HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Laboratory of Theoretical Computer Science
T-79.298 Postgraduate Course in Digital Systems Science
Fall Semester 2001

Satu Virtanen

# Algorithms for Model Checking and Modeling of Reactive Systems

Course report on Sections 8 and 9 of *Model Checking* by Clarke and Schlingloff [CS01]
DRAFT

# Contents


**1 Basic Model Checking Algorithms** **1**

1.1 Global Model Checking for Branching Time Logics . . . . . 2

1.1.1 Modal Logic . . . . . 2

1.1.2 Computation Tree Logic (CTL) . . . . . 4

1.1.3 Fairness . . . . . 12

1.2 Local Model Checking for Linear Time Logics . . . . . 12

1.2.1 Modal Logic with a Single Accessibility Relation . . . . . 13

1.2.2 Linear Temporal Logic (LTL) . . . . . 17

1.3 Model Checking for Propositional $\mu$-Calculus . . . . . 24

1.3.1 Global Model Checking for $\mu$TL . . . . . 25

1.3.2 Local Model Checking for $\mu$TL . . . . . 28

**2 Modeling of Reactive Systems** **31**

2.1 Distributed Systems . . . . . 31

2.1.1 Communication by Message Passing . . . . . 32

2.1.2 Communication by Shared Variables . . . . . 32

2.2 Concurrent Systems . . . . . 33

2.2.1 Synchronization According to Time . . . . . 34

2.3 Formalisms for Finite State Systems . . . . . 34

2.3.1 Parallel Transition Systems . . . . . 35

2.3.2 Elementary Petri Nets . . . . . 35

2.3.3 Equivalence of Transition Systems and Elementary Petri Nets 36

2.3.4 Shared Variable Programs . . . . . 36

2.4 Examples of Reactive System Applications . . . . . 37
2.4.1 The Grid Puzzle . . . . . 37
2.4.2 A Shift Register for Interfacing a Parallel Data Bus . . . . . 40
2.4.3 A Communication Protocol for a Cellular Phone . . . . . 42

**Bibliography** **46**

**Index** **47**

# Notations and Acronyms

| | |
|---|---|
| $\mathcal{M}$ | A Kripke-model of a system, $\mathcal{M} = (U, \mathcal{I}, w_0)$ |
| $\varphi$ | A specification formula to be verified |
| $U$ | A universe, i.e. a set of states |
| $\mathcal{P}$ | Set of atomic propositions |
| $\mathcal{R}$ | Set of transition relations $U \times U$ |
| $\mathcal{I}$ | An interpretation on $\mathcal{P}$ and $\mathcal{R}$ |
| $\mathcal{F}$ | A frame consisting of a universe and an interpretation |
| $\emptyset$ | The empty set, denoted by {} in [CS01] |
| $\mathcal{O}(f(n))$ | Worst-case complexity proportional to $f(n)$ |
| **A** | The path quantifier "all" |
| **E** | The path quantifier "exists" |
| **F** | The temporal operator "future" |
| $\mathbf{F}^*$ | The reflexive temporal operator "future" |
| $\mathbf{F}^+$ | The non-reflexive temporal operator "future" |
| **G** | The temporal operator "global" |
| $\mathbf{G}^*$ | The reflexive temporal operator "global" |
| $\mathbf{G}^+$ | The non-reflexive temporal operator "global" |
| **X** | The temporal operator "next" |
| **U** | The temporal operator "until" |
| $\mathbf{U}^*$ | The reflexive temporal operator "until" |
| $\mathbf{U}^+$ | The non-reflexive temporal operator "until" |
| BDD | Binary decision diagram |
| CTL | Computation Tree Logic |
| LTL | Linear Temporal Logic |
| $\mu$TL | Propositional $\mu$-Calculus |
| SCC | Strongly connected component |
| SDL | Specification and Description Language |
| SMV | Symbolic Model Verifier |

# Chapter 1

# Basic Model Checking Algorithms

The model checking problem is to *decide whether* $\mathcal{M} \models \varphi$ applies for a given model $\mathcal{M}$ and a given formula $\varphi$. The model $\mathcal{M}$ could theoretically be encoded as a set of assumptions $\Phi$, after which it could be decided whether the formula $\varphi$ can be deduced from these, i.e. $\Phi \vdash \varphi$. These assumptions or *program axioms* in $\Phi$ usually have a special form, such as $(\mathtt{state_i} \rightarrow (\mathbf{X}\,\mathtt{succ_{i1}} \vee \ldots \vee \mathbf{X}\,\mathtt{succ_{in}}))$ for linear time modeling, or $(\mathtt{state_i} \rightarrow (\langle a_i \rangle\,\mathtt{succ_{i1}} \wedge \ldots \wedge \langle a_n \rangle\,\mathtt{succ_{in}}))$ for branching time modeling. This special form cannot in general be taken into account by the decision procedure for $\Phi \vdash \varphi$. Therefore every assumption will be broken down into its basic propositional components, which is very inefficient. Only systems of very small scale can be verified in this manner, and thus it is not a practical approach to verification of real-world systems.

The model checking algorithms presented here will avoid any decomposition or encoding of the models in sets of axioms, but attempt to work on the models directly. Sometimes the model is represented by an automaton or several automata, which are then used in the model checking algorithm. The algorithms inspect the given model $\mathcal{M}$ and examine whether a given specification formula $\varphi$ can be satisfied by either a *computation tree* (for branching time semantics) or a *computation path* (for linear approaches) that can be somehow generated from $\mathcal{M}$. The only parameters necessary for the algorithms are the model $\mathcal{M}$ and the formula $\varphi$ to check.

Models are often represented as either *Kripke-models*, which consist of a universe $U$, which is a set of states, an initial state $w_0 \in U$, which is often called the *current state* (or current point), and a set of *accessibility relations* $\mathcal{I}$. Thus a Kripke-model is usually denoted as the 3-tuple $\mathcal{M} = (U, \mathcal{I}, w_0)$. Kripke-models are in a sense directed graphs as there can be several successors to each state in each of the transition relations defined by $\mathcal{I}$.

There are two different tasks in model checking, depending whether *universal* or *initial* satisfiability is required. The universal satisfiability question asks whether $(U, \mathcal{I}) \models \varphi$, where $\mathcal{F} = (U, \mathcal{I})$ is called a *frame*[1]. This means that $\varphi$ is to be satisfied in all of the states of the universe $U$ of the frame $\mathcal{F}$:

$$(U, \mathcal{I}) \models \varphi \;\leftrightarrow\; \forall w_0 \in U : (U, \mathcal{I}, w_0) \models \varphi.$$

[1] Usually the term "frame" is used to denote a set of states and a transition relation, not fixing the valuations of the propositions $\mathtt{p} \in \mathcal{P}$ as it does here

An equivalent form of stating this would be requiring the set of all "states" $w \in U$ for which $w \models \varphi$, denoted with $\varphi^{\mathcal{F}}$ to coincide with the universe U, i.e. $U = \varphi^{\mathcal{F}} = \{w \in U \mid w \models \varphi\}$.

If there is an algorithm for calculating the set $\varphi^{\mathcal{F}}$, it can be directly used to solve initial satisfiability, i.e. whether the formula is valid in a specified initial state (sometimes a set of initial states) $(U, \mathcal{I}, w_0) \models \varphi$, by checking whether $w_0 \in \varphi^{\mathcal{F}}$. The trick also works the other way around: with a procedure to check the satisfiability in the initial version, simple iteration over the possible states $w_i$, inspecting with the procedure whether $(U, \mathcal{I}, w_i) \models \varphi$, allows the set $\varphi^{\mathcal{F}}$ can be constructed.

Model checking algorithms that perform iteration on the structure of the formula $\varphi$ (e.g. by examining sub-formulas in an iterative fashion) and traverse the entire model $\mathcal{M}$ per each iteration step are called *global* model checking algorithms. The *local* model checking algorithms extend the portion of $\mathcal{M}$ to be examined per each round in an iterative manner, examining the truth value of the entire $\varphi$ (e.g. evaluating all of the sub-formulas of $\varphi$). The choice of how to proceed does not affect the worst-case time complexity, but there can be significant differences in the average case computational complexity.

Thus there are three choices to be made when selecting a model checking algorithm for verification purposes:

(1) whether to use *linear* or *branching* time logic

(2) whether to examine the *universal* or *initial* satisfiability of $\varphi$

(3) whether to use a *global* or *local* method of proceeding

In principle, these three selections are independent of each other, but in practice, the branching time logics are mostly bound to global approaches with universal validity, whereas linear time logics usually have local algorithms with initial validity checking.

## 1.1 Global Model Checking for Branching Time Logics

A branching time logic is such where each state can have more than one successor state in each transition relation $R \in \mathcal{R}$ and the decision between the possible computation sequences is non-deterministic at each time step. This section will present an algorithm for modal logic (interpreted in branching time) as a set of rules and a pseudocode model checking algorithm for CTL (Computation Tree Logic).

### 1.1.1 Modal Logic

The motivation of *modal logic* is to formalize temporal phrases such as "eventually" and "always". In modal logic, an atomic proposition $\mathtt{p} \in \mathcal{P}$ is interpreted as "$\mathtt{p}$ *is valid now*". Instead of using some kind of general validity of the proposition, a notion of time or current state is bound to the formulas with the temporal operators. The

grammar of the multimodal and temporal logic is the here limited[2] to the following:

$$\mathbf{ML} ::= \mathcal{P} \mid \bot \mid (\mathbf{ML} \rightarrow \mathbf{ML}) \mid \langle \mathcal{R} \rangle \mathbf{ML}.$$

For a Kripke-model $\mathcal{M} = (U, \mathcal{I}, w_0)$, the *interpretation* $\mathcal{I}$ assigns a subset $\mathcal{I}(\mathtt{p})$ of the universe $U$ to every $\mathtt{p} \in \mathcal{P}$, and to every $R \in \mathcal{R}$, a subset of the Cartesian product of the states, i.e. $\mathcal{I}(R) \subseteq U \times U$.

The validity of a multimodal formula $\varphi$ is evaluated according to the following rule set. A point to be noted on the notation is that $w \models \varphi$ is a shorthand for $(U, \mathcal{I}, w) \models \varphi$ when the frame $\mathcal{F} = (U, \mathcal{I})$ is assumed implicitly known.

(i) $\mathcal{M} \models \mathtt{p}$ if and only if $w_0 \in \mathcal{I}(\mathtt{p})$

(ii) $\mathcal{M} \not\models \bot$

(iii) $\mathcal{M} \models (\varphi \rightarrow \psi)$ if and only if $\mathcal{M}\varphi$ implies $\mathcal{M} \models \psi$

(iv) $\mathcal{M} \models \langle R \rangle \varphi$ if and only if there exists a $w_1 \in U$ such that $(w_0, w_1) \in \mathcal{I}(R)$ and $(U, \mathcal{I}, w_1) \models \varphi$

The set of points validating a multimodal formula $\varphi$, denoted by $\varphi^{\mathcal{F}} = \{w \in U \mid w \models \varphi\}$, can be calculated by a *recursive descent* on the structure of $\varphi$ for a given Kripke-frame $\mathcal{F} = (U, \mathcal{I})$ according to the following rules:

(i) for an *atomic proposition* $\mathtt{p}$, $\mathtt{p}^{\mathcal{F}} \triangleq \mathcal{I}(\mathtt{p})$

(ii) $\bot^{\mathcal{F}} \triangleq \emptyset$

(iii) $(\phi \rightarrow \psi)^{\mathcal{F}} \triangleq (U \setminus \phi^{\mathcal{F}}) \cup \psi^{\mathcal{F}}$

(iv) $(\langle R \rangle \psi)^{\mathcal{F}} \triangleq \{w \in U \mid \exists w' \in \psi^{\mathcal{F}}, (w, w') \in \mathcal{I}(R)\}$

The first rule defines the set of states in which an atomic proposition $\mathtt{p}$ is valid to be the set defined by $\mathcal{I}$. The second rule requires a symbol $\bot$ ("false") not to be valid in any state of the universe. Rule (iii) defines the set in which the implication $\psi \rightarrow \varphi$ is valid to be those states of the universe in which the "condition" $\psi$ is not valid, combined with the set of states where the "consequence" $\varphi$ is valid. This is because of the definition of the truth value for an implication. The fourth rule states that the set of states in which a formula $\langle R \rangle \psi$ is valid is the set of those states that have an immediate "successor" $w'$ such that $(w, w') \in \mathcal{I}(R)$ for which $w' \in \psi^{\mathcal{F}}$, i.e. there is an arc connecting each state in $(\langle R \rangle \psi)^{\mathcal{F}}$ to a state in $\psi^{\mathcal{F}}$.

The set of states where $\langle R \rangle \psi$ is valid can be calculated from $\psi^{\mathcal{F}}$ in two ways. First, by checking whether for each $w \in U$, $\psi^{\mathcal{F}} \cap R(w) \neq \emptyset$, i.e. whether the intersection is non-empty. Another possibility is to define the *inverse image* of a state $w$ under the relation $R$ as $R^{-1}(w') = \{w \mid (w, w') \in \mathcal{I}(R)\}$ and then calculate $\bigcup\{R^{-1}(w') \mid w' \in \psi^{\mathcal{F}}\}$. Thus the latter version can be calculated by traversing all arcs $(w, w') \in \mathcal{I}(R)$ and checking whether $w' \in \psi^{\mathcal{F}}$. If so, then also $w \in (\langle R \rangle \psi)^{\mathcal{F}}$.

[2] In fact, this is not a limited modal logic in any way, merely presented briefly. Varying the relation $R$ allows quite many different types of situations to be expressed by this logic.

**Example 1: Verifying a modal logic formula**

A simple example of calculating the validity set consists of a model $\mathcal{M} = (U, \mathcal{I}, w_0)$ on the set of two atomic propositions $\mathcal{P} = \{\mathtt{p}, \mathtt{q}\}$ and a relation $R$, with $U = \{s_1, s_2, s_3\}$, $\mathcal{I}(\mathtt{p}) = \{s_1, s_2\}$, $\mathcal{I}(\mathtt{q}) = \{s_1, s_3\}$, $\mathcal{I}(R) = \{(s_1, s_2), (s_2, s_3)\}$ and $w_0 = s_1$. Figure 1.1 corresponds to the situation represented by the above sets.

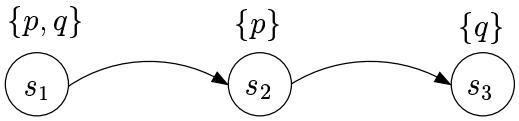


FIGURE 1.1: The model $\mathcal{M}$ with sets of valid propositions for each state and the accessibility relation $\mathcal{I}(R)$

For a multimodal formula $\varphi \triangleq (\mathtt{q} \rightarrow \langle R \rangle \mathtt{p})$, the set $\varphi^{\mathcal{F}} \subseteq U$ is calculated as follows: There are three subformulas: $\mathtt{q}$, $\langle R \rangle \mathtt{p}$ and $\mathtt{p}$. For the atomic propositions, sets $\mathtt{q}^{\mathcal{F}} = \{s_1, s_3\}$ and $\mathtt{p}^{\mathcal{F}} = \{s_1, s_2\}$ are exactly the sets $\mathcal{I}(\mathtt{q})$ and $\mathcal{I}(\mathtt{p})$ respectively. The set $(\langle R \rangle \mathtt{p})^{\mathcal{F}}$ is obtained from these sets using rule (iv) above: all states that are immediate predecessors of the states in $\mathtt{p}^{\mathcal{F}}$ in $\mathcal{I}(R)$ belong to $(\langle R \rangle \mathtt{p})^{\mathcal{F}}$, which gives the set $\{s_1\}$.

Now only the implication and therefore the formula $\varphi$ are left to handle. Applying rule (iii), all states in $U \setminus \mathtt{q}^{\mathcal{F}}$ (namely $s_2$) are combined with all those in $(\langle R \rangle \mathtt{p})^{\mathcal{F}}$ (namely $s_1$) to obtain $\varphi^{\mathcal{F}} = \{s_1, s_2\}$. As $w_0 = s_1 \in \varphi^{\mathcal{F}}$, the conclusion $\mathcal{M} \models \varphi$ is drawn.

In order to avoid calculation of the same sub-formula more than once, it is necessary to store for each sub-formula $\psi$ the set $\psi^{\mathcal{F}}$ into a some data structure. This structure should be able to efficiently handle large state spaces, as for each of the sets, $|\psi^{\mathcal{F}}|$ may have the same order of magnitude as the size of $U$.

What comes to the *computational complexity* of an algorithm implementing the above rules (i) to (iv), it is *linear* both in respect to the size of the model and to the size of the formula, i.e. the number of different sub-formulas. This suggests that handling infinite models would be out of the question, but there have been cases of successful model checking with $\psi^{\mathcal{F}}$ being of infinite size. The sets $\psi^{\mathcal{F}}$ were represented symbolically instead of resorting to explicit storage as discussed above. This however requires for the infinite model to have some symbolic representation, such as Turing machines or Petri nets.

### 1.1.2 Computation Tree Logic (CTL)

Often the point of interest is a set of possible computations, which is the focus of branching time logics such as the Computation Tree Logic, CTL, which allows formulas that argue on the existence of paths fulfilling certain criteria. The validity of formulas is state-based and correctness of a program involves examination of sets

of *maximal paths* in a tree.

As a reminder, a *path* is a nonempty sequence of points (or states or steps), which may be either finite or infinite. Formally, for a path $\sigma = (w_0, w_1, \ldots)$, it must apply $\forall i : 0 \leq i < |\sigma|$, where $|\sigma|$ is the length of the path, $\exists R_i \in \mathcal{R}$ such that $(w_i, w_{i+1}) \in \mathcal{I}(R_i)$.

A path is said to be *maximal* either if whenever for a node $w$ $\exists w', R_i$ such that $(w, w') \in R_i$ and the path does not end in node $w'$, or $\nexists w' : w \prec w'$. Thus maximal paths are either infinite or end in points for which no successors exist.

The relation "$<$" here is the *transitive closure* of the states $w \in U$, i.e. there must exist a path from the former to the latter on the computation path (or a tree) presently under discussion. Another relation is defined for *accessibility in one step*, which is denoted by "$\prec$". Thus "$<$" is actually the transitive closure of the relation "$\prec$".

The syntax of CTL is presented below:

$$\mathbf{CTL} ::= \mathcal{P} \mid \bot \mid (\mathbf{CTL} \rightarrow \mathbf{CTL}) \mid \mathbf{E}(\mathbf{CTL}\,\mathbf{U}^{+}\mathbf{CTL}) \mid \mathbf{A}(\mathbf{CTL}\,\mathbf{U}^{+}\mathbf{CTL}).$$

The interpretation of the *path quantifiers* illustrated by Figure 1.2 is quite straightforward: **E** requires the "until"-property from any *one* of the possible computations paths starting from the root (or the point of evaluation in general), whereas **A** requires it from *all* computation paths of the tree.

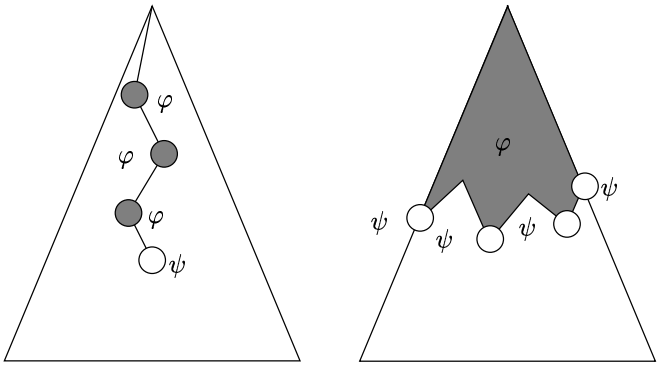


FIGURE 1.2: Interpretations of the path quantifiers **A** and **E**

For CTL, the model $\mathcal{M} = (U, \mathcal{I}, w_0)$ is a *tree* rooted at the initial state $w_0$, and the interpretation of $v$ (the transitive closure of the successor relation $\prec$) is a tree-order: for $w_1 < w_2$ to hold, $w_1$ must be on the path from the root $w_0$ to the node $w_2$ in the tree. For a CTL-formula $\varphi$, $\mathcal{M} \models \varphi$ holds for a given Kripke-model $\mathcal{M} = (U, \mathcal{I}, w_0)$ if and only if the *maximal tree* generated from $\mathcal{M}$ at the *unambiguous* initial state $w_0$ satisfies $\varphi$.

The CTL model checking algorithm is in principle very similar to the algorithm for modal logic presented above as four rules. It proceeds through each state $w \in U$ of the model $\mathcal{M}$, calculating for each state the set of sub-formulas $\psi_i$ of $\varphi$ that are valid at that state. This information is stored together with the state.

If the marking has already been done for sub-formulas $\psi_1$ and $\psi_2$, that is: the set

of states in which the sub-formula is satisfied is known for both of them, then it is possible to determine the points satisfying formulas of the form $\mathbf{E}(\psi_2 \mathbf{U}^+ \psi_1)$ or the form $\mathbf{A}(\psi_2 \mathbf{U}^+ \psi_1)$. The calculation is based on two equivalences, which in turn is based on the definition of the non-reflexive "until"-operation:

**Definition 1.1.** $w_0 \models (\phi \mathbf{U}^+ \psi)$ if and only if $\exists w_1 \in U$ such that $w_0 < w_1$ (i.e. $w_1$ appears later than $w_0$ in the computation sequence) and $w_1 \models \psi$, also $\forall w_2 \in U$ such that $w_0 < w_2 < w1$, applies that $w_2 \models \phi$.

The necessary equivalences are:

$$\mathbf{E}(\psi_2 \mathbf{U}^+ \psi_1) \leftrightarrow \mathbf{EX}(\psi_1 \vee (\psi_2 \wedge \mathbf{E}(\psi_2 \mathbf{U}^+ \psi_1)))$$
$$\mathbf{A}(\psi_2 \mathbf{U}^+ \psi_1) \leftrightarrow \mathbf{AX}(\psi_1 \vee (\psi_2 \wedge \mathbf{A}(\psi_2 \mathbf{U}^+ \psi_1)))$$

In practice this means that for a formula $\varphi = \mathbf{E}(\psi_2 \mathbf{U}^+ \psi_1)$, all states which have at least one successor (due to the non-reflexivity of $\mathbf{U}^+$) labeled with either $\psi_1$ or with both $\varphi$ and $\psi_2$, are labeled with $\varphi$. As this is a recursive definition in respect to $\varphi$'s appearances, it must be repeated until nothing further can be labeled with $\varphi$, i.e. *stabilization* is reached. The successors are examined because of the appearance of the "next" operator $\mathbf{X}$, and only one such successor suffices for the marking to proceed is due to the appearance of the "there is at least one path such that..." operator $\mathbf{E}$.

In the case of $\varphi = \mathbf{A}(\psi_2 \mathbf{U}^+ \psi_1)$, it is necessary for the state to be labeled to have again at least one successor, and due to the operator $\mathbf{A}$ ("for all paths originating from this point applies..."), all of the successors must fulfill the above requirement used for $\mathbf{E}$: each successor is labeled with either $\psi_1$ or with both $\varphi$ and $\psi_2$. This also must be repeated until stabilization.

This procedure is similar to a basic algorithm that marks the transitive closure by traversing a graph. A recursive formulation for the algorithm that marks both cases $\mathbf{E}$ and $\mathbf{A}$ is given below in pseudocode:

```
function eval (formula φ) : set of states
    case φ of
        p: return I(p)
        ⊥: return ∅
        (ψ1 → ψ2): return ((U \ eval(ψ1)) ∪ eval(ψ2))
        E(ψ2 U+ ψ1): E1 := eval(ψ1), E2 := eval(ψ2), E := ∅
            repeat until stabilization // UNTIL E DOES NOT GROW
                E := E ∪ {w | (succ(w) ∩ (E1 ∪ (E2 ∩ E))) ≠ ∅}
            return E
        A(ψ2 U+ ψ1): E1 := eval(ψ1), E2 := eval(ψ2), E := ∅
            repeat until stabilization // UNTIL E DOES NOT GROW
                E := E ∪ {w | ∅ ≠ succ(w) ⊆ (E1 ∪ (E2 ∩ E))}
            return E
```

The following procedure is used to initialize the iteration on the structure of the formula. The function `succ` simply "traverses" the accessibility relation "$\prec$" one step forward.

**procedure** `CTLcheck` (model $\mathcal{M} = (U, \mathcal{I}, w_0)$, formula $\varphi$) {

  **if** $w_0 \in$ `eval`$(\varphi)$
    `print` "$\varphi$ *is* satisfied at $w_0$ of the frame $\mathcal{F} = (U, \mathcal{I})$"
  **else**
    `print` "$\varphi$ *is not* satisfied at $w_0$ of the frame"

**function** `succ` (state $w$) : set of states
  **return** $\{w' \mid (w, w') \in \mathcal{I}(\prec)\}$ // RETURNS STATES REACHABLE BY "$\prec$"

---

### Example 2: Verifying a CTL formula

The execution of the algorithm can be demonstrated by the following over-simplified model shown in Figure 1.3 and some simple formula $\varphi$ to be examined. The purpose of the example if to illustrate all of the rules implemented in the algorithm, not to provide an exciting view on CTL. The formula whose validity is examined is $\varphi \triangleq (\mathbf{E}(\neg \mathtt{q}\, \mathbf{U}^+(\mathtt{p} \wedge \mathtt{r})) \rightarrow \mathbf{A}(\mathtt{p}\, \mathbf{U}^+ \mathtt{q}))$.

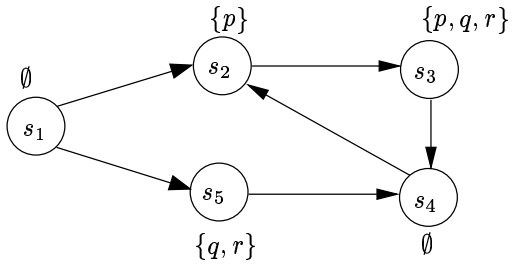


FIGURE 1.3: The model $\mathcal{M}$ with sets of valid propositions for each state and the immediate successor relation "$\prec$"

As the algorithm starts executing, the **case** catches the implication $\psi_1 \rightarrow \psi_2$, where $\psi_1 \triangleq \mathbf{E}(\neg \mathtt{q}\, \mathbf{U}^+(\mathtt{p} \wedge \mathtt{r}))$ and $\psi_2 \triangleq \mathbf{A}(\mathtt{p}\, \mathbf{U}^+ \mathtt{q})$. The first recursive call is thus $E_1 :=$ `eval`$(\psi_1)$.

The **case** of the recursive call `eval`$(\psi_1)$ catches the path quantifier $\mathbf{E}$, where $\mathbf{E}(\psi_4\, \mathbf{U}^+ \psi_3)$ with $\psi_4 \triangleq \neg \mathtt{q}$ and $\psi_3 \triangleq \mathtt{p}$. The next recursive call `eval`$(\psi_3)$ can be seen as a simple call (although $\neg \mathtt{q}$ is not exactly an atomic proposition, but rather of the form $\mathtt{q} \rightarrow \bot$) that returns $\mathcal{I}(\neg \mathtt{q}) = \{s_1, s_2, s_4\} = E_{2'}$. This is obtained by subtracting $\mathcal{I}(\mathtt{q}) = \{s_3, s_5\}$ from $U$, as defined for the implication $\neg \mathtt{q} \rightarrow \bot$. This is substituted as the set $E_{2'}$ in the calculation of the "exists"-subformula.

Using the following equivalences, the result set given below is what the algorithm defines:

$$\begin{array}{lllll} \mathtt{p} \wedge \mathtt{r} & \triangleq & \neg(\neg \mathtt{p} \vee \neg \mathtt{r}) & \triangleq & \neg(\mathtt{p} \rightarrow \neg \mathtt{r}) \\ & \triangleq & \neg(\mathtt{p} \rightarrow (\mathtt{r} \rightarrow \bot)) & \triangleq & (\mathtt{p} \rightarrow (\mathtt{r} \rightarrow \bot)) \rightarrow \bot \end{array}$$

Therefore, $E_{1'} = \mathcal{I}(\mathtt{p} \wedge \mathtt{r}) = \{s_3\}$ is obtained by a similat recursive call to $\psi_3$ and

calculated as below:

$$\begin{aligned} E_{1'} &= (U \setminus (\mathtt{p} \rightarrow (\mathtt{r} \rightarrow \bot)^{\mathcal{F}}) \cup \emptyset \\ &= U \setminus ((U \setminus \mathcal{I}(\mathtt{p})) \cup (\mathtt{r} \rightarrow \bot)^{\mathcal{F}}) \\ &= U \setminus ((U \setminus \mathcal{I}(\mathtt{p})) \cup ((U \setminus \mathcal{I}(\mathtt{r})) \cup \emptyset)) \\ &= U \setminus ((U \setminus \mathcal{I}(\mathtt{p}) \cup (U \setminus \mathcal{I}(\mathtt{r})))) \\ &= U \setminus (\{s_1, s_4, s_5\} \cup \{s_1, s_2, s_4\}) \\ &= U \setminus \{s_1, s_2, s_4, s_5\} = \{s_3\}. \end{aligned}$$

Now the iteration on the set of states satisfying $\psi_1$. Initially, the result set $E_1$ is empty. Each iteration step sets $E_1 \cup \{w \mid \mathtt{succ}(w) \cap (E_{1'} \cup (E_{2'} \cap E))\}$ as the next value of $E$. This requires computing the successor sets with the simple function $\mathtt{succ}(w)$. These are fortunately fixed and small, as are the sets $E_{1'}$ and $E_{2'}$:

$$\begin{aligned} \mathtt{succ}(s_1) &= \{s_2, s_5\} \\ \mathtt{succ}(s_2) &= \{s_3\} \\ \mathtt{succ}(s_3) &= \{s_4\} \\ \mathtt{succ}(s_4) &= \{s_2\} \\ \mathtt{succ}(s_5) &= \{s_4\} \\ E_{1'} &= \{s_3\} \\ E_{2'} &= \{s_1, s_2, s_4\} \end{aligned}$$

The below table shows the iteration process which produces the set $E_1$ returned by this recursive call.

| $E_1$ | $E_{2'} \cap E$ | $E_{1'} \cup (E_{2'} \cap E)$ | $\Rightarrow E$ |
|---|---|---|---|
| $\emptyset$ | $\emptyset$ | $\{s_3\}$ | $\{s_2\}$ |
| $\{s_2\}$ | $\{s_2\}$ | $\{s_2, s_3\}$ | $\{s_1, s_2, s_4\}$ |
| $\{s_1, s_2, s_4\}$ | $\{s_1, s_2, s_4\}$ | $\{s_1, s_2, s_3, s_4\}$ | $\{s_1, s_2, s_3, s_4, s_5\}$ |
| $\{s_1, s_2, s_3, s_4, s_5\}$ | $\{s_1, s_2, s_4\}$ | $\{s_1, s_2, s_3, s_4\}$ | $\{s_1, s_2, s_3, s_4, s_5\}$ |

The process stabilizes quite quickly, due to the small size of the model. Obviously, the set $E_1$ is the set of states where $\psi_1$ is valid, as from all those states there clearly exists one path in which $\neg\mathtt{q}\,\mathbf{U}^+(\mathtt{p} \wedge \mathtt{r})$ applies. The only possible cause of worry is the state $s_5$, but as its predecessor $s_1$ also has another path starting from it, and the other path fulfills the criterion, the subformula $\psi_1$ is valid everywhere in the model.

This result is then returned to the first call of the procedure `eval` as the value of $E_1$. The algorithm continues by calculating $E_2$ with a recursive call: $\mathtt{eval}(\psi_2)$ where $\psi_2 = \mathbf{A}(\psi_6\,\mathbf{U}^+\psi_5)$. Here again the calculations $E_{2''} = \mathtt{eval}(\psi_6) = \mathtt{eval}(\mathtt{p}) = \{s_2, s_3\}$ and $E_{1''} = \mathtt{eval}(\psi_5) = \mathtt{eval}(\mathtt{q}) = \{s_3, s_5\}$ are straightforwardly $\mathcal{I}(\mathtt{p})$ and $\mathcal{I}(\mathtt{q})$ respectively.

The iteration step here is defined by $E_2 \cup \{w \mid \emptyset \neq \mathtt{succ}(w) \subseteq (E_{1''} \cup (E_{2''} \cap E))\}$. The successor sets of the states $s_i$ are as in the above calculation of $E_1$, but now from above $E_{1''} = \{s_3, s_5\}$ and $E_{2''} = \{s_2, s_5\}$.

| $E_2$ | $E_{2''} \cap E$ | $E_{1''} \cup (E_{2''} \cap E)$ | $\Rightarrow E$ |
|---|---|---|---|
| $\emptyset$ | $\emptyset$ | $\{s_3, s_5\}$ | $\{s_2\}$ |
| $\{s_2\}$ | $\{s_2\}$ | $\{s_2, s_3, s_5\}$ | $\{s_1, s_2\}$ |
| $\{s_1, s_2\}$ | $\{s_2\}$ | $\{s_1, s_2, s_3, s_5\}$ | $\{s_1, s_2\}$ |

Again the iteration stabilizes quickly to the set $E_2 = \{s_1, s_2\}$ which is the set of states from which all possible paths always satisfy $\mathtt{p}\,\mathbf{U}^+\mathtt{q}$.

The final step of the algorithm is to evaluate the implication $\psi_1 \rightarrow \psi_2$. This can be done now that the necessary sets $E_1 = U$ and $E_2 = \{s_1, s_2\}$ are known. The set $\varphi^{\mathcal{F}}$ consists of those states in $U$ that are not in $E_1$ combined with those that are in $E_2$. This produces the set $\{s_1, s_2\}$. As $w_0 = s_1 \in \varphi^{\mathcal{F}}$, $\mathcal{M} \models \varphi$ for the initial definition of validity. As $\varphi^{\mathcal{F}} \neq U$, the formula is not universally satisfied.

---

To analyze the complexity of the presented algorithm, note that each of the **repeat** cycles stabilizes after at most $|U|$ iterations, as the Kripke-model $\mathcal{M}$ contains only a finite number of states. The worst-case scenario is that each of those iterations has to search the entire model, traversing the $\mathcal{O}(|U|^2)$ transitions. Thus the computational complexity of the algorithm can be expressed as $\mathcal{O}(|\varphi| \cdot |U|^3)$, where $|\varphi|$ denotes the number of different sub-formulas of $\varphi$.

### Inverse Reachability problem

By reorganizing the search procedure, the complexity may be reduced to being only quadratic to $|U|$ and thus linear to $|\mathcal{M}|$ (as $\mathcal{M}$ contains $\mathcal{O}(|U|^2)$ transitions). The following algorithm marks all the states for which a formula $\mathbf{EF}^+\varphi$ is valid. This is equivalent to the *inverse reachability problem*, formulated as follows: for a graph $G = (E, V)$, for a given set of *target nodes* $T \subseteq E$, mark all such nodes from which there is at least some finite path leading into $T$. The below algorithm assumes that it is decidable in constant time whether there is a vertex $v \in V$ between any two nodes (e.g. a incidence matrix representation of the states and the transitions between them). The algorithm is clearly $\mathcal{O}(|E|)$.

```
function inverseReachability(set of points T) : set of points
    <set> S := ∅ // THE RESULTING SET OF NODES IS INITIALLY EMPTY
    <set> T' = T // A WORKING SET FOR THE ALGORITHM
    while T' ≠ ∅ do
        T' := predecessors(T') \ S
        S := S ∪ T'
    return S;

function predecessors(point w) : set of points
    return {w' | (w', w) ∈ I(≺)} // SET OF POINTS PRECEDING w IN ONE STEP
```

---

### Example 3: Finding "ancestors" by inverse reachability

Figure 1.4 shows a directed graph where the edges mark the successor relation "$\prec$". The target set of the calculation is $T = \{g, j\}$. Obviously, the first iteration of the algorithm has $S = \emptyset$ and $T' = T$. The function `predecessors` returns the set $\{c, e\} \cup \{g, h\}$, the union of the successor sets of the target nodes $g$ and $j$ respectively. Thus the first round of iteration ends with $T' = \{c, e, g, h\} \setminus \emptyset$ and $S = \emptyset \cup T'$.

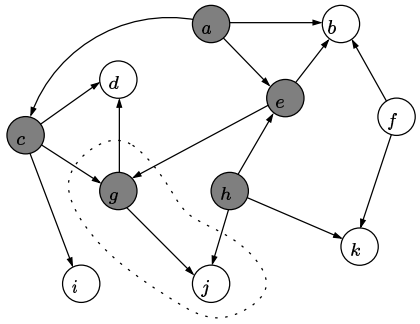


FIGURE 1.4: Inverse reachability calculation for target set $T = \{g, j\}$ with the "inversely reachable" nodes in gray

Thus the second round calculates states reachable in one step from the "expanded" target set $T'$, namely $\{a\} \cup \{e\}$. After this, no more nodes are predecessors of the nodes in $S$, so the calculation finishes as $\texttt{predecessors}(T') \setminus S = \emptyset$, and the resulting set $S = \{a, c, e, g, h\}$.

---

The worst-case bound for the algorithm is based on the observation that every point is included in the set $S$ in the **while** loop at most once. All the set operations can be performed in linear time with respect to the size of the sets, which are all bounded by $|E|$.

The idea of $\mathbf{EF}^+$ can be refined to suit $\mathbf{EU}^+$ as $\mathbf{EF}^+\varphi \triangleq \mathbf{E}(\top\, \mathbf{U}^+\varphi)$. Also, as the $\mathbf{AU}^+$ operator can be expressed with the operators $\mathbf{EU}^+$ and $\mathbf{EG}^+$by the following equivalence, stating that for all paths, $\psi_2$ holds until $\psi_1$ is valid if there does not exists such a path where $\psi_1$ would not be valid until neither is valid, or there exists a path for which $\psi_1$ is never valid:

$$\mathbf{A}\,(\psi_2\,\mathbf{U}^+\psi_1) \leftrightarrow \neg(\mathbf{E}\,(\psi_1\,\mathbf{U}^+(\neg(\psi_1 \wedge \psi_2))) \vee \mathbf{EG}^+\neg\psi_1)$$

So the only missing piece from the more efficient CTL model checking algorithm is a procedure for marking all states for which $\mathbf{EG}^+\varphi$ holds. This can be achieved with the following steps:

(i) exclude all states in which $\varphi$ does not hold

(ii) denote the remaining set with $V_\varphi \subseteq U$

(iii) mark all states $w \in U$ from which a state $w' \in V_\varphi$ without any successors at all that can be reached[3] traversing only through states of $V_\varphi$

(iv) find the maximal strongly connected components (SCCs) of $V_\varphi$

(v) mark all states that are members of SCCs

(vi) mark all states $w \in U$ from which a non-trivial SCC of $V_\varphi$ can be reached by a path in $V_\varphi$

[3] A state in which $\varphi$ does not hold may be included here, as long as the validity of $\varphi$ begins from an immediate successor and continues for a maximal path.

A *strongly connected component* of a graph $G = (E, V)$ is a set of nodes $C \subseteq E$ such that $\forall c_1, c_2 \in C$ exists a path connecting $c_1$ and $c_2$. A non-trivial SCC is generally viewed as a set $C$ as defined above but with more than one node. Calculating these is a basic graph theoretic problem with efficient algorithms.

All these operations can be done in $\mathcal{O}(|\mathcal{M}|)$ time. Thus the CTL model checking can be done with the quite naïve algorithm presented above in $\mathcal{O}(|\varphi| \cdot |\mathcal{M}|)$.

---

**Example 4: Calculating $\psi^{\mathcal{F}}$ for $\psi = \mathbf{EG}^{+}\varphi$ in a given model $\mathcal{M}$**

Figure 1.5 shows a model $M$ with a set of states $U = \{a, \ldots, m\}$ and the transition relation $\mathbf{X}$. The set $V_\varphi$ defined by step (i) consists of all white nodes, i.e. states in which $\varphi$ has been marked to be valid. The set $\psi^{\mathcal{F}}$ is initially set to $\{b, e, j\}$ by step (ii), as the nodes $a, f \in V_\varphi$ have no successors in the graph and $b$, $e$ and $j$ have a path to $a$ and $f$ respectively, traversing only through nodes in $V_\varphi$. Node $g$ has no successors in $V_\varphi$, but that does not help here, due to the definition of the $\mathbf{G}^{+}$ operator.

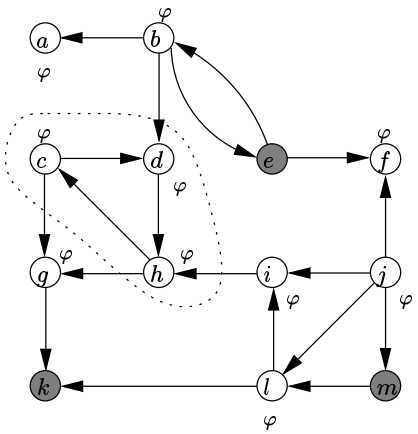


FIGURE 1.5: Calculating the set of states satisfying $\mathbf{EG}^{+}\varphi$ from a known marking of $\varphi$

Next, $\psi^{\mathcal{F}}$ is by steps (iv) and (v) expanded to include the states in nontrivial SCCs of $V_\varphi$. In the graph as a whole there are two such components: $\{b, e\}$ and $\{c, d, h\}$. In both of these sets, all nodes are reachable from one another by a path. The former set is not however a SCC of $V_\varphi$, as $e \not\models \varphi$. Thus now $\psi^{\mathcal{F}} = \{b, c, d, e, h, j\}$ by step (v).

The last step includes in $\psi^{\mathcal{F}}$ those states in $U$ that can reach a nontrivial SCC by a path in $V_\varphi$. This gives three more states, $i$, $l$ and $m$. The final result is $\psi^{\mathcal{F}} = \{b, c, d, h, i, j, l, m\}$.

---

### 1.1.3 Fairness

In some model checking tools, such as SMV (Symbolic Model Verifier), it is possible to provide a set of additional *constraints* together with the Kripke-model $\mathcal{M}$. During model checking. only those states in which all of the provided constraints are valid are examined when verifying the given formula $\varphi$.

The most commonly occurring need for constraints is due to *fairness* assumptions in the system specification. It is often desirable to restrict the evaluation of the path quantifiers **A** and **E** only to those paths that are in some sense fair.

Constraints of the form $\mathbf{F}^+\psi$, with $\psi$ consisting of boolean connectives and atomic propositions, are called *simple fairness constraints*. For example, if only infinite paths are included in the model verification process, a constraint can be written to require that for every state on the computation path, there must be another future state: $\mathbf{F}^+\top$.

Constraints of the form $(\mathbf{G}^+\mathbf{F}^+\psi_1 \rightarrow \mathbf{G}^+\mathbf{F}^+\psi_2)$ are called *Streett fairness constraints*. Such constraints restrict the model checking to *strongly fair* computations. A common example is that if a process requests a resource infinitely often, it must also be granted that resource infinitely often. *Weak fairness* requires that if a process keeps requesting without interruptions infinitely, it will eventually be granted the resource in question [BBF+01]. Note that strong fairness always implies weak fairness but not vice versa.

The above fairness constraints are clearly not expressible in CTL: there is no path quantification preceding each of temporal operator. They are formulas of LTL, Linear Temporal Logic. In verification systems, constraints are mostly specified in the same language then the formula $\varphi$, but also approaches using different languages are used, such as verifying a CTL formula with LTL constraints.

It is in general not possible to express fairness properties with the normal CTL grammar defined in Section 1.1.2; some extensions are necessary for using CTL for that purpose [BBF+01]. The problem is evident in the written interpretation of fairness properties: they effectively state that *a certain condition will apply infinitely often* in the possible computations, which implies semantics different from all the CTL operators.

Including LTL formulas as fairness constraints in the algorithm of Section 1.1.2 can be achieved by building a tableau for the LTL assumptions and calculating a product of $\mathcal{M}$ and the tableau. Model checking of the CTL formula $\varphi$ is then done on the product instead of the model directly. A complexity increase follows, but the magnitude depends on the type of the LTL formulas.

## 1.2 Local Model Checking for Linear Time Logics

For linear time logics, validity $\mathcal{M} \models \varphi$ does not involve the *maximal tree* of computations as for CTL, but instead requires that the set of *maximal sequences* generated from the Kripke-model $\mathcal{M} = (U, \mathcal{I}, w_0)$ starting at the initial state $w_0$ satisfy the given formula $\varphi$. This is called *sequence validity*.

An alternative approach to generating all sequences is to examine whether $\neg\varphi$ is valid for even one sequence, i.e. *natural model* generated by $\mathcal{M}$. A natural model is one in which the transition relation is isomorphic to the set of natural numbers, that is: each state has an immediate successor and there cannot "exist" anything between those states. No simple marking procedure of the states does the trick here as it is necessary to examine the entire *set* of natural models, i.e. all of the possible maximal paths generated by the model.

CTL-formulas can be viewed to be valid *in a state*, but linear time logics such as LTL consider validity on complete and most often *infinite paths*. For example, a fairness constraint $\mathbf{F}^{+}\psi$ may be valid for some of the sequences generated from $\mathcal{M}$, but not all. In CTL, the sub-formulas of the verified formula are either valid in a state or not and the state space is limited. Thus model checking is easier for the state-based approach than that of all possible sequences.

### 1.2.1 Modal Logic with a Single Accessibility Relation

The discussion of model checking for linear time logic begins with a simplification: a modal logic where there is only one accessibility relation $R$. The simplified grammar is given below:

$$\mathbf{ML} ::= \mathcal{P} \mid \bot \mid (\mathbf{ML} \rightarrow \mathbf{ML}) \mid \langle R \rangle \mathbf{ML}.$$

For given $\mathcal{M}$ and $\varphi$, the task is to determine whether a maximal sequence can be generated from $\mathcal{M}$ starting at $w_0$ such that $\varphi$ is satisfied at $w_0$. This can be done by constructing the *propositionally maximal consistent sets* for $\varphi$ and then performing a depth-first search on the product of $S$ and $U$, which is the model's set of states.

A set $m$ of subformulas of a formula $\varphi$ is said to be *maximal* if the following holds for the elements $\psi \in SF(\varphi)$: for each element, either the element or its negation is included in the set $m$, i.e. $\{\psi \mid \psi \in m\} \cup \{\neg\psi \mid \psi \notin m\}$. Such a set is denoted as $m \in SF(\varphi)$ .

The definition of propositional consistency follows: a set $m \subseteq SF(\varphi)$ is said to be *propositionally consistent* if both of the requirements below hold:

(i) $\bot \notin m$

(ii) if $(\psi_1 \rightarrow \psi_2) \in SF(\varphi)$, then the it must hold that $(\psi_1 \rightarrow \psi_2) \in m$ if and only if either $\psi_1 \notin m$ or $\psi_2 \in m$

The rule (ii) essentially states the validity condition of implication. These two rules can be expanded to handle also other boolean connectives, as it holds that

- $(a \rightarrow \bot) \triangleq \neg a$
- $(a \rightarrow b) \triangleq (\neg a \vee b)$
- $\neg(a \vee b) \triangleq (\neg a \wedge \neg b)$

Using these equivalences with the above rules for propositional consistency, it is clear that the following applies if $m \subseteq SF(\varphi)$ is propositionally consistent:

(a) if $\neg \in SF(\varphi)$, then $\neg\psi \in m \leftrightarrow \psi \notin m$

(b) if $(\psi_1 \vee \psi_2) \in SF(\varphi)$, then $(\psi_1 \vee \psi_2) \in m$ if and only if either $\psi_1 \in m$ or $\psi_2 \in m$

(c) if $(\psi_1 \wedge \psi_2) \in SF(\varphi)$, then $(\psi_1 \wedge \psi_2) \in m$ if and only if both $\psi_1 \in m$ and $\psi_2 \in m$

The rules again correspond to the definitions of validity for the boolean connectives used in each one.

---

**Example 5: Finding propositionally consistent sets**

For $\varphi = (\mathtt{q} \rightarrow \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q}))$, the set $SF(\varphi)$ includes all formulas that can be obtained from $\varphi$ by decomposing it with respect to the grammar of modal logic: $SF(\varphi) = \{\mathtt{q}, \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q}), (\mathtt{p} \rightarrow \mathtt{q}), \mathtt{p}, \varphi\}$. Note that the set of subformulas is also thought to contain the formula itself. The subsets of this set are

$$
\begin{aligned}
2^{SF(\varphi)} \;=\; \{\; & \emptyset, \{\mathtt{p}\}, \{\mathtt{q}\}, \{(\mathtt{p} \rightarrow \mathtt{q})\}, \{\langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})\}, \{\mathtt{p}, \mathtt{q}\}, \{\mathtt{p}, (\mathtt{p} \rightarrow \mathtt{q})\}, \\
& \{\mathtt{p}, \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})\}, \{\mathtt{q}, (\mathtt{p} \rightarrow \mathtt{q})\}, \{\mathtt{q}, \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})\}, \\
& \{(\mathtt{p} \rightarrow \mathtt{q}), \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})\}, \{\mathtt{p}, \mathtt{q}, (\mathtt{p} \rightarrow \mathtt{q})\}, \{\mathtt{p}, \mathtt{q}, \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})\}, \\
& \{\mathtt{p}, (\mathtt{p} \rightarrow \mathtt{q}), \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})\}, \{\mathtt{q}, (\mathtt{p} \rightarrow \mathtt{q}), \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})\}, SF(\varphi) \quad \}
\end{aligned}
$$

A subset is propositionally consistent if it does not contain any "conflicting" formulas, such as $\mathtt{p}$ and $(\mathtt{p} \rightarrow \mathtt{q})$ without the presence of $\mathtt{q}$. In evaluating the propositional consistency, modal formulas are treated as atomic propositions. Thus $\langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})$ will be denoted as $\mathtt{r}$ below. Each of these "atomic" propositions must have a truth value because of the maximality requirement in each of the maximal propositionally consistent sets, and each implication that follows from these truth values must also be included for propositional consistency.

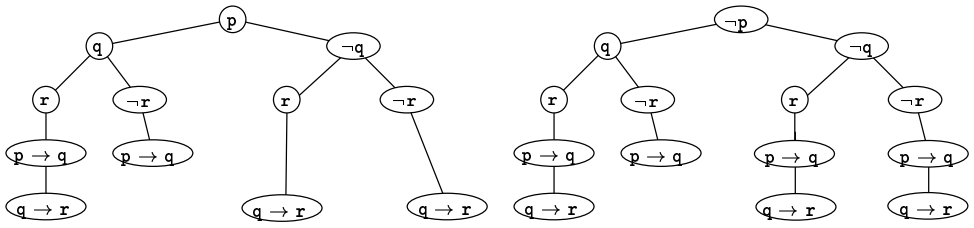


FIGURE 1.6: The tree generating maximal propositionally consistent subsets of $SF(\varphi)$ (the negations are explicitly shown only for atomic propositions)

From Figure 1.6, the set of maximal propositionally consistent sets can be interpreted as in the table below. Note here that $(\mathtt{q} \rightarrow \mathtt{r}) \triangleq \varphi$

$$
\begin{aligned}
m \;\in\; \{\; & \{\mathtt{p}, \mathtt{q}, \mathtt{r}, (\mathtt{p} \rightarrow \mathtt{q}), (\mathtt{q} \rightarrow \mathtt{r})\}, && \{\mathtt{p}, \mathtt{q}, \neg\mathtt{r}, (\mathtt{p} \rightarrow \mathtt{q})\} \\
& \{\mathtt{p}, \neg\mathtt{q}, \mathtt{r}, (\mathtt{q} \rightarrow \mathtt{r})\} && \{\mathtt{p}, \neg\mathtt{q}, \neg\mathtt{r}, (\mathtt{q} \rightarrow \mathtt{r})\} \\
& \{\neg\mathtt{p}, \mathtt{q}, \mathtt{r}, (\mathtt{p} \rightarrow \mathtt{q}), (\mathtt{q} \rightarrow \mathtt{r})\}, && \{\neg\mathtt{p}, \mathtt{q}, \neg\mathtt{r}, (\mathtt{p} \rightarrow \mathtt{q})\} \\
& \{\neg\mathtt{p}, \neg\mathtt{q}, \mathtt{r}, (\mathtt{p} \rightarrow \mathtt{q}), (\mathtt{q} \rightarrow \mathtt{r})\}, && \{\neg\mathtt{p}, \neg\mathtt{q}, \neg\mathtt{r}, (\mathtt{p} \rightarrow \mathtt{q}), (\mathtt{q} \rightarrow \mathtt{r})\} \quad \}
\end{aligned}
$$

---

To define the model checking procedure, some new terminology must be introduced. The model checking algorithm will attempt to find a counterexample for $\mathcal{M} \models \psi$, and thereby must construct a structure in which to look for the counterexample. This is done by using the negation of the specification formula, $\varphi \triangleq \neg\psi$.

An *atom* is any pair $(w, m)$, where $w \in U$ is a state in the model $\mathcal{M}$ and $m \subseteq SF(\varphi)$, where $SF(\varphi)$ is a propositionally consistent set of sub-formulas of $\varphi$. Note that here $\varphi$ must be the negation of the formula $\psi$ actually being verified, as the procedure searches for a counterexample by attempting to satisfy the negation.

An atom is said to be *admissible* if $w$ and $m$ *agree on the interpretation of propositions*, i.e. if for an atomic proposition $\mathtt{p}$, $\mathtt{p} \in SF(\varphi)$, then $\mathtt{p} \in m \leftrightarrow w \in \mathcal{I}(p)$. If the state of the atom is the initial state $w_0$ of the model $\mathcal{M}$, the atom $\alpha = (w_0, m_0)$ is said to be an *initial atom*, where $\varphi \in m_0$. A relation $X_R$ is defined between admissible atoms in the following manner:

**Definition 1.2.** $X_R((w, m), (w', m'))$ holds for a formula $\varphi$ that is in *positive normal form* (that is: all negations preceding atomic propositions) if and only if the following conditions (i) to (iv) apply:

(i) $(w, w') \in \mathcal{I}(R)$

(ii) if $\langle R\rangle\psi \in SF(\varphi)$ and $\psi \in m'$, then $\langle R\rangle\psi \in m$

(iii) if $\langle R\rangle\psi \in m$, then $\psi \in m'$

(iv) some $\langle R\rangle\psi \in m$

According to the definition of $X_R$, the steps of the generated sequence must be such that are proper in the Kripke-model $\mathcal{M}$ (i), the $\langle R\rangle$ operator is defined to be in a sense a "next" operator similar to the temporal operator $\mathbf{X}$ and is defined consistently both "forward" (ii) and "backward" (iii) in the sequence. Also, the generated sequence may be finite if no $\langle R\rangle\psi$ is "contained" in an atom (iv).

The requirement of positive normal form is due to the lack of rules for handling formulas of the form $[R]\,\varphi$, the dual of $\langle R\rangle$ ($\langle R\rangle\psi \triangleq \neg[R]\neg\psi$). They could be included in the definition by the following refinements: in rule (iv), also the presence of some $\neg\langle R\rangle\psi$, or equivalently $[R]\psi' \in m$, will suffice. An *additional rule* (iii)' is required to take possible negative future obligations into account: $\neg\langle R\rangle\psi \in m \Leftrightarrow \langle R\rangle\neg\psi \in m$. The rules (ii) and (iii) could also be combined for the sake of clarity: they essentially state that $\langle R\rangle\psi \in m \Leftrightarrow \psi \in m'$, also ensuring for the additional rule that $\langle R\rangle\neg\psi \in m \Leftrightarrow \psi \notin m'$.

A *forest* of atoms may now be constructed with the following recursive procedure: the *root nodes* of the forest are the initial atoms, and any node $\alpha$ has all of the atoms $\alpha'$ that are reachable from it as its children, that is: all atoms $\alpha'$ for which $X_R(\alpha, \alpha')$ are child nodes of $\alpha$. Each of the branches in this forest is of finite length if backward arcs are used in cyclic situations. This is because for a finite Kripke model there can be only a finite number of atoms corresponding to the model.

A leaf in the forest, i.e. a node $\alpha = (w, m)$ with no children, is called *open* if it has no formulas that start with a modal operator (namely the operator $\langle R\rangle$) in its $m$, and *closed* if it does. Note that if the requirement of positive normal form is avoided with the additions made to the forest construction rule set, a leaf $\alpha = (w, m)$ is also

*open* when it has a successor $\alpha' = (w', m')$ for which no $[R]\psi \in m'$.

A path in the forest that is either infinite or ends in an open leaf is called an *accepting path*. Any such path corresponds to a sequence generated from the Kripke-model that satisfies the formula $\varphi$, and is thereby a counterexample to the validity of the specification formula $\psi$.

---

**Example 6: A forest of atoms for $\varphi = (\mathtt{q} \rightarrow \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q}))$ and a model $\mathcal{M}$**

To verify the negation of the previously discussed formula $\neg\varphi = \neg(\mathtt{q} \rightarrow \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q}))$ in a model $\mathcal{M} = (U, \mathcal{I}, w_0)$, the forest of atoms need to be constructed for the negation of the specification, which is simply $\varphi$. The model $\mathcal{M}$ in shown in Figure 1.7 with the universe $U = \{s_1, s_2\}$, initial state $w_0 = s_1 \in U$, and valuations for the propositions as shown. The model was intentionally chosen to be very small so it would produce a small forest to act as an example of the construction procedure.

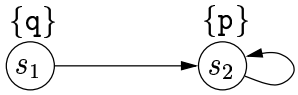


FIGURE 1.7: A simple non-branching Kripke-model $\mathcal{M} = (U, \mathcal{I}, w_0)$ with $\mathcal{I}(\mathtt{q}) = \{s_1\}$ and $\mathcal{I}(\mathtt{p}) = \{s_2\}$

Now the possible atoms $\alpha = (w, m)$ for the forest can be constructed by combining a $w \in U$ with a $m$ that is a maximal propositionally consistent set, which were listed for $\varphi$ in the previous example. Of these $2 \times 8$ atoms only those that are admissible suffice for the forest construction. Thus $s_1$ may be combined with only those sets for which $\mathtt{q} \in m$ and $\neg\mathtt{p} \in m$, as only $\mathtt{q}$ is true in $s_1$. Similarly, only $\mathtt{q}$ but not $\mathtt{p}$ may appear in the set combined with $s_2$ to produce an admissible atom. $\mathtt{r} \triangleq \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})$ may take both of the truth values for all states in $U$. The below table lists the admissible atoms of the forest, with the denotation $\mathtt{r}$[4] expanded back to $\langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})$ for positive occurrences (the negative occurrences are excluded). Therefore also $(\mathtt{q} \rightarrow \mathtt{r})$ again marked with $\neg\varphi$.

| $\alpha$ | $w$ | $m \in SF(\varphi)$ |
|---|---|---|
| $\alpha_1$ | $s_1$ | $\{\mathtt{p}, \neg\mathtt{q}, \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q}), \varphi\}$ |
| $\alpha_2$ | $s_1$ | $\{\mathtt{p}, \neg\mathtt{q}, \varphi\}$ |
| $\alpha_3$ | $s_2$ | $\{\neg\mathtt{p}, \mathtt{q}, \langle R \rangle (\mathtt{p} \rightarrow \mathtt{q}), (\mathtt{p} \rightarrow \mathtt{q}), \varphi\}$ |
| $\alpha_4$ | $s_2$ | $\{\neg\mathtt{p}, \mathtt{q}, (\mathtt{p} \rightarrow \mathtt{q})\}$ |

Form these atoms, the initial ones are those for which $w = w_0 = s_1$ and $\varphi \in m$. Both atoms $\alpha_1$ and $\alpha_2$ fulfill these requirements. Thus the forest will consist of two trees with $\alpha_1$ and $\alpha_2$ as roots.

Now the relation $X_R(\alpha_i, \alpha_j)$ needs to be defined for the atoms. The first requirement deals with the model only: two atoms $\alpha_i = (w_i, m_i)$ and $\alpha_j = (w_j, m_j)$ must be connected in the model, i.e. the states $w_i$ and $w_j$ are connected by $R$ in the model $\mathcal{M}$. The model of Figure 1.7 shows the relation $\mathcal{I}(R) = \{(s_1, s_2), (s_2, s_2)\}$. Thus the

[4] In the construction of the maximal propositionally consistent sets, model formulas were treated as atomic propositions.

following arcs are possible by rule (i) in the relation definition:

$$X_R \triangleq \{(\alpha_1, \alpha_3), (\alpha_1, \alpha_4), (\alpha_2, \alpha_3), (\alpha_2, \alpha_4), (\alpha_3, \alpha_4), (\alpha_4, \alpha_3)\}.$$

Rule (ii) of the relation construction is related to the formula $\varphi$ and requires that as $\langle R \rangle (\mathtt{p} \rightarrow \mathtt{q}) \in SF(\varphi)$, if $(\mathtt{p} \rightarrow \mathtt{q})$ holds at $\alpha_j$, the statement $\langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})$ must hold in $\alpha_i$ in order to have $(\alpha_i, \alpha_j) \in X_R$.

The implication $(\mathtt{p} \rightarrow \mathtt{q})$ holds in atoms $\alpha_3$ and $\alpha_4$, whereas $\langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})$ holds in atoms $\alpha_1$ and $\alpha_3$. Rule (iii) states that if $\langle R \rangle (\mathtt{p} \rightarrow \mathtt{q})$ holds in $\alpha_i$, the implication $(\mathtt{p} \rightarrow \mathtt{q})$ must hold in $\alpha_j$.

This reduces the relation $X_R$ to only two arcs, $\{(\alpha_1, \alpha_3), (\alpha_3, \alpha_4)\}$. Here the atom $\alpha_4$ is an open leaf, as it does not require any modal formula[5] to be true. As there is a path (although only traversing one arc) from the initial atom $\alpha_1$ to an open leaf $\alpha_4$ (that is: an accepting path), a counterexample has been found to the validity of $\varphi$ and thus the algorithm correctly concludes $\mathcal{M} \not\models \varphi$, as obviously $(\mathtt{p} \rightarrow \mathtt{q})$ does not hold in $s_2$.

---

The search for accepting paths can be arranged in a depth-first manner from the initial atoms to their successors, using backtracking techniques. Some storage must be available for the nodes encountered during the search in order to detect cycles. In practice, the choice of the storage facility seems to favor hashtables, where the hash function does not need to take all of the components of the node into account. The reason for this is that the valuation for the atomic proposition determined by the $w$-component already determines the values of all the boolean formulas in $m$. Thus from $m$, storing only the value of those formulas that contain the $\langle R \rangle$-operator suffices.

During the search it is also necessary to identify closed atoms and mark these during backtracking — if a marked atom appears, the search procedure will know not to recurse there again. Also other improvements based on the corresponding decision procedure are available. These are direct consequences of the improvements available for the basic decision procedure of modal logic.

The search can normally be terminated at the discovery of the first accepting path, as only one is needed to make the counterexample. With an appropriate ordering on the child nodes of the current node, finding a counterexample can be quite quick (thus it might be useful to experiment with a couple of different ordering if the optimal one is not known), even though the worst-case scenario naturally includes traversing the entire forest. This occurs whenever $\varphi$ is valid: there is no counterexample at all to be found.

### 1.2.2 Linear Temporal Logic (LTL)

LTL is a temporal logic on *natural models* , that is: models where every step on a computation sequence has an unambiguous successor and nothing in between. The

[5] Here, a modal formula contains also at least one modal operation in addition to possible boolean connectives.

general syntax of LTL is

$$\mathbf{LTL} ::= \mathcal{P} \mid \bot \mid (\mathbf{LTL} \rightarrow \mathbf{LTL}) \mid (\mathbf{LTL}\, \mathbf{U}^{+} \mathbf{LTL}) \mid (\mathbf{LTL}\, \mathbf{U}^{-} \mathbf{LTL}).$$

The operator $\mathbf{U}^{+}$ is a "future until" operator demanding that the preceding formula will hold until the latter becomes valid in some future state. The operator $\mathbf{U}^{-}$ is a "since" operator, referring to past time with the following semantics: $w_0 \models (\varphi\, \mathbf{U}^{-} \psi)$ if and only if $\exists w_1 \in U$ such that $w_1 < w_0 \wedge w_1 \models \psi$ when for all intermediate states $w_2 \in U$ where $w_1 < w_2$ and $w_2 < w_0$, $w_2 \models \varphi$.

It is worthwhile to note that each LTL-formula can be translated to an equivalent first order formula. Another interesting feature in LTL is its *separation property*: a temporal formula is said to be

- *pure future* if it has the form $(\varphi\, \mathbf{U}^{+} \psi)$, and neither $\varphi$ nor $\psi$ contains the operator $\mathbf{U}^{-}$,
- *pure present* if it does not contain any occurrence of either of the operators $\mathbf{U}^{+}$ and $\mathbf{U}^{-}$, and
- *pure past* if it has the form $(\varphi\, \mathbf{U}^{-} \psi)$, and neither $\varphi$ nor $\psi$ contains the operator $\mathbf{U}^{+}$.

For each LTL-formula, there is an equivalent *separated formula*, which is a formula consisting of boolean combinations of past, present and future formulas, without any nesting of the "until" operators.

Also, every LTL-formula has an equivalent formula, which does not use any past temporal operators. Thus for the sake of simplicity, only the *future fragment* of LTL is attended to in this discussion — the *past-temporal operators* such as the "since" operator $\mathbf{U}^{-}$ are not addressed further in this text.

To check a whether a formula is valid for all possible sequences, it again suffices to examine whether the negation is valid even for just one sequence, i.e. search for a counterexample to the validity of the original specification formula. It can be shown that examining the validity of a LTL formula $\varphi$ in a model $\mathcal{M}$ is in a sense equivalent[6] to finding a directed Hamiltonian path[7] in a graph $G$. The below algorithm is presented in full detail by Clarke, Grumberg, and Peled in [CGP99].

The grammar of LTL is rewritten for the forthcoming model checking procedure; each formula can be thought to consist of a conjunction of eventualities[8] of atomic propositions $\mathbf{F}^{*}\mathtt{p}$, and global implications $\mathbf{G}^{*}(\mathtt{p} \rightarrow \mathbf{XG}^{*}\neg\mathtt{p})$. This is in positive conjunctive normal form:

$$\mathbf{F}^{*}\mathtt{p}_1 \wedge \ldots \wedge \mathbf{F}^{*}\mathtt{p}_n \wedge \mathbf{G}^{*}(\mathtt{p}_1 \rightarrow \mathbf{XG}^{*}\neg\mathtt{p}_1) \wedge \ldots \mathbf{G}^{*}(\mathtt{p}_n \rightarrow \mathbf{XG}^{*}\neg\mathtt{p}_n).$$

For an arbitrary directed graph $G = (V, E)$, acting as the foundation of the model $\mathcal{M}$, the number of nodes $v \in V$ is directly bound to the number of atomic propositions

[6] There exists a reduction from HAMILTON PATH to the model checking problem.

[7] A Hamiltonian path is a sequence of edges that forms a path visiting each vertex of the graph exactly once.

[8] An eventuality is any formula that requires something to be valid in a future state.

that appear in the formula $\varphi$ of the above form that is to be checked for validity in a model: $|V| = n$. There is a node $v_i \in V$ for each $\mathtt{p}_i \in SF(\varphi)$, where $SF(\varphi)$ is again the set of subformulas of $\varphi$.

Two additional nodes, a source $s \in V$ for which $\forall v_i \in V : (s, v_i) \in E \wedge (v_i, s) \notin E$, and a sink $t \in V$ for which $\forall v_i \in V : (v_i, t) \in E \wedge (t, v_i) \notin E$, are attached to $V$. The source node $s$ acts as the initial node, i.e. the starting point of the Hamiltonian path, from which all of the nodes $v_i \in V$ are reached by an edge in $E$. Thus any node $v_i \in V$ may be the first node to visit on the path. Also, $t$ must be the end point of the Hamiltonian path, as it certainly cannot appear in the middle of the path, as there are no outgoing vertices from it. To make the transition relation total (allowing infinite computation sequences), a self-loop $(t, t)$ is needed for the sink node.

Now define $\mathcal{M} = (U, \mathcal{I}, w_0)$ as follows:

- the set of states $U = V \cup \{s, t\}$, where $s, t \notin V$; initial state $w_0 = s$
- $\mathcal{I}(R) = E \cup \{(s, v_i) \mid v_i \in V\} \cup \{(v_i, t) \mid v_i \in V\} \cup \{(t, t)\}$
- $\mathcal{I}(\mathtt{p})$ for all atomic propositions $\mathtt{p} \in \mathcal{P}$ is such that
  - $v_i \in \mathcal{I}(\mathtt{p}_i)$ if and only if $1 \leq i \leq |V|$
  - $v_i \notin \mathcal{I}(\mathtt{p}_j)$ if and only if $1 \leq i, j \leq |V|$, $i \neq j$
  - $\forall 1 \leq i \leq |V| : s \notin \mathcal{I}(\mathtt{p}_i)$
  - $\forall 1 \leq i \leq |V| : t \notin \mathcal{I}(\mathtt{p}_i)$

Now $\varphi$ is valid (i.e. satisfiable) on some path clearly if and only if there exists an infinite computation sequences for $\mathcal{M}$ starting at $s$ and visiting all states $v_i$ exactly once, ending in the self-loop of $t$. Note that $\mathcal{M}$ is now a linear and serial model.

Here the size of the model is bound by the size of the formula. For cases where the formula is much smaller than the model, another approach is necessary. An algorithm that relies on explicit tableau construction by Lichtenstein and Pnueli is summarized by Clarke, Grumberg and Peled in [CGP99]. The algorithm decomposes the formula into a tableau, from which a model can be obtained if and only if the formula is satisfiable.

In this algorithm, the set of temporal operators needed is $\mathbf{X}$ and $\mathbf{U}^*$ Again, the others may be defined by the means of these two: $\mathbf{F}^*\psi \triangleq (\top \mathbf{U}^* \psi)$ and $\mathbf{G}^*\psi \triangleq \neg\mathbf{F}^*\neg\psi$.

The *closure* of a formula $\varphi$ is denoted by $CL(\varphi)$ and defined as the minimal set of formulas containing $\varphi$ itself where the following applies for $\psi_i \in SF(\varphi)$:

(i) $\neg\psi \in CL(\varphi)$ if and only if $\psi \in CL(\varphi)$

(ii) if $\psi_1 \vee \psi_2 \in CL(\varphi)$, then both $\psi_1, \psi_2 \in CL(\varphi)$

(iii) if $\mathbf{X}\psi \in CL(\varphi)$, then $\psi \in CL(\varphi)$

(iv) if $\neg\mathbf{X}\psi \in CL(\varphi)$, then $\mathbf{X}\neg\psi \in CL(\varphi)$

(v) if $(\psi_2 \,\mathbf{U}^* \psi_1) \in CL(\varphi)$, then all $\psi_1, \psi_2, \mathbf{X}(\psi_2 \,\mathbf{U}^* \psi_1) \in CL(\varphi)$

The size of the resulting closure set can be shown to be $\mathcal{O}(|\varphi|)$. Similarly to the construction for modal logic in Section 1.2.1, an atom is defined for a model $\mathcal{M}$ and a formula $\varphi$ as a pair $\alpha_i = (w_i, c_i)$, where $w_i \in U$ and $c_i \in CL(\varphi) \cup \mathcal{P}$ is a maximal propositionally consistent set of formulas that agrees with the valuations of the atomic propositions at $w_i$. This can be formalized as follows:

- $\forall \mathtt{p} \in \mathcal{P} : \mathtt{p} \in c_i$ if and only if $w_i \in \mathcal{I}(\mathtt{p})$
- $\forall \psi \in CL(\varphi) : \psi \in c_i$ if and only if $\neg\psi \notin c_i$
- $\forall (\psi_1 \vee \psi_2) \in CL(\varphi) : (\psi_1 \vee \psi_2) \in c_i$ if and only if either $\psi_1 \in c_i$ or $\psi_2 \in c_i$
- $\forall \neg\mathbf{X}\psi \in CL(\varphi) : \neg\mathbf{X}\psi \in c_i$ if and only if $\mathbf{X}\neg\psi \in c_i$
- $\forall (\psi_2 \,\mathbf{U}^*\psi_1) \in CL(\varphi)$, $(\psi_2 \,\mathbf{U}^*\psi_1) \in c_i$ if and only if either $\psi_1 \in c_i$ or both $\psi_2 \in c_i$ and $\mathbf{X}(\psi_2 \,\mathbf{U}^*\psi_1) \in c_i$

Again, a graph $G = (V, E)$ is constructed using the atoms as nodes,$\alpha \in V$, very similarly to the atom forest of the previous section. The transition relation between two atoms $\alpha = (w, c)$ and $\alpha' = (w', c')Q$, , i.e. the edges $(\alpha, \alpha') \in E$ of the graph, is defined as follows:

- $(\alpha, \alpha') \in E$ if and only if $(w, w') \in \mathcal{I}(R)$ and
- $\forall \mathbf{X}\psi \in CL(\varphi) : \mathbf{X}\psi \in c$ if and only if $\psi \in c'$

An *eventuality sequence* is defined as an infinite path $\pi$ in a graph $G$ such that if for some atom $\alpha = (w, c)$ on the path $\pi$, $(\psi_2 \,\mathbf{U}^*\psi_1) \in c$, then there must exist another atom $\alpha' = (w', c')$ such that $\psi_1 \in c'$. In other words, all "until" formulas on the path $\pi$ must be possible to satisfy as $\psi_1$ will eventually be valid.

This enables the formulation of the following lemma from [CGP99], remembering that initial atoms are such atoms where the state $w$ in an initial state (if unambiguous, $w_0$) and the verified formula $\varphi$ is in $c$. Also note that the formula verified by this algorithm is the negation of the original specification formula, as the algorithm searches for some counterexample. As one computation sequence validating the negation of the specification formula is a sufficient counterexample to the validity of the specification, the path quantifier $\mathbf{E}$ is used in the lemma. However strange the notation may seem when used in an LTL formula, the symbol $\mathbf{E}$ is used to denote initial validity and correspondingly $\mathbf{A}$ for universal validity.

**Lemma 1.1.** $\mathcal{M}, w_0 \models \mathbf{E}\,\varphi$ if and only if there exists an eventuality sequence starting at an initial atom $\alpha = (w_0, c_0)$.

The proof of this, as well as the proof of the following lemma, is given in [CGP99].

The *strongly connected components* of the generated graph need to be calculated in order to examine the fulfillment of the eventualities. An SCC of atoms is said to be *self-fulfilling* if for any atom $\alpha = (w, c)$ of that particular SCC for which $(\psi_2 \,\mathbf{U}^*\psi_1) \in c$, there exists some $\alpha' = (w', c')$ in the same SCC such that $\psi_1 \in c'$. This leads to another lemma from [CGP99]:

**Lemma 1.2.** An eventuality sequence starts from an atom $\alpha = (w, c)$ if and only if there there is a path in the atom graph $G$ such that a self-fulfilling SCC can be reached by it from $\alpha$.

To evaluate the satisfiability of the formula $\varphi$ in the model $\mathcal{M}$ used in this calculation, it is only required by Lemma 1.2 that some self-fulfilling SCC is reachable from one of the initial atoms. Thus $\mathcal{M} \models \varphi$ if and only if there exists at least one path from an initial atom $(w_0, m_0)$ to a self-fulfilling SCC of the atom forest. This is so, as a natural model given by such a sequence of atoms either ends in a terminal node or circulates the entire self-fulfilling SCC infinitely. When a maximal SCC is identified, the required and fulfilled eventualities of its nodes can be collected. If the set of those eventualities required is a subset of the set of those fulfilled, the SCC is self-fulfilling by definition.

---

**Example 7: An atom forest for an LTL formula**

The formula to be verified is an LTL formula $\neg\varphi = \neg((\mathtt{p} \vee \mathtt{r})\,\mathbf{U}^*\mathbf{X}\mathtt{q}))$ and thus the negation needed for the construction is $\varphi = ((\mathtt{p} \vee \mathtt{r})\,\mathbf{U}^*\mathbf{X}\mathtt{q}))$. The model of Figure 1.8 is the model used for the atom graph construction.

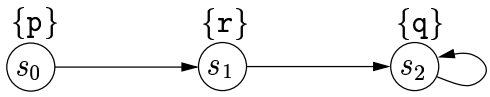


FIGURE 1.8: A simple non-branching Kripke-model $\mathcal{M} = (U, \mathcal{I}, w_0)$ with $\mathcal{I}(\mathtt{p}) = \{s_0\}$, $\mathcal{I}(\mathtt{r}) = \{s_1\}$, and $\mathcal{I}(\mathtt{q}) = \{s_2\}$

From the set of subformulas $SF(\varphi) = \{\mathtt{p}, \mathtt{q}, \mathtt{r}, \mathbf{X}\mathtt{q}, \mathtt{p} \vee \mathtt{r}, \varphi\}$, the closure can be obtained:

$$\begin{aligned} CL(\varphi) \;=\; \{\;& \mathtt{p}, \neg\mathtt{p}, \mathtt{q}, \neg\mathtt{q}, \mathtt{r}, \neg\mathtt{r}, \mathbf{X}\mathtt{q}, \neg\mathbf{X}\mathtt{q}, \mathbf{X}\neg\mathtt{q}, \\ & \mathtt{p} \vee \mathtt{r}, \neg(\mathtt{p} \vee \mathtt{r})\,\mathbf{X}((\mathtt{p} \vee \mathtt{r})\,\mathbf{U}^*\mathbf{X}\mathtt{q}), \\ & \neg\mathbf{X}((\mathtt{p} \vee \mathtt{r})\,\mathbf{U}^*\mathbf{X}\mathtt{q}), \mathbf{X}\neg((\mathtt{p} \vee \mathtt{r})\,\mathbf{U}^*\mathbf{X}\mathtt{q}), \varphi\neg\varphi \quad \} \end{aligned}$$

From these, the maximal propositionally consistent sets are constructed as earlier, again treating temporal formulas as they were atomic propositions.

All of these sets are combined with both of the states of the model to obtain the atoms. Of these, those that do not agree with the state on the valuation of atomic propositions are excluded, possibly already during the construction phase. With the remaining admissible atoms, an accessibility relation is defined by using the five rules above. The initial atoms are identified as possible starting points for the eventuality sequences.

Another example of LTL model checking by this algorithm is presented in [CGP99].

---

The algorithm for LTL model checking presented by Clarke and Schlingloff in [CS01] has much in common with the respective satisfiability algorithm. Again, a forest

of atoms is defined for $\varphi$ and $\mathcal{M}$. The relation $X_R$ between admissible atoms is redefined for LTL as follows[9]:

**Definition 1.3.** $X_R(w,m),(w',m'))$ applies if all of the following applies:

(i) $(w,w') \in \mathcal{I}(\mathbf{X})$

(ii) if $\mathbf{X}\psi \in SF(\varphi)$ and $\psi \in m'$, then $\mathbf{X}\psi \in m$.

(iii) if $\mathbf{X}\psi \in m$, then $\psi \in m'$

(iv) $\exists \mathbf{X}\psi \in m$

(v) if $\mathbf{F}^*\psi \in SF(\varphi)$ then $\mathbf{F}^*\psi \in m$ if and only if either $\psi \in m$ or $\mathbf{XF}^*\psi \in m$

The rules (i) to (iv) here are very similar to the ones given in Definition 1.2 in the previous section for modal logic; $\langle R \rangle$ has been replaced with $\mathbf{X}$. The fifth rule enforces the recursion axiom $\vdash \mathbf{F}^*\psi \leftrightarrow \psi \vee \mathbf{XF}^*\psi$. Note that there are no rules defined to handle the dual $\langle R \rangle$, $[R]$. Again the goal is to find an accepting path through the forest constructed from the atoms arising from the above definition, only accepting those paths that have *all eventualities* fulfilled.

### Finding maximal self-fulfilling SCCs

The construction of maximal strongly connected components for model checking is best done with *Tarjan's algorithm*, which uses enumeration of strong components in a graph while backtracking from a depth-first search. The pseudo-code presented here is from [CS01] and intended to be used on the atom forest of the latter $SF(\varphi)$-based construction presented, but can with modification also suit the earlier construction basing on $CL(\varphi)$. The function `children` is assumed to follow the accessibility relation of the graph of atoms used in either case. First, the procedure dealing with roots of the found (SCCs is provided, and after that, the actual depth-first search routine.

```
procedure processRoot (atom α)

    <set> required := ∅ // SET OF REQUIRED EVENTUALITIES
    <set> fulfilled := ∅ // SET OF FULFILLED EVENTUALITIES

    repeat
        β := pop(SCC)
        storage[β] := ∞ // PUT "BELOW" EVERYTHING ELSE
        required := required ∪ {ψ1 | (ψ2 U+ ψ1) ∈ β}
        fulfilled := fulfilled ∪ {ψ | ψ ∈ β}
    until (α = β) // ALL ELEMENTS IN THE SCC OF α ARE PROCESSED
    if (required ⊆ fulfilled) // THE SCC OF α IS SELF-FULFILLING
        print "φ is satisfiable in M"
        exit
```

[9] Keeping in mind that $\mathbf{F}^*\varphi \triangleq (\top \mathbf{U}^* \varphi)$ and $\mathbf{X}\varphi$ means that $\varphi$ is valid is a successor, this is easier to relate to the grammar given for LTL

```
procedure dfs(node α) // DEPTH FIRST SEARCH

    if (storage does not contain α) // A "FRESH" NODE
        <integer> currentDepth = recursionDepth
        recursionDepth++ // INCREMENT DEPTH BY ONE
        // INITIAL VALUE AT CURRENT RECURSION DEPTH
        storage[α] = currentDepth
        push(SCC, α) // TO THE TOP OF THE STACK
        // EXAMINE THE ACCESSIBILITY RELATION E:
        <set> successorAtoms := successor(α)

        for all (β ∈ successorAtoms) do
            dfs(β)
            // CHECK WHETHER β IS "ABOVE" α
            storage[α] = min(storage[α], storage[β])
        // IF NOTHING WAS ABOVE ⇒ α IS A ROOT OF AN SCC

        if (storage[α] = currentDepth)
            processRoot(α)
```

The `dfs` procedure together with the procedure `processRoot` constitute an implementation of Tarjan's algorithm including a test for self-fulfillment of the located SCCs. In the forward phase, the algorithm recursively traverses the tree of all atoms reachable from an atom $\alpha$, storing these on a stack. During backtracking the check is made whether $\alpha$ is the root of an SCC: if there are no cycles starting from above $\alpha$ (i.e. a node $\beta$ which is both "below" and "above" $\alpha$ in the traversing order), $\alpha$ is a root of an SCC consisting of all the nodes below it, that is: all the nodes stored above it on the stack.

When a SCC is found, it is inspected for self-fulfillment by iterating over all of the atoms of the SCC (i.e. anything above $\alpha$ on the stack and finally the root itself). This involves building two sets of eventualities: those required and those fulfilled, and checking whether the latter includes the former as a subset.

The hashtable is initialized to some invalid value, and as an atom is first encountered. Its recursion depth is stored as the initial value for that atom in the hashtable. The higher in the traversing tree an atom is, the smaller is its hashtable entry. After returning from a recursive call to an successor atom, the hashtable entries of the successor and the root candidate are compared. The smaller of those two values is set as the hashtable entry of the root candidate. Therefore, if the hashtable entry of the root candidate differs from its actual recursion depth after making all of the recursive calls of the successor atoms, it cannot be a root of an SCC. This is so because it has a (possibly not immediate) successor that is also a predecessor.

A main procedure needs to call `dfs` once per each initial atom, i.e. a pair $(w_0, m_0)$ where $w_0$ is the initial state of $\mathcal{M}$ and $\varphi \in m_0$. For the algorithm to report non-satisfiability, it must go through all the initial atoms and all atoms reachable from them.

**Computational complexity of LTL model checking**

The algorithm for LTL model checking described above is exponential with respect to the number of formulas with the $\mathbf{U}^+$ operator, as the sets of those formulas determine the propositionally consistent sets. With respect to the size of the Kripke-model $\mathcal{M}$, the algorithm is linear, $\mathcal{O}(\mathcal{M})$. It has been shown that LTL model checking with the past-temporal operators included is complete in **PSPACE** with respect to $|\varphi|$, and **NLOGSPACE** in size of the model $\mathcal{M}$.

The high complexity in regard to the size of the formula is normally not an obstacle, as in practice specification formulas tend to be quite short. Problems arise from the state space explosion: even with a linear complexity in regard to the model, the number of atoms to traverse in the algorithm can be quite large. Only magnitudes such as $10^5$ to $10^6$ for the number of reachable atoms are technically possible with current technology, according to Clarke and Schlingloff [CS01]. However, methods exist to exceed such limits but are not discussed in this context.

Another algorithm, with explicit tableau construction is presented by Clarke, Grumberg and Peled in [CGP99].

## 1.3 Model Checking for Propositional $\mu$-Calculus

Propositional $\mu$-calculus ($\mu$TL) is a very expressive logic, capable of representing everything expressible by CTL*, which is a language combining CTL and LTL that allows temporal operators to be nested without placing a path quantifier before each one. $\mu$TL can be said to be the "assembly language of temporal logic", as most of programmable logics are expressible by it. This is a natural consequence of tha fact that $\mu$TL fully characterized the behavior of any finite-state process [Cle90].

The idea of $\mu$TL is to allow quantification over certain subsets of the universe $U$, which is not a first order notion. For reasons of increasing computational complexity, quantifying over any arbitrary subset is a not a practical concept, but the *fix-point* quantification of $\mu$TL works quite well. The grammar of $\mu$TL is defined as follows:

$$\mu\mathbf{TL} ::= \mathcal{P} \mid \mathcal{Q} \mid \bot \mid (\mu\mathbf{TL} \rightarrow \mu\mathbf{TL}) \mid \langle\mathcal{R}\rangle\, \mu\mathbf{TL} \mid \nu\mathcal{Q}\, \mu\mathbf{TL}.$$

In addition to the "diamond"-operator $\langle\mathcal{R}\rangle$, its dual, the "box" operator $[\mathcal{R}]\varphi = \neg\langle\mathcal{R}\rangle\neg\varphi$, is often used in $\mu$TL-formulas. The quantifier (i.e. *recursion operator*)$\nu$ is a *restricted existential quantifier* on sets of points. Another quantifier $\mu$ is defined by $\nu$ as $\mu q\, \varphi \triangleq \neg\nu q\, \neg(\varphi\{q := \neg q\})$, where $\varphi\{q := \psi\}$ denotes the formula obtained from $\varphi$ by substituting $\psi$ in place of every *free occurrence* of $q$. Free occurrences are such appearances of $q$ that are not themselves under the scope of a quantifier. $\mu$ is interpreted as a *restricted universal second order quantifier*. A noteworthy equivalence in discussing $\mu$TL is that $\mathcal{M} \models (\varphi\, \mathbf{U}^+\psi) \leftrightarrow \mathcal{M} \models \mu q\, \mathbf{X}(\psi \vee (\varphi \wedge q))$.

Propositional $\mu$-calculus has fairly fluent algorithms both for global and local model checking. After making the following definitions, the model checking algorithm of CTL presented in Section 1.1.2 can be used as a starting point for a global model checking algorithm for $\mu$TL. The correspondence is quite strong, as $\mu$-calculus is interpreted not only on natural models, but also branching models.

**Definition 1.4.** A function $f : 2^U \mapsto 2^U$ is *monotonic* if $P \subset Q \rightarrow f(P) \subseteq f(Q)$.

**Definition 1.5.** A set $Q \subseteq U$ is called a *fixed point* of the function $f$ if $Q = f(Q)$.

**Definition 1.6.** $gfp(f) = \bigcup\{Q \mid Q \subseteq f(Q)\}$

**Definition 1.7.** $lfp(f) = \bigcap\{Q \mid f(Q) \subseteq Q\}$

**Definition 1.8.** A function $f : 2^U \mapsto 2^U$ is *union-continuous* if $f(\bigcup_{i\in I}\{x_i\}) = \bigcup_{i\in I} f(\{x_i\})$ for any index set $I$.

In other words, a function $f$ is union-continuous if $P_1 \subseteq P_2 \subseteq P_3 \subseteq \ldots$ implies that $f(\bigcup_i P_i) = \bigcup_i f(P_i)$, where $\forall i : P_i \subseteq U$. For a finite $U$, a monotonic function $f$ is necessarily union-continuous[10]. Thus monotonicity and finiteness together imply continuity.

Using the above definitions, Theorem 1.1 known as the Knaster-Tarski fix-point theorem states that for a monotonic function, the duals *gfp* and *lfp* are the unambiguous *greatest* and *least fixed point* of the function, respectively.

**Theorem 1.1.** Let $f : 2^U \mapsto 2^U$ be a monotonic function. Then

(a) $gfp(f) = f(gfp(f))$ and $lfp(f) = f(lfp(f))$, and

(b) $(Q = f(Q)) \rightarrow ((Q \subseteq gfp(f)) \wedge (lfp(f) \subseteq Q))$

Cleaveland [Cle90] gives the following enlightening rewritings from $\mu$TL to the temporal operators "always" **G** and "eventually" **F**:

$$\begin{array}{lcl} \mathbf{G}^*\psi & \triangleq & \nu q\,(\psi \wedge [\mathcal{R}]q) \\ \mathbf{F}^*\psi & \triangleq & \mu q\,(\psi \vee (\langle\mathcal{R}\rangle\top \wedge [\mathcal{R}]q)) \end{array}$$

### 1.3.1 Global Model Checking for $\mu$TL

According to Theorem 1.1, the following equivalences are defined:[11]

$$\begin{array}{lcl} (U,\mathcal{I},w) \models \nu q & \leftrightarrow & w \in \bigcup\{Q \mid Q \subseteq \varphi^{\mathcal{F}}\{q := Q\}\} \\ (U,\mathcal{I},w) \models \mu q & \leftrightarrow & w \in \bigcap\{Q \mid \varphi^{\mathcal{F}}\{q := Q\} \subseteq Q\} \end{array}$$

Also, $(\nu q\,\varphi)^{\mathcal{F}} = gfp(\varphi^{\mathcal{F}}_q)$ and $(\mu q\,\varphi)^{\mathcal{F}} = lfp(\varphi^{\mathcal{F}}_q)$, which states that $\nu q\,\varphi$ is in a sense obtained by starting with the entire universe and iteratively bounding the set with $\varphi$, where as $\mu q\,\varphi$ is obtained by expanding the "scope" by iterative application of $\varphi$ on an initially empty set.

A *functional* of a formula $\psi$ is denoted as $\psi^i$. A functional $\psi^i$ means that $\psi$ is applied to some argument $i$ times: $\psi^i(x) = \psi(\psi(\psi\ldots\psi(\psi(x)))\ldots))$. If the functional

[10] It is also intersection-continuous, i.e. $P_1 \supseteq P_2 \supseteq \ldots$ implies $f(\bigcap_i P_i) = \bigcap_i f(P_i)$ when $P_i \subseteq U$.

[11] Here a notion like $\psi\{x_0 := x\}$ denotes the formula obtained by replacing every *free occurrence* of $x_0$ by $x$ in $\psi$. A free occurrence of a variable is again an occurrence not bound by a quantifier.

defined by $\varphi$ is union-continuous (Definition 1.8), the fixed points $\nu q\,\varphi$ and $\mu q\,\varphi$ can be obtained with the following limits:

$$\begin{aligned} \nu q\,\varphi &= \lim_{i\to\infty} \varphi^i(\top) \\ \mu q\,\varphi &= \lim_{i\to\infty} \varphi^i(\bot). \end{aligned}$$

This means that in order to obtain $\nu q\,\varphi$, the formula $\varphi$ is applied repeatedly to "everything", thus limiting the space with $\varphi$ until *stabilization*. Here stabilization means that nothing more is excluded during an iteration. Generally, stabilization of an iterative process means that no modifications result from an iteration any longer and the process may stop.

This process in effect produces the greatest fixed point of the functional of $\varphi$ as stabilization is reached. Similarly for $\mu q\,\varphi$, the iteration begins from "nothing" and expands the space by applying $\varphi$ until stabilization, i.e. nothing is added in an iteration anymore, which leads to the least fixed point of the functional of $\varphi$.

For a finite universe $U$, every monotonic function is also union-continuous, and it is also sufficient to use $|U|$ as a bound to constrain the above limits instead of approaching infinity as above:

$$\begin{aligned} \nu q\,\varphi &= \lim_{i\to|U|} \varphi^i(\top) \\ \mu q\,\varphi &= \lim_{i\to|U|} \varphi^i(\bot) \end{aligned}$$

Thus the global model checking algorithm for branching time $\mu$TL can be written from the corresponding recursive algorithm for CTL on page 6 with minor modifications, as the handling of the quantifiers $\mu$ and $\nu$ must also be implemented. This global algorithm could be efficiently implemented e.g. using BDDs.

**function** `eval` (formula $\varphi$, sets of states $H_i$) : set of states
  // $H_i$ ARE THE VALUATIONS CURRENTLY USED FOR THE PROPOSITION VARIABLES
  **case** $\varphi$ **of**
    p: **return** $\mathcal{I}(\mathtt{p})$ // ATOMIC PROPOSITION
    $q$: **return** $v(q)$ // VALUATION OF A PROPOSITION VARIABLE
    $\bot$: **return** $\emptyset$
    $(\psi_1 \to \psi_2)$: **return** $((U \setminus \mathtt{eval}(\psi_1)) \cup \mathtt{eval}(\psi_2))$
    $\langle R\rangle\psi$: **return** $R^{-1}(\mathtt{eval}(\psi))$
    $\nu q\,\psi$ : $H_1 := U$
      **repeat until** stabilization // UNTIL $H_1$ DOES NOT SHRINK
        $H_1 := \mathtt{eval}(\psi\{\mathtt{q} := H_1\})$
      **return** $H_1$
    $\mu q\,\psi$ : $H_2 := \emptyset$
      **repeat until** stabilization // UNTIL $H_2$ DOES NOT GROW
        $H_2 := \mathtt{eval}(\psi\{\mathtt{q} := H_2\})$
      **return** $H_2$

---

**Example 8: A run of the recursive function `eval` explained**

For a $\mu$TL formula $\mu q_1(\mathbf{X}q_1 \vee (\mathtt{p}_1 \wedge \mu q_2(\mathbf{X}q_2 \vee \mathtt{p}_2)))$, the appearing boolean connectives can be transformed into implications and negations, more easily interpreted with the above algorithm.

Using two simple equivalences $(a \vee b) \triangleq (\neg a \to b)$ and $(a \wedge b) \triangleq \neg(\neg a \vee \neg b) \triangleq \neg(a \to \neg b)$, the formula can be rewritten as

$$\psi_1(q_1, q_2) = \mu q_1((\mu q_2(\neg \mathtt{p}_2 \to \mathbf{X} q_2) \to \neg \mathtt{p}_1) \to \mathbf{X} q_1).$$

Interpreting negations in the above algorithm is again done by the observation that $(a \to \bot) \triangleq \neg a$. Also note that the $X$ here is represented by $\langle R \rangle$ in the algorithm description

Thus the evaluation of $\psi_1(q_1, q_2) = \mu q_1 \, \psi_2(q_1, q_2)$ begins with repeating the evaluation of $\psi_2$ with different substitutions for $q_1$, starting with the empty set as stated by the above algorithm. In the calculations following this step in recursion depth thus always obey the current substitution given by the iteration step, $\{q_1 := H_{2i}\}$.

Each of these iterations is a recursive call on $\psi_2(q_1, q_2) = \psi_3(q_2) \to \psi_4(q_1) = (\mu q_2(\neg \mathtt{p}_2 \to \mathbf{X} q_2) \to \neg \mathtt{p}_1) \to \mathbf{X} q_1$, which is turn makes first the recursive call for the condition of the implication: $\psi_3(q_2) = \mu q_2(\neg \mathtt{p}_2 \to \mathbf{X} q_2) \to \neg \mathtt{p}_1$, which is again an implication.

Evaluating $\psi_3(q_2) = \psi_5(q_2) \to \psi_6$, where the call to $\psi_5(q_2) = \mu q_2(\neg \mathtt{p}_2 \to \mathbf{X} q_2)$ is evaluated first. This starts another repetitive evaluation, now fixing $q_2$ to some subset of $U$, starting from the empty set.

Each of these repetitions thus evaluates the subformula $\psi_7 = \neg \mathtt{p}_2 \to \mathbf{X} q_2 = (\mathtt{p}_2 \to \bot) \to \mathbf{X} q_2$. The evaluation of this innermost implication is quite straightforward given the current substitution set for $q_2$. The implications are evaluated by using the following three sets obtainable from the model $\mathcal{M}$: $\mathcal{I}(\bot) = \emptyset$ for the negation, $\mathcal{I}(\mathtt{p}_2)$ for the proposition, and the successor sets of the quantified set $q_2$ as $(\mathbf{X} q_2)^{\mathcal{F}}$ by $\mathcal{I}(R)$.

After this repetition returns, the consequence-part of the middle implication is evaluated: $\psi_6 = \neg \mathtt{p}_1 = (\mathtt{p}_1 \to \bot)$ and thus $\psi_6^{\mathcal{F}} = U \setminus \mathcal{I}(\mathtt{p}_1)$. Now the result of the middle implication can be returned and the repetition for $\{q_2 := H_{2i}\}$ can be continued.

After the repetition stabilizes, the consequence of the outermost implication $\psi_4(q_1) = \mathbf{X} q_1$ is evaluated. The evaluation of $\psi_4$ is done as above by moving one step in $\mathcal{I}(R)$.

---

As in the CTL version, the **repeat** cycles stabilize after at most $|U|$ rounds, giving a computational complexity $\mathcal{O}\left(|\varphi| \cdot |U|^{qd(\varphi)}\right)$, where the exponent $qd(\varphi)$ is the *nesting depth* of fixed-point operators in the formula $\varphi$. This is because the computation of an inner fixed-point formula always needs to be restarted from scratch for each iteration on the enclosing formula.

Translating the CTL-formula $\mathbf{EF}^*(\mathtt{p}_1 \wedge \mathbf{EF}^* \mathtt{p}_2)$ into $\mu$TL gives the formula of the previous example: $\mu q_1(\mathbf{X} q_1 \vee (\mathtt{p}_1 \wedge \mu q_2(\mathbf{X} q_2 \vee \mathtt{p}_2)))$. This is an example of an *alternation-free formula*. Alternation-freedom is due to the fact that the proposition variable $q_1$ does not appear in the inner fixed point formula $\mu q_2(\mathbf{X} q_2 \vee \mathtt{p}_2)$. A consequence of this is that when $\mu q_1$ is evaluated, the inner formula can be regarded as a constant. Therefore the model checking of an alternation-free formula can be done in linear time.

The possibility of high computational complexity can be demonstrated with the

following $\mu$TL formula: $\mu q_1(\mathrm{p}_1 \wedge \mu q_2(\mathbf{X}q_1 \vee \mathbf{X}q_2 \vee \mathrm{p}_2))$. With this type of nesting, for each iteration of $q_1$, the inner formula $\mu q_2(\mathbf{X}q_1 \vee \mathbf{X}q_2 \vee \mathrm{p}_2)$ is re-evaluated.

That is: for $\psi(q_1, q_2) = \mu q_2(\mathbf{X}q_1 \vee \mathbf{X}q_2 \vee \mathrm{p}_2)^{\mathcal{F}}$ and $\varphi(q_1) = (\mathrm{p}_1 \wedge \mu q_2\, \psi(q_1, q_2))^{\mathcal{F}}$, the calculation of $\mu q_1\, \varphi(q_1)$ can be written out as the following iteration, with the repeated occurrences of the sub-formula underlined:

$$
\begin{array}{lcll}
\varphi^0 & \triangleq & \bot & \\
\psi^{0,0} & \triangleq & \bot & \\
\psi^{0,1} & \triangleq & \psi(\varphi^0, \psi^{0,0}) & \triangleq \quad (\mathbf{X}\bot \vee \mathbf{X}\bot \vee \mathrm{p}_2), \\
\psi^{0,2} & \triangleq & \psi(\varphi^0, \psi^{0,1}) & \triangleq \quad (\mathbf{X}\bot \vee \mathbf{X}(\mathbf{X}\bot \vee \mathrm{p}_2) \vee \mathrm{p}_2), \\
\psi^{0,3} & \triangleq & \psi(\varphi^0, \psi^{0,2}) & \triangleq \quad (\mathbf{X}\bot \vee \mathbf{X}(\mathbf{X}\bot \vee (\mathbf{X}\bot \vee \mathrm{p}_2) \vee \mathrm{p}_2) \vee \mathrm{p}_2), \\
\vdots & & & \\
\psi^{0,n+1} & \triangleq & \psi(\varphi^0, \psi^{0,n}) & \triangleq \quad \underline{\mu q_2(\mathbf{X}\bot \vee \mathbf{X}q_2 \vee \mathrm{p}_2)}, \text{ if } \psi^{0,n+1} \triangleq \psi^{0,n}, \\
\varphi^1 & \triangleq & \varphi(\varphi^0) & \triangleq \quad (\mathrm{p}_1 \wedge \underline{\mu q_2(\mathbf{X}\bot \vee \mathbf{X}q_2 \vee \mathrm{p}_2)}) \vee \mathbf{X}\bot \vee \mathrm{p}_2), \\
\psi^{1,0} & \triangleq & \bot & \\
\psi^{1,1} & \triangleq & \psi(\varphi^1, \psi^{1,0}) & \triangleq \quad (\mathbf{X}(\mathrm{p}_1 \wedge \underline{\mu q_2(\mathbf{X}\bot \vee \mathbf{X}q_2 \vee \mathrm{p}_2)}) \vee \mathbf{X}\bot \vee \mathrm{p}_2), \\
\psi^{1,2} & \triangleq & \psi(\varphi^1, \psi^{1,1}) & \triangleq \quad (\mathbf{X}\varphi^1 \vee \mathbf{X}\psi^{1,1} \vee \mathrm{p}_2), \\
\vdots & & &
\end{array}
$$

A more sophisticated approach exists, in which each sequence of nested fixed-point[12] operators of the same type ($\nu q_1 \ldots \nu q_n$ or $\mu q_1 \ldots \mu q_n$) are calculated in the same loop. As $\varphi^0 \subseteq \varphi^1$ and $\psi$ is monotonic, also $\psi^{0,n} \subseteq \psi^{1,n}$

The computation of a least fixed point *lfp* can begin from any value below the result, as the process is expanding. Thus instead of initializing $\psi^{1,0}$ with $\bot$, also $\psi^{0,n}$ can be used. This generalizes to always using the last approximation as the starting value when restarting computation of an inner fixed point of the same type and therefore the value of the *lfp* cannot increase more than $|U|$ times, as nothing more can be added. With this improvement, the exponent of the complexity function may be replaced with the *alteration depth* of the fixed-point operators of $\varphi$ instead of using the direct nesting depth, resulting in $\mathcal{O}(|\varphi| \cdot |U|^{ad(\varphi)})$,

Other improvements are available as well: by storing more intermediate values during the computation, the evaluation time requirement can be reduced to $\mathcal{O}(|U|^{\lfloor ad/2 \rfloor + 1})$. No lower bound for the time complexity of $\mu$TL model checking is presently known, but the problem has been shown to be in $\mathbf{NP} \cap \mathbf{coNP}$.

### 1.3.2 Local Model Checking for $\mu$TL

Several approaches have been proposed for the local model checking of propositional $\mu$-calculus. Clarke and Schlingloff [CS01] have chosen to present an outline of a proof system using top-down decomposition: a tableau method, which explores a portion of the model using depth-first search. Also a method based on Gaussian elimination has been suggested, in which the formula is identified with a system of linear inequalities.

[12] "Fixed point" means here that something is required from one or several fixed states or points of time in the future or in the past.

The nodes of the tableau are sequences of the form $\Delta, w \models \psi$, where $w \in U$ is a state in given the Kripke-model $\mathcal{M}$, $\psi$ is a sub-formula of the given formula $\varphi$, and $\Delta$ is a list of definitions. The definition list contains a sequence of declarations $(q_1 = \psi_1, \ldots, q_n = \psi_n)$ where each proposition variable $q_i$ appears only once and each $\psi_i$ may only contain variables $q_j$ such that $j < i$.

The following assumptions are made for the sake of simplicity: only $\vee$, $\wedge$, $\langle R \rangle$, $[R]$, $\mu$ and $v$ are used as basic operations, negations are assumed to appear only in literals;i.e. the formula must be in *positive normal form*. In addition each $\mu$ or $\nu$ quantification in the formula $\varphi$ is expected to bind a different propositional variable. The tableau is constructed using the following seven steps for decomposition of the initial node $\emptyset, w_0 \models \varphi$:

(i) a node $\Delta, w \models (\psi_1 \wedge \psi_2)$ has two children: $\Delta, w \models \psi_1$ and $\Delta, w \models \psi_2$

(ii) a node $\Delta, w \models (\psi_1 \vee \psi_2)$ has one child: either $\Delta, w \models \psi_1$ or $\Delta, w \models \psi_2$

(iii) a node $\Delta, w \models \langle R \rangle \psi$ has one child: $\Delta, w' \models \psi$

(iv) a node $\Delta, w \models [R]\, \psi$ has $n$ children: $\Delta, w_1 \models \psi, \ldots, \Delta, w_n \models \psi$

(v) a node $\Delta, w \models \mu q\, \psi$ has one child: $\Delta', w \models \psi$

(vi) a node $\Delta, w \models \nu q\, \psi$ has one child: $\Delta', w \models \psi$

(vii) a node $\Delta, w \models q$ has one child: $\Delta, w \models \psi$

When using these rules of decomposition, the following four restrictions apply:

- rule (iii) can only be applied when $w' \in R(w)$
- when applying rule (iv), it must hold that $R(w) = \{w_1, \ldots, w_n\}$
- for rules (v) and (vi), $\Delta' = \Delta \cup \{q = \psi\}$ and $\Delta' = \Delta \cup \{q = \neg\psi\}$ respectively
- rule (vii) is only applicable when $(q = \psi) \in \Delta$ and no ancestor node is of the form $\Delta', w \models \psi$ with the same $w$ and $\psi$

Rule (v) states that in order to check whether $\mu q\, \psi \models w$, $q$ is recorded as a fixed point of $\psi(q)$ and check if $w \models \psi$. According to rule (vii), when decomposing $\psi(q)$ and later in the decomposition a $q$ occurs, it may be "unfolded" to $\psi$. When doing this, it must be noted that each proposition variable $q$ may only be unfolded once per each branch of the tableau and once in each state of the model; otherwise the unfolding does not necessarily terminate. This practice keeps the tableau of a finite model also finite.

Due to the branching time interpretation of $\mu$TL, the tableau rules for the algorithm have to be non-deterministic. This results in the same pair $(w, \varphi)$ to have several completed tableaus that are not identical; $w \models \varphi$ if and only if *at least one* of these tableaus is successful, i.e. has only successful leaves.

When no rule cannot be applied for any leaf, the tableau is said to be *maximal*. A leaf $\Delta, w \models \psi$ of a maximal tableau is said to be *successful* if the following holds:

(1) $(\psi = \mathtt{p} \in \mathcal{P} \wedge w \in \mathcal{I}(\mathtt{p})) \vee (\psi = \neg\mathtt{p} \wedge w \notin \mathcal{I}(\mathtt{p}))$

(2) $\psi = q \in \mathcal{Q}, q \notin \Delta, w \in v(q)$, or $\psi = \neg q, q \notin \Delta, w \notin v(q)$, or

(3) rule (iv) produces no children, that is: $\psi = [R]\,\psi' \wedge R(w) = \emptyset$

(4) $\psi = q \in \mathcal{Q}$ and $q$ was included in $\Delta$ by rule (vi)

Therefore, for an unsuccessful leaf holds the following:

(1) $(\psi = \mathtt{p} \in \mathcal{P} \wedge w \notin \mathcal{I}(\mathtt{p})) \vee (\psi = \neg\mathtt{p} \wedge w \in \mathcal{I}(\mathtt{p}))$

(2) $\psi = q \in \mathcal{Q}, q \notin \Delta, w \notin v(q)$, or $\psi = \neg q, q \notin \Delta, w \in v(q)$, or

(3) rule (iii) cannot be applied, that is: $\psi = \langle R\rangle\psi' \wedge R(w) = \emptyset$

(4) $\psi = q \in \mathcal{Q}$ and $q$ was included in $\Delta$ by rule (v)

The above definitions lead the following theorem which states that the tableau method is both sound (does not allow tableaus to be constructed when $w \notin \varphi^{\mathcal{F}}$) and complete (produces a successful tableau whenever $w \in \varphi^{\mathcal{F}}$):

**Theorem 1.2.** $w \in \varphi^{\mathcal{F}}$ if and only if there exists a successful tableau with root $\emptyset, w \models \varphi$.

# Chapter 2

# Modeling of Reactive Systems

In most real-life systems, the system specification is naturally divided into modules or other smaller constructs that cannot be in any apparent or reasonable way combined into a single Kripke-model. Thus the model checking algorithms described in Chapter 1 cannot be directly applied. This does not fortunately mean that model checking would be of no use in real-life; only that it is often more fluent to model the system to be verified in some number of sub-components rather than a single system model. This is increasingly common, due amongst other things to the wide-spread use of object-oriented design and implementation techniques and the preference of creating reusable software or hardware design.

Composing the system under inspection from the modules is not a trivial task. Questions arise concerning the order of execution of the processes and the communication between them. In practice, modular systems are without exception also *parallel* systems, where the problems of concurrent execution need to be addressed. The means of *communication* and *synchronization* are numerous and so are the methods of presenting them in a formal model. The issue of process communication is generally investigated in regard to *distributed* systems discussed in Section 2.1 and process synchronization in regard to *concurrent* systems discussed in Section 2.2, although many practical systems can be seen as both distributed and concurrent.

## 2.1 Distributed Systems

A *distributed system* is generally considered to consist of *spatially separate processes*, e.g. processes that are being executed simultaneously on different computers. Need for communication between these processes makes these separate modules parts of the same system. The channel of communication makes them in a way joint together, although independent in other aspects. Examples of larger systems that can also be considered as distributed systems are the Internet, any intranet or many applications of mobile computing [CDK00]. The main requirement for a system to be classified as a distributed system is not the spatial separation but the need for independent processes to communicate or share resources in some controlled manner.

There are two main implementation techniques for inter-process communication: *message passing* and *shared variables*. Communication by message passing is "pure"

distribution, but the use of shared memory requires much attention on the concurrency of reads and writes as well as the communication itself.

### 2.1.1 Communication by Message Passing

As a paradigm, message passing is quite simple although often tedious to define for an actual system. The idea in *message passing* is that the processes send each other messages, where the messages and the order of sending them are specified by some protocol. Each process may *send* or *receive* a message. The messages may be directed to specific *channels* with specified recipients or *broadcasted* to any process within the messaging space.

In *synchronized* communication, the `send` and `receive` primitives are *blocking* operations, i.e. the sending process must wait for the recipient to react before proceeding further [CDK00]. In *asynchronous* communication there is no such delay: the sending process may proceed immediately after `send` has been performed, regardless of the recipient condition [And00]. Sometimes `receive` is considered a blocking operation even in asynchronous communication.

Often the communication passed in these messages in some way controls the execution of the processes of the system. One process may for example ask for permission to use the printer, and the permission is granted if the other processes reply with an accepting message.

The term *synchronous* communication is also used to indicate that a process (or more generally a system module) may not proceed until all the communication partners defined for the particular task have expressed willingness to participate in that action. Thus processes can cooperate to perform a certain task requiring specific sub-tasks to be accomplished by the participants. Similarly, *asynchronous* communication can be viewed as the processes deciding themselves whether they wish to wait, usually by using some sort of buffer for messages that are not immediately reacted to. In this aspect, synchronous communication is simply a special case with buffer size being one and all of the processes deciding to wait of the buffer to be full or empty again, depending on their action.

When non-determinism is required for the inter-process communication, *guards* may be placed on the messaging operations. A guard consists of a boolean expression and communication statements. It will allow for some statements to be executed if the expression is true and the execution of the communication would not break some defined rules, e.g. cause delay [And00].

### 2.1.2 Communication by Shared Variables

When two or more processes interact by reading from and writing to a shared memory space, the may exchange information by using *shared variables* for separate issues on which to communicate. The semantic of possible values for each of the variables must be defined, similarly to defining a grammar or a protocol for the message passing paradigm.

When using shared variables as the channel of inter-process communication, it must be ensured that the accesses to the shared variables are *legal*: two processes should not be able to modify the value of a variable at the same time, and no process should read a value after some other process has reserved it for modification. The means of achieving such mutual exclusion are viewed as methods of concurrent programming. When shared variables are used in distributed systems, the processes must regularly check the values of the variables and synchronize their execution on the basis of that information, whereas in employing message passing, the processes receive notification messages from each other and make the execution decisions basing on those messages.

It is often difficult to separate the two means of communication from each other, as both can be seen as special cases of the other: messages are in a sense values of shared variables where as the variables are in a sense communication channels. The distinction needs to made only because many verification systems are only capable of dealing with one paradigm, excluding the other. Also many modeling languages do not support both approaches.

## 2.2 Concurrent Systems

In a *concurrent system*, multiple executions are considered to take place simultaneously, e.g. on a multiprocessor machine. Despite the definition, concurrent systems may also be running on a single-processor computer: each of the processes is interleaved in some specified fashion, controlled by the operating system.

The important issue is the controlled usage of system resources, such as files or printers: no two processes can print a document at the same time, as the output would interleave on the printed in a non-desirable manner. If one process is handling a system resource that can be accessed by only one process at a time, it is said to be in a *critical section* of the system. That particular resource is thereby *locked* from the other processes. The processes wishing to access a locked resource normally *wait* on that lock in some orderly fashion, but not necessarily in the order of arrival. Often the ordering is at least partially determined by *priorities* assigned to the processes: a process with high priority is more likely to "pass through" quickly.

As a certain operation of a process may require the possession of more than one resource, a number of locks may need to be obtained. The order in which this is done by separate processes is significant, as a *deadlock* may result: if both processes $A$ and $B$ need to use both of the system resources $\alpha$ and $\beta$, and process $A$ has acquired the lock of $\alpha$ first, whereas process $B$ possesses the lock of $\beta$, both processes will wait infinitely on the second lock, possessed by the other process.

Also other unwanted situations are possible: should processes $C$ and $D$ both require access to resource $\delta$ repeatedly, process $C$ may be in a sense "faster" (i.e. have higher priority) and get the resource every time, whereas process $D$ stays waiting infinitely. Such configuration is called a *livelock*.

The design of concurrent system needs to take into account these and other problematic situations of assigning resources to processes. Should some of the necessary resources be shared among processes that can be viewed as spatially separate, the system is also considered to be a distributed one.

### 2.2.1 Synchronization According to Time

There are two meanings to the term *synchronization* in concurrent systems. The task of locking resources — blocking processes — is called *synchronization* as well as the task of *timing* events taking place at the separate system modules. Here the latter is discussed in more detail.

In order to discuss any timing properties, it must be determined how time is represented in the system model. Time is considered *discrete* if a computation of a process is considered to consist of a *sequence of steps*, between which exists nothing, in the sense that no action can be taken in between the steps. Thus points of time can be represented as integers and clocks as integer counters. The other possible approach is *continuous* time, which is more rarely used in system modeling. In a continuous time system it is possible to let the state changes happen gradually. Systems that include a possibility of both timing schemes are called *hybrid* systems.

The selection of the timing scheme is based on the properties of the system being modeled and verified. Some model verification tools only allow one or the other, most often supporting discrete time. This is quite sufficient for many practical modeling tasks.

When the system modules are said to be *synchronized*, for each of the discrete time steps, each of the modules performs a task. The tasks are generally independent on each other, but it can also be stated that only when certain conditions apply, certain transitions may be taken by the processes. One process may have to stay in an *idle loop*, performing essentially a `no-op` (stands for *no-operation*) until the conditions for taking the next actual step are reached. In *asynchronous* computation, for each time step, some but not necessarily all modules advance. If performing a `no-op` is possible for every process in every state of the computation, then the synchronous and asynchronous computation modes coincide. The idle step is also called a "stutter".

Both of the execution modes may be implemented in the verification system by *interleaving* the execution of the modules. In synchronized mode each module can be advanced per each interleaved "round", but in asynchronous computation some processes may be "skipped" during the round corresponding to a specific time step. Similar interleaving takes place when several processes are run on the same processor with some scheduling that determines which process to run — here some fairness constraints need to be applied in order to avoid behavior that leads to *starvation*, i.e. some processes are not served in reasonable time although they are ready for execution. Also other problems besides starvation may appear in the scheduling, preventing some or all of the processes to progress desirably.

## 2.3 Formalisms for Finite State Systems

There are several different formalisms for parallel systems with a finite set of states, three of which are presented below. These are quite abstract — as natural for formalisms — and each characterizes several different implementations in actual modeling systems.

### 2.3.1 Parallel Transition Systems

A *transition system* is characterized by an *alphabet* $\Sigma$, a non-empty and finite set of states marked with $S$. A subset $S_0 \subseteq S$ (possibly containing only one state) are considered to be *initial states*, i.e. places where the computation may begin.The *transition relation* $\Delta \subseteq S \times \Sigma \times S$ describes which states in $S$ are reachable in one step from one another for the symbols of the alphabet $\Sigma$. Thus a transition system can be written as the tuple $(\Sigma, S, \Delta, S_0)$. Transition systems are important representations of Kripke-models, as for any Kripke-model $\mathcal{M} = (U, \mathcal{I}, w_0)$ exists a transition system that accepts the same language (i.e. paths) that the model generates.

A *parallel transition system* is an $n$-tuple of transition systems, marked with $T = (T_1, \ldots, T_n)$, in which there are no shared states between the systems, namely $\forall i, j$ such that $i < j$, applies that $S_i \cap S_j = \emptyset$. The *global* transition system of $T$ in association to the $n$-tuple is defined as follows:

- the *global alphabet* is the union of all alphabets, $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$
- the *global state space* is the Cartesian product of all the state spaces, $S = S_1 \times \ldots \times S_n$
- the set of *global initial states* is similarly the Cartesian product of all the initial state sets, $S_0 = S_{10} \times \ldots \times S_{n0}$
- for the *global transition relation*, each subsystem only makes transitions that are valid for its own alphabet, that is: a transition $\delta = ((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ if and only if $\forall T_i$ the following applies:

  (i) if $a \in \Sigma_i$, then $(s_1, a, s'_1) \in \Delta_i$, and

  (ii) if $a \notin \Sigma_i$, then $s_i = s'_i$

### 2.3.2 Elementary Petri Nets

Petri nets are tools for modeling distributed systems both mathematically and graphically. A Petri net consists of places ($P$), transitions ($T$) and arcs connecting the places and the transitions. (A more formal discussion will follow shortly.) Places may contain *tokens*, which define a *marking* on the net. There may be more than one token in a place and there can be different types of tokens. When a transition *fires*, the tokens move according to the arcs, thus changing the marking of the net.

An *elementary Petri net* consists of a finite set $P$ of *places* (similar to the set of states in the previous formalism) and a set of *transitions* $T$ such that $P \cap T = \emptyset$. A *flow relation* $F$ of a Petri net describes the relations of the places and the transitions: $F \subseteq (P \times T) \cup (T \times P)$ and there is an *initial marking* $m_0 \subseteq P$ imposed on the net. Thus an elementary Petri net is a 4-tuple $N = (P, T, F, m_0)$.

A marking function $\mathcal{L}$ is a function from the set of places $P$ to the set of $P$'s subsets $2^P$ .Any subset of $P$ is a *marking* of the Petri net. For each transition, a *preset* and a *postset* is defined based on the places "reachable" before and after the transition, respectively, according to $F$. The *preset* can be written as $\bullet t \doteq \{p \mid (p, t) \in F\}$, and likewise for the *postset*, $t\bullet \doteq \{p \mid (t, p) \in F\}$.

A transition of a Petri net is said to be *enabled at a marking* $m \subseteq P$ if its preset is included in the marking and the intersection of the postset with the marking includes nothing but the postset: $\bullet t \subseteq m \wedge t\bullet \cap m \subseteq \bullet t$. This means that all of the "input places" of the transition $t$ are *occupied* at $m$ and all of the "output places" that are not also input places are empty.

A marking $m'$ is said to be the *result of firing a transition* $t$ from some marking $m$, if the transition $t$ is enabled at $m$ and the marking $m'$ consists of the places that are either in $m$ but not in the preset of $t$ or alternatively in the postset of $t$, $m' = (m \setminus \bullet t) \cup t\bullet$.

*Place-transition Petri nets* are more complicated than the finite-state, elementary Petri nets: they allow several tokens to occupy the same place.

### 2.3.3 Equivalence of Transition Systems and Elementary Petri Nets

For each elementary Petri net there exists an equivalent transition system. The construction begins by identifying the alphabet $\Sigma$ of the transition system as the set of transitions $T$ of the Petri net. Then the set of states $S$ for the transition system is fluently the set of markings possible for the Petri net, and the initial state set $S_0$ contains only one initial state, namely the initial marking $m_0$. It is noteworthy that the size of the set of states $|S|$ hereby constructed of exponential size in respect to the number of places $|P|$ in the Petri net. Here the transition relation $\Delta$ of the transition system contains a transition $(m, t, m')$ if and only if $m'$ results in firing $t$ at $m$.

It is also possible to construct a transition system that is of the same order of magnitude than the underlying elementary Petri net by requiring that $\forall p \in P$ there is a transition system with two states $p^0$ and $p^1$, with 1 declaring $p$ to be occupied by a token and 0 stating it to be empty. The transition relation is defined as follows:

$$\forall t \in T : \left((p^1, t, p^0) \in \Delta \Leftrightarrow p \in \bullet t \setminus t\bullet\right) \wedge \left((p^0, t, p^1) \in \Delta \Leftrightarrow p \in t\bullet \setminus \bullet t\right).$$

Plainly said, this means that a transition $\delta \in \Delta$ that causes a place $p$ to be occupied by transition $t$ must be such that the place belongs only the the preset of $t$ but not the postset. Also, if a transition $\delta \in \Delta$ causes a place $p$ to lose its token (i.e. become empty), $p$ must be in the postset of $t$ but not the preset. From this transition systems for the places $p$ a global, parallel transition system can be constructed such that the language of the construction represents the set of firing sequences of the Petri net. A construction is possible also vice versa, converting a parallel transition system into an elementary Petri net of the same order of magnitude.

### 2.3.4 Shared Variable Programs

A *shared variable program* consists of a set of *program variables* $V = \{v_1, \ldots, v_n\}$ and a *state space* $D = D_1 \times \ldots \times D_n$ in which each $D_i = \{d_{i1}, \ldots, d_{im_i}\}$ is a *finite domain* of the corresponding variable $v_i$. A *transition relation* $T$ is defined as the Cartesian product of the state space, $T \subseteq D \times D$ and there is one *initial state*

$s_0 = (d_{11}, \ldots, d_{n1})$. This definition makes the shared variables program presentable with the tuple $(V, D, T, s_0)$.

As can be interpreted from the representation of the initial state, each state of a shared variables program is a tuple $(d_1, \ldots, d_n)$ where each $d_i \in D_i$. This defines the size of the state space as the product of the domain sizes, $|D| = \prod_{i=1}^{n} |D_i|$, which is apparent from the definition of $D$ above.

The transition relation $T$ of the program is defined by a *propositional formula* $\varphi$, where the set of atomic propositions is $\mathcal{P} = \{(x = y) \,|\, x, y \in (V \cup V' \cup D_i)\}$, where $V$ is set set of program variables as above, $V' = \{v'_1, \ldots, v'_n\}$. A transition $(s, s')$ is in $T$ if and only if the proposition $\varphi_T$ is a logical implication of the valuation $\mathcal{I}$, i.e. $\mathcal{I} \models \varphi_T$, where $s = (d_1, \ldots, d_n)$, $s' = (d'_1, \ldots, d'_n)$, $\mathcal{I}(v_i) = d_i$, and $\mathcal{I}(v'_i) = d'_i$.

For almost any model of concurrency, a shared variables program may be obtained. This makes shared variables programs a usable model for reactive systems.

## 2.4 Examples of Reactive System Applications

The possible examples of the usage of modeling and verification for reactive systems are numerous. Clarke and Schlingloff [CS01] have chosen to present three quite different examples:

- a purely algorithmic example : a combinatorial puzzle game of moving tiles
- a hardware-oriented example: a shift register for data bus interfacing
- an example of distributed software: a communication protocol of a cellular phone

### 2.4.1 The Grid Puzzle

This example is not very connected to the real-life usage of model checking techniques, but serves to demonstrate some of the properties of the current modeling technology. The problem is to perform combinatorial search on the set of possible configurations of a grid puzzle invented by Sam Loyd in 1870: the puzzle is a rack with a $h \times v$ grid with one empty slot and the rest of the slots filled with distinguishable square tiles of equal size.

The tiles can be moved upwards, downwards or sideways by employing the blank space as the destination slot. Naturally only those tiles can be moved that are adjacent to the empty slot. Here a sufficient method of distinguishing the tiles is labeling them with numbers, normally the tiles form an image of some sort. The goal of the game is to return the tiles into the default ordering (e.g. the numerical ordering or to a configuration that correctly restores the image) from some other configuration by moving the tiles one by one.

To represent the puzzle as some formalism, it is fluent to compose a shared variables program by assigning a variable for each tile $i = 1, \ldots, (h \cdot v - 1)$. The variable must

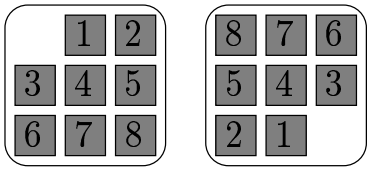


FIGURE 2.1: The initial (left) and desired (right) configurations of a $3 \times 3$ grid puzzle

be a two-component vector $t_i = (v_{ih}, v_{iv})$, representing the position of the tile on the puzzle rack on both the horizontal and the vertical axes. A separate variable is needed to determine the direction of the next move: $m = \{up, down, left, right\}$. Should the attempted move cause a tile to cross the border of the puzzle rack, no change is made to the configuration. Otherwise the positions of the tile and the empty slot represented by $e = (e_h, e_v)$ are swapped.

This can be written as a shared variables program by inspecting the possible orbits of the tiles on the rack. The goal can be set as a conjunction of the positions reaching their desired values, and the model checker can examine the "solvability" of the puzzle by the CTL (Computation Tree Logic) clause $\neg$**EF**`goal`. The SMV code is presented at the end of this section.

The space requirement to compute the transition relation for a $3 \times 3$ –puzzle is 3 KB. The number of states becomes $4 \cdot (h \cdot v)! \approx 1.4 \cdot 10^6$, and half of those are reachable from all possible initial states. When SMV is used to check this model against the CTL-specification, claiming that there can be no solution, the tool attempts to contradict the claim by constructing a counter example, namely a sequence of moves leading to the configuration fulfilling `goal`. This is successful for the model provided above; a couple of minutes of processor time required on a 133 MHz Pentium with 32 MB of main memory.

For a larger case, $h = 4$ while $v$ remains at 3, the size of the state space grows to approximately $10^9$. It is fairly easy to detect that a solution must exist, but the process of constructing the counter example for a state space that large requires partitioning it into strongly connected components, which requires several days of CPU time even on Sparc Ultra using 1 GB of main memory. If the state space would not fit into the main memory entirely, the performance would suffer significantly, as using virtual memory results in constant swapping.

Clarke and Schlingloff [CS01] state that currently an exhaustive search on the case of $h = v = 4$ is not realistic with existing tools, but combinations of model checking and the usage of heuristics enables automatic construction of solutions to some combinatorial games, including this puzzle.

```
MODULE main
DEFINE v := 3, h := 3; -- Board size, 3 × 3 tiles
VAR move:  u, d, l, r; -- Possible moves
   -- Horizontal positions of the tiles
   hpos: array 0 ..  (h * v - 1) of 1 ..  h
   -- Vertical positions of the tiles
   vpos: array 0 ..  (h * v - 1) of 1 ..  v

ASSIGN -- Defining the legal block swaps
next(hpos[0]) := case -- The empty slot is (0, 0) in the array
   -- If going left and not on the left edge, swap there
   (move = l) & !(hpos[0] = 1) :  hpos[0] - 1;
   -- If going right and not on the right edge, swap there
   (move = r) & !(hpos[0] = h) :  hpos[0] + 1;
   -- If not moving sideways, the horizontal position does not change
   1 :  hpos[0]; esac;

next(hpos[1]) := case
   -- The horizontal position of the block with label "1"
   (move = l) & !(hpos[0] = 1) &
   -- If empty slot not the leftmost and moving left
      -- If block "1" is on the right side of the empty slot on the same row
      vpos[1] = vpos[0] & hpos[1] = hpos[0] + 1 |
      -- or if going right and empty slot is not on the right edge
      (move = r) & !(hpos[0] = h) & vpos[1] = vpos[0] &
      -- If block "1" is on the left side of the empty slot on the same row
      hpos[1] = hpos[0] & vpos[1] = vpos[0] + 1 :  hpos[0]; -- Swap
   1 :  hpos[1]; esac; -- No change otherwise on the horizontal position
next(vpos[1]) := case -- The vertical position of the block with label "1"
   (move = u) & !(vpos[0] = 1) &
   -- If empty slot not the uppermost and moving up
      -- If block "1" is below the empty slot on the same column
      hpos[1] = hpos[0] & vpos[1] = vpos[0] - 1 |
      -- or if going down and empty slot is not on the lower edge
      (move = d) & !(vpos[0] = v) & hpos[1] = hpos[0] &
      -- If block "1" is above the empty slot on the same column
      vpos[1] = vpos[0] & hpos[1] = hpos[0] - 1 :  hpos[0]; -- Swap
   1 :  vpos[1]; esac; -- No change otherwise on the vertical position

   -- The same is repeated for all the other labeled tiles "2" to "8"
   ⋮

   init(vpos[0]) := 1; init(hpos[0]) := 1;
   init(vpos[1] := 1); init(hpos[1] := 2;
   -- Also initialization for the positions of the other 7 tiles
   ⋮

DEFINE goal := (vpos[0] := 3 & hpos[0] = := 3 & vpos[1] := 3 & hpos[2] &
   -- And other definitions of the desired final configuration of the puzzle
   ⋮
SPEC !EF goal
```

### 2.4.2 A Shift Register for Interfacing a Parallel Data Bus

The hardware verification example is a *shift register* for exchanging data between the data bus and a serial device, acting as a parallel-serial converter and vice versa. The selection between parallel and serial access modes is made with a *mode control input*, and for both of the modes, there is an input clock. These shall referred to by the following abbreviations: `mc` for the mode control input, `pc` for the parallel input clock, and `sc` for the serial input clock.

For a "well-behaving" circuit — that is: for a circuit where an ordering can be imposed on the wires and the value of each write only depends on the value of it "neighbors", i.e. wires that are close to it in the ordering — automatic verification is possible for much larger circuits also. Well-behaving circuits with several hundreds of storage units have been verified automatically.

When `mc` is *high* and a `pc` clock pulse arrives, parallel loading is performed: data is read from the bus to the flip-flops. Consequently, for a *low* `mc`, serial data input takes place with every `sc` clock pulse: the state of all the flip-flops is "shifted" to the right one step per each clock pulse to move it on the parallel output. $n$ steps are required before the data is ready to be sent to the bus by an `oc` command.

If the *serial input* `inp` is held low, a right shift takes place. After $n$ of these, data obtained in parallel from the bus can be written in a serial manner to the output port `out`. Figure 2.2 should clarify the situation for those familiar with circuits.

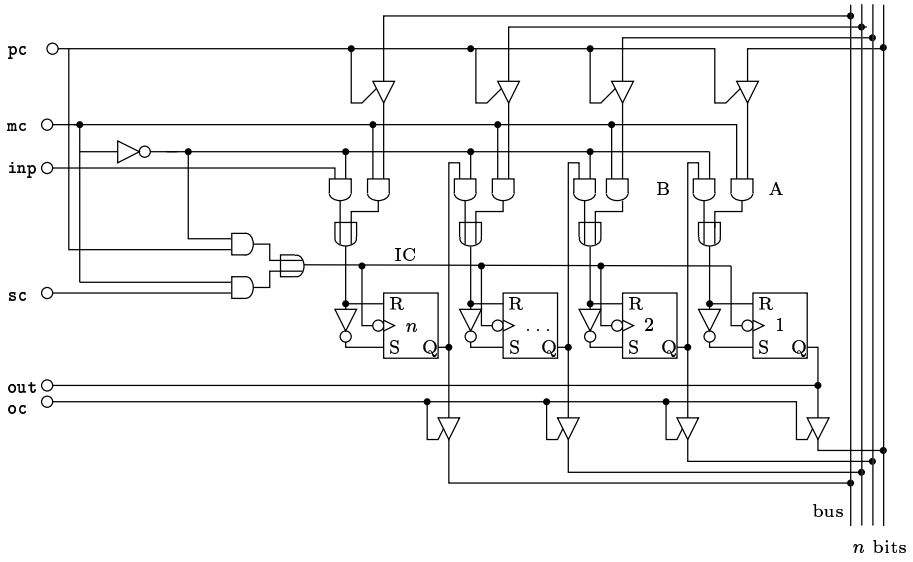


FIGURE 2.2: A functional diagram of an $n$-bit shift register for data bus interfacing

The register is implemented with SR-*bistables*. Bistable mechanism is one that has two stable equilibrium states, requiring no power input to maintain either one of these equilibrium states. Thus they make good switches. When discussing circuit design, the two states are commonly marked with 0 (low) and 1 (high). In this register, the SR-bistable has the following characteristic function for the changes from state `Q` to the next state `Q'`:

- when both inputs S and R are *low*, no state change occurs
- should both inputs be *high*, the state is undefined
- if only S is high, the output is set to 1
- if only R is high, the output is reset (to 0)

| S | R | Q' | description |
|---|---|---|---|
| 0 | 0 | Q | maintain |
| 1 | 0 | 1 | set |
| 0 | 1 | 0 | reset |
| 1 | 1 | - | undefined |

The undefined state can be modeled by making a non-deterministic choice between 0 and 1 for the value of the next state Q' internally. The output here changes only when the clock line changes from high to low. To model the state changes accurately (necessary e.g. for asynchronous circuits), timing information for the gates would need to be included, but as the clock is merely a trigger here, event-based modeling is more appropriate.

Using the event-based approach, the change of "direction" in the clock line is considered an event, which may or may not occur in each of the system states. For every infinite run, an infinite number if clock pulses is required as a fairness constraint in order to prevent executions in which clocks can be infinitely blocked.

To make a model of the system, a representation of the circuit's *truth table* suffices: the outputs are presented a s a boolean function of the inputs and the latch states. Many tools are capable of constructing this representation automatically from a standardized hardware description language. The correctness requirements are expressed by the following two formulas, with $n$ representing the width of the data bus (i.e. the number of bits there):

$$\mathbf{AG}^*(\mathtt{mc} \wedge \mathtt{pc} \rightarrow \bigvee_{i=1}^{n} (\mathtt{bus}[\mathtt{i}] \leftrightarrow \mathbf{A}((\mathtt{oc} \rightarrow \mathbf{AX}(\mathtt{bus}[\mathtt{i}]\, \mathbf{U}^+ \mathtt{ic})))$$

$$\mathbf{AG}^*(\neg\mathtt{mc} \wedge \mathtt{sc} \rightarrow \bigvee_{i=2}^{n} (\mathtt{Q}[\mathtt{i}] \leftrightarrow \mathbf{A}(\mathtt{Q}[\mathtt{i}-\mathtt{1}]\, \mathbf{U}^+ \mathtt{ic}))),$$

where ic is the result of the bit-wise operation $\mathtt{ic} = ((\neg\mathtt{mc} \wedge \mathtt{pc}) \vee (\mathtt{mc} \wedge \mathtt{sc}))$. These formulas state that if data is input to the register, it will stay there until a new input occurs. For the parallel control mode and a pulse of pc, data from the bus position $i$ is delivered per each pulse of the output clock oc, until new input arrives. Similarly, for the serial control mode and a pulse of sc, the latches maintain their state until the arrival of the next input.

The following SMV-input can be used to verify these formulas for $n = 32$ in less than a second. More formulas of a similar construct could be written e.g. to ensure that after a sequence of $n$ sequential load operations, a subsequent output pulse would cause the correct data to be put onto the bus.

```
MODULE main
VAR Q, bus:  array 1 ..  n of boolean; -- n SR-LATCHES AND n DATA-BITS
   inp, mc, pc, sc, oc:  boolean; -- INPUT LINES
DEFINE out := Q[1]; ic := ((!mc & pc) | (mc & sc));
      -- FOR ALL OF THE n BITS INDEXED WITH 0,...,n-1:
      A[0] := mc & pc & bus[0]; B[0] := !mc & Q[1];
      R[0] := !(A[0] | B[0]); S[0] := !R[0];
      ⋮
      -- ALSO ASSIGNMENT RULES FOR ALL BITS, AN EXAMPLE HERE:
ASSIGN next(Q[0]) := case ic:  case
            !S[0] & !R[0]:  Q[0]; -- MAINTAIN
            S[0] & !R[0]:  1; -- SET
            !S[0] & R[0]:  0; -- RESET
            S[0] & R[0]:  {0, 1}; esac; -- UNDEFINED
      next(bus[i]) := case oc:  Q[0]; !oc:  {0, 1}; esac;
      ⋮
      -- SAME FOR THE REST
FAIRNESS ic FAIRNESS oc
```

### 2.4.3 A Communication Protocol for a Cellular Phone

The last example presents a model for a *bounded buffer* used by a set of communicating processes within the operating system of a Siemens cellular phone. The inter-process communication is implemented by having the processes pass priority message to each other. Each process can be represented as a finite state machine that can be described by a set of SDL (Specification and Description Language) diagrams. The basic operation is that a process waits in some specific state for a message to arrive from some other process, to which it then reacts by performing certain operations, sending further messages to other processes, and performing a state transition. A part of a sample transition graph is shown in Figure 2.3 and the corresponding SDL diagram in Figure 2.4, both adapted from Clarke and Schlingloff [CS01].

The presented sample is the implementation of the following requirement presented in the international GSM standard:

> "Initially the mobile station looks for a cell which satisfies tha suitability constraints by checking cells in descending order of received signal strength. If a suitable cell is found, the mobile station camps on it and performs any registration necessary."

The this system should be inherently deadlock-free, which can be verified with the following formula: **AG*****EF***`init`, i.e. there is no sequence of user action possible that could cause the phone to reach such a state from which there is no path to the initial state. In plain English the property states that it must always be possible to reset the device. This is an important property marketwise, as the number of units produced is expected to be quite high. For this particular Siemens cellular phone, a number of design flaws that could have led to problems in operating the device were identified by formal verification before the implementation phase.

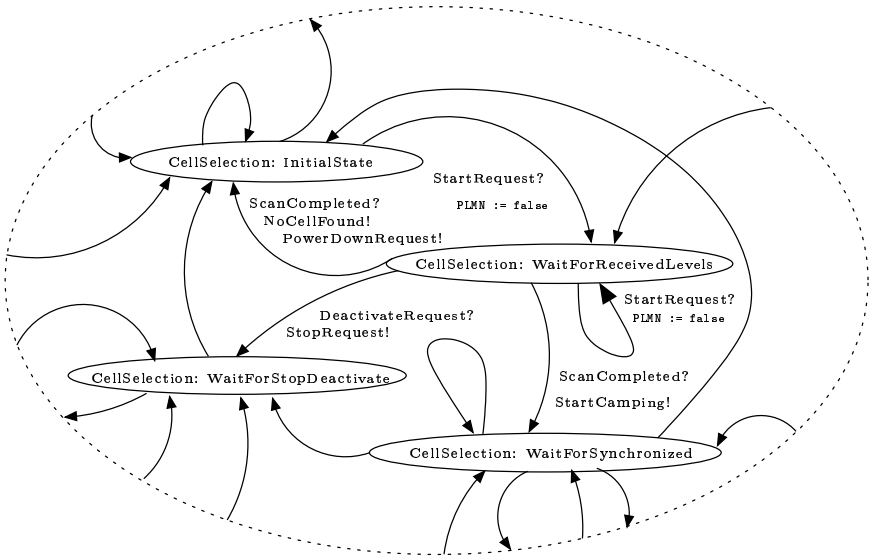


FIGURE 2.3: A portion of the transition graph of a process

The model discussed here consists of five processes and the operating system kernel process. The size of the state space per process is approximately 10 to 20 states and there are approximately 50 different types of messages for the inter-process communication. The processes are scheduled by the operating system based on a priority scheme; also the storage and delivery of the messages is handled by the operating system. Thus it must maintain some sort of a buffering mechanism for the messages of each process.

In the verification process, the most significant factor turns out to be the buffer size $n$. For the mobile phone system at hand, the buffer size $n$ was fixed on the interval 15–20. As each of the $n$ position of the buffer could contain any of the appr. 50 messages, the possible combinations of the buffer contents for the buffers of the five processes are numerous.

It can be safely assumed (that is: formulated into a fairness assumption) that a buffer overflow does not need to be considered, as it would be a clear signal of an implementational error. This could be the result of some high-proroty process constantly resending the same message, consequentially filling up any limited-size buffer. The possibility of state explosion due to the number of possible message combinations is avoided by observing that the buffer contents are a result of some regular pattern repeating in the communication. In practise, if the number of reachable states were to grow exponentially, it would most often be a indication of an error in the specification.

A transition relation can be written into a transition table like the one below if all of the messages are of the same priority. The right side represents the state of the three variables `input`, `buffer`, and `output` before the transition and the left side the corresponding values afterwards A buffer is represented as a sequence $\langle x_1, \ldots, x_v \rangle$, where $v < n$ and $n$ is the buffer size. The empty buffer is denoted with $\langle\,\rangle$.

On the right side, empty entries indicate that the value will be set by the environment, whereas the input value **nil** corresponds to no message being sent at the time and the

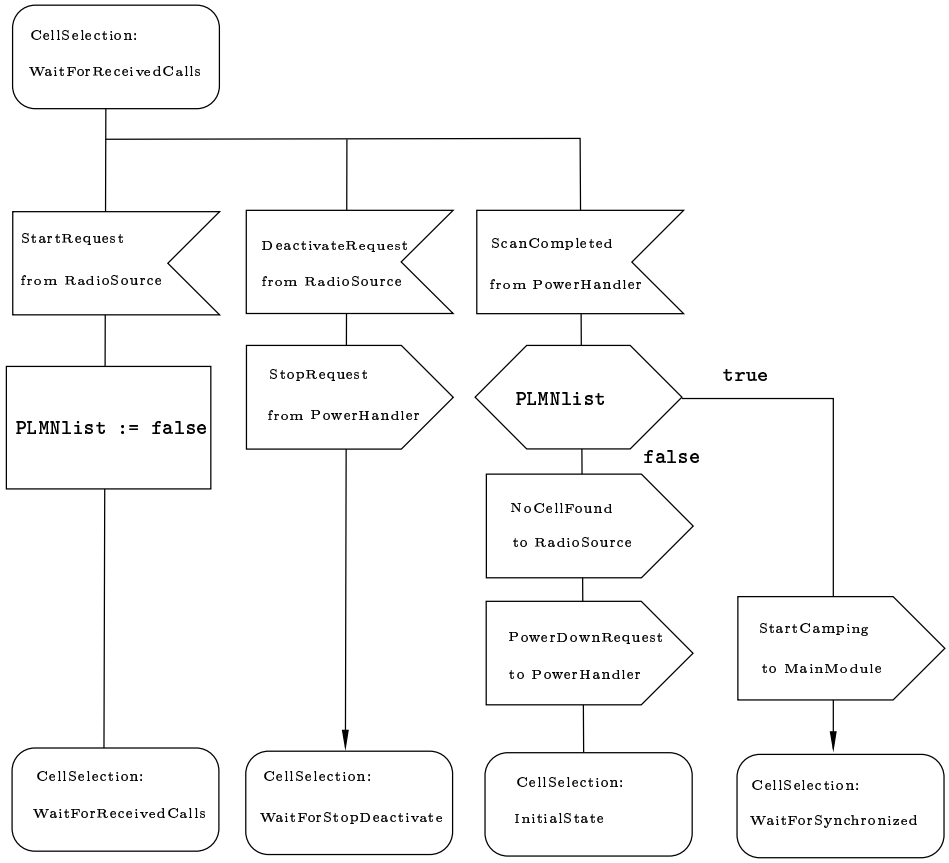


FIGURE 2.4: The SDL diagram corresponding to the graph of Figure 2.3

next value of the input is determined by the sender. When the sender puts a value $x$ into the input, this value is appended to the buffer and the input is set to **nil**as the message was handled. When the output is **nil** and there is a message for delivery (i.e. something comes out of the buffer), the element extracted from the buffer is set as the value of the output. As the operating system delivers a message $y$ from the process's output, the output value is reset to **nil** so that the message will not be redelivered at a later step.

The buffer overflow is represented by the last line of the table: the buffer is already full (i.e. contains $n$ entries), but inserting another one is attempted. Here the input and the buffer remain as they are, with only the output effected by the step. Thus the situtation of buffer overflow can be described with the following formula: $\mathbf{AG}^*(i \neq \mathbf{nil} \rightarrow \mathbf{X}(i = \mathbf{nil}))$.

Such buffer sequences can be modeled in various forms, one of which is shown in SMV syntax below the table. The code uses an array of length $n$ to represent the buffer, with the first element of the array being the front of the message queue. Incoming messages are put to the smallest empty buffer slot, that is, a slot with the value **nil**and the position $v$ being the first of those slots to appear from left to right.

| input | buffer | output | input | buffer | output |
|---|---|---|---|---|---|
| **nil** | $\langle\,\rangle$ | **nil** | | $\langle\,\rangle$ | **nil** |
| $x$ | $\langle\,\rangle$ | **nil** | **nil** | $\langle\,\rangle$ | $x$ |
| **nil** | $\langle x_1, \ldots, x_v \rangle$ | **nil** | | $\langle x_1, \ldots, x_{v-1} \rangle$ | $x_v$ |
| $x$ | $\langle x_1, \ldots, x_v \rangle$ | **nil** | **nil** | $\langle x, x_1, \ldots, x_{v-1} \rangle$ | $x_v$ |
| **nil** | $\langle\,\rangle$ | $y$ | | $\langle\,\rangle$ | |
| $x$ | $\langle\,\rangle$ | $y$ | **nil** | $\langle x \rangle$ | |
| **nil** | $\langle x_1, \ldots, x_v \rangle$ | $y$ | | $\langle x_1, \ldots, x_v \rangle$ | |
| $x$ | $\langle x_1, \ldots, x_v \rangle$ | $y$ | **nil** | $\langle x, x_1, \ldots, x_v \rangle$ | |
| $x$ | $\langle x_1, \ldots, x_v \rangle$ | $y$ | $x$ | $\langle x_1, \ldots, x_n \rangle$ | |

```
-- FOR EACH SLOT j SEPARATELY, HERE SHOWN FOR SLOT j = 2
ASSIGN next(buffer[2]) := case -- ACCORDING TO THE TABLE ABOVE,
            (input = nil) & !(output = nil) :  buffer[2];
            (input = nil) & (output = nil) :  buffer[3];
            !(input = nil) & !(output = nil) :
                      if !(buffer[1]=nil) & buffer[2]=nil then input
                      else buffer[2] fi;
            !(input = nil) & (output = nil) :
                      if buffer[2]=nil then nil
                      else if buffer[3]=nil then i
                      else buffer[3] fi fi;
            esac;
      ⋮
```

Similarly, variables need to be defined for the input `i` and the output `o` with respect to the intended behavior described by the table above.

The SMV code expects the filling of the buffer to be done in an orderly fashion such that if a slot in the position $j$ is empty, also all positions $k \leq j$ contain the value **nil**. This is assumed to hold for states reachable from an initially empty buffer using the above code for the buffer handling. This model does not exclude transitions from illegal configurations to other illegal configurations.

This code can also be represented in compact from with BDDs. As in this model the value of the buffer slot depends only on the values of the neighboring slots, a representation of linear to the buffer length $n$ is possible to construct.

# Bibliography


[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[BBF+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Ptrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools.* Springer Verlag, 2001.

[CDK00] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, third edition, 2000.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

[Cle90] Rance Cleaveland. Tableau-based model checking in the propositional $\mu$-calculus. *Acta Informatica*, 27(8):725–747, September 1990.

[CS01] Edmund M. Clarke and Bernd-Holger Schlingloff. *Model Checking.* Elsevier Science Publishers, 2001.

# Index


$\mu$, 24
$\nu$, 24
$\prec$, 5

accepting path, 16
admissible atom, 15
alternation depth, 28
alternation-free formula, 27
asynchronous communication, 32
atom, 15

bistable, 40
bounded buffer, 42
branching time logic, 2

closed leaf, 15
closure of a formula, 19
Computation Tree Logic, 5
concurrent system, 33
constraint, 12
critical section, 33

deadlock, 33
discrete time, 34
distributed system, 31

elementary Petri net, 35
eventuality, 18
eventuality sequence, 20

fairness, 12
fixed point, 25
frame, 1
free occurrence, 25
functional, 25

gfp, 25
global model checking, 2
greatest fixed point, 25
guard, 32

Hamiltonian path, 18

initial satisfiability, 2
interpretation of a model, 3
inverse image, 3
inverse reachability problem, 9

Knaster-Tarski fix-point theorem, 25
Kripke-model, 1

least fixed point, 25
lfp, 25
Linear Temporal Logic, 12, 17
livelock, 33
local model checking, 2

marking, 35
maximal path, 5
maximal set, 13
maximal tableau, 29
message passing, 32
modal logic, 2
model checking problem, 1
monotonic function, 25

natural model, 13, 17
nesting depth, 27
no-op, 34

open leaf, 15

parallel system, 34
parallel transition system, 35
path, 5
path quantifier, 5
Petri net, 35
positive normal form, 15
postset, 35
preset, 35
process priority, 33
propositional consistency, 13

recursion operator, 24
recursive descent, 3

self-fulfilling SCC, 20
separated formula, 18
separation property, 18
sequence valid, 12

shared variable program, 36
shared variables, 32
simple fairness, 12
spatially separate processes, 31
stabilization, 6
starvation, 34
Streett fairness, 12
strong fairness, 12
strongly connected component, 11
stuttering, 34
successful leaf, 29
synchronization, 34
synchronized communication, 32
synchronized system modules, 34
synchronous communication, 32

Tarjan's algorithm, 22
token, 35
transition system, 35
transitive closure, 5

union-continuous function, 25
universal satisfiability, 1
unsuccessful leaf, 30

weak fairness, 12