporal logic**, which can state how the behavior of the systems evolves over time.

Specification

Important issues in specification are:

Consistency Is the given specification consistent?

Completeness Does the given specification cover all the properties that the system should satisfy?

These issues are **not** directly addressed in model checking.

Verification

Ideally verification is completely automatic. However, in practice, it often involves human assistance.

One such manual activity is the analysis of the verification results. In the case of a negative result, the user is provided with an **error trace**. This can help the designer in tracking down where the error occurred.

In this case, analyzing the trace error may require a modification of the system and reapplication of the model checking algorithm.

Verification

An error trace can also result from a **false negative**, that is from:

- incorrect modeling of the system, or
- incorrect formalization of the specification, or
- inconsistent specification.

A final possibility is that the verification task will fail to terminate normally, due to the size of the model, which is too large to fit into the computer memory.

Outline - Class I

1. The need for formal methods
2. Hardware and software verification
3. The process of model checking
4. **Temporal logic and model checking**
5. Symbolic algorithms
6. Partial order reduction

Temporal logic and model checking

Temporal logics have proved to be useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly.

They are often classified in **linear time logics** (like LTL) and **branching time logics** (like CTL and CTL*), according to whether time is assumed to have a linear or branching structure.

A brief history of temporal logic

60's Prior defines temporal logics and uses them in order to investigate philosophical and theological questions like the controversial between free willing and determinism.

1977 Pnueli adopts for the very first time the same temporal logics for analyzing the behaviour of concurrent computer programs.

1983 Clarke, Emerson and Sistla prove that the model checking problem for Computation Tree Logic CTL* is PSPACE-complete.

1985 Clarke and Sistla prove that the model checking problem for Linear Temporal Logic LTL is PSPACE-complete.

A brief history of temporal logic

1986 Vardi and Wolper propose the use of automata over infinite words for linear time model checking.

1986 Clarke, Emerson and Sistla devise a linear time model checker for Computation Tree Logic CTL.

1987 McMillan realizes the effectiveness of using symbolic algorithms to perform model checking.

The destiny of human beings and computer programs have both been investigated with temporal logics!

The state explosion problem

The major drawback of model checking is the **state explosion problems**. The parallel composition of two concurrent components is modeled by taking the Cartesian product of the corresponding state spaces.

This means that the global state space of a concurrent system has size **exponential** in the number of concurrent processes running in the system.

The exploration a huge state space may be prohibitive even for an algorithm running in linear time in the size of the model.

The state explosion problem

Two main approaches have been proposed to cope with the state explosion problem:

- Symbolic algorithms
- Partial order reduction

Outline - Class I

1. The need for formal methods
2. Hardware and software verification
3. The process of model checking
4. Temporal logic and model checking
5. **Symbolic algorithms**
6. Partial order reduction

Symbolic algorithms

The idea of **symbolic model checking** is to encode each system state as an assignment of Boolean values (0 or 1) to the set of state variables associated with the system.

A set of states is hence a set of assignments, and may be implicitly represented as a **Boolean formula** over the set of state variables.

The transition relation is a mapping from states to states, and hence may be implicitly represented as a Boolean formula in terms of two sets of variables: one set encoding the old state and the other encoding the new.

Symbolic algorithms

Logical formulas may be associated with the set of states that validate the formula, and hence they may be implicitly represented as Boolean formulas.

Finally, Boolean formulas may be compactly represented by using **Ordered Binary Decision Diagrams** (OBDDs).

The symbolic model checking algorithm performs the verification by manipulating the OBDD representations of the system and of the formula. Efficient algorithms exist to manipulate OBDDs.

Outline - Class I

1. The need for formal methods
2. Hardware and software verification
3. The process of model checking
4. Temporal logic and model checking
5. Symbolic algorithms
6. **Partial order reduction**

Partial order reduction

Verifying software causes some problems for model checking. Software tends to be less structured than hardware.

In addition, concurrent software is usually asynchronous, that is, most of the activities taken by different processes are performed independently, without a global synchronizing clock.

For these reasons, the state explosion problem is particularly serious for software.

Partial order reduction

The most successful technique for dealing with the state explosion problem for software is the **partial order reduction**.

A common model for representing concurrent software is the **interleaving model**, in which all of the events is a single execution are arranged in a linear order called an **interleaving sequence**. Concurrently executed events appear arbitrarily ordered with respect to one other.

In the **partial model** of program execution, concurrently executed events are not ordered. Each partial order execution can correspond to multiple interleaving sequences.

Partial order reduction

Two events are **independent** of each other when executing them in either order results in the same global state.

It makes no sense to distinguish between two interleaving sequences in which two independent events are executed in different order. Only one of them may be selected.

The partial order reduction technique reduces the state space by selecting only a subset of the ways one can interleave independently executed transitions.

Outline - Class II

1. Modal logic ML
2. Model checking for ML
3. Computation Tree Logic CTL
4. The microwave oven example
5. Model checking for CTL

Modal Logics - Syntax

Modal logics extend Propositional Calculus by adding a modality **F** (usually denoted by $\Diamond$) which locally shifts the evaluation perspective.

More formally, let $\mathsf{PROP} = \{p, q, \ldots\}$ be a set of proposition variables. The **syntax** of modal logic is as follows:

$$\varphi := \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \mathbf{F}\varphi$$

The dual modality **G** (usually denoted by $\Box$) is defined as:

$$\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$$

Modal Logics - Structures

Modal logic is interpreted over **Kripke structures** of the form $\langle M, R, V \rangle$, where

- M is a set of worlds (or states, points, ...);
- $R \subseteq M \times M$ is a binary relation on M;
- $V : \mathsf{PROP} \rightarrow 2^M$ is a valuation function mapping propositions to set of states.

Modal Logics - Semantics

Let $\mathfrak{M} = \langle M, R, V \rangle$ be a Kripke structures and $m \in M$. The **semantics** of modal logic is as follows:

$$
\begin{array}{lll}
\mathfrak{M}, m \models \top & & \\
\mathfrak{M}, m \models p & \text{iff} & m \in V(p),\ p \in \mathsf{PROP} \\
\mathfrak{M}, m \models \neg\varphi & \text{iff} & \mathfrak{M}, m \not\models \varphi \\
\mathfrak{M}, m \models \varphi \wedge \psi & \text{iff} & \mathfrak{M}, m \models \varphi \text{ and } \mathfrak{M}, m \models \psi \\
\mathfrak{M}, m \models \mathbf{F}\varphi & \text{iff} & \text{there is } m' \text{ such that } Rmm' \text{ and } \mathfrak{M}, m' \models \varphi
\end{array}
$$

Model and satisfiability checking

Model checking: Given a finite Kripke structure $\mathfrak{M}$, a world m in $\mathfrak{M}$, and a formula φ, does $\mathfrak{M}, m \models \varphi$?

Satisfiability checking: Given a formula φ, is there a Kripke structure $\mathfrak{M}$ and a world m in $\mathfrak{M}$ such that $\mathfrak{M}, m \models \varphi$?

Procedure `ModalModelChecker`$(\langle M, R, V\rangle, \alpha)$
for **every** $i = 1, \ldots, |\alpha|$ **do**
  for **every** $\beta \in Sub(\alpha)$ **such that** $|\beta| = i$ **do**
    case on the form of β
      $\star$ $\beta = p$
      $\star$ $\beta = \beta_1 \wedge \beta_2$
      for **every** $w \in M$ **do**
        if $\beta_1 \in V(w)$ **and** $\beta_2 \in V(w)$ **then**
          $V(w) = V(w) \cup \{\beta\}$
      $\star$ $\beta = \neg\beta_1$
      for **every** $w \in M$ **do**
        if **not** $\beta_1 \in V(w)$ **then**
          $V(w) = V(w) \cup \{\beta\}$
      $\star$ $\beta = \mathbf{F}\gamma$
      for **every** $w \in M$ **do**
        if $\exists v.\ Rwv \wedge \gamma \in V(v)$ **then**
          $V(w) = V(w) \cup \{\beta\}$

Computational complexity

Let $n = |M|$ be the **number of nodes**, $m = |R|$ be the **number of edges**, and k be the **length of** α (the number of operators plus the number of propositions in α).

Note that $|Sub(\alpha)| = k$. Hence, the main for loop runs for k times. The Boolean cases cost $O(n)$, and the modal case costs $O(n + m)$.

Hence, the model checker for modal logic runs in time $O(k \cdot (n + m))$ in the worst-case. The complexity is **linear** in the product of the length of the formula and the size of the model.

Outline - Class II

1. Modal logic ML
2. Model checking for ML
3. **Computation Tree Logic CTL**
4. The microwave oven example
5. Model checking for CTL

Computation Tree Logic CTL

CTL is a logic designed to reason about properties holding along **computation paths** of structures.

It extends propositional logic with unary temporal operators **EX**, **AX** and binary temporal operators **EU**, **AU**. The informal semantics of the temporal operators is as follows:

- **EX**α means "α holds at **some successor**";
- **AX**α means "α holds at **every successor**";
- **EU**(α, β) means "along **some path** α holds until β will hold";
- **AU**(α, β) means " along **every path** α holds until β will hold".

Computation Tree Logic CTL

CTL formulas are defined as follows:

- $\top$ is a CTL-formula;
- a proposition p is a CTL-formula;
- if α and β are CTL-formulas, then $\neg\alpha$ and $\alpha \wedge \beta$ are CTL-formulas;
- if α is a CTL-formula, then $\mathbf{EX}\alpha$ and $\mathbf{AX}\alpha$ are CTL-formulas;
- if α and β are CTL-formulas, then $\mathbf{EU}(\alpha, \beta)$ and $\mathbf{AU}(\alpha, \beta)$ are CTL-formulas.

Computation Tree Logic CTL

CTL formulas are interpreted over **total Kripke structures** $\mathfrak{M} = \langle M, R, V \rangle$. A structure is total if every state has at least one successor.

A **computation path** of $\mathfrak{M}$ is an infinite sequence of states $\pi = s_0, s_1, \ldots$ such that Rs_i, s_{i+1} for every $i \geq 0$. We denote by π_i the ith state s_i of π.

Computation Tree Logic CTL

Given a total Kripke structures $\mathfrak{M} = \langle M, R, V \rangle$ and a state $s \in M$, the semantics of CTL is as follows:

$$\mathfrak{M}, s \models \mathbf{EX}\alpha \quad \text{iff} \quad \exists s'.Rss' \wedge \mathfrak{M}, s' \models \alpha$$

$$\mathfrak{M}, s \models \mathbf{AX}\alpha \quad \text{iff} \quad \forall s'.Rss' \rightarrow \mathfrak{M}, s' \models \alpha$$

$$\mathfrak{M}, s \models \mathbf{EU}(\alpha, \beta) \quad \text{iff} \quad \exists \pi \text{ starting from } s.\ \exists j \geq 0.\ \mathfrak{M}, \pi_j \models \beta \wedge \forall\ 0 \leq i < j.\ \mathfrak{M}, \pi_i \models \alpha$$

$$\mathfrak{M}, s \models \mathbf{AU}(\alpha, \beta) \quad \text{iff} \quad \forall \pi \text{ starting from } s.\ \exists j \geq 0.\ \mathfrak{M}, \pi_j \models \beta \wedge \forall\ 0 \leq i < j.\ \mathfrak{M}, \pi_i \models \alpha$$

Computation Tree Logic CTL

Shorthands are:

- $\mathbf{EF}\alpha = \mathbf{EU}(\top, \alpha)$;
- $\mathbf{AF}\alpha = \mathbf{AU}(\top, \alpha)$;
- $\mathbf{EG}\alpha = \neg\mathbf{AF}\neg\alpha$;
- $\mathbf{AG}\alpha = \neg\mathbf{EF}\neg\alpha$.

Outline - Class II

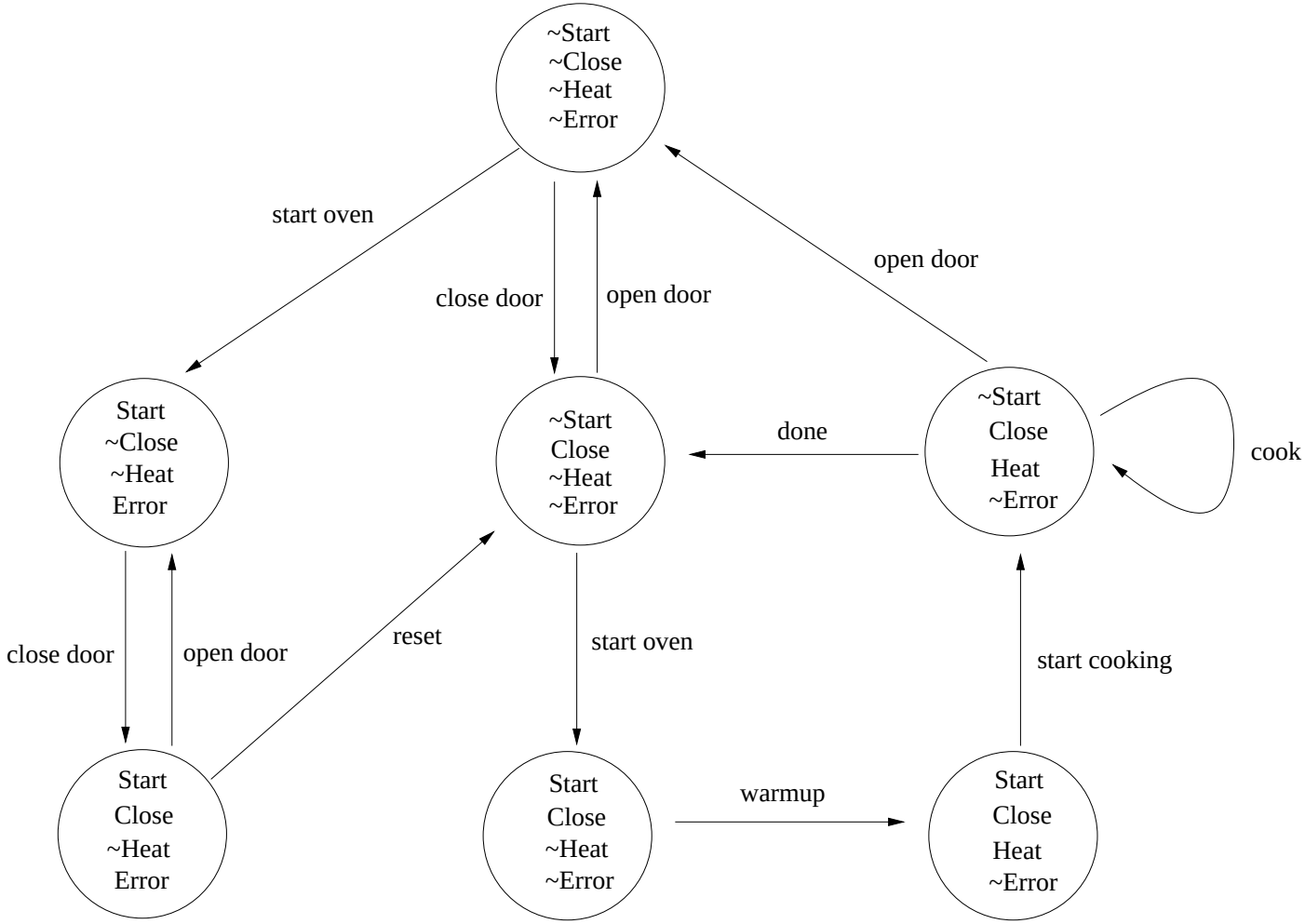
1. Modal logic ML
2. Model checking for ML
3. Computation Tree Logic CTL
4. **The microwave oven example**
5. Model checking for CTL

The microwave oven example

Informal specification of a microwave oven functioning:

> To cook food in the oven, open the door, put the food inside, and close the door. Do not put metal containers in the oven. Press the start button. The oven will warmup for 30 seconds, and then it will start cooking. When the cooking is done, the oven will stop. The oven will stop also whenever the door is opened during cooking. If the oven is started while the door is open, an error will occur, and the oven will not heat. In such a case, the reset button may be used.

Modelling


~Start
~Close
~Heat
~Error
start oven
close door
open door
open door
Start
~Close
~Heat
Error
~Start
Close
~Heat
~Error
done
~Start
Close
Heat
~Error
cook
close door
open door
reset
start oven
start cooking
Start
Close
~Heat
Error
Start
Close
~Heat
~Error
warmup
Start
Close
Heat
~Error

Specification

1. If the oven heats, then the door is closed:

$$\mathbf{AG}(\texttt{Heat} \rightarrow \texttt{Close})$$

Specification

1. If the oven heats, then the door is closed:

$$\mathbf{AG}(\texttt{Heat} \rightarrow \texttt{Close})$$

2. Whenever the start button is pushed, eventually the oven will heat:

$$\mathbf{AG}(\texttt{Start} \rightarrow \mathbf{AF}\texttt{Heat})$$

Specification

1. If the oven heats, then the door is closed:

$$\mathbf{AG}(\texttt{Heat} \rightarrow \texttt{Close})$$

2. Whenever the start button is pushed, eventually the oven will heat:

$$\mathbf{AG}(\texttt{Start} \rightarrow \mathbf{AF}\texttt{Heat})$$

3. Whenever the oven is correctly started, eventually the oven will heat:

$$\mathbf{AG}((\texttt{Start} \wedge \neg\texttt{Error}) \rightarrow \mathbf{AF}\texttt{Heat})$$

Specification

1. If the oven heats, then the door is closed:

$$\mathbf{AG}(\texttt{Heat} \rightarrow \texttt{Close})$$

2. Whenever the start button is pushed, eventually the oven will heat:

$$\mathbf{AG}(\texttt{Start} \rightarrow \mathbf{AF}\texttt{Heat})$$

3. Whenever the oven is correctly started, eventually the oven will heat:

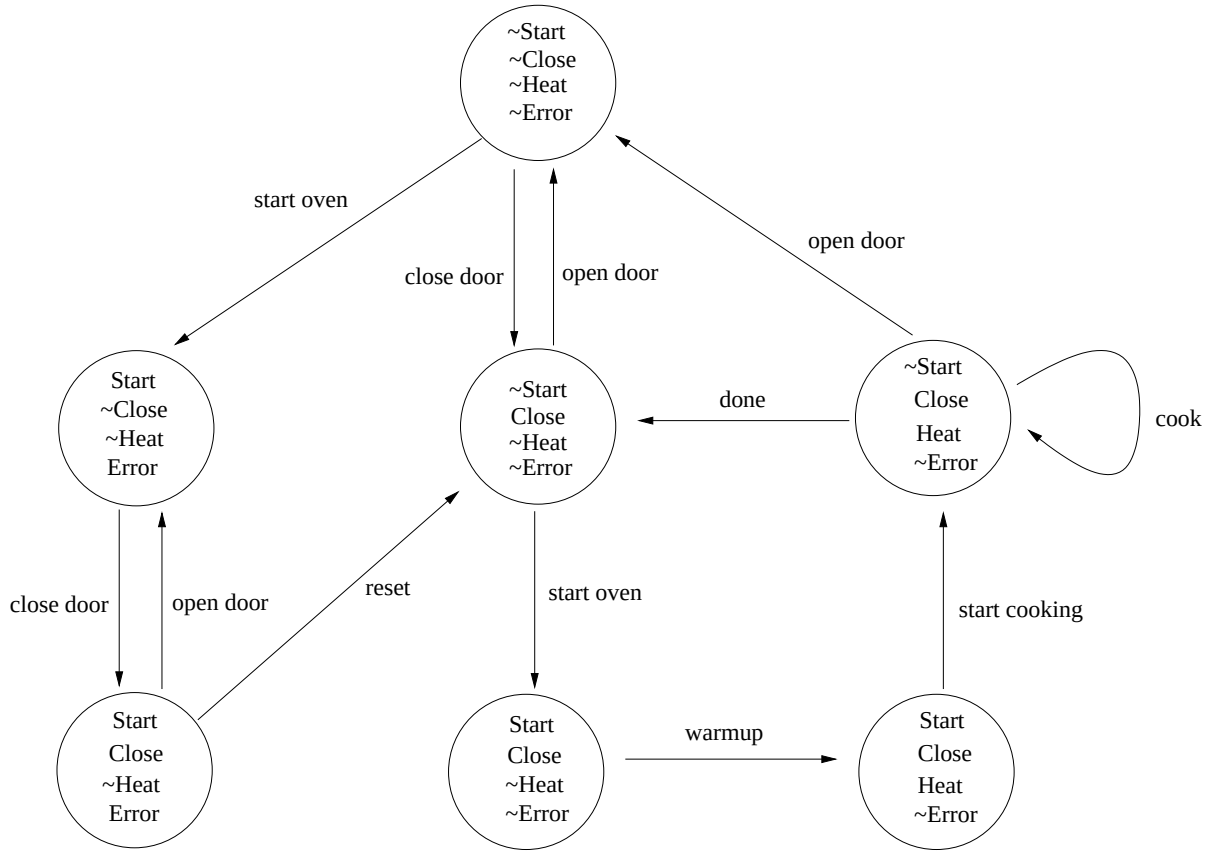
$$\mathbf{AG}((\texttt{Start} \wedge \neg\texttt{Error}) \rightarrow \mathbf{AF}\texttt{Heat})$$

4. Whenever an error occur, it will be still possible to cook:

$$\mathbf{AG}(\texttt{Error} \rightarrow \mathbf{EF}\texttt{Heat})$$

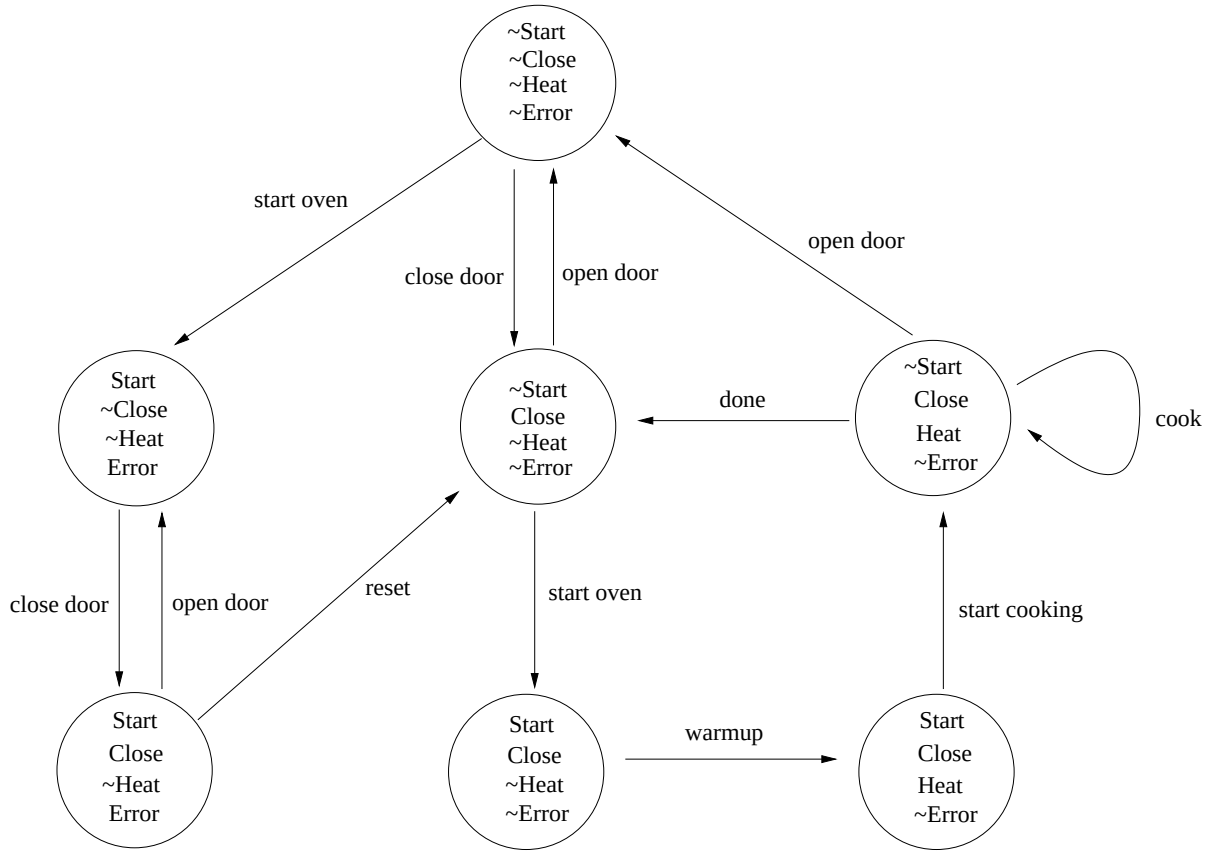
Verification

AG(`Heat` $\rightarrow$ `Close`) is ?



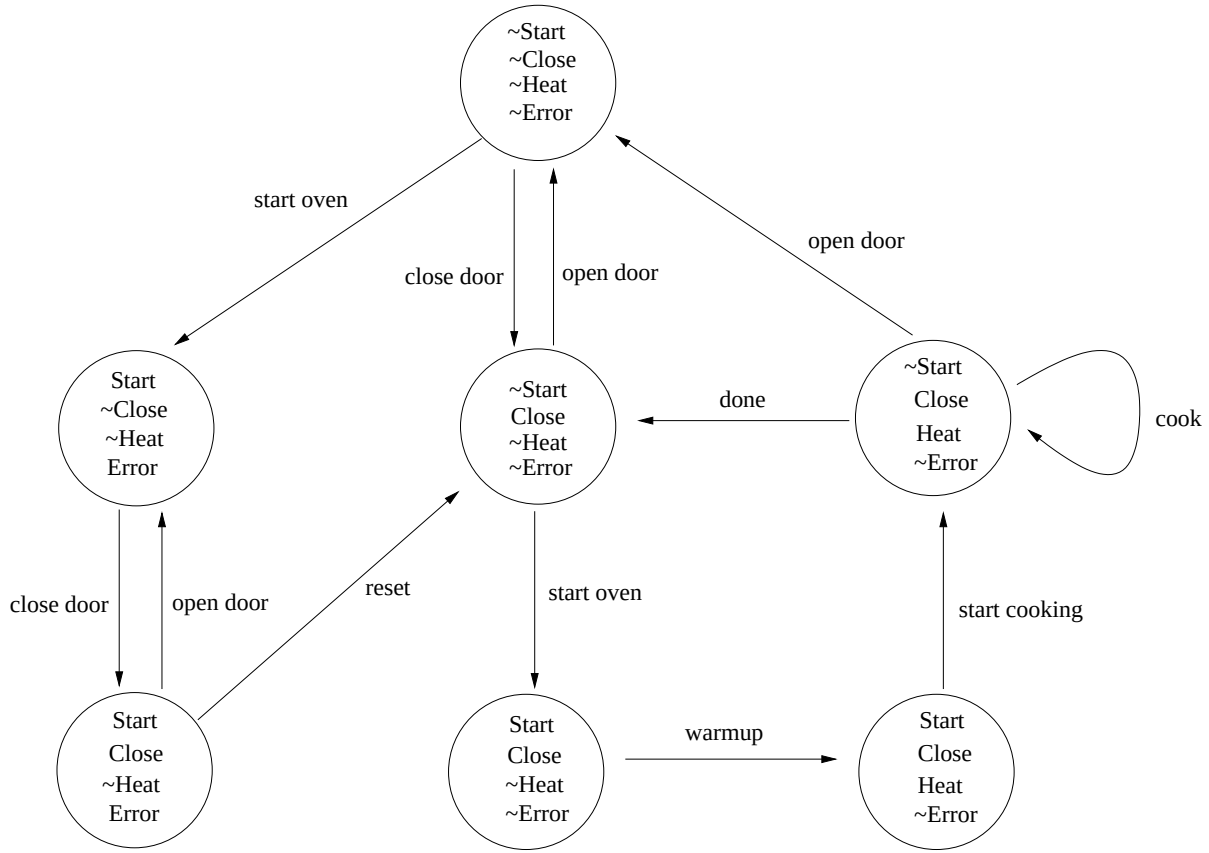
Verification

AG(`Heat` → `Close`) is true!



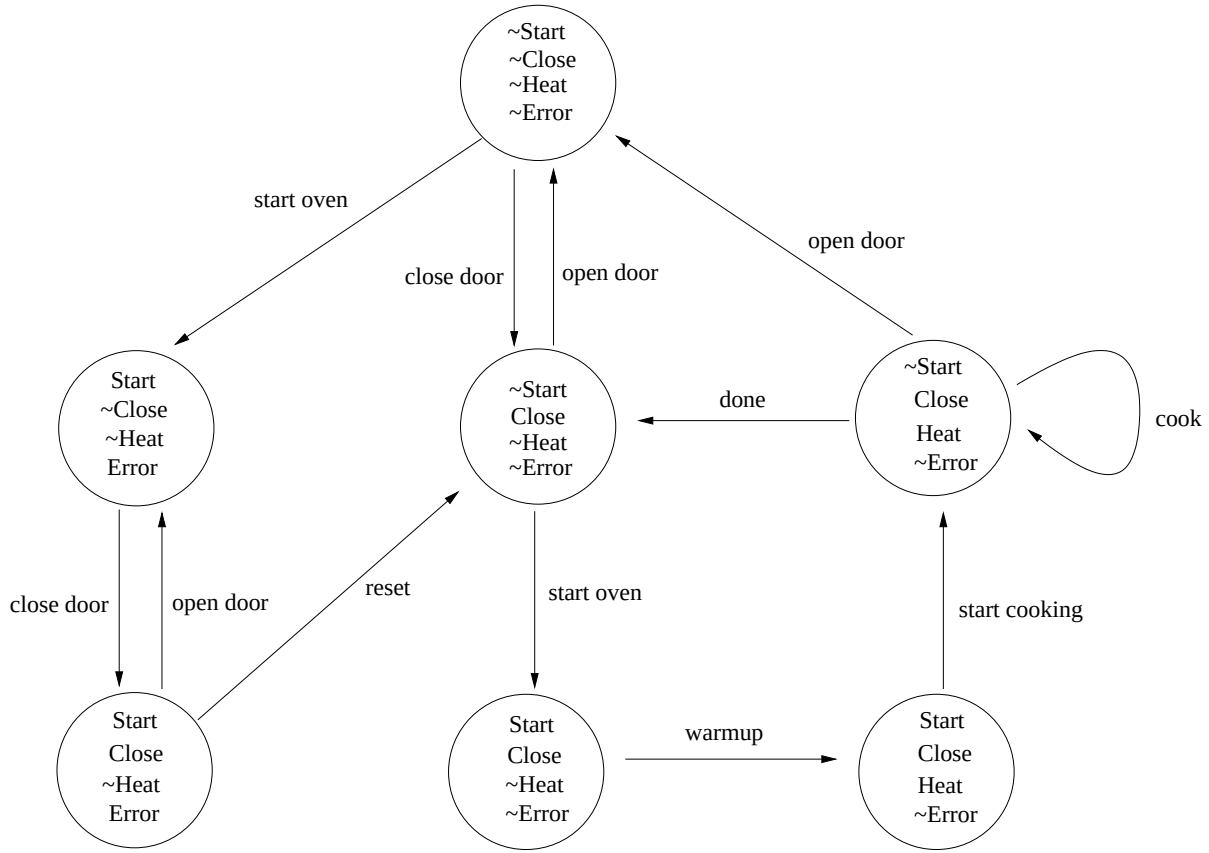
Verification

$\mathbf{AG}(\texttt{Start} \rightarrow \mathbf{AF}\texttt{Heat})$ is ?



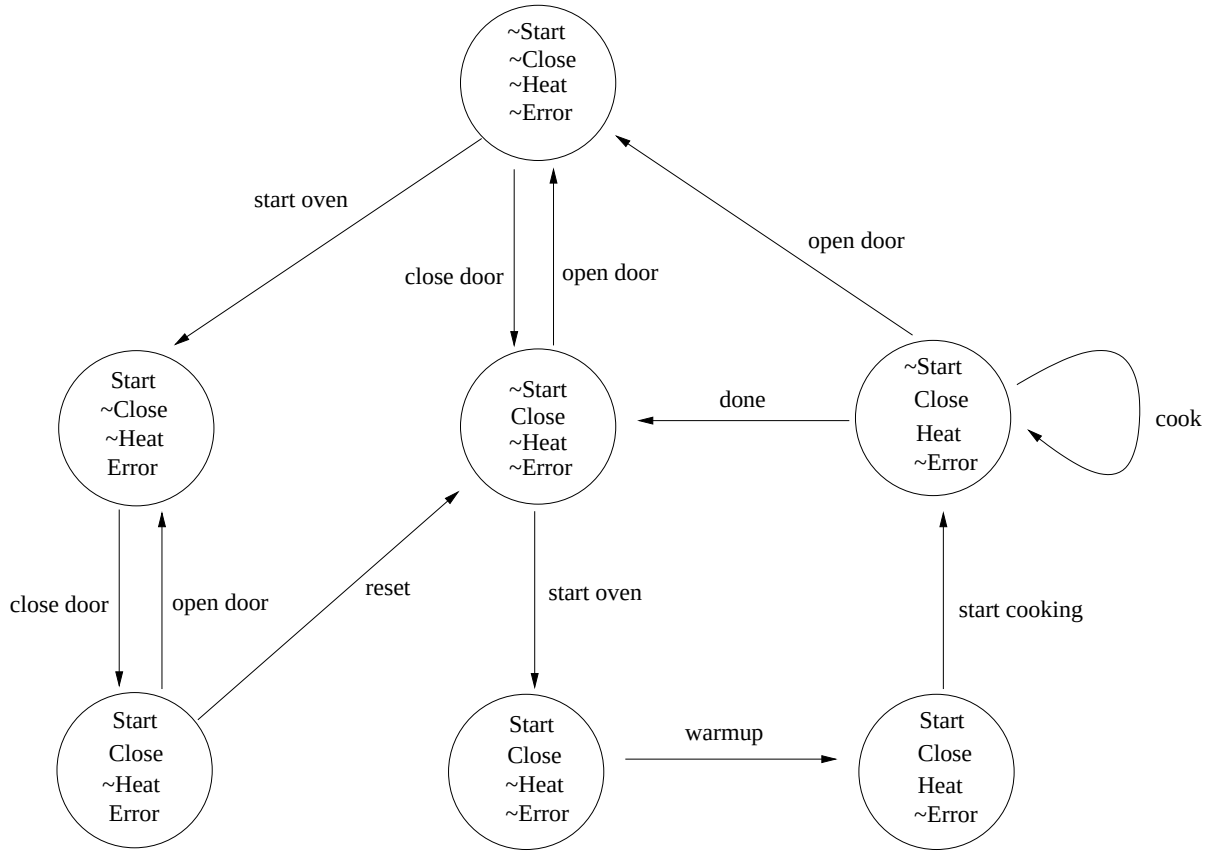
Verification

$\mathbf{AG}(\texttt{Start} \rightarrow \mathbf{AF}\texttt{Heat})$ is false!



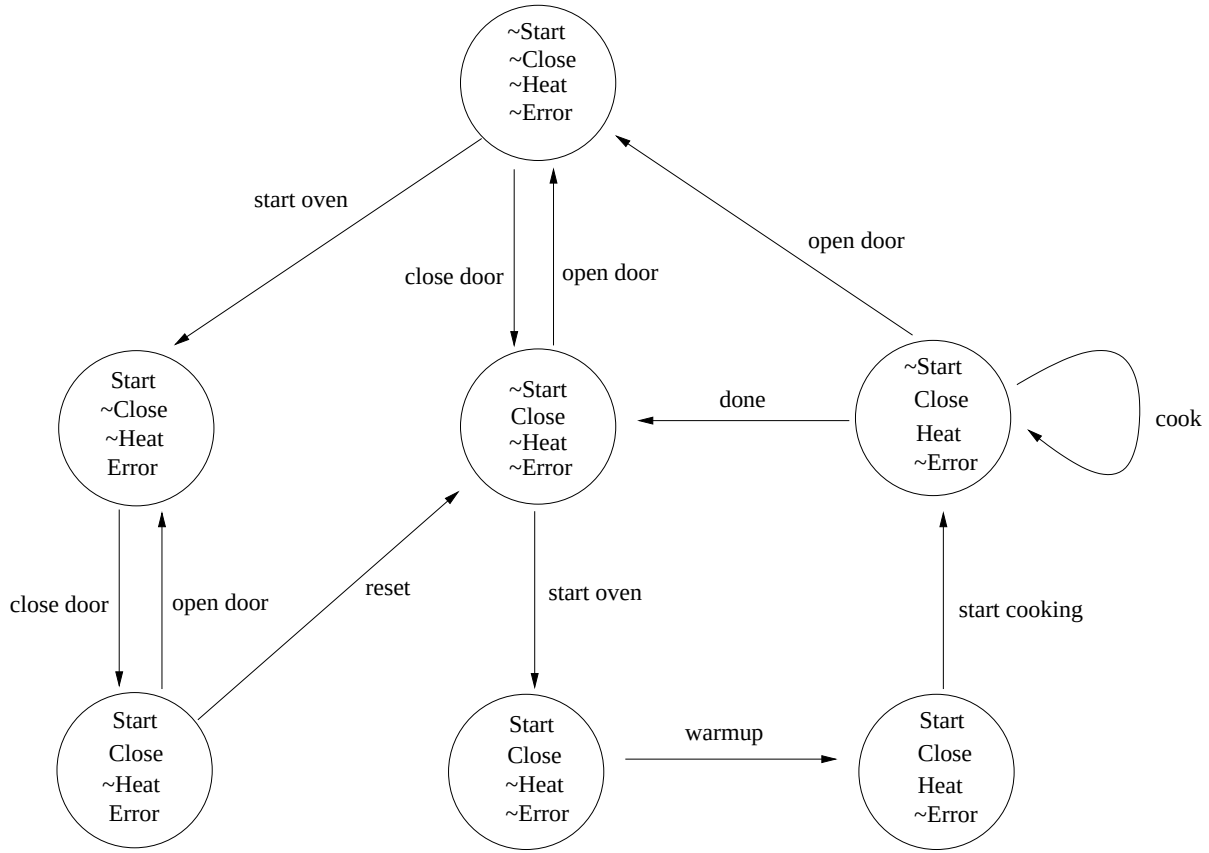
Verification

$\mathbf{AG}((\texttt{Start} \wedge \neg\texttt{Error}) \rightarrow \mathbf{AF}\texttt{Heat})$ is ?



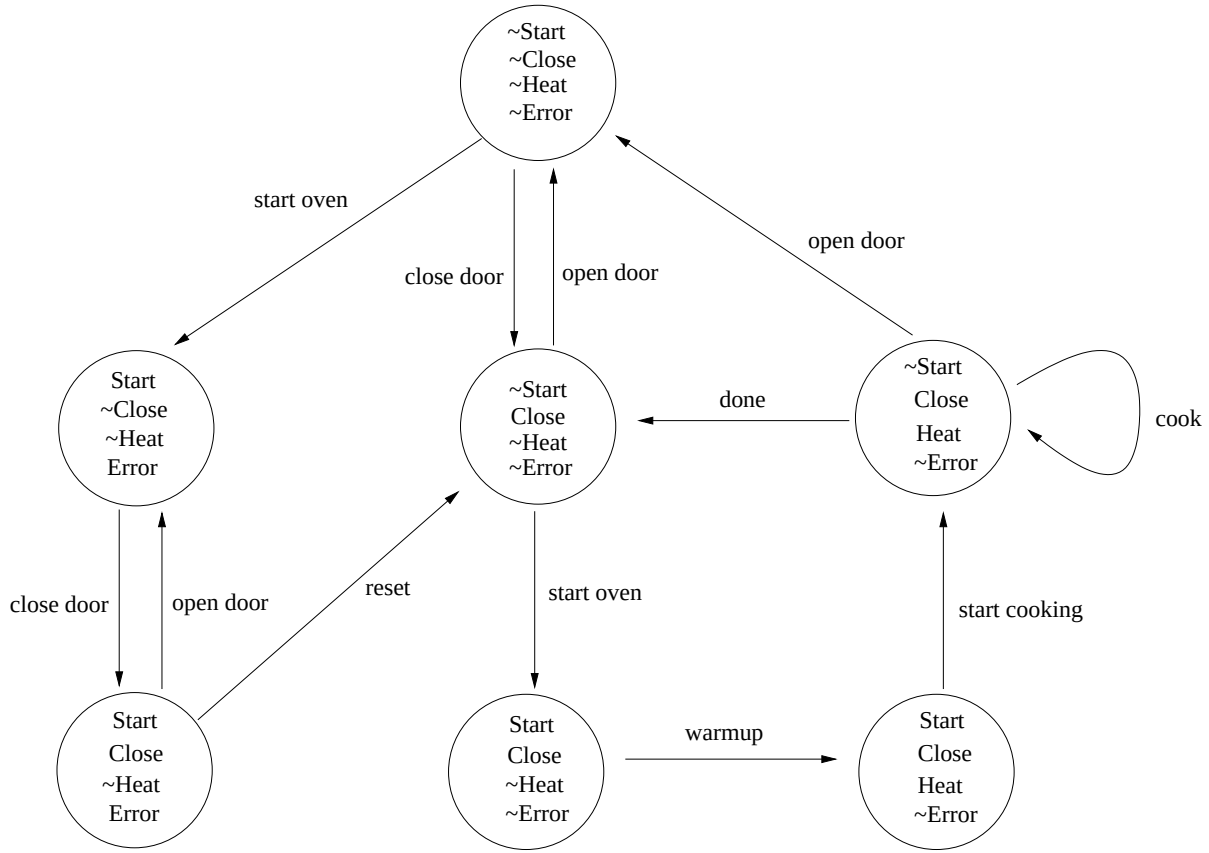
Verification

$\mathbf{AG}((\texttt{Start} \wedge \neg\texttt{Error}) \rightarrow \mathbf{AF}\texttt{Heat})$ is true!



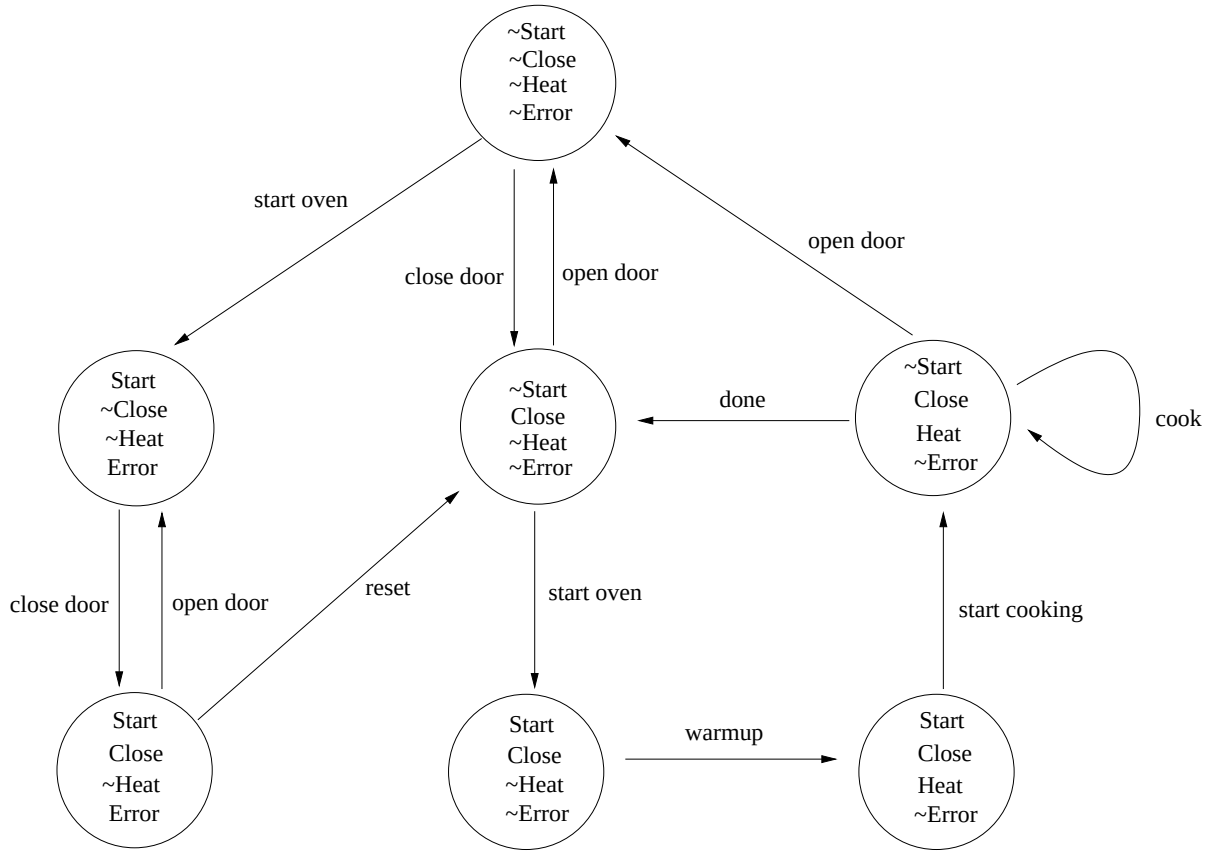
Verification

AG(`Error` $\rightarrow$ **EF**`Heat`) is ?



Verification

AG(`Error` → **EF**`Heat`) is true!



Outline - Class II

1. Modal logic ML
2. Model checking for ML
3. Computation Tree Logic CTL
4. The microwave oven example
5. **Model checking for CTL**

Model checking for CTL

We elaborate subformulas in increasing length order (like in the modal case).

Cases **EX** and **AX** are resolved by visiting the successors of the current state. However, cases **EU** and **AU** impose us to visit, in the worst-case, all the nodes reachable by a path from the current state. We will take advantage of the following **recursive** definitions:

$$\begin{aligned} \mathbf{EU}(\alpha, \beta) &= \beta \vee (\alpha \wedge \mathbf{EXEU}(\alpha, \beta)) \\ \mathbf{AU}(\alpha, \beta) &= \beta \vee (\alpha \wedge \mathbf{AXAU}(\alpha, \beta)) \end{aligned}$$

Procedure `CTL Model Checker`$(\mathfrak{M} = \langle M, R, V\rangle, \alpha)$
for **every** $i = 1, \ldots, |\alpha|$ **do**
  for **every** $\beta \in Sub(\alpha)$ **such that** $|\beta| = i$ **do**
    case on the form of β
      $\star$ propositional cases
      $\star$ $\beta = \mathbf{EX}\gamma$
      for **every** $w \in M$ **do**
        if $\exists v.\ Rwv \wedge \gamma \in V(v)$ **then**
          $V(w) = V(w) \cup \{\beta\}$
      $\star$ $\beta = \mathbf{AX}\gamma$
      for **every** $w \in M$ **do**
        if $\forall v.\ Rwv \rightarrow \gamma \in V(v)$ **then**
          $V(w) = V(w) \cup \{\beta\}$

$\star\ \beta = \mathbf{EU}(\gamma, \delta)$
for **every** $w \in M$ **do** $\texttt{marked}(w) = \texttt{False}$
for **every** $w \in M$ **do**
    if $w \in V(\delta)$ **then** $\texttt{CheckEU}(\mathfrak{M}, w, \gamma, \delta)$
$\star\ \beta = \mathbf{AU}(\gamma, \delta)$
for **every** $w \in M$ **do** $\texttt{marked}(w) = \texttt{False}$
for **every** $w \in M$ **do**
    if not $\texttt{marked}(w)$ **then** $\texttt{CheckAU}(\mathfrak{M}, w, \gamma, \delta)$

Procedure `CheckEU`$(\mathfrak{M}, w, \gamma, \delta)$
if not `marked`(w) **then**
`marked`$(w) =$ `True`
$V(w) = V(w) \cup \{\mathbf{EU}(\gamma, \delta)\}$
for every v **such that** Rvw **do**
if $v \in V(\gamma)$ **then** `CheckEU`$(\mathfrak{M}, v, \gamma, \delta)$

Function `CheckAU`$(\mathfrak{M}, w, \gamma, \delta)$
if `marked`(w) **then**
  if $\mathbf{AU}(\gamma, \delta) \in V(w)$ **then return** `True` **else return** `False`
`marked`$(w) =$ `True`
if $\delta \in V(w)$ **then**
  $V(w) = V(w) \cup \{\mathbf{AU}(\gamma, \delta)\}$
  return `True`
if $\gamma \notin V(w)$ **then**
  return `False`
for **every** v **such that** Rwv **do**
    if not `CheckAU`$(\mathfrak{M}, v, \gamma, \delta)$ **then**
      return `False`
$V(w) = V(w) \cup \{\mathbf{AU}(\gamma, \delta)\}$
return `True`

Computational complexity

Let $n = |M|$ be the **number of nodes**, $m = |R|$ be the **number of edges**, and k be the **length of** α.

The main for loop runs for k times. The Boolean cases cost $O(n)$. The temporal cases cost $O(n + m)$. Hence, the model checker for CTL runs in time $O(k \cdot (n + m))$ in the worst-case.

The complexity is **linear** in the product of the length of the formula and the size of the model.

References

- E. M. Clarke, O. Grumberg and D. A. Peled. Model Checking. The MIT Press. 1999;
- M. R. A. Huth and M. D. Ryan. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, 2000.