1 Introduction

Since the birth of the electronic computer in the late 1940s, mathematicians and computer scientists have researched theoretical and practical techniques to establish that a given computer program behaves according to a given specification. In general terms, this research topic is known as program verification. Alan Turing, for example, presented a paper at the inaugural EDSAC conference in 1949 about establishing the correctness of a large routine — see Lockwood Morris and Cliff Jones reproduction [MJ84]. But it wasn’t until the mid 1960s with Floyd’s work on correctness of flowchart programs that research in program verification began to make progress. Floyd’s activity certainly kick-started a number of prominent researchers in the field: Dijkstra, Hoare, Milner, Manna and Pnueli, to name just a few. At a similar time, researchers were also looking at ways in which a computer can be used to support a human to undertake logical proofs. Indeed from that age, Alan Robinson is renowned for his resolution principle [Rob65] that underlies most first order logic provers of today, albeit in rather more sophisticated forms, e.g. [Vor95].

Great strides were also made with developing implementations of decidable logics, and being able to combine the resulting decision procedures, as they are known, for certain combinations of decidable logics. Such decidable logics held the following promise. If one can encode both the program and the program’s specification as formulas, say \( \phi \) and \( \psi \) in the logic, then the associated decision procedure can be applied to determine the validity, or otherwise, of the correctness formula \( \phi \Rightarrow \psi \) in a fully automated fashion. For example, propositional logic is decidable, and all finite state programs can be encoded as a propositional formula. Unfortunately, at the time, all but the tiniest of programs and specification defeated such approaches. The intractable (exponential and beyond) complexities of the decision procedures were simply too high.

However, all was not lost. Following on from the seminal work of Pnueli, [Pnu77], and subsequent work of the duo Manna and Pnueli, see [MP92, MP95, MP96] for summary books, the computational advantage of comparing (in some way) a temporal description of program properties directly with a state machine model of the program was soon realised. And thus, during the 1980s, the research area of what has become known as model
checking was initiated. Model checking research has produced what many believe to be the most useful technology from the whole of the formal methods community within computer science over the past quarter of a century. It has led to automated verification tools that can be used to verify behavioural properties of significantly sized models. The first major successes were applications in hardware verification, i.e. establishing the correctness of properties of gate-level (and higher) models (circuit descriptions) of synchronous digital hardware components. Today, the field commands considerable world-wide research supported by many international conferences and journals. Current hot topics focus on techniques to obtain greater scalability through abstraction, compositionality and parallelism, for example, and be more applicable to software, e.g. model checking Java programs.

Underlying all model checking techniques are a few simple ideas and over the course of the following few lectures, I will provide you with the essence of automata-based program verification. In this first section, I will introduce you to a (simple form of) temporal logic, to its models, show the relation between the temporal logic models and state machine models (finite state automata), and reachability analyses over the automata.

1.1 Background Reading

In addition to these brief notes (and slides), I thoroughly recommend you look at Chapter 9 of Clarke, Grumberg and Peled’s classic (but quite recent) book on Model Checking [CGP99]. Ed Clarke was one of the founders and prime movers of the model checking research community. In the early 1980s, together with Alan Emerson, they proposed the restricted branching time logic CTL (standing for Computation Tree Logic) and its satisfiability procedures and use for synthesis and model checking. Subsequently, Clarke has remained one of the leaders in the field, developing with his students algorithms and implementations for symbolic model checking, abstraction mechanisms and more recently, CEGAR, counter example guided abstraction refinement methods. Pnueli, amongst many others, have argued that linear time temporal logics are often better suited for specification, despite the more costly satisfiability procedures. In fact, we will only consider the linear time case. Moshe Vardi and Pierre Wolper have been key in the development of automata based approaches to temporal logic model checking. And it is largely that approach I introduce. The literature lacks good background books, however, you may wish to peruse the papers [VW86] and other related papers which can be found on Moshe’s web page http://www.cs.rice.edu/~vardi/papers.

2 What is Model Checking?

Model checking is what is loosely called a Push button verification technology. The user supplies a formal description of some system model and a property formally expressed in some logic (or equivalent representation). The user then only has to push a button to determine whether the given property does indeed hold for the given model. In other words, there is an algorithm underlying model checking that gives a yes/no answer. The system model will typically represent an abstraction of all the executions of some actual hardware/software system (often called behaviours. The properties to be verified may characterise both desired (often referred to as required properties, or requirements), as well as undesirable, sets of behaviours.
A Simple Example

Suppose we wish to verify some properties of a network-based message switching system. We will consider, however, not the actual message switching system but a very high-level, abstract, model of such a system. Let us assume that the Network comprises a collection of Nodes, pairs of which are connected by communication channels. Each node has an Address. Messages contain a Destination address and some content (which is of no interest to us here). We assume that Nodes contain routing control information and are able to route incoming messages towards their destination.

A rather desirable (dynamic) property of this network might be:

\[
\begin{align*}
\text{a correctly addressed message input at any node in the network will,} \\
\text{in due course (time), reach its destination node and be routed no} \\
\text{further, i.e. delivered.}
\end{align*}
\]

An undesirable property might be:

\[
\begin{align*}
\text{a correctly addressed message singly input to the network (at some} \\
\text{node) gets delivered many times, i.e. more than once, to the node} \\
\text{with destination address. More informally, this characterises that} \\
\text{no message is delivered more than once to its destination.}
\end{align*}
\]

Desirable properties, such as the first, we clearly want to be true for all possible executions, i.e. runs, of the system; on the other hand, undesirable properties, such as the second one, should be false for all possible runs of the system.

Three key points can be observed. Firstly, for important classes of models (of systems), we can represent the set of all its possible executions in a finite way. Secondly, there are decidable logics that are expressive enough to be able to describe the kinds of properties given above. Thirdly, the logics can readily be interpreted over the structures used to express the behaviours of some system model and hence decision procedures for determining the whether formalised properties hold for given models can be applied.
2.1 Success and Limitations

As I mentioned in the introduction, model checking based verification (and associated tooling) is probably the most successful verification technology to have arisen and transferred to industry from the formal methods research community over the past quarter of a century. A number of factors have influenced this success, but undoubtedly the following aspects have been key to the success. Model checking is a pushbutton technology; the user of the tooling requires no knowledge or understanding of formal, i.e. mathematical, proof. Model checking is well suited to establishing properties of concurrent systems (very often non-terminating ones), systems that are extremely difficult to reason about via other, especially paper and pencil, methods. Model checking delivers more than a yes/no answer to the verification question. If the property fails to hold, the tooling can deliver a counterexample, effectively an execution path that will lead to the point of failure of the required property. The counterexample can be input to other animation tools to show in a graphical manner what went wrong. Of course, more than one counterexample may exist, in which case all can be generated, if so desired. Users can apply formal methods, thereby meeting parts of various ISO/MOD/DOD standards for system modelling and construction, without detailed knowledge of the methods. Many production quality model checkers now exist, several freely available in the public domain (having been initially developed / prototyped from publicly funded research programmes. There have been great success stories in hardware verification. Major chip manufacturers, for example Intel, use model checking technology to help increase confidence in the hardware design, prior to chip manufacture. Over the past five years there has been growing interest and research in the application of model checking to software, in which the role of the model checker shifts more towards being used as highly sophisticated debugging tool.

Perhaps this all sounds too good to be true. Well, there are drawbacks (and of course it is these drawbacks that keep the research community busy). To begin with, the verification problem being solved by these techniques is computationally intractable, and naive application most likely (usually) leads to resource exhaustion, both in terms of space, i.e. memory utilisation, and time. The basic approach is restricted to finite state models. For example, in general terms, it is not possible to apply the model checking approach to recursive programs, which may, over infinite time, use an infinite amount of memory. Much research work over the past decade has been focussed on various forms of model abstraction techniques, aiming to reduce the verification problem to one over smaller, finite, structures over which model checking technology works well. However, finding the right form of abstraction is very hard and usually does require specialist knowledge. Model checking can be extended to handle (limited forms of) real-time and stochastic information, but this again further restricts the scale of application that can be solved. More often than not, the formal logics used to described the properties to be checked are too complex for typical users, who are not usually highly trained logicians or theoretical computer scientists.

In the next section we give a very brief introduction to a linear-time temporal logic, often used as a property language for model checkers.
Temporal Logic

We only have time for the briefest of introductions to temporal logic, a subject to which we could, in reality, devote several whole course units. But it is important for me to show you a little of this formal logical formalism and its expressiveness, and the way it can be connected semantically to programs and associated models characterising their execution behaviours.

Temporal logic introduced to the computer science community in 1977 by Amir Pnueli for describing (concurrent) program properties [Pnu77] (although not recognised as ground-breaking piece of work at the time, it is now well recognised as a seminal, landmark, paper which significantly contributed to Pnueli winning the Turing Award — the CS equivalent to a Nobel Prize — Pnueli is certainly the father of temporal logic in computer science). Temporal logics are a form of modal logic, an area of logic and philosophy that has been studied for many centuries and dates back to Aristotle. Modal logics give recognition to the possibility and necessity of truth. The possible worlds view of modal logic is easiest to understand and relate to temporal logic. Propositions are interpreted in worlds connected by some accessibility relation. A proposition which is necessarily true in some given world is one that is true in all worlds accessible from that given world. A proposition which is possibly true in some given world is one that is true in some world accessible from the given world. Two modal operators are introduced: □ for necessity; and ◊ for possibility. Then, given a proposition $p$, the modal formula $\square p$ ($◊ p$) denotes that the proposition $p$ is necessarily (possibly) true, i.e. true in all accessible worlds from the one in which the formula is being interpreted. Various forms of modal logic can then be constructed according to how the various worlds are related, i.e. via the properties of the accessibility relation.

For a temporal view of a possible worlds model, we view the different worlds as representing different times and treat the accessibility relation as an earlier than / later than relation. The way in which the earlier/later than relation behaves dictates the temporal flow, i.e. flow of time. We will only consider linear (and, in fact, discrete) flows of time. For such flows, the $\square$ modality then means in every future moment of time from (and including) now and the $◊$ modality means at some future moment from now.

For notational ease, we will not use these usual symbols for the modalities. Instead we will use descriptive words:

$$\text{Always} \quad \text{Sometime} \quad \text{Next} \quad \text{Until}$$

A propositional temporal logic can be formed by augmenting propositional logic with the temporal modalities (operators) named above. Let us formalise a property of the network example, just using the Always and Sometime modalities — we will define the other two in the next subsection. Assume that $\text{sndto}(m,d)$ represents a proposition that a message $m$ is sent into the network at some node destined for node address $d$ and that $\text{rdvfrom}(m,d)$ represents the proposition that message $m$ is received from the network at the node addressed $d$. Then, the linear temporal logic formula

$$\text{Always}(\text{sndto}(m,d) \Rightarrow \text{Sometime} \text{rdvfrom}(m,d))$$

captures the desirable property we expressed informally in natural language in the simple network example of section 2. A direct natural language reading is “Always in the
future if $snd(m, d)$ holds then at sometime later $rcvfrom(m, d)$ will hold”. Suppose the original formula is said to hold at some time point $t$, then the formula $sndto(m, d) \Rightarrow$ *Sometime* $rcvfrom(m, d)$ has to hold at every moment in the future of $t$. Thus suppose the proposition $sndto(m, d)$ holds at point $t_1 \geq t$, then we will have that *Sometime* $rcvfrom(m, d)$ also holds at time $t_1$, which, according to the semantics we will provide for *Sometime* will mean that $rcvfrom(m, d)$ holds at some time $t_2 \geq t_1$.

Temporal logics are able to express important classes of concurrency properties; the two most famous ones are

- *safety* properties, which express that “nothing bad will happen”, and
- *liveness* properties, which express that “something good will happen”.

The formula

$$sndto(m, d) \Rightarrow \text{Sometime} \; rcvfrom(m, d)$$

is an example of a liveness property; it expresses that something good will happen, i.e. correct delivery of the message (assuming appropriate interpretations of the propositions $sndto(m, d)$ and $rcvfrom(m, d)$).

### 3.1 A semantics for a linear temporal logic

Let us briefly formalise a semantics for a linear temporal logic formed by augmenting propositional logic with temporal modalities, *Always*, *Sometime*, *Next* and *Until*. Syntactically, the temporal logic is defined inductively as follows:

If $\phi$ is a formula of propositional logic, then it is a formula of temporal logic.

If $\phi$ and $\psi$ are formulas of temporal logic, then so are:

$$\neg \phi, \; \phi \land \psi, \; \phi \lor \psi, \; \phi \Rightarrow \psi, \; \phi \Leftrightarrow \psi$$

*Always* $\phi$, *Sometime* $\phi$, *Next* $\phi$, $\phi$ *Until* $\psi$

We define a semantics in the usual way by defining a structure in which formulas can be interpreted. The structures in which propositional linear temporal logic formulas are evaluated can be thought of as sequences of states, each state providing interpretation to the atomic propositions used. We will assume that the temporal flow is both linear and discrete, namely, for any given time point, there will be a unique “next” timepoint. The temporal flow can then be represented by the natural numbers with the usual less than ordering. Thus, given some time point $t$, then $t+1$ will denote the next (future) point in time.

Let $\sigma = s_0, s_1, s_2, \ldots, s_n, s_{n+1}, \ldots$, denote a sequence of states $s_i$, for indices $i \in \mathbb{N}$. It is usual for $\sigma$ to be of infinite length. We will then use the notation $\sigma, i \models \varphi$ to denote the evaluation of the formula $\varphi$ at position $i$ in the sequence $\sigma$. 

6
\( \sigma, i \models p \) \quad \text{iff} \quad p \text{ is true in state } s_i \text{ of } \sigma, \text{ for atomic propositions } p

\( \sigma, i \models \phi \land \psi \) \quad \text{iff} \quad \sigma, i \models \phi \text{ and } \sigma, i \models \psi

\( \sigma, i \models \phi \lor \psi \) \quad \text{iff} \quad \sigma, i \models \phi \text{ or } \sigma, i \models \psi

\( \sigma, i \models \neg \phi \) \quad \text{iff} \quad \text{it is not the case that } \sigma, i \models \phi

\( \sigma, i \models \text{Next } \phi \) \quad \text{iff} \quad \sigma, i + 1 \models \phi

\( \sigma, i \models \text{Always } \phi \) \quad \text{iff} \quad \forall j \geq i. \sigma, j \models \phi

\( \sigma, i \models \text{Sometime } \phi \) \quad \text{iff} \quad \exists j \geq i. \sigma, j \models \phi

\( \sigma, i \models \phi \text{Until } \psi \) \quad \text{iff} \quad \exists k \geq i. \sigma, k \models \psi

\text{and } \forall j \in \{i..k - 1\}. \sigma, j \models \phi

The meanings for formulas of the form \( \phi \Rightarrow \psi \) and \( \phi \Leftrightarrow \psi \) are given by the definitions

\[ \phi \Rightarrow \psi = \neg \phi \lor \psi \]

\[ \phi \Leftrightarrow \psi = (\phi \Rightarrow \psi) \land (\psi \Rightarrow \phi) \]

**Example 3.1 Formula evaluation:** Assume atomic propositions \( p, q \) and \( r \). Consider an infinite sequence \( \sigma \) such that \( s_{i+3} = \{p, q\}, s_{i+3+1} = \{p\} \) and \( s_{i+3+2} = \{p, r\} \) for all \( i \geq 0 \).

(i) \( \sigma, 0 \models p \) holds as \( p \) is true in state \( s_0 \), i.e. \( p \in s_0 \).

(ii) \( \sigma, 1 \models \neg q \) holds as \( q \) is false in state \( s_1 \), i.e. \( q \notin s_1 \).

(iii) \( \sigma, 0 \models \text{Next } \neg q \) holds as \( \sigma, 1 \models \neg q \).

(iv) \( \sigma, 0 \models \text{Always } p \) holds as \( \sigma, i \models p \) holds for all \( i \geq 0 \).

(v) \( \sigma, 0 \models \text{Sometime } r \) holds as \( \sigma, 2 \models r \) and \( 2 > 0 \).

(vi) \( \sigma, 0 \models \text{Always Sometime } q \) holds as \( \sigma, i \models \text{Sometime } q \) holds for all \( i \geq 0 \).

**Exercise 3.1 Formula evaluation:** Given atomic propositions \( p, q \) and \( r \), together with an infinite sequence \( \sigma \) such that \( s_{i+3} = \{p\}, s_{i+3+1} = \{p, q\} \) and \( s_{i+3+2} = \{p, q, r\} \) for all \( i \geq 0 \). Determine the truth or otherwise of:

(i) \( \sigma, 0 \models (p \land \neg q) \)

(ii) \( \sigma, 0 \models \text{Next Next } r \)

(iii) \( \sigma, 0 \models (p \land \neg q) \Rightarrow \text{Next Next } r \)

(iv) \( \sigma, 0 \models \text{Always } ((p \land \neg q) \Rightarrow \text{Next Next } r) \)

### 3.2 Kripke structures

In the above, models for propositional linear, discrete, temporal logic were given as infinite state sequences. One might think this is a problem from the point of view of determining whether a formula is satisfiable, i.e. whether a model for the formula exists. Fortunately, however, this logic has a very special property, called the finite model property, also often referred to as the small model property. This property means that any satisfiable (temporal logic) formula has a model that can be represented in a finite way. As an example, consider the infinite sequence model of Example 3.1. It can be represented by a directed graph as in Figure 2. The nodes of the graph represent states of the infinite state sequence. The nodes have been further labelled by the sets of atomic propositions that hold in the state that the node represents. One of the nodes is marked as an initial node. From the initial node, one can trace an infinite path in the graph that corresponds, via the proposition labelling, to the infinite sequence model of Example 3.1.

These graph-like structures are referred to as Kripke structures.
Definition 3.1 Formally, a Kripke structure $K$ is a four element tuple:-

$$K = (\ S \quad \text{set of states}, \ R \quad \text{a total binary relation over } S, \ i.e. \subseteq (S \times S), \ I \quad \text{an interpretation for atomic propositions}, \ S_0 \quad \text{a set of initial states, i.e. } S_0 \subseteq S)$$

where $AP$ is the set of atomic propositions.

Definition 3.2 A path in a Kripke structure $K = (S,R,I,S_0)$ is an infinite sequence of states, $s_0,s_1,\ldots,s_i,\ldots$ s.t. for all $i$, $(s_i,s_{i+1}) \in R$.

Example 3.2 Another Kripke structure: Consider the following graph.

The nodes labelled Sa, Sb, Sc, and Sd represent the same named states of the Kripke structure. The node labelled by Se denotes the fully connected subgraph formed by the set of states $\{Se_1, Se_2, Se_3, Se_4\}$; this abstraction node Se thus represents a state with any assignment to the propositions $p$ and $q$, hence the truth value $T$ appearing as the set of
propositions. This abstraction simply avoids the need to draw out the detailed subgraph.

\[ K = (\{Sa, Sb, Sc, Sd, Se_1, Se_2, Se_3, Se_4\}, \]
\[ \{(Sa, Sa), (Sa, Sb), (Sa, Sc), (Sa, Sd),
(Sb, Se_1), (Sb, Se_2), (Sb, Se_3), (Sb, Se_4),
(Sc, Sc), (Sc, Sd), (Sc, Sa), (Sc, Sb),
(Sd, Se_1), (Sd, Se_2), (Sd, Se_3), (Sd, Se_4),
(Se_1, Se_1), (Se_1, Se_2), (Se_1, Se_3), (Se_1, Se_4),
(Se_2, Se_2), (Se_2, Se_3), (Se_2, Se_4), (Se_2, Se_1),
(Se_3, Se_3), (Se_3, Se_4), (Se_3, Se_1), (Se_3, Se_2),
(Se_4, Se_4), (Se_4, Se_1), (Se_4, Se_2), (Se_4, Se_3)\},
\{(Sa \mapsto \{p\}), [Sb \mapsto \{q\}], [Sc \mapsto \{}], [Sd \mapsto \{p, q\}],
[Se_1 \mapsto \{}], [Se_2 \mapsto \{p\}], [Se_3 \mapsto \{q\}], [Se_4 \mapsto \{p, q\}]),
\{Sa, Sb, Sc, Sd\})\]

The given Kripke structure contains all the models for the temporal formula

\[ p \Rightarrow \text{Sometime } q \]

(holding in an initial state). The four initial states Sa, Sb, Sc and Sd characterise all the possible assignments one may have for the propositions p and q. Clearly, if the given temporal formula is to hold in state Sa then, as p is true in that state, there must be infinite paths from Sa which pass through a state in which q holds; this is the case as there are paths that visit Sb or Sd. Similarly, for state Sd, another initial state that creates an obligation to satisfy \text{Sometime } q, which of course is satisfied immediately in Sd. For the two initial states that don’t have p present, then all possible infinite paths should be traceable, which indeed they are. Thus, this structure certainly contains all the models for the given formula. In fact, the structure also contains paths that are not models. We will see later how restrictions can be placed on the Kripke structure so that can be used to represent exactly the models of a given formula.

Exercise 3.2

1. Show that the Kripke structure depicted by the graph

![Diagram](image)

assuming that the set of initial states is \{Sa\} and contains models for the temporal formulas

(i) \text{Sometime } p

(ii) \text{Always Sometime } p

2. Construct an infinite sequence model for

\text{Always Sometime } p

that is not a path of the above graph.
3. Show that the Kripke structure depicted by the graph

\[ \text{assuming that the set of initial states is } \{ \text{Sa} \} \text{ and contains models for the temporal formula} \]

\textbf{Sometime Always } p \\
\textit{but doesn't contain models for the formula} \\
\textbf{Sometime Always } \neg p \\

4 Graph representation of Programs

In the above section we have seen how graphs can be viewed as containing the models for temporal properties, although we have yet to see how we can restrict the set of paths of a graph to be exactly those models of a given temporal formula. Next, I want to show you how we can represent concurrent programs as similar structures. Once we’ve done so, we’ll have a means for comparing the behaviours of programs.

We will work with a very simple notion of program. We will not consider any particular programming language, principally because we don’t have the time, but also because the detail is not necessary to gain an understanding of the underlying ideas. We will consider concurrent programs that compute over a set of shared variables. Although we don’t consider any real programming language (say Java), we do require some notation. So, for example, the pseudo program

\begin{verbatim}
 cobegin
  11:  x = x + 1;
 ||
  12:  y = y - 2;
 coend
\end{verbatim}

denotes a program that, when run, will concurrently execute the two assignment statements \( x = x + 1 \) and \( y = y - 2 \). We will assume an interleaving model of concurrent execution. Thus the behaviour of a concurrent program will be represented by the possible interleavings of the execution traces of the sequential subprograms (in the above case, the two assignment statements).

Let \( CP \) be a concurrent program over a finite set of variables \( V \) ranging over a finite data set \( D \). A state of the concurrent program \( CP \) can be represented by mapping the variables \( V \) to values from \( D \). The restriction to finite sets for variables and data is important for ensuring that our program has a finite upper bound on the number of possible states in may be in. We further assume that the set of variables contains a program counter \( pc \), uniquely determining the set of program instructions that are next to be executed.

For example, an initial state of the above program might be

\[ [x \mapsto 1, y \mapsto 3, \text{pc} \mapsto 0] \]
capturing that the initial value of $x$ is 1, that of $y$ is 3 and that the $pc$ is 0. An concurrent execution state sequence might then be

$$\langle [x \mapsto 1, y \mapsto 3, pc \mapsto 0], [x \mapsto 2, y \mapsto 3, pc \mapsto 1], [x \mapsto 2, y \mapsto 1, pc \mapsto 3]\rangle$$

in which the assignment to $x$ happens before the assignment to $y$, but another execution sequence is

$$\langle [x \mapsto 1, y \mapsto 3, pc \mapsto 0], [x \mapsto 1, y \mapsto 1, pc \mapsto 2], [x \mapsto 2, y \mapsto 1, pc \mapsto 3]\rangle$$

The behaviour of the concurrent program is taken as the set of all the possible execution state sequences.

**Exercise 4.1** What are the possible interleaved execution sequences of the concurrent program

```plaintext
cobegin
  \ell_{11}: x = x + 1;
  \ell_{12}: y = x + y;
||
  \ell_{2}: y = y - 2;
coend
```

assuming initial values of $x$ and $y$ as 0 and 3 respectively. You may assume that the assignments states are executed atomically. Hint: There are three possible interleavings which are formed by executing the $\ell_{2}$ statement either before $\ell_{11}$, or after $\ell_{11}$ and before $\ell_{12}$, or after the $\ell_{12}$ statement.

### 4.0.1 Logical modelling of a concurrent program

We now show how to represent a concurrent program as a 1st order logic formula. We treat each step of (concurrent) program as a state transformer. Repeated application of the state transformer can then be thought of as building a state sequence corresponding to some particular execution sequence.

States can easily be represented by logical formulas, for example, the state

$$[x \mapsto 1, y \mapsto 3, pc \mapsto 0]$$

and can be represented by 1st order formula

$$x == 1 \land y == 3 \land pc == 0$$

Thus, we will let a 1st order formula $S_0$ over variables $V$ represent the set of initial states of a given concurrent program $CP$.

We will represent the state transformation, corresponding to some atomic step of the concurrent program, as a relation between the old values of variables and the new, transformed values. Think of the set of unprimed variables $V$ as denoting the current state variables of the program $CP$. We introduce a corresponding set of primed variables $V'$ (i.e. for each variable $x$ there is a primed version $x'$) to denote the values of variables $V$ in the next state; we refer to the variable $V'$ as the next state variables. For example, a
formula $x' = x + 1$ represents a state transition where the next value of $x$ is its current value plus 1.

We can now write down a 1st order formula, which we will name as $\mathcal{R}$, over the current state variables $V$ and their next state correspondents $V'$. The valuations of the formula $\mathcal{R}$ can be viewed as a set of pairs of states, i.e. a transition relation.

**Example 4.1** As an example, consider the sequential program

\begin{align*}
\ell_1: & \ x = x + 1; \\
\ell_2: & \ y = y - 2; \\
\ell_3: & \ \text{if} \ (y > 0) \ \text{goto} \ \ell_1;
\end{align*}

which starts execution from the statement labelled by $\ell_1$ with initial values for $x$ and $y$ as 1 and 3, respectively. Assume a program counter variable $pc$, which has values 1, 2 and 3 corresponding to statement labels $\ell_1$, $\ell_2$ and $\ell_3$. Remember that the primed variables denote the next state values of the unprimed variables. The transition relation, encoded as a logical formula over unprimed and primed variables, is then given as follows.

\begin{align*}
(pc == 1) \Rightarrow ((x' == x + 1) \land (y' == y) \land (pc' == 2)) \\
\land \\
(pc == 2) \Rightarrow ((y' == y - 2) \land (x' == x) \land (pc' == 3)) \\
\land \\
(pc == 3) \land (y > 0) \Rightarrow ((x' == x) \land (y' == y) \land (pc' == 1))
\end{align*}

The initial state is characterised by the logical formula

\[(pc == 1) \land (x == 1) \land (y == 3)\]

**Example 4.2** Consider the program

\begin{verbatim}
ℓ0: cobegin
    ℓ11: x = x + 1;
    ℓ12: y = y - 2;
    ℓ13: ||
    ℓ21: y = y + 1;
    ℓ22: coend
ℓ3: if (y > 0) goto ℓ0;
\end{verbatim}

which starts execution at location $\ell_0$ with initial values of $x$ and $y$ as 1 and 3.

The `cobegin coend` statement has its substatements executed concurrently. We assume that the assignment statements are executed atomically. Then, for an interleaved mode of execution we have the following possible pairs of statement labels for the `cobegin coend`:
The transition relation for the program is then given by:

\[
(pc == 0) \Rightarrow ( ((x' == x + 1) \land (y' == y) \land (pc' == 1221)) \lor
((x' == x) \land (y' == y + 1) \land (pc' == 1122)))
\]

\[
(pc == 1221) \Rightarrow ( ((y' == y - 2) \land (x' == x) \land (pc' == 1321)) \lor
((y' == y + 1) \land (x' == x) \land (pc' == 1222)))
\]

\[
(pc == 1321) \Rightarrow ( (y' == y + 1) \land (x' == x) \land (pc' == 1322))
\]

\[
(pc == 1122) \Rightarrow ( (x' == x + 1) \land (y' == y) \land (pc' == 1222))
\]

\[
(pc == 1222) \Rightarrow ( (y' == y - 2) \land (x' == x) \land (pc' == 3))
\]

\[
(pc == 3) \land (y > 0) \Rightarrow ( (x' == x) \land (y' == y) \land (pc' == 0))
\]

and the initial state formula is

\[
(pc == 0) \land (x == 1) \land (y == 3)
\]

4.0.2 From 1st order logic representation to Kripke structure

Given the logical representation, as a formula characterising a state transformation relation (or state transition relation) and a formula characterising the initial states of the program, we can now directly compute a corresponding Kripke structure, whose paths will correspond to legal execution traces of the program (from the initial states).

We build a Kripke structure \(K = (S, R, I, S_0)\) over a proposition alphabet \(AP\) from the initial state formula \(S\) and state transformation relation \(R\).

We take the set of atomic propositions \(AP\) for the Kripke structure be the set of identities \(v == d\), for variables \(v \in V\) and data values \(d \in D\).

We take a state as above, i.e. a mapping from \(V\) to \(D\). Then given a proposition, i.e. an identity \(v == d\), we define its valuation as true in a state \(s\) if and only if \(s(v) == d\).

The components of the structure \(K = (S, R, I, S_0)\) are then defined as:

- set of states \(S\) is set of all valuations for \(V\);
- the initial state set \(S_0\) is the subset of \(S\) that satisfy \(S_0\);
- for states \(s, s', (s, s') \in R\) holds if \(R\) is true for assignments to variables \(v\) and \(v'\) according to \(s\) and \(s'\), respectively;
- The interpretation function \(I : S \mapsto 2^{AP}\) is defined so that \(I(s)\) gives the subset of all atomic propositions true in \(s\);
Finally, the relation \( R \) is extended to be total, i.e. \((s, s)\) is added to \( R \) for all states \( s \) that have no successor in \( R \).

**Example 4.3** Consider the following simple concurrent program.

\[
\ell_0: \text{while } (y > 0) \\
\ell_1: \text{cobegin} \\
\quad \ell_{11}: x = x + 1; \\
\quad \ell_{12}: \\
\quad | | \\
\quad \ell_{21}: y = y - 2; \\
\ell_{22}: \text{coend}
\]

Similar to Example 4.1, we encode the label pairs, occurring through the concurrent execution of the \textit{cobegin coend} statement, with a single \textit{pc} value. Note that the label pair \((\ell_{11}, \ell_{21})\) is identified with \(\ell_0\), and similarly the label pair \((\ell_{12}, \ell_{22})\) is identified with \(\ell_0\).

<table>
<thead>
<tr>
<th>Label pair</th>
<th>\textit{pc} value</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\ell_{11}, \ell_{21}))</td>
<td>1</td>
</tr>
<tr>
<td>((\ell_{11}, \ell_{22}))</td>
<td>1122</td>
</tr>
<tr>
<td>((\ell_{12}, \ell_{21}))</td>
<td>1221</td>
</tr>
<tr>
<td>((\ell_{12}, \ell_{22}))</td>
<td>0</td>
</tr>
</tbody>
</table>

The logical characterisation of the transition relation is then:

\[
(pc == 0) \land (y > 0) \Rightarrow (((pc' == 1) \land (x' == x) \land (y' == y)) \land
((pc' == 1221) \land (x' == x + 1) \land (y' == y)) \lor
((pc' == 1122) \land (x' == x) \land (y' == y - 2)))
\]

\[
(pc == 1221) \Rightarrow ((pc' == 0) \land (y' == y - 2) \land (x' == x))
\]

\[
(pc == 1122) \Rightarrow ((pc' == 0) \land (x' == x + 1) \land (y' == y))
\]

Given the initial state formula as:

\[
(pc == 0) \land (x == 10) \land (y == 1)
\]

we can generate the Kripke structure, with transition structure graphically represented as
where the interpretation function $I$ is defined as below.

\[
I(s_0) = \{ pc = 0, x = 10, y = 1 \}
I(s_1) = \{ pc = 1122, x = 11, y = 1 \}
I(s_2) = \{ pc = 1221, x = 10, y = -1 \}
I(s_3) = \{ pc = 0, x = 11, y = -1 \}
\]

The generated Kripke structure corresponds to a single execution of the loop body. The state $s_3$ loops on itself due to the extension of $R$ to be total.

If the initial condition had been

\[(pc == 0) \land (x == 10) \land (y == 3)\]

then there would have been two executions of the loop body and the following graph would be generated.

![Graph](image)

\[
I(s_0) = \{ pc = 0, x = 10, y = 3 \}
I(s_1) = \{ pc = 1122, x = 11, y = 3 \}
I(s_2) = \{ pc = 1221, x = 10, y = 1 \}
I(s_3) = \{ pc = 0, x = 11, y = -1 \}
I(s_4) = \{ pc = 1122, x = 12, y = 1 \}
I(s_5) = \{ pc = 1221, x = 11, y = -1 \}
I(s_6) = \{ pc = 0, x = 12, y = -1 \}
\]

**Exercise 4.2** Construct a Kripke structure that corresponds to the following program when started with the initial values of $x$ and $y$ both set as $1$.

\[
\ell_0: \text{while } (y > 0)
\]

\[
\ell_1: \text{cobegin}
\ell_{11}: x = y + 1;
\ell_{12}:
||
\ell_{21}: y = x - 2;
\ell_{22}:
\text{coend}
\]

If we assumed that the set of values $D$ was the set of integers, what happens if the initial value of $y$ had been $2$ instead of $1$. Compute two iterations of the body and see what happens.

**Exercise 4.3** Now construct a Kripke structure that corresponds to the following program when started with the initial values of $x$ and $y$ set as $1$ and $2$, respectively.
ℓ0: while (y > 0)  
ℓ1: cobegin  
ℓ11: x = (y + 1)%3;  
ℓ12:  
| |  
ℓ21: y = x - 2;  
ℓ22:  
coend

Note: the addition in the statement labelled ℓ11 is undertaken modulo 3.

4.1 Reachability Analyses

Now we have a graph-based representation of both programs and temporal (behavioural) properties, we can begin to consider formal means for analysis. For example, we would like to determine whether all paths of the program Kripke structure are contained within the set of valid paths of the temporal property Kripke structure — this is model-checking. For the moment, we will delay on that since we have not yet built precisely the right set of models. However, we can perform various reachability analyses on the program Kripke structure. Indeed, we might characterise a set of “bad” or “unsafe” states — BAD and then check that no state in BAD is reachable from any initial state.

Definition 4.1 A state $s_g$ is said to be reachable from a state $s$ in a Kripke structure $K = (S, R, I, S_0)$ if there is a path of $K$ from state $s$ that passes through $s_g$, i.e. there is a sequence of states $s_i$, for $i = 0..n$ for some $n > 0$ such that for all $i > 0$, $(s_{i-1}, s_i) \in R$ and $s_0 = s$ and $s_n = s_g$.

Example 4.4 Consider again the state graph

![State Graph]

with interpretation

$I(s_0) = \{ pc = 0, x = 10, y = 3 \}$  
$I(s_1) = \{ pc = 1122, x = 11, y = 3 \}$  
$I(s_2) = \{ pc = 1221, x = 10, y = 1 \}$  
$I(s_3) = \{ pc = 0, x = 11, y = 1 \}$  
$I(s_4) = \{ pc = 1122, x = 12, y = 1 \}$  
$I(s_5) = \{ pc = 1221, x = 11, y = -1 \}$  
$I(s_6) = \{ pc = 0, x = 12, y = -1 \}$

We use a depth-first search algorithm to determine whether some state is reachable from another. However, it is trivial to see in the above graph that all states identified in the diagram are in fact reachable from the initial $s_0$, however, only states in the set $\{s_1, s_3, s_4, s_5, s_6\}$ are reachable from $s_1$. Obviously, $s_1$ can not reach $s_2$, and vice-versa.
Pseudo code for a forwards reachability algorithm is presented in Figure 3. A call of \( f_{\text{Reachable}}(s, \text{targetSet}, \{\}) \) will return true if and only there is a path from the state \( s \) to some state in the set \( \text{targetSet} \). As you can see, it is just a version of depth-first graph search.

```java
boolean f_Reachable ( s, targetSet, visitedSet) {
    boolean found = false;
    if (s ∈ targetSet) return true;
    if (s ∈ visitedSet) return false;
    else {
        for each successor s’ of s and while not found do
            found = f_Reachable(s’, targetSet, visitedSet ∪ {s});
        return found;
    }
}
```

Figure 3: Forwards Reachability Algorithm

Alternatively, one can compute the set of reachable states from a given state. Pseudo code for achieving that is presented in Figure 4.

```java
Set f_ReachableStates ( s, visitedSet) {
    if (s ∈ visitedSet) return visitedSet;
    else {
        newVisitedSet = visitedSet ∪ {s}
        for each successor s’ of s do
            newVisitedSet =
                newVisitedSet ∪ f_ReachableStates(s’, newVisitedSet );
        return newVisitedSet;
    }
}
```

Figure 4: Forwards Reachable States Algorithm

Finally, instead of computing forwards through the graph, one might wish to determine whether there is a path to a given state, i.e. search backwards from the given state. To move backwards, we thus use a predecessor function, instead of the successor function. Figure 5 contains pseudo code for computing the set of states backwards reachable from a given state.

**Exercise 4.4**

1. Write out a pseudo code description of the backwards version of the algorithm in Figure 3.

2. Now write a corresponding Java method for the class Graph given in the appendix.
Set $b_{\text{ReachableStates}}(s, \text{visitedSet})$ {

if ($s \in \text{visitedSet}$) return $\text{visitedSet}$;
else {
    newVisitedSet = visitedSet $\cup \{s\}$
    for each predecessor $s'$ of $s$ do
        newVisitedSet =
            newVisitedSet $\cup b_{\text{ReachableStates}}(s', \text{newVisitedSet})$;
    return newVisitedSet;
}
}

Figure 5: Backwards Reachable States Algorithm

5 Summary - so far

Following a very brief introduction, covering a little background to the area of automated program verification, we have show in this section how one can reduce the verification of logical properties of program models to graph problems. This is the approach taken by explicit state model checking, tools for which are now regularly used in industries concerned with safety critical computing applications. However, you must appreciate the very high computational complexity, i.e. cost, of undertaking explicit state model checking. Remember the question, what’s the maximum number of states of a single process program over ten 32-bit variables? Fortunately, simple reachability analyses are usually linear, in the worst case, in size of graph structure. But when we come to consider the analysis of more general temporal logic properties, we will see the complexity rise; the problem is indeed much harder, computationally speaking.

The next section of these notes will consider a restricted form of model checking based on finite word automata.
References


import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

public class Graph{

    private Set<State> nodes;
    private Set<Pair<State>> edges;

    public Graph(){
        nodes = new HashSet<State>();
        edges = new HashSet<Pair<State>>();
    }

    public void addNode(State state){
        nodes.add(state);
    }

    public void addEdge(Pair<State> edge){
        edges.add(edge);
    }

    public Set<State> successors(State s){
        Set<State> successors = new HashSet<State>();
        Iterator<Pair<State>> edge_iterator = edges.iterator();
        while (edge_iterator.hasNext()){  
            Pair<State> edge = edge_iterator.next();
            if (edge.fst().equals(s)){
                successors.add(edge.snd());
            }
        }
        return successors;
    }

    public Set<State> predecessors(State s){
        Set<State> predecessors = new HashSet<State>();
        Iterator<Pair<State>> edge_iterator = edges.iterator();
        while (edge_iterator.hasNext()){  
            Pair<State> edge = edge_iterator.next();
            if (edge.snd().equals(s)){
                predecessors.add(edge.fst());
            }
        }
        return predecessors;
    }
}
public boolean fReachable(State s,
    Set<State> targetStates,
    Set<State> visitedStates){
    if (targetStates.contains(s)) return true;
    if (visitedStates.contains(s)) return false;
    else {
        Set<State> successors = successors(s);
        Iterator<State> successor_iterator = successors.iterator();
        boolean found = false;
        while (!found && successor_iterator.hasNext()){ 
            State successorState = successor_iterator.next();
            Set<State> newVisitedStates = new HashSet<State>(visitedStates);
            newVisitedStates.add(s);
            found = fReachable(successorState, targetStates, newVisitedStates);
        }
        return found;
    }
}

public Set<State> fReachableStates(State s, Set<State> visitedStates){
    if (visitedStates.contains(s)) return visitedStates;
    else {
        Set<State> newVisitedStates = new HashSet<State>(visitedStates);
        newVisitedStates.add(s);
        Set<State> successors = successors(s);
        Iterator<State> successor_iterator = successors.iterator();
        while (successor_iterator.hasNext()) {
            State successor = successor_iterator.next();
            newVisitedStates.addAll(fReachableStates(successor,newVisitedStates));
        }
        return newVisitedStates;
    }
}

public Set<State> bReachableStates(State s, Set<State> visitedStates){
    if (visitedStates.contains(s)) return visitedStates;
    else {
        Set<State> newVisitedStates = new HashSet<State>(visitedStates);
        newVisitedStates.add(s);
        Set<State> predecessors = predecessors(s);
        Iterator<State> predecessor_iterator = predecessors.iterator();
        while (predecessor_iterator.hasNext()) {
            State predecessor = predecessor_iterator.next();
            newVisitedStates.addAll(bReachableStates(predecessor, newVisitedStates));
        }
        return newVisitedStates;
    }
}
public class State{

    private String name;

    public State(String givenName){
        name = givenName;
    }

    public boolean equals(Object o){
        State state = (State)o;
        return name.equals(state.name);
    }
} //class State

public class Pair<T>{

    private T fst;
    private T snd;

    public Pair(T givenFst, T givenSnd){
        fst = givenFst;
        snd = givenSnd;
    }

    public T fst(){
        return fst;
    }

    public T snd(){
        return snd;
    }
} //class T