The explanation of this phenomenon is that the unrolling technique allows us to use prefixed points as starting points for the Tarski/Knaster fixpoint iteration. To see this, consider again the expansion of the equation system $[x \overset{\mu}{=} f(x,y,z), y \overset{\nu}{=} g(x,y,z), z \overset{\mu}{=} h(x,y,z)]$. When the outermost block, i.e. the equation $z \overset{\mu}{=} h(x,y,z)$ invokes a new iteration, we normally have to start the fixpoint iteration for $y \overset{\nu}{=} g(x,y,z)$ from $y = 1$. During the iteration, we would then obtain the values of the unrolled equation system, following the equations of the y_k's. However, as the right hand side g is monotonic, the values would now be greater than in the previous iteration. Hence, there is no need to restart the iteration, instead we can start from a prefixed point to evaluate the y_k's.

The effect is therefore very similar to the improvement of Theorem 3.22 where we considered the alternation of fixpoints as a criterion to restart an iteration. A complete unrolling of the fixpoints is however not useful in practice, and normally the fixpoints are reached much earlier. An implementation should therefore generate the unrolled blocks on-the-fly, and should evaluate these whenever necessary.

3.7 Computing Fair States and Fair Paths (Counterexamples)

In the previous sections, we have developed algorithms for checking μ-calculus formulas on Kripke structures. For example, we have seen that the set of states $\mathcal{S}_{\mathsf{inf}}$ of a structure $\mathcal{K}$ that have at least one infinite path can be computed by evaluation of a μ-calculus formula, namely $\Phi_{\mathsf{inf}} :\equiv \nu x.\Diamond x$ (page 106). The set $\mathcal{S}_{\mathsf{inf}}$ is important for the evaluation of many other formulas (cf. Lemma 2.43 on page 81). In particular, we have the following equations that show how EX and AX can be defined by $\Diamond$ and $\Box$, respectively:

- $[\![\mathsf{EX}\varphi]\!]_{\mathcal{K}} = \mathsf{pre}^{\mathcal{R}}_{\exists}(\mathcal{S}_{\mathsf{inf}} \cap [\![\varphi]\!]_{\mathcal{K}})$, hence, $\mathsf{EX}\varphi = \Diamond(\Phi_{\mathsf{inf}} \wedge \varphi)$
- $[\![\mathsf{AX}\varphi]\!]_{\mathcal{K}} = \mathsf{pre}^{\mathcal{R}}_{\forall}((\mathcal{S} \setminus \mathcal{S}_{\mathsf{inf}}) \cup [\![\varphi]\!]_{\mathcal{K}})$, hence, $\mathsf{AX}\varphi = \Box(\Phi_{\mathsf{inf}} \rightarrow \varphi)$

Using the modified transition relation $\mathcal{R}_{\mathsf{inf}} := \mathcal{R} \cap (\mathcal{S} \times \mathcal{S}_{\mathsf{inf}})$, we can equivalently use the following equations:

- $[\![\mathsf{EX}\varphi]\!]_{\mathcal{K}} = \mathsf{pre}^{\mathcal{R}_{\mathsf{inf}}}_{\exists}([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\mathsf{AX}\varphi]\!]_{\mathcal{K}} = \mathsf{pre}^{\mathcal{R}_{\mathsf{inf}}}_{\forall}([\![\varphi]\!]_{\mathcal{K}})$

This means that, model checking CTL (see page 340) and other fixpoint-based temporal logics (without past modalities) like CTL^2 (see page 342) can be done by restricting the Kripke structure to the states $\mathcal{S}_{\mathsf{inf}}$ and then performing the usual fixpoint iterations, where it no longer matters if EX and $\Diamond$ is used.

The difference between EX and $\Diamond$ is therefore that EX considers only states that have at least one infinite path, while $\Diamond$ considers all states, including those that only have finite paths. The restriction of EX is due to the semantics

of the path quantifiers: recall that $\mathsf{E}\varphi$ holds in a state s iff there is an infinite path starting in s such that φ holds on that path.

In general, it is possible to further *restrict the set of paths that should be considered for quantification by particular path formulas* Φ: we simply have to replace $\mathsf{E}\varphi$ with $\mathsf{E}(\Phi \wedge \varphi)$. We then often write $\mathsf{E}_\Phi\varphi$ and $\mathsf{A}_\Phi\varphi$ as abbreviations for $\mathsf{E}(\Phi \wedge \varphi)$ and $\mathsf{A}(\Phi \rightarrow \varphi)$, respectively (cf. Theorem 5.13 on page 340).

The most important restriction of this kind is the restriction to *fair paths*. Recall that a path π is fair w.r.t. a set of fairness constraints $\mathcal{F} = \{F_1, \ldots, F_f\}$ iff every $F_i \in \mathcal{F}$ contains a state that appears infinitely often on π (cf. Definition 2.3 on page 47). Moreover, the set of fair states $\mathcal{S}_{\mathsf{fair}}$ is the set of states that have at least one fair path. The computation of the fair states $\mathcal{S}_{\mathsf{fair}}$ is a fundamental problem for the verification of reactive systems.

In this section, we consider this computation in detail. In the next section, it is first explained that there is no simple way to eliminate fairness constraints, e.g., by restricting the state sets of the Kripke structures. Then, we show how to specify the fair states by equivalent μ-calculus formulas, so that the model checking algorithms of this chapter can be used for their computation. Refined complexity analysis will moreover show that the complexities that we have derived in the previous sections are often too pessimistic. We then consider SCC enumeration algorithms that compute the strongly connected components (SCCs) of a Kripke structure one after another. In particular, we consider algorithms that represent the SCCs in a symbolic manner and show how the fair states problem can be solved by their aid. Finally, we apply these algorithms to the generation of counterexamples, which means at this stage to compute a fair path of a Kripke structure. The translations of temporal logics and automata to the μ-calculus as discussed in the following chapters can make use of these procedures to generate counterexamples in case these specifications do not hold on a structure.

3.7.1 Kripke Structures with Fairness Constraints

In the previous section, we have considered the restriction of the future modal operators $\Diamond$ and $\Box$ to infinite paths as required by the macro operators EX and AX, respectively. We have seen that the difference between these operators can be eliminated by cutting off all deadend paths of the Kripke structure, i.e., restricting the Kripke structure to the states $\mathcal{S}_{\mathsf{inf}}$.

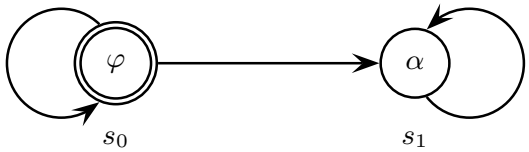

Fig. 3.21. Existence of unfair paths after restriction to $\mathcal{S}_{\mathsf{fair}}$

Unlike the restriction to infinite paths, the restriction to fair paths can not be simply solved by restricting the Kripke structure to the fair states. This can be seen by the example in Figure 3.21, where we assume that state s_1 should be infinitely often visited. The path that remains in the initial state s_0 is certainly an unfair one, but nevertheless we have $\mathcal{S}_{\mathsf{fair}} = \{s_0, s_1\} = \mathcal{S}$. For this reason, the formula $\mathsf{EG}\varphi$ holds in the structure of Figure 3.21, but is false when we add the fairness constraint $\{s_1\}$ (which equivalently means that we change the formula to $\mathsf{E}_{\mathsf{GF}\alpha}\mathsf{G}\varphi$).

Having seen that fairness constraints really add a new feature to Kripke structures, many authors introduced the notion of *fair Kripke structures*. As an alternative, other authors still put fairness restrictions to the specification side, i.e., as further restrictions to the path quantifiers. As the latter allows to consider specifications with and without fairness constraints on the same structure, we follow the second (and thus more general) approach.

In the remainder of this section, we consider several algorithms to compute $\mathcal{S}_{\mathsf{fair}}$. We consider a given structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ together with a set of fairness constraints $\mathcal{F} = \{\alpha_1, \ldots, \alpha_f\}$ where all α_i are propositional formulas. A path is then fair iff it satisfies the formula $\bigwedge_{i=1}^{f} \mathsf{GF}\alpha_i$, and therefore we have $\mathcal{S}_{\mathsf{fair}} = \left[\!\left[\mathsf{E}\bigwedge_{i=1}^{f} \mathsf{GF}\alpha_i\right]\!\right]_{\mathcal{K}}$.

There are also other notions of fairness that have been motivated by scheduling the execution of processes [179]. For example, an infinite computation is *unconditionally fair* [327] if every process is executed infinitely often, which can be expressed as $\bigwedge_{i=1}^{f} \mathsf{GF}\alpha_i$. A computation is *weakly fair* [317, 327] if every process that is persistently enabled is infinitely often executed. This can be expressed as $\bigwedge_{i=1}^{f} (\mathsf{FG}\mathit{enabled}_i \rightarrow \mathsf{GF}\mathit{execute}_i)$, or equivalently as $\bigwedge_{i=1}^{f} \mathsf{GF}(\mathit{enabled}_i \rightarrow \mathit{execute}_i)$. A computation is *strongly fair* [317, 327] if every process that is infinitely often enabled is also infinitely often executed. This can be expressed as $\bigwedge_{i=1}^{f} (\mathsf{GF}\mathit{enabled}_i \rightarrow \mathsf{GF}\mathit{execute}_i)$, or equivalently as $\bigwedge_{i=1}^{f} \mathsf{FG}\neg\mathit{enabled}_i \vee \mathsf{GF}\mathit{execute}_i$. Other names [327] for unconditional fairness, weak fairness, and strong fairness are *impartiality* , *justice* , and simply *fairness*.

It is easily seen that unconditional and weak fairness can both be written as a formula of the form $\bigwedge_{i=1}^{f} \mathsf{GF}\alpha_i$. In particular, we are interested in those states that have a path that satisfies this property, i.e. the states $\mathcal{S}_{\mathsf{fair}}$ as defined above. Strong fairness can not be written in this form and is a strictly stronger notion of fairness, as we will see in the chapter about ω-automata. Nevertheless, we can adapt the algorithms and techniques used to deal with weak fairness also to the case of strong fairness.

In the following, it is important to distinguish between *fair cycles*, *fair states*, and *fair SCCs*. We have already explained the definition of the fair states, i.e., those states that have at least one fair path. Fair SCCs and fair cycles are defined below.

Definition 3.43 (Strongly Connected Components and SCC-Quotient). *For every Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, we write $s \rightsquigarrow_{\mathcal{K}} s'$ if there is a (finite) path from state s to state s' in $\mathcal{K}$ or if $s = s'$. Clearly, the relation $\rightsquigarrow_{\mathcal{K}}$ is a preorder (i.e., it is reflexive and transitive), so that the following relation $\approx_{\mathsf{SCC}}$ is the corresponding equivalence relation*[6]*:*

$$s \approx_{\mathsf{SCC}} s' :\Leftrightarrow s \rightsquigarrow_{\mathcal{K}} s' \wedge s' \rightsquigarrow_{\mathcal{K}} s$$

The equivalence classes of $\approx_{\mathsf{SCC}}$ are called strongly connected components (SCCs), and in particular, we denote the equivalence class of a state s as $\mathsf{SCC}(s)$. Moreover, we define the forward and backward sets of a set $S \subseteq \mathcal{S}$ as $\mathsf{FW}(S) := \{s \in \mathcal{S} \mid \exists s' \in S.s' \rightsquigarrow_{\mathcal{K}} s\}$ and $\mathsf{BW}(S) := \{s \in \mathcal{S} \mid \exists s' \in S.s \rightsquigarrow_{\mathcal{K}} s'\}$, respectively.

As for every equivalence relation, we can build the quotient of $\mathcal{S}$ with respect to $\approx_{\mathsf{SCC}}$, which is used to define the SCC-quotient of the Kripke structure. Formally, this is an acyclic graph whose vertices are the SCCs of the Kripke structure, and there is an edge between two SCCs C_1 and C_2 if there are states $s_1 \in C_1$ and $s_2 \in C_2$ such that $(s_1, s_2) \in \mathcal{R}$.

For fairness constraints $\mathcal{F} = \{F_1, \ldots, F_f\} \subseteq 2^{\mathcal{S}}$, we say an SCC $\mathsf{SCC}(s)$ is fair w.r.t. $\mathcal{F}$ iff (1) $\mathsf{SCC}(s) \cap F_i \neq \{\}$ holds for every set $F_i \in \mathcal{F}$ and (2) if $\mathsf{SCC}(s)$ is a singleton, then additionally $(s, s) \in \mathcal{R}$ must hold. A tuple of states $(s_0, \ldots, s_{n-1})$ with $(s_i, s_{(i+1) \bmod n}) \in \mathcal{R}$ for $i \in \{0, \ldots, n-1\}$ is called a cycle of $\mathcal{K}$. A cycle $(s_0, \ldots, s_{n-1})$ of $\mathcal{K}$ is fair if $\{s_0, \ldots, s_{n-1}\} \cap F_i \neq \{\}$ holds for every set $F_i \in \mathcal{F}$.

Note that $\rightsquigarrow_{\mathcal{K}}$ induces a preorder on the set of states of $\mathcal{K}$, since $\rightsquigarrow_{\mathcal{K}}$ is reflexive and transitive. Hence, it follows that $\approx_{\mathsf{SCC}}$ is an equivalence relation. Moreover, it follows by definition of $\approx_{\mathsf{SCC}}$ and $\mathsf{SCC}(\{s\})$ that $\mathsf{SCC}(\{s\}) = \mathsf{FW}(\{s\}) \cap \mathsf{BW}(\{s\})$.

Moreover, as usual for the equivalence relation of a preorder, the SCC-quotient of $\mathcal{K}$ is a partially ordered set, where reachability is the partial order relation. For this reason, the SCC-quotient is a directed acyclic graph. The SCCs that do not have predecessors and successors in the SCC-quotient are called *initial* and *terminal* SCCs.

If $\mathsf{SCC}(s) \cap F_i \neq \{\}$ holds for every set $F_i \in \mathcal{F}$, it is easily seen that we can find a cycle in $\mathsf{SCC}(s)$ that visits every F_i infinitely often. To see this, simply choose a state $s_i \in \mathsf{SCC}(s) \cap F_i$ for all i, and note that since all s_i belong to $\mathsf{SCC}(s)$ there is a path through the states s_1, s_2, …, s_f, and back to s_1. In case $\mathsf{SCC}(s) = \{s\}$ holds, since is also the case, since we additionally required that then $(s, s) \in \mathcal{R}$ holds. Hence, checking containment of a fair cycle is fairly easy for weak fairness:

Lemma 3.44 (Fair Cycle Containment for Weak Fairness). *For every SCC C of a structure $\mathcal{K}$ with fairness constraints $\mathcal{F} = \{F_1, \ldots, F_f\} \subseteq 2^{\mathcal{S}}$, the following holds: C contains a fair cycle iff C is fair, i.e., if $F_i \cap C \neq \{\}$ for all $i \in \{1, \ldots, f\}$.*

[6] For every preorder, one can define an equivalence relation in the above way. See also page 56-59.

Hence, $\mathcal{S}_{\mathsf{fair}}$ contains at least all fair SCCs. Additionally, $\mathcal{S}_{\mathsf{fair}}$ contains those 'unfair' SCCs that can reach a fair SCC. As reachability is also a property of SCCs, it follows that having a fair path is a property of an entire SCC: either all states of a SCC have a fair path or none of them has a fair path. As a consequence, $\mathcal{S}_{\mathsf{fair}}$ *is a union of SCCs*. The same holds for forward and backward sets.

Lemma 3.45 (SCC-Closed Sets). *For every Kripke structure $\mathcal{K}$ with states $\mathcal{S}$, fairness constraints $\mathcal{F} = \{F_1, \ldots, F_f\} \subseteq 2^{\mathcal{S}}$, and a set of states $S \subseteq \mathcal{S}$, the sets $\mathcal{S}_{\mathsf{fair}}$, $\mathsf{FW}(S)$, and $\mathsf{BW}(S)$, are all SCC-closed, i.e., these sets are unions of SCCs of $\mathcal{K}$. Moreover, the terminal SCCs contained in $\mathcal{S}_{\mathsf{fair}}$, i.e., those SCCs contained in $\mathcal{S}_{\mathsf{fair}}$ that do not reach other SCCs in $\mathcal{S}_{\mathsf{fair}}$, are all fair SCCs.*

Clearly, it is possible to define *fair cycles and fair SCCs also for strong fairness constraints* $\bigwedge_{i=1}^{f} \mathsf{GF}\varphi_i \vee \mathsf{FG}\psi_i$ as follows:

$(*)$ $C \subseteq [\![\psi_i]\!]_{\mathcal{K}}$ or $C \cap [\![\varphi_i]\!]_{\mathcal{K}} \neq \{\}$ for every $i \in \{1, \ldots, f\}$

Clearly, we have $C \subseteq \left[\!\!\left[\mathsf{E}\bigwedge_{i=1}^{f} \mathsf{GF}\varphi_i \vee \mathsf{FG}\psi_i\right]\!\!\right]_{\mathcal{K}}$ for every fair cycle in the above sense: The path $\pi^{(t)} := s_{t \bmod n}$ is a witness to prove the inclusion. Of course, the inclusion $C \subseteq \left[\!\!\left[\mathsf{E}\bigwedge_{i=1}^{f} \mathsf{GF}\varphi_i \vee \mathsf{FG}\psi_i\right]\!\!\right]_{\mathcal{K}}$ may also hold for unfair cycles C, if the states of C can reach a fair cycle C' in the sense of $(*)$ (this is not different to weak fairness).

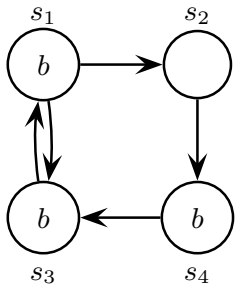

Fig. 3.22. Using strong fairness, unfair SCCs may still contain fair cycles

However, checking whether a SCC contains a fair cycle can not be directly reduced to the above condition $(*)$. As an example, consider the Kripke structure given in Figure 3.22 over the variables $\{a, b\}$ and the fairness constraint $\mathsf{GF}a \vee \mathsf{FG}b$. Since neither $\{s_1, s_2, s_3, s_4\} \subseteq [\![b]\!]_{\mathcal{K}} = \{s_2\}$ nor $\{s_1, s_2, s_3, s_4\} \cap [\![a]\!]_{\mathcal{K}} \neq \{\}$ holds, the entire SCC $\{s_1, s_2, s_3, s_4\}$ is not fair. Nevertheless, this unfair SCC contains the fair cycle (s_1, s_3).

It is however possible to still make use of condition $(*)$ to check containment of a fair cycle in a SCC C for strong fairness. If $(*)$ holds and C is not a singleton, then the SCC contains a fair cycle. If $(*)$ holds and $C = \{s\}$ holds

```
function HasWeaklyFairCycle(C, F)
  fair := true;
  while (F ≠ {}) ∧ fair do
    F := selectfrom(F);
    F := F \ {F};
    fair := fair ∧ (F ∩ C ≠ {})
  end;
  if |C| = 1 then {s} :≡ C; return fair ∧ (s, s) ∈ R end;
  return fair
end

function HasStronglyFairCycle(C, F)
  fair := true;
  C' := C;
  for all (F_φ, F_ψ) ∈ F do
    check := (C ⊆ F_ψ) ∨ (C ∩ F_φ ≠ {});
    fair := fair ∧ check;
    if ¬check then C' := C' ∩ F_ψ end;
  end;
  if |C| = 1 then {s} :≡ C; return fair ∧ (s, s) ∈ R end;
  if fair then return true end;
  C := PartitionSCC(C');
  hasCycle := true;
  while (C ≠ {}) ∧ hasCycle do
    C_i := selectfrom(C);
    C := C \ {C_i};
    hasCycle := hasCycle ∧ HasStronglyFairCycle(C_i, F)
  end;
  return hasCycle
end
```

Fig. 3.23. Checking containment of fair cycles in an SCC C

for some state s, then (s, s) is a fair cycle iff $(s, s) \in \mathcal{R}$ holds. Finally, if neither $|C| = 1$ nor $(*)$ holds, C may still contain a fair cycle C''. If condition $(*)$ failed due to a pair (φ_i, ψ_i), then we have $C \subseteq [\![\neg\varphi_i]\!]_{\mathcal{K}}$ and $C \cap [\![\neg\psi_i]\!]_{\mathcal{K}} \neq \{\}$. If a fair cycle C'' exists in C, then we conclude that $C'' \cap [\![\varphi_i]\!]_{\mathcal{K}} = \{\}$, and therefore that $C'' \subseteq [\![\psi_i]\!]_{\mathcal{K}}$ must hold. We therefore continue the search for a fair cycle in the intersection of C with all states $[\![\varphi_i]\!]_{\mathcal{K}}$ that contributed to the failure of condition $(*)$. This set is computed by the variable C' in the algorithm given in Figure 3.23. However, before the recursive descent, we have to partition C' into SCCs to establish the precondition of the algorithm.

Hence, we see that checking containment of a (strongly) fair cycle in a SCC can be done by partitioning the state set into SCCs. The fair states are then obtained by computing the union of those SCCs that can reach at least one SCCs that has a fair cycle.

3.7.2 SCC Partitioning by Depth-First Traversals

As outlined in the previous section, the set of states that have a fair path (regardless whether strong or weak fairness is considered) can be computed by partitioning the state set of a Kripke structure into its SCCs and checking which of these SCCs has a fair cycle according to the algorithms given in Figure 3.23. After this, we have to compute the union of all SCCs that can reach those SCCs that have a fair cycle.

For this reason, partitioning the state set into its strongly connected components is important for the computation of the fair states (although we will see in Section 3.7.4 that there are methods that can do without the SCC partitioning). It is straightforward to compute $\mathsf{SCC}(s)$ for a particular state: Assume that a state s is represented by the formula φ_s, i.e., that $[\![\varphi_s]\!]_{\mathcal{K}} = \{s\}$ holds. Then, we also write $\mathsf{SCC}(\varphi_s)$ for a μ-calculus formula that evaluates to $\mathsf{SCC}(s)$. We moreover extend the definitions of the forward and backward sets for arbitrary formulas φ and ψ (or state sets) as follows:

- $\mathsf{BW}(\psi) := \mu y.\, \psi \vee \Diamond y$
- $\mathsf{FW}(\psi) := \mu y.\, \psi \vee \overleftarrow{\Diamond}\, y$
- $\mathsf{BW}_\varphi(\psi) := \mu y.\, \psi \vee \varphi \wedge \Diamond y$
- $\mathsf{FW}_\varphi(\psi) := \mu y.\, \psi \vee \varphi \wedge \overleftarrow{\Diamond}\, y$

Note that $\mathsf{BW}(\psi)$ holds on the set of states that have a possibly finite path that leads to a state where ψ holds. $\mathsf{BW}_\varphi(\psi)$ additionally requires that φ must hold on every state of the (possibly finite) path up to the state where ψ holds. Similarly, $\mathsf{FW}(\psi)$ corresponds with the set of states that can be reached from a state where ψ holds, and $\mathsf{FW}_\varphi(\psi)$ additionally requires that φ must hold on every state of the path from the state where ψ held.

According to the definition of strongly connected components, it is easily seen that $\mathsf{SCC}(\varphi_s) = \mathsf{BW}(\varphi_s) \cap \mathsf{FW}(\varphi_s)$ holds. An minor alternative is $\mathsf{SCC}(\varphi_s) = \mathsf{FW}_{\mathsf{BW}(\varphi_s)}(\varphi_s)$, which is based on the observation that due to $\mathsf{SCC}(\varphi_s) \subseteq \mathsf{BW}(\varphi_s)$, one can restrict the additional computation of $\mathsf{FW}(\varphi_s)$ to the states in $\mathsf{BW}(\varphi_s)$ after this set has been computed (similarly, we have $\mathsf{SCC}(\varphi_s) = \mathsf{BW}_{\mathsf{FW}(\varphi_s)}(\varphi_s)$).

In any case, we can compute $\mathsf{SCC}(\varphi_s)$ via an alternation-free μ-calculus formula, which can therefore be done in time $O(|\mathcal{K}|)$. However, having only the SCC of one state is not sufficient, since we need the complete partitioning of the state set, or at least a method to enumerate all SCCs until we find one that has a fair cycle and that is reachable from the initial states.

A naive approach to compute the SCC-quotient of a structure $\mathcal{K}$ would be to repeatedly select a state $s \in \mathcal{S}$, compute $\mathsf{SCC}(\varphi_s)$ in time $O(|\mathcal{K}|)$ via the above fixpoint characterizations, and to recursively apply this procedure to the structure restricted to the states $\mathcal{S} \setminus \mathsf{SCC}(\varphi_s)$. A rough estimation of the runtime would then yield an overall complexity of $O(|\mathcal{K}|\,|\mathcal{S}|)$. However,

if symbolic representations should be used, we can not use the Cleaveland-Steffen algorithm and rather obtain $O(|\mathcal{K}|\,|\mathcal{S}|^2)$ as an upper bound for the runtime.

A simple way to obtain the SCC-partition is to apply iterative squaring to compute the transitive closure $\mathcal{R}^*$ of the transition relation. This is done by computing the relations $\mathcal{R}^0 := \mathcal{R}$ and $\mathcal{R}^{2i} := \mathcal{R}^i \circ \mathcal{R}^i := \{(s,s') \mid \exists s''.(s,s'') \in \mathcal{R}^i \wedge (s'',s') \in \mathcal{R}\}$ until $\mathcal{R}^{2i} = \mathcal{R}^i =: \mathcal{R}^*$ holds (as this is a fixpoint over a transition relation, we can not represent this fixpoint by a μ-calculus formula). Then, it is easily seen that $\mathsf{FW}(\varphi_s) = \mathsf{pre}_\exists^{\mathcal{R}^*}(\varphi_s)$ and $\mathsf{BW}(\varphi_s) = \mathsf{suc}_\exists^{\mathcal{R}^*}(\varphi_s)$, and therefore that $\mathsf{SCC}(\varphi_s) = \mathsf{pre}_\exists^{\mathcal{R}^*}(\varphi_s) \wedge \mathsf{suc}_\exists^{\mathcal{R}^*}(\varphi_s)$ holds. This procedure has been used in [500, 501] but is commonly considered to be inefficient [431], at least when used with BDDs[7].

Concerning the best case complexity, there are better algorithms to compute all SCCs of a structure[8]. These are based on depth first traversals of the structure, where each state is attached with certain numbers to identify the SCCs. The most popular ones of these algorithms are those of Tarjan [256, 491], Kosaraju/Sharir (published first in [462]), and of Jiang [272]. These algorithms are all based on one or more depth first searches, therefore run in time $O(|\mathcal{K}|)$, and are therefore optimal.

To understand these algorithms, consider the general depth first traversal procedure given in Figure 3.24. During the depth first traversal of the reachable states of a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ the procedure attaches two numbers, namely *Start*$[s]$ and *Term*$[s]$ to every reachable state s. These numbers are obtained by maintaining two counters c_1 and c_2, that are incremented each time the depth first traversal is started for a new state (incrementing c_1) or terminates the call for a state (incrementing c_1) after its reachable states have been explored. *Start*$[s]$ is the content of c_1 (after incrementation) c_1 when state state s is first entered, and *Term*$[s]$ is the the content of c_2 (after incrementation) when all states that are reachable from state s have been explored. Note that a state s has already been visited iff *Start*$[s] > 0$ holds, and the states that are reachable from s have all been explored iff *Term*$[s] > 0$ holds. Hence, after the depth first traversal, the arrays *Start* and *Term* can finally be viewed as functions that map the (reachable) states $\mathcal{S}$ to numbers $\{1, \ldots, |\mathcal{S}|\}$. For this reason, both numbers induce a total order on the states.

During the depth first traversal, the algorithm moreover computes a partition $\mathcal{R} = \mathcal{R}_{\mathsf{tree}} \cup \mathcal{R}_{\mathsf{fwd}} \cup \mathcal{R}_{\mathsf{bwd}} \cup \mathcal{R}_{\mathsf{cross}}$ of the transition relation $\mathcal{R}$. $\mathcal{R}_{\mathsf{tree}}$ is the set of transitions that have been followed by the depth first traversal, and therefore the restriction of $\mathcal{K}$ to $\mathcal{R}_{\mathsf{tree}}$ yields a so-called *spanning forest* $\mathcal{K}_{\mathsf{tree}}$ of $\mathcal{K}$ that corresponds with the recursion tree of the depth first traversal. We

[7] One reason for this inefficiency is that three sets of variables are required to compute the relation product with BDDs: $\mathcal{R}^{2i}(\mathbf{x},\mathbf{y}) := \exists \mathbf{z}.\mathcal{R}^i(\mathbf{x},\mathbf{z}) \wedge \mathcal{R}^i(\mathbf{z},\mathbf{y})$.

[8] We prefer to still talk about Kripke structures, although the problems are normally stated for arbitrary graphs.

```
function DFS(s, c1, c2, Start, Term, R_tree, R_fwd, R_bwd, R_cross)
  c1 := c1 + 1;
  Start[s] := c1;
  for all s' ∈ suc_∃^R({s}) do
    if Start[s'] = 0 then
      R_tree := R_tree ∪ {(s, s')};
      (c1, c2, Start, Term, R_tree, R_fwd, R_bwd, R_cross)
          := DFS(s', c1, c2, Start, Term, R_tree, R_fwd, R_bwd, R_cross)
    elseif Term[s'] = 0 then
      R_bwd := R_bwd ∪ {(s, s')};
    elseif Start[s] < Start[s'] then
      R_fwd := R_fwd ∪ {(s, s')};
    else
      R_cross := R_cross ∪ {(s, s')};
    end
  end;
  c2 := c2 + 1;
  Term[s] := c2;
  return (c1, c2, Start, Term, R_tree, R_fwd, R_bwd, R_cross)
end

function DepthFirstSearch()
  for all s ∈ S do
    Start[s] := 0;
    Term[s] := 0
  end;
  c1 := 0; c2 := 0;
  Init := {};
  R_tree := {}; R_fwd := {}; R_bwd := {}; R_cross := {};
  for all s ∈ I do
    if Start[s] = 0 then
      Init := Init ∪ {s};
      (c1, c2, Start, Term, R_tree, R_fwd, R_bwd, R_cross)
          := DFS(s, c1, c2, Start, Term, R_tree, R_fwd, R_bwd, R_cross)
    end
  end;
  return (Init, Start, Term, R_tree, R_fwd, R_bwd, R_cross)
end
```

Fig. 3.24. Depth-First Search with First-Visit and Completion Numbers

write $s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$ if state s' can be reached from state s by transitions of $\mathcal{R}_{\mathsf{tree}}$, i.e. s' lies on a branch of the subtree starting in s. Using this notation, the following holds for a transition $(s, s') \in \mathcal{R}$:

- $(s, s') \in \mathcal{R}_{\mathsf{fwd}} \Leftrightarrow \exists s''.s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'' \wedge s'' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$
- $(s, s') \in \mathcal{R}_{\mathsf{bwd}} \Leftrightarrow s' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s \vee (s = s')$
- $(s, s') \in \mathcal{R}_{\mathsf{cross}} \Leftrightarrow s \not\rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s' \wedge s' \not\rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$

Hence, $(s, s') \in \mathcal{R}_{\mathsf{fwd}}$ abbreviates a sequence of $\mathcal{R}_{\mathsf{tree}}$ transitions, $(s, s') \in \mathcal{R}_{\mathsf{bwd}}$ either points the end s back to the beginning s' of such a sequence or is a self-loop, and $(s, s') \in \mathcal{R}_{\mathsf{cross}}$ connects different branches of $\mathcal{K}_{\mathsf{tree}}$. To see that the algorithm correctly classifies transitions into $\mathcal{R}_{\mathsf{tree}}$, $\mathcal{R}_{\mathsf{fwd}}$, $\mathcal{R}_{\mathsf{bwd}}$ and $\mathcal{R}_{\mathsf{cross}}$, we have to inspect the code of the for-loop in function DFS in the exploration of state s:

- $(s, s') \in \mathcal{R}$ is inserted in $\mathcal{R}_{\mathsf{tree}}$ if $\mathit{Start}[s'] = 0$ holds. In this case, DFS is recursively called for state s', and as this is the only case that may lead to recursive calls, we conclude that $s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$ holds iff the call $\mathsf{DFS}(s', \ldots)$ has been initiated by $\mathsf{DFS}(s, \ldots)$. Moreover, we conclude that $\mathit{Start}[s] < \mathit{Start}[s']$ and $\mathit{Term}[s] > \mathit{Term}[s']$ will finally hold, since $\mathsf{DFS}(s', \ldots)$ *is called after* $\mathsf{DFS}(s, \ldots)$ and $\mathsf{DFS}(s', \ldots)$ *will terminate before* $\mathsf{DFS}(s, \ldots)$.
- $(s, s') \in \mathcal{R}$ is inserted in $\mathcal{R}_{\mathsf{bwd}}$ if $\mathit{Start}[s'] > 0$ and $\mathit{Term}[s'] = 0$ holds. The latter means that the call $\mathsf{DFS}(s', \ldots)$ is still active. One reason for this could be that $s = s'$ holds, otherwise the current call $\mathsf{DFS}(s, \ldots)$ has been caused by $\mathsf{DFS}(s', \ldots)$ so that we have $s' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$. In each case, the transition $(s, s') \in \mathcal{R}$ points back to a state s' that is on the recursion stack. Moreover, we conclude that $\mathit{Start}[s] \geq \mathit{Start}[s']$ and $\mathit{Term}[s] \geq \mathit{Term}[s']$ will finally hold, since $\mathsf{DFS}(s, \ldots)$ *is called after* $\mathsf{DFS}(s', \ldots)$ and $\mathsf{DFS}(s, \ldots)$ *will terminate before* $\mathsf{DFS}(s', \ldots)$ (equality is due to the self-loop case).
- $(s, s') \in \mathcal{R}$ is inserted in $\mathcal{R}_{\mathsf{fwd}}$ if $\mathit{Start}[s'] > 0$, $\mathit{Term}[s'] > 0$, and $\mathit{Start}[s] < \mathit{Start}[s']$ holds. The first two conditions mean that there has already been a call $\mathsf{DFS}(s', \ldots)$ that already terminated, and the third condition means that $\mathsf{DFS}(s', \ldots)$ has been called after $\mathsf{DFS}(s, \ldots)$. As the call $\mathsf{DFS}(s, \ldots)$ is still active, we conclude that $s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$ must hold, i.e., that during the traversal of the subtree of s the procedure already visited s'. Since this can not have been done due to transition (s, s') (which is explored right now), there must be another state s'' with $s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s''$ and $s'' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$. In addition to $\mathit{Start}[s] < \mathit{Start}[s']$, we conclude that $\mathit{Term}[s] > \mathit{Term}[s']$ will finally hold, since $\mathsf{DFS}(s', \ldots)$ already terminated.
- $(s, s') \in \mathcal{R}$ is inserted in $\mathcal{R}_{\mathsf{cross}}$ if $\mathit{Start}[s'] > 0$, $\mathit{Term}[s'] > 0$, and $\mathit{Start}[s] > \mathit{Start}[s']$ holds. The first two conditions mean that there has already been a call $\mathsf{DFS}(s', \ldots)$ that already terminated, and the third condition means that $\mathsf{DFS}(s', \ldots)$ has been called before $\mathsf{DFS}(s, \ldots)$. We can conclude that there must be some state s'' with $s'' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$, $s'' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$, but neither $s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$ nor $s' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$ holds. Clearly, we moreover will finally have $\mathit{Term}[s] > \mathit{Term}[s']$.

An example for the depth first traversal is given in Figure 3.25, and Figure 3.26 show the general situations for forward, backward and cross transitions. Of course, the depth first traversal depends on the representation of the Kripke structure, since the representation may determine the ordering for the enumeration of the states and transitions of $\mathcal{K}$.

Note that it is not possible that $\mathit{Start}[s] < \mathit{Start}[s']$ and $\mathit{Term}[s] < \mathit{Term}[s']$ holds. There may be states s, s' with these properties (consider s_1 and s_4 in

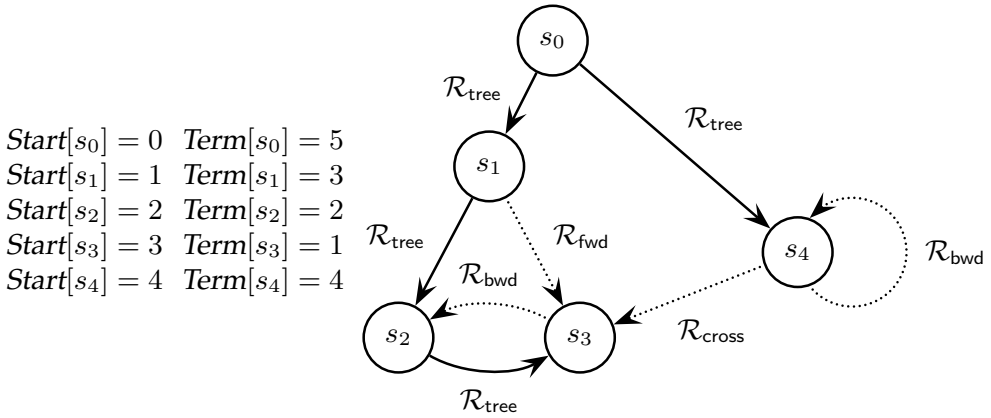

Fig. 3.25. Illustration of the Depth First Traversal

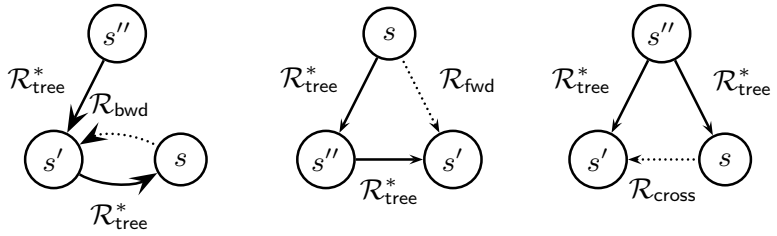

Fig. 3.26. Illustration of $\mathcal{R}_{\mathsf{fwd}}$, $\mathcal{R}_{\mathsf{bwd}}$ and $\mathcal{R}_{\mathsf{cross}}$ transitions

Figure 3.25), but it is not possible that these states are connected by a transition.

By the above considerations, we have proved the first eight points of the lemma below. The remaining two points are consequences of the fact that $\mathcal{R}_{\mathsf{tree}}$ forms a forest of spanning trees that directly corresponds with the recursions tree of the depth first traversal. Since each state is explored exactly once, it follows that each state has at most one predecessor in $\mathcal{R}_{\mathsf{tree}}$ (there may be none iff the state is a root).

Lemma 3.46 (Correctness of DepthFirstSearch**).** *If procedure* DepthFirstSearch *as given in Figure 3.24 has been applied to a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, the following holds for arbitrary transitions $(s, s') \in \mathcal{R}$ between reachable states s, s', where the Kripke structure $\mathcal{K}_{\mathsf{tree}}$ is the structure $\mathcal{K}$ restricted to the transitions $\mathcal{R}_{\mathsf{tree}}$:*

- $\mathit{Start}[s] < \mathit{Start}[s'] \wedge \mathit{Term}[s] > \mathit{Term}[s'] \Leftrightarrow (s, s') \in \mathcal{R}_{\mathsf{fwd}} \cup \mathcal{R}_{\mathsf{tree}}$
- $\mathit{Start}[s] \geq \mathit{Start}[s'] \wedge \mathit{Term}[s] \leq \mathit{Term}[s'] \Leftrightarrow (s, s') \in \mathcal{R}_{\mathsf{bwd}}$
- $\mathit{Start}[s] > \mathit{Start}[s'] \wedge \mathit{Term}[s] > \mathit{Term}[s'] \Leftrightarrow (s, s') \in \mathcal{R}_{\mathsf{cross}}$
- $\mathit{Start}[s] < \mathit{Start}[s'] \wedge \mathit{Term}[s] < \mathit{Term}[s']$ *can not hold for* $(s, s') \in \mathcal{R}$
- $(s, s') \in \mathcal{R}_{\mathsf{fwd}} \cup \mathcal{R}_{\mathsf{tree}} \Leftrightarrow s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$
- $(s, s') \in \mathcal{R}_{\mathsf{fwd}} \Leftrightarrow \exists s''. s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'' \wedge s'' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$
- $(s, s') \in \mathcal{R}_{\mathsf{bwd}} \Leftrightarrow s' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s \vee (s = s')$
- $(s, s') \in \mathcal{R}_{\mathsf{cross}} \Leftrightarrow s \not\rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s' \wedge s' \not\rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$
- $s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$ *implies* $s' \not\rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$
- *for every state s', there is at most one state s with* $(s, s') \in \mathcal{R}_{\mathsf{tree}}$

The last point is proved as follows: assume that $(s', s) \in \mathcal{R}_{\mathsf{tree}}$ and $(s'', s) \in \mathcal{R}_{\mathsf{tree}}$ hold. By the above characterization of $\mathcal{R}_{\mathsf{tree}}$, we then have $\mathit{Start}[s'] < \mathit{Start}[s]$, $\mathit{Term}[s'] > \mathit{Term}[s]$, $\mathit{Start}[s''] < \mathit{Start}[s]$, and $\mathit{Term}[s''] > \mathit{Term}[s]$. Without loss of generality, assume that $\mathit{Start}[s''] < \mathit{Start}[s']$ holds, which implies $\mathit{Term}[s''] > \mathit{Term}[s']$ (by the fourth point of the above lemma). Hence, we have by the first point $s'' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$. However, this implies $(s'', s) \in \mathcal{R}_{\mathsf{fwd}}$, and thus $(s'', s) \notin \mathcal{R}_{\mathsf{tree}}$.

In the following, we show how the above information can be used to compute the SCCs of a structure. Clearly $(s, s') \in \mathcal{R}_{\mathsf{bwd}}$ directly implies that s and s' belong to the same SCC. However, the converse may not hold, so that membership in $\mathcal{R}_{\mathsf{bwd}}$ is not the SCC equivalence relation. To cope with these problems, we need the following definitions:

Definition 3.47. *If procedure* DepthFirstSearch *as given in Figure 3.24 has been applied to a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, we have computed the starting and terminating numbers of every state. We extend these numbers to arbitrary sets of states as follows:*

- $\mathit{Start}(S) := \min\{\mathit{Start}[s] \mid s \in S\}$
- $\mathit{Term}(S) := \max\{\mathit{Term}[s] \mid s \in S\}$

For every SCC C of $\mathcal{K}$, there is a uniquely determined state $s \in C$ such that $\mathit{Start}(C) = \mathit{Start}[s]$ holds (s is that state of C that has been visited first). In the following, we denote this state as $\mathit{Root}(C)$, and also write $\mathit{Root}(s)$ instead of $\mathit{Root}(\mathsf{SCC}(s))$ for arbitrary states s.

$\mathit{Start}(S)$ is recursion step where the first state of S is explored, and $\mathit{Term}(S)$ is the last termination step, where the exploration of a state in S is explored. The above definition is motivated by the way DFS traverses an SCC C of $\mathcal{K}$. Clearly, there is a uniquely determined state $\mathit{Root}(C)$ of C that is visited first. It may be the case that DFS fully explores C before exploring another state $s' \notin C$. In this case, the states of $C = \{s_1, \ldots, s_n\}$ form a segment of a branch in $\mathcal{K}_{\mathsf{tree}}$, i.e., we have $(s_i, s_{i+1}) \in \mathcal{R}_{\mathsf{tree}}$ for $i \in \{1, \ldots, n-1\}$. Moreover, the call of DFS to $\mathit{Root}(C)$ remains active until all states of C have been visited.

If not all states of a SCC C are explored one after the other, DFS may explore some states of C after entering the SCC via $\mathit{Root}(C)$, but may leave C by an $\mathcal{R}_{\mathsf{tree}}$ transition from a state $s \in C$ to a state $s' \notin C$ before C has been fully explored. Note that the call $\mathsf{DFS}(s', \ldots)$ can not explore any further states of C, since otherwise a path from s' to a state in C would exists, and therefore $s' \in C$ would hold. Hence, if the search $\mathsf{DFS}(s', \ldots)$ terminates, no further states of C have been explored, and therefore $\mathsf{DFS}(\mathit{Root}(s), \ldots)$ can still not terminate. However, it may be the case that $\mathsf{DFS}(s, \ldots)$ terminates: As an example, consider Figure 3.28 with $s = s_3$, $s' = s_4$. Nevertheless, as C has not yet been fully explored, there must be some state $s'' \in C$ whose call $\mathsf{DFS}(s'', \ldots)$ is still active. Hence, DFS can continue the search in C via this state $s'' \in C$. This means that a transition (s'', s''') is added to $\mathcal{R}_{\mathsf{tree}}$ with $s''' \in C$.

The above situation may be repeated several times, i.e., after adding $\mathit{Root}(C)$ to the recursion tree $\mathcal{K}_{\mathsf{tree}}$, a branch from $\mathit{Root}(C)$ over states in C is followed. This branch of $\mathcal{R}_{\mathsf{tree}}$ may be continued to states that do not belong to C. In this case, the branch will not contain further states of C. If the depth first traversal comes back, it may create further branches starting with a prefix in C to the already available branches. At the end, the states of the SCC form a prefix tree of a subtree with root $\mathit{Root}(C)$, i.e., all branches of the spanning tree rooted in $\mathit{Root}(C)$ have a possibly empty prefix that belongs to C. As examples, consider Figures 3.27 and 3.28.

Lemma 3.48 (SCC-Segmentation of Branches of the Spanning Trees). *If procedure* DepthFirstSearch *as given in Figure 3.24 has been applied to a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, the following holds for arbitrary states $s, s' \in \mathcal{S}$ and the Kripke structure $\mathcal{K}_{\mathsf{tree}}$ restricted to the transitions $\mathcal{R}_{\mathsf{tree}}$:*

- *$\mathit{Root}(C) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$ for every state $s \in C$*
- *$\mathit{Term}(C) = \mathit{Term}[\mathit{Root}(C)]$*
- *$s \in C$ iff $\mathit{Root}(C) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$ and that path does not visit other roots.*

Hence, every SCC C forms a segment of some branch of $\mathcal{K}_{\mathsf{tree}}$.

Proof. The first two points are essentially proved by the explanations given before the lemma. The key observation is that once the exploration of a SCC C has been started by calling DFS for $\mathit{Root}(C)$, the DFS search of C may be interrupted in that DFS leaves C from a state $s \in C$ to a state $s' \notin C$. However, the exploration of s' can not visit any state in C, since otherwise $s' \in C$ would follow. Hence, when DFS returns from s', it can continue the search from some state $s'' \in C$ by adding further edges (s'', s''') to $\mathcal{R}_{\mathsf{tree}}$. As a result, the first two points of the lemma follow.

The last point is a bit more difficult to prove. We consider two implications:

$\Rightarrow$: Suppose $s \in C$. By the first point, we already know that $\mathit{Root}(C) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$ holds. Now assume, that this path would visit another root s'. Thus, we have $\mathit{Root}(C) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$. However, then $s' \in C$ follows, since due to $s, \mathit{Root}(C) \in C$ there is a path from s to $\mathit{Root}(C)$. As each SCC C has a uniquely determined root, it follows that $\mathit{Root}(C) = s'$ must hold.

$\Leftarrow$: Suppose $\mathit{Root}(C) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$ holds and that this path does not visit any other root than $\mathit{Root}(C)$. We have to prove that then $s \in C$ holds, i.e., that $\mathit{Root}(C) = \mathit{Root}(\mathsf{SCC}(s))$ holds. As $\mathit{Root}(C)$ is the only root on $\mathit{Root}(C) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$, either $\mathit{Root}(C) = \mathit{Root}(\mathsf{SCC}(s))$ holds or $\mathit{Root}(\mathsf{SCC}(s))$ is not on this path. We prove that the second case leads to a contradiction, and thus the first case must hold. By the first point of the lemma, we have $\mathit{Root}(\mathsf{SCC}(s)) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$, and since every state s has a unique predecessor in $\mathcal{R}_{\mathsf{tree}}$, we must either have $\mathit{Root}(\mathsf{SCC}(s)) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} \mathit{Root}(C) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$ or $\mathit{Root}(C) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} \mathit{Root}(\mathsf{SCC}(s)) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$. The latter can not hold, since we

assumed that the path $\mathit{Root}(C) \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$ does not visit other root. However, also the former case leads to a contradiction, since it implies that $\mathit{Root}(C) \in \mathsf{SCC}(s)$, and thus that $\mathit{Root}(C) = \mathit{Root}(\mathsf{SCC}(s))$, which we assume to be false.

□

Hence, $\mathit{Root}(C)$ is the state that is entered first by DFS and whose call to DFS terminates last (compared to the other states in C). The last point of the above lemma is the crucial observation to the computation of SCCs via the depth first traversal: SCCs are prefixes of the branches of the subtree rooted in $\mathit{Root}(C)$. An illustration of this is given in Figure 3.27, where the upper part shows a Kripke structure $\mathcal{K}$ (some transitions were dotted to make the SCCs more visible), and the lower part shows a spanning tree obtained by a depth first traversal. It can be seen that the SCCs correspond with segments of the branches. Figure 3.27 shows that SCCs may not only be a segment of single branch, but may form a finite subtree of the spanning tree.

The idea of Tarjan's algorithm to compute the SCCs is to check before termination of DFS *whether the currently explored state is a root: If we can identify roots on the branches, we find the SCCs in between these roots.*

In the following, we will establish a criterion that allows us to check whether a state that has been currently explored is a root or not. Ideally, we would like to assign the number $\mathit{RootStart}[s] = \mathit{Start}[\mathit{Root}(s)]$ to every state s. If we could do this, a state s could be recognized to be a root iff $\mathit{RootStart}[s] = \mathit{Start}[s]$ holds. Unfortunately, it is not straightforward to compute $\mathit{RootStart}[s]$ during the depth first traversal. In particular, the definition $\mathit{RootStart}[s] := \mathit{Start}[\mathit{Root}(s)]$ makes use of $\mathit{Root}(s)$ itself, so that it is useless for our purpose. Below, we find another way to identify roots of SCCs in that we define another number $\mathit{Low}[s]$:

Definition 3.49 (Root Criterion for SCCs). *Using the numbers $\mathit{Start}[s]$ of the states $s \in \mathcal{S}$ of a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ that are computed by procedure* DepthFirstSearch*, we define:*

$$\mathit{Low}[s] = \min \left(\begin{array}{l} \{\mathit{Start}[s]\} \cup \\ \{\mathit{Start}[s'] \mid (s, s') \in \mathcal{R}_{\mathsf{bwd}}\} \cup \\ \{\mathit{Start}[s'] \mid (s, s') \in \mathcal{R}_{\mathsf{cross}} \wedge s' \in \mathsf{SCC}(s)\} \cup \\ \{\mathit{Low}[s'] \mid (s, s') \in \mathcal{R}_{\mathsf{tree}}\} \end{array} \right)$$

It is clear that for every state $\mathit{Low}[s] \leq \mathit{Start}[s]$ holds due to the above definition. Also, by definition, we conclude from $s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$ that $\mathit{Low}[s] \leq \mathit{Low}[s']$ must hold. As roots do neither have cross transitions to other states of their SCC nor backward transitions (since their *Start* number is minimal), it follows that for a root s, we have $\mathit{Low}[s] = \min\{\mathit{Low}[s'] \mid s' \in \mathsf{SCC}(s)\}$.

In the next lemma, we will prove that the equation $\mathit{Low}[s] = \mathit{Start}[s]$ holds for roots, and that this equation only holds for roots. For non-root states, $\mathit{Low}[s]$ may not be equal to $\mathit{Start}[\mathit{Root}(s)]$. Instead, we have the inequations $\mathit{Start}[\mathit{Root}(s)] \leq \mathit{Low}[s] < \mathit{Start}[s]$ (see Figure 3.28).

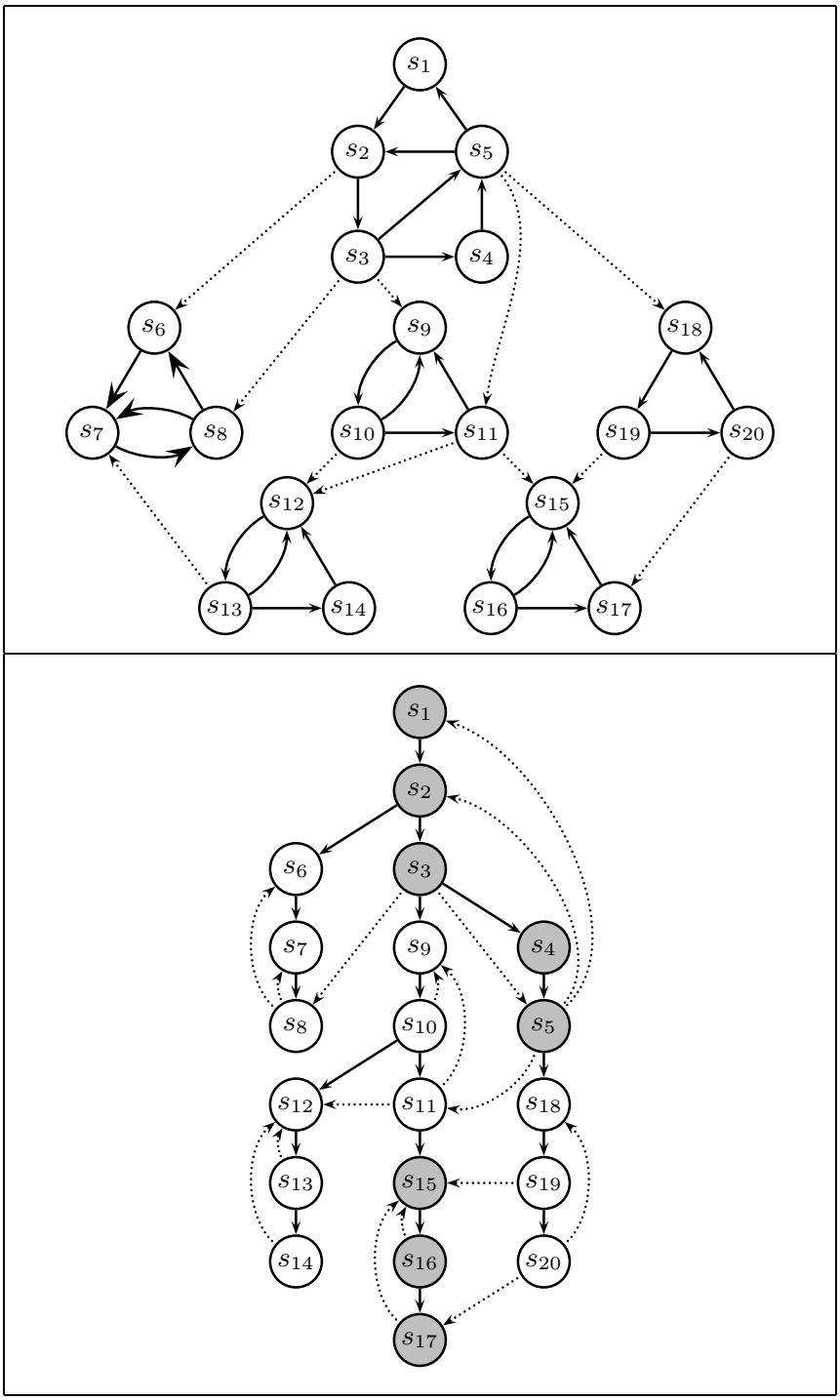

Fig. 3.27. Segmentation of branches of $\mathcal{K}_{\mathsf{tree}}$ by SCCs

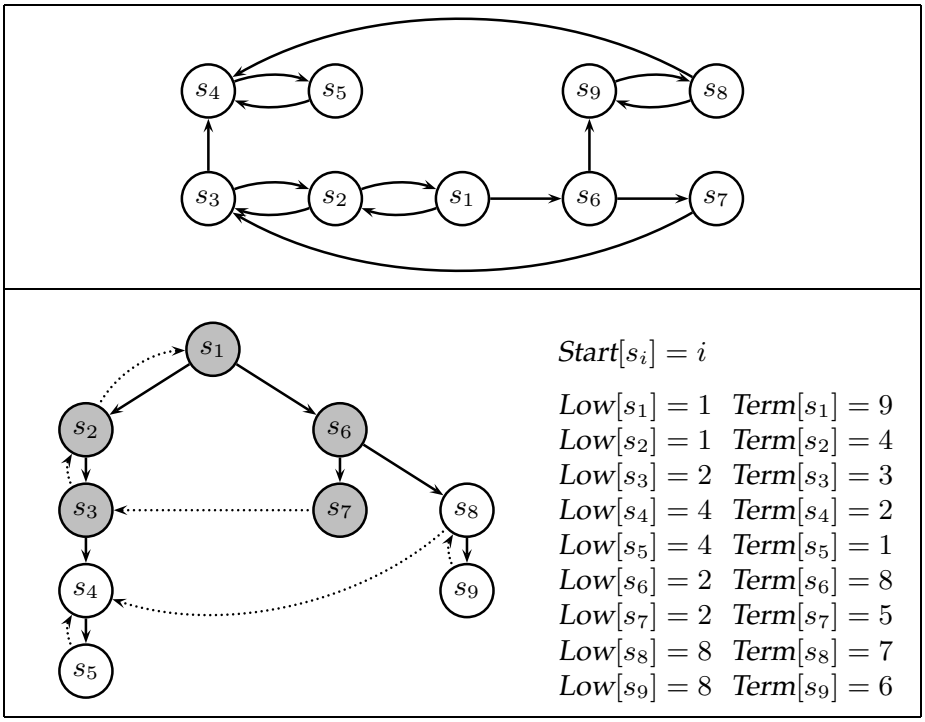

Fig. 3.28. Segmentation of branches of $\mathcal{K}_{\mathsf{tree}}$ by SCCs

Lemma 3.50 (Root Criterion for SCCs). *The following holds for all states s and their the numbers $Start[s]$ and $Low[s]$ as computed by procedure* DepthFirstSearch *and as defined in the above definition:*

- $s \neq Root(s)$ *implies that* $Low[s] < Start[s]$
- $Start[Root(s)] \leq Low[s]$
- $Low[s] = Start[s]$ *holds iff s is the root of its SCC.*

Proof. Suppose $s \neq Root(s) =: r$. By Lemma 3.48, we know that $r \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$ holds, and since $s \in \mathsf{SCC}(r)$, there is also a path from s to r. This path may consist of transitions of $\mathcal{R}_{\mathsf{tree}} \cup \mathcal{R}_{\mathsf{bwd}} \cup \mathcal{R}_{\mathsf{cross}}$ (note that we can neglect transitions in $\mathcal{R}_{\mathsf{fwd}}$, since these can be replaced by a sequence of transitions of $\mathcal{R}_{\mathsf{tree}}$). Let s' be the state where the first transition not in $\mathcal{R}_{\mathsf{tree}}$ must be taken (hence, we have $s \not\rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s''$). Then, we have

$$r \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s' \rightarrow_{\mathcal{K}_{\mathsf{bwd}} \cup \mathcal{K}_{\mathsf{cross}}} s'' \rightsquigarrow_{\mathcal{K}} r$$

with $(s', s'') \in \mathcal{R}_{\mathsf{bwd}} \cup \mathcal{R}_{\mathsf{cross}}$. We draw several consequences of the above relations: By Lemma 3.46, it follows that (1) $Start[s] < Start[s']$, (2) $Term[s] > Term[s']$, and (3) $Start[s''] < Start[s']$ (we can neglect the equality $Start[s''] = Start[s']$, when we assume a shortest path). Moreover, we have (4) $Start[s''] < Start[s]$, which is seen as follows: assume $Start[s] < Start[s'']$ would hold, then by (3) we would have $Start[s] < Start[s''] < Start[s']$. Due to point four in Lemma 3.46, we moreover have $Term[s'] \leq Term[s''] \leq Term[s]$, and thus

$s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s''$, which is against our assumption that (s', s'') is a transition that does not belong to $\mathcal{R}_{\mathsf{tree}}$. Hence, (4) holds. From the definition of *Low*, we directly derive

(5) $\forall s. \mathit{Low}[s] \leq \mathit{Start}[s]$
(6) $\forall s_1, s_2.\ (s_1, s_2) \in \mathcal{R}_{\mathsf{tree}} \rightarrow \mathit{Low}[s_1] \leq \mathit{Low}[s_2]$
(7) $\forall s_1, s_2.\ (s_1, s_2) \in \mathcal{R}_{\mathsf{bwd}} \rightarrow \mathit{Low}[s_1] \leq \mathit{Start}[s_2]$
(8) $\forall s_1, s_2.\ (s_1, s_2) \in \mathcal{R}_{\mathsf{cross}} \wedge s_2 \in \mathsf{SCC}(s_1) \rightarrow \mathit{Low}[s_1] \leq \mathit{Start}[s_2]$

Next, we obtain (9) $\mathit{Low}[s] \leq \mathit{Low}[s']$ from $s \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$ and (6). Moreover, we obtain (10) $\mathit{Low}[s'] \leq \mathit{Start}[s'']$ from $(s', s'') \in \mathcal{R}_{\mathsf{bwd}} \cup \mathcal{R}_{\mathsf{cross}}$ and (7) or (8), since $s'' \in \mathsf{SCC}(s')$. Hence, by (9), (10), and (4), we conclude $\mathit{Low}[s] < \mathit{Start}[s]$, which proves the proposition. Hence, if s is not a root, then $\mathit{Low}[s] < \mathit{Start}[s]$ holds, and thus $\mathit{Low}[s] = \mathit{Start}[s]$ can only hold if s is a root. That this is the case is proved next.

To prove that $\mathit{Start}[\mathit{Root}(s)] \leq \mathit{Low}[s]$ holds, we use induction on $\mathit{Term}[s]$. In the induction base, we consider the leftmost leaf s of the spanning tree. As there are no outgoing transitions of $\mathcal{R}_{\mathsf{tree}} \cup \mathcal{R}_{\mathsf{cross}}$, we have $\mathit{Low}[s] = \min(\{\mathit{Start}[s]\} \cup \{\mathit{Start}[s'] \mid (s, s') \in \mathcal{R}_{\mathsf{bwd}}\})$. We prove the proposition in this case by the first two cases below. Note that we do not make use of the induction hypothesis there. For the induction step, we consider all four cases that can determine the value $\mathit{Low}[s]$, and use the induction hypothesis in the fourth case only:

$\mathit{Low}[s] = \mathit{Start}[s]$: By definition of $\mathit{Root}(s)$, we have $\mathit{Start}[s] \geq \mathit{Start}[\mathit{Root}(s)]$, so that $\mathit{Start}[\mathit{Root}(s)] \leq \mathit{Low}[s]$ holds.

$\mathit{Low}[s] = \mathit{Start}[s']$: where $(s, s') \in \mathcal{R}_{\mathsf{bwd}}$ holds. By Lemma 3.48, we conclude (1) $\mathit{Root}(s') \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$, and by Lemma 3.46, we derive (2) $s' \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s$ from $(s, s') \in \mathcal{R}_{\mathsf{bwd}}$. Due to $(s, s') \in \mathcal{R}_{\mathsf{bwd}}$, we know that $s \in \mathsf{SCC}(s')$, and thus (3) $\mathit{Root}(s') = \mathit{Root}(s)$ holds. By Lemma 3.46, we conclude from (1) that (4) $\mathit{Start}[\mathit{Root}(s')] \leq \mathit{Start}[s']$ holds. As our case assumption is (5) $\mathit{Low}[s] = \mathit{Start}[s']$, we have

$$\mathit{Start}[\mathit{Root}(s)] \overset{(3)}{=} \mathit{Start}[\mathit{Root}(s')] \overset{(4)}{\leq} \mathit{Start}[s'] \overset{(5)}{=} \mathit{Low}[s]$$

$\mathit{Low}[s] = \mathit{Start}[s']$: where $(s, s') \in \mathcal{R}_{\mathsf{cross}}$ and $s' \in \mathsf{SCC}(s)$ holds. By Lemma 3.48, we conclude (1) $\mathit{Root}(s') \rightsquigarrow_{\mathcal{K}_{\mathsf{tree}}} s'$. By our assumption $s' \in \mathsf{SCC}(s)$, we have (2) $\mathit{Root}(s') = \mathit{Root}(s)$. By Lemma 3.46 and (1), we obtain (3) $\mathit{Start}[\mathit{Root}(s')] \leq \mathit{Start}[s']$ holds. As our case assumption is (4) $\mathit{Low}[s] = \mathit{Start}[s']$, we have again

$$\mathit{Start}[\mathit{Root}(s)] \overset{(2)}{=} \mathit{Start}[\mathit{Root}(s')] \overset{(3)}{\leq} \mathit{Start}[s'] \overset{(4)}{=} \mathit{Low}[s]$$

$\mathit{Low}[s] = \mathit{Low}[s']$: where $(s, s') \in \mathcal{R}_{\mathsf{tree}}$. Since we can exclude the first case above, we can furthermore assume (1) $\mathit{Low}[s] < \mathit{Start}[s]$. By Lemma 3.46, we conclude (2) $\mathit{Start}[s] < \mathit{Start}[s']$ and (3) $\mathit{Term}[s] > \mathit{Term}[s']$,

so that the induction hypothesis applies to state s'. Hence, we have (4) $\mathit{Start}[\mathit{Root}(s')] \leq \mathit{Low}[s']$, and to sum up

$$\mathit{Start}[\mathit{Root}(s')] \overset{(4)}{\leq} \mathit{Low}[s'] = \mathit{Low}[s] \overset{(1)}{<} \mathit{Start}[s] \overset{(2)}{<} \mathit{Start}[s']$$

Since $(s, s') \in \mathcal{R}_{\mathsf{tree}}$ holds, by Lemma 3.48, we either have $\mathit{Root}(s) = \mathit{Root}(s')$ or $s' = \mathit{Root}(s')$, since the branches of $\mathcal{K}_{\mathsf{tree}}$ are segmented by the SCCs. The second possibility, i.e. $s' = \mathit{Root}(s')$ can not hold, since then, the contradiction $\mathit{Start}[s'] < \mathit{Start}[s']$ follows from the above facts. Hence, we have $\mathit{Root}(s) = \mathit{Root}(s')$, and therefore, our proof is done by the above inequations.

Finally, the third point follows from the first two, since $\mathit{Low}[s] \leq \mathit{Start}[s]$ holds by definition of $\mathit{Low}[s]$. Hence, the third point can be split into the implications (1) $s \neq \mathit{Root}(s) \rightarrow \mathit{Low}[s] < \mathit{Start}[s]$ and (2) $s = \mathit{Root}(s) \rightarrow \mathit{Low}[s] \geq \mathit{Start}[s]$. It is easily seen that these are equivalent to the first to points.

□

The above lemma gives us a criterion to check during the depth first traversal whether a state s is the root of its SCC after its direct successor states have been explored. To this end, we compute the numbers $\mathit{Low}[s]$ as given in the above definition. If we compute these numbers in post-order, i.e., before termination of the depth first traversal, all the required information is available. The only problem is to care about the condition $s' \in \mathsf{SCC}(s)$ when a transition $(s, s') \in \mathcal{R}_{\mathsf{cross}}$ is processed. To guarantee this condition, Tarjan's algorithm as given in Figure 3.29 maintains a stack where the already explored states are stored in post order. If a root node is detected, we know due to the segmentation lemma that the SCC of this state is a prefix of the stack. Hence, whenever a root is found, we delete all states from the top of this stack down to the root. For this reason, if a transition $(s, s') \in \mathcal{R}_{\mathsf{cross}}$ is processed, we can check $s' \in \mathsf{SCC}(s)$ by the then equivalent condition $s' \in \mathit{Stack}$.

Theorem 3.51 (Correctness and Complexity of Tarjan**).** *Algorithms* Tarjan *and* Tarjan2 *given in Figure 3.29 and Figure 3.30, respectively, compute all strongly connected components of a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$. Moreover,* Tarjan2 *runs in time $O(|\mathcal{K}|)$.*

It is easily seen that algorithm Tarjan performs a depth first traversal, where additionally the numbers $\mathit{Low}[s]$ are computed according to their definition. Additionally, if the currently explored state s is a root, we know that the states above this state on the stack are the SCC of this state. Hence, we delete them from the stack and store the found SCC.

There are several variants of Tarjan's algorithm. One of these variants is given in Figure 3.30. In this version, the numbers *Term* are unnecessary. Their only usage in algorithm TarjanDFS was to classify a transition as a

```
function TarjanDFS(s, c1, c2, Start, Term, Low, Stack)
  c1 := c1 + 1;
  Start[s] := c1;
  Low[s] := c1;
  for all s' ∈ suc⁼_R({s}) do
    if Start[s'] = 0 then
      Stack := (s' ▷ Stack);
      (C, c1, c2, Start, Term, Low, Stack)
          := TarjanDFS(s', c1, c2, Start, Term, Low, Stack);
      Low[s] := min(Low[s], Low[s'])
    end;
    if Term[s'] = 0 then
      Low[s] := min(Low[s], Start[s'])
    if Start[s] ≥ Start[s'] ∧ s' ∈ Stack then
      Low[s] := min(Low[s], Start[s'])
    end
  end;
  c2 := c2 + 1;
  Term[s] := c2;
  if Start[s] = Low[s] then
    C := {};
    do
      s' := HD (Stack);
      C := C ∪ {s'};
      Stack := TL (Stack)
    while s' ≠ s;
  end
  return (C ∪ {C}, c1, c2, Start, Term, Low, Stack)
end

function Tarjan()
  for all s ∈ S do
    Start[s] := 0; Term[s] := 0; Low[s] := 0
  end;
  c1 := 0; c2 := 0; Stack := [ ]; SCC := {};
  for all s ∈ I do
    if Start[s] = 0 then
      Stack := (s ▷ Stack);
      (C, c1, c2, Start, Term, Low, Stack)
          := TarjanDFS(s, c1, c2, Start, Term, Low, Stack);
      SCC := SCC ∪ C
    end
  end;
  return SCC
end
```

Fig. 3.29. Tarjan's algorithm to compute strongly connected components

```
function Tarjan2DFS(s, c_1, Start, Other, Low, Stack)
  c_1 := c_1 + 1;
  Start[s] := c_1;
  Low[s] := c_1;
  for all s' ∈ suc_∃^R({s}) do
    if Start[s'] = 0 then
      Stack := (s' ▷ Stack);
      (C, c_1, Start, Other, Low, Stack)
          := Tarjan2DFS(s', c_1, Start, Other, Low, Stack)
    end;
    if Other[s'] = false then
      Low[s] := min(Low[s], Low[s'])
    end
  end;
  if Start[s] = Low[s] then
    C := {};
    do
      s' := HD (Stack);
      Other[s'] := true;
      C := C ∪ {s'};
      Stack := TL (Stack)
    while s' ≠ s;
  end
  return (C ∪ {C}, c_1, Start, Other, Low, Stack)
end

function Tarjan2()
  for all s ∈ S do
    Start[s] := 0; Other[s] := false; Low[s] := 0
  end;
  c_1 := 0; Stack := [ ]; SCC := {};
  for all s ∈ I do
    if Start[s] = 0 then
      Stack := (s ▷ Stack);
      (C, c_1, Start, Other, Low, Stack)
          := Tarjan2DFS(s, c_1, Start, Other, Low, Stack);
      SCC := SCC ∪ C
    end
  end;
  return SCC
end
```

Fig. 3.30. Tarjan's algorithm without *Term* numbers

backward transition. Instead, the algorithm makes use of a Boolean valued array *Other*$[s]$ that is initialized by false. Each time a SCC is found the value *Other*$[s]$ of the states that belong to the SCC is turned to true. This has moreover the advantage that the test $s' \in$ *Stack* can be avoided.

To see that this algorithm still computes the same numbers *Low*$[s]$, note that *Other*$[s] =$ true *holds iff* $\mathsf{SCC}(s)$ *has already been taken from the stack*. Hence, in the if-statement of the for-loop of Tarjan2DFS where *Other*$[s'] =$ false is tested, we have *Other*$[s'] =$ false iff $s' \in \mathsf{SCC}(s)$. In particular, *Other*$[s'] =$ false holds for all $(s, s') \in \mathcal{R}_{\mathsf{bwd}}$. Hence, the conditional statement filters the relevant states s' for determining *Low*$[s]$. Note that it is no harm to consider also states s' with $(s, s') \in \mathcal{R}_{\mathsf{fwd}}$, as long as these belong to $\mathsf{SCC}(s)$.

Hence, also Tarjan2 correctly computes the numbers *Low*$[s]$ and the SCCs of the structure. Other variants of Tarjan's algorithm can be found in [398]. These additionally improve Tarjan2 in that less states are stored on the stack. In particular, trivial SCCs are not stored on the stack.

Another algorithm to compute SCCs by depth first traversals due to Kosaraju and Sharir [7, 462] is given in Figure 3.31. The algorithm employs a first depth first traversal $\mathsf{DepthFirstSearch}_1$ to sort the reachable states of the spanning trees of $\mathcal{K}$ in lists L_T according to their *Term* numbers, although these numbers are not explicitely computed. By Lemma 3.48, we know that *Term*$[$*Root*$(C)] = \max\{$*Term*$[s] \mid s \in C\}$ holds for every SCC C. Hence, *for every SCC C, the root* *Root*(C) *is placed before the other states of C in the list L_T.* These lists are used in the second depth first traversal in the driving loop of function Kosaraju that initiates DFS_2: By our observation, that the *Root*(C) is put before the other states of C, it follows that the first element of L_T is the root of its SCC. Hence, we can find the SCC by a backward traversal from the root, which is done by DFS_2 called from Kosaraju. During this backward traversal, the numbers *Start*$_2[s]$ are set > 0 for the states that belong to the SCC. After that, the list L_T is searched from left to right until a state s with *Start*$_2[s] = 0$ is found, hence, this state does not belong to already found SCCs. Moreover, it must be a root, since it is the leftmost state with the property *Start*$_2[s] = 0$. For this reason, we can again find its SCC by a backward traversal, and can repeat this procedure until all SCCs are found.

For example, the depth first traversal given in Figure 3.28 yields

$$L_T = [s_1, s_2, s_8, s_9, s_7, s_2, s_3, s_4, s_5]$$

Theorem 3.52 (Correctness and Complexity of Kosaraju**).** *Algorithm* Kosaraju *given in Figure 3.31 computes all strongly connected components of a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ in time $O(|\mathcal{K}|)$.*

Kosaraju's algorithm employs two depth first traversals, and is therefore viewed to be less efficient as Tarjan's algorithm. Its advantage is that its correctness is easier to understand, since we do not need to directly check for roots. The disadvantage, however, is that additionally to $\mathsf{suc}^{\mathcal{R}}_{\exists}(\{s\})$, we also need $\mathsf{pre}^{\mathcal{R}}_{\exists}(\{s\})$, which is normally not stored in adjacency lists.

```
function DFS_1(s, c, Start_1, L_T)
  c := c + 1; Start_1[s] := c;
  for all s' ∈ suc^R_∃({s}) do
    if Start_1[s'] = 0 then
      (c, Start_1, L_T) := DFS(s', c, Start_1, L_T)
    end
  end;
  L_T := (s ▷ L_T);
  return (c, Start_1, L_T)
end

function DepthFirstSearch_1()
  for all s ∈ S do Start_1[s] := 0 end;
  c := 0; L_T := {};
  for all s ∈ I do
    if Start_1[s] = 0 then
      (c, Start_1, L_T) := DFS(s, c, Start_1, [ ]);
      L_T := L_T ∪ {L_T}
    end
  end;
  return L_T
end

function DFS_2(s, c, C, Start_2)
  c := c + 1; Start_2[s] := c;
  C := C ∪ {s};
  for all s' ∈ pre^R_∃({s}) do
    if Start_2[s'] = 0 then
      (c, C, Start_2) := DFS(s', c, C, Start_2);
    end
  end;
  return (c, C, Start_2)
end

function Kosaraju()
  L_T := DepthFirstSearch_1();
  for all s ∈ S do Start_2[s] := 0; Term_2[s] := 0 end;
  c := 0; C := {};
  for all L_T ∈ L_T do
    do
      s := HD (L_T); L_T := TL (L_T);
      if Start_2[s] = 0 then
          (c, C, Start_2) := DFS_2(s, c, {}, Start_2)
          C := C ∪ {C}
      end
    end
  end;
  return C
end
```

Fig. 3.31. Kosaraju's algorithm to compute strongly connected components

3.7.3 SCC Partitioning by Breadth-First Traversals

In the previous section, we considered algorithms to compute all SCCs of a Kripke structure by depth first traversal through the structure. However, based on depth first traversals, none of the previously discussed algorithms can benefit from implicit representation of state sets, which limits their practical usage. In this section, we consider recent advances to apply symbolic methods for the enumeration of SCCs.

As already mentioned at the beginning of the previous section, the simplest way to use symbolic representations for the computation of the SCCs, is to compute the transitive closure of the transition relation by iterative squaring [500, 501], i.e., by computing the relations $\mathcal{R}^0 := \mathcal{R}$, and $\mathcal{R}^{2i} := \mathcal{R}^i \circ \mathcal{R}^i = \{(s, s') \mid \exists s'' \in \mathcal{S}.(s, s'') \in \mathcal{R}^i \wedge (s'', s') \in \mathcal{R}^i\}$. Clearly, this iteration stops with a fixpoint $\mathcal{R}^*$ that is the transitive closure of $\mathcal{R}$. Hence, $(s, s') \in \mathcal{R}^*$ holds iff there is a path from s to s'. It is clear how strongly connected components can then be found: we have $\mathsf{SCC}(s) = \mathsf{pre}_\exists^{\mathcal{R}^*}(\varphi_s) \cap \mathsf{suc}_\exists^{\mathcal{R}^*}(\varphi_s)$ if φ_s is an expression that represents $\{s\}$, i.e, if $[\![\varphi_s]\!]_\mathcal{K} = \{s\}$ holds.

However, computing the transitive closure is commonly considered to be inefficient [431], since three sets of variables are required. In the recent years, better symbolic algorithms to compute strongly connected components have been developed [52, 212, 536, 537]. It has to be remarked here that the complexities mentioned in these references refer to *'symbolic steps'*, which are predecessor, successor, and Boolean operations. As these operations require time $O(|\mathcal{K}|)$ with our implementation of Boolean arrays, we have to multiply the results by this factor to obtain a bound of the execution time.

A simple approach to compute the SCCs of a Kripke structure by symbolic methods is to select an arbitrary state $s \in \mathcal{S}$, and to compute its SCC via the following fixpoints, where we again assume that $[\![\varphi_s]\!]_\mathcal{K} = \{s\}$ holds:

$$\mathsf{SCC}(\varphi_s) = \left[\!\!\left[(\mu x.\varphi_s \vee \Diamond x) \wedge \left(\mu y.\varphi_s \vee \overleftarrow{\Diamond}\, y \right) \right]\!\!\right]_\mathcal{K}$$

The correctness is obvious due to the definition of SCCs, and the complexity to compute the SCC requires at most $O(|\mathcal{S}|)$ iterations. However, the above fixpoints only compute the SCC of one particular state s, so that we next remove the states in $\mathsf{SCC}(\varphi_s)$ of the Kripke structure and select another state to compute its SCC. In the worst case, we have to repeat the procedure for all states, thus yielding an upper bound of $O(|\mathcal{K}|\,|\mathcal{S}|)$ for the overall runtime. If we use symbolic methods to compute the fixpoints rather than Cleaveland-Steffen, we even yield an upper bound of $O(|\mathcal{K}|\,|\mathcal{S}|^2)$ for the complexity.

In 1999, Xie and Beerel [536, 537] presented an algorithm that essentially works in the above described way (see Figure 3.32): it selects a state s, computes its backward and forward sets $\mathcal{B}$ and $\mathcal{F}$, respectively, so that the SCC has been found by symbolic methods, and proceeds with another state, until no states are left. They additionally improved this naive SCC computation in

```
function XieBeerel(S_φ)
  if S_φ = {} then return {} end;
  s := selectfrom(S_φ);
  B := {s}; B_front := {s};
  while B_front ≠ {} do
    B_front := (S_φ ∩ pre∃^R(B_front)) \ B;
    B := B ∪ B_front
  end;
  F := {s}; F_front := {s};
  while F_front ≠ {} do
    F_front := (B ∩ suc∃^R(F_front)) \ F;
    F := F ∪ F_front
  end;
  C := F ∩ B;
  S_1 := F \ C;
  C_1 := XieBeerel(S_1);
  S_2 := S_φ \ F;
  C_2 := XieBeerel(S_2);
  return C_1 ∪ C_2 ∪ {C}
end
```

Fig. 3.32. Symbolic computation of SCCs due to Xie and Beerel [536, 537]

two ways: first, they restricted the computation of the forward set $\mu y.\varphi_s \vee \overleftarrow{\Diamond} y$ to the previously computed backward set, due to the following equivalence:

$$\mathsf{fix}\begin{bmatrix} x \overset{\mu}{=} \varphi_s \vee \Diamond x \\ y \overset{\mu}{=} \varphi_s \vee \overleftarrow{\Diamond} y \end{bmatrix} \mathsf{in}\ x \wedge y\ \mathsf{end} \Leftrightarrow \mathsf{fix}\begin{bmatrix} x \overset{\mu}{=} \varphi_s \vee \Diamond x \\ y \overset{\mu}{=} \varphi_s \vee x \wedge \overleftarrow{\Diamond} y \end{bmatrix} \mathsf{in}\ x \wedge y\ \mathsf{end}$$

This saves iterations for the computation of the forward set, since now only the subset of the forward set is computed that contributes to the SCC. Their second improvement is due to the observataion that forward and backward sets are SCC-closed (see Lemma 3.45), and that intersection, union, and complements of SCC-closed sets are again SCC-closed. In particular, the sets $\mathcal{F} \setminus \mathsf{SCC}(s)$ and $\mathcal{S}_\varphi \setminus \mathcal{F}$ are SCC-closed (provided that $\mathcal{S}_\varphi$ is SCC-closed), so that the recursion may proceed with these sets instead of $\mathcal{S} \setminus \mathsf{SCC}(s)$. It has been shown that XieBeerel requires $O(|\mathcal{S}|^2)$ symbolic operations, and thus has an overall runtime of $O(|\mathcal{K}|\,|\mathcal{S}|^2)$.

To be precise, Figure 3.32 only presents a simplified version of the algorithm given in [536, 537]. The original algorithm also *eliminates the trivial SCCs that are directly reachable from a found SCC without passing another nontrivial SCC*. To this end, the notion of *finite maximum distance predecessors* was defined as the set of states that can reach another set of states, but only by paths of a limited finite maximal length (hence, in particular, without passing cycles). This is particularly useful for symbolic procedures since the trivial SCCs destroy the essential idea of symbolic representations, i.e., that large

sets are stored by small data structures. As this improvement can be made for all the procedures described in this section, we do not list it in the algorithms, but recommend its use for all symbolic implementations.

```
function LockStep(S_φ)
  if S_φ = {} then return {} end;
  s := selectfrom(S_φ);
  F := {s}; F_front := {s};
  B := {s}; B_front := {s};
  while F_front ≠ {} ∧ B_front ≠ {} do
    F_front := (S_φ ∩ suc*_∃(F_front)) \ F;
    B_front := (S_φ ∩ pre*_∃(B_front)) \ B;
    F := F ∪ F_front;
    B := B ∪ B_front
  end;
  if F_front = {} then
    Converged := F;
    B := B ∩ F; B_front := B_front ∩ F;
    while B_front ≠ {} do
      B_front := (F ∩ pre*_∃(B_front)) \ B;
      B := B ∪ B_front
    end;
    C := B
  else
    Converged := B;
    F := F ∩ B; F_front := F_front ∩ B;
    while F_front ≠ {} do
      F_front := (B ∩ suc*_∃(F_front)) \ F;
      F := F ∪ F_front
    end;
    C := F
  end;
  S_1 := Converged \ C;
  S_2 := S_φ \ Converged;
  C_1 := LockStep(S_1);
  C_2 := LockStep(S_2);
  return C_1 ∪ C_2 ∪ {C}
end
```

Fig. 3.33. Symbolic computation of SCCs due to Bloem, Gabow, and Somenzi [52]

In 2000, Bloem, Gabow, and Somenzi [52] proposed algorithm LockStep given in Figure 3.33 as an improvement of Xie and Beerel's algorithm. As XieBeerel, LockStep first selects an arbitrary state $s \in \mathcal{S}$, but then starts to *simultaneously compute the backward and forward sets.* As soon as the first iteration converges, the found set is stored in the variable *Converged*. Moreover,

the fixpoint iteration of the second set is completed where now the restriction to *Converged* can be made. After that, the SCC C is found, and the algorithm is recursively applied to $\mathcal{S}_1 := \mathit{Converged} \setminus C$ and $\mathcal{S}_2 := \mathcal{S} \setminus \mathit{Converged}$. Note that LockStep does not compute the entire backward set if the forward set is obtained in the first iteration, since the second iteration is then restricted to the forward set. Analogously, LockStep does not compute the entire forward set if the backward set is obtained in the first iteration, since the second iteration is then restricted to the backward set.

To determine the complexity of the algorithm, we virtually introduce for every state $s \in \mathcal{S}$ a number $\mathit{Charge}[s] \in \mathbb{N}$. At the beginning all these numbers are 0, and each time a predecessor or successor operation is performed, we increase one of the numbers $\mathit{Charge}[s]$. It does not matter which one of these numbers is increased, the important issue is that after each recursive call, the sum $\sum_{s \in \mathcal{S}} \mathit{Charge}[s]$ is the number of predecessor or successor operation that have been applied so far. The conditions to increase the numbers $\mathit{Charge}[s]$ are obtained by the following observations:

(1) After the fixpoint iterations, the sets C, $\mathcal{S}_1$, *and* $\mathcal{S}_2$ *partition the set* $\mathcal{S}_\varphi$, i.e., $\mathcal{S}_\varphi = C \cup \mathcal{S}_1 \cup \mathcal{S}_2$, and C, $\mathcal{S}_1$, and $\mathcal{S}_2$ are pairwise disjoint.

(2) After the fixpoint iterations, we have $|\mathcal{S}_1| \leq \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$ or $|\mathcal{S}_2| \leq \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$: If this would not hold, we would have $|\mathcal{S}| = |C| + |\mathcal{S}_1| + |\mathcal{S}_2| > 1 + 2 \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor \geq |\mathcal{S}|$, which is a contradiction.

(3) *There are at most* $|C| + \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$ *iterations of the first while loop:* To see this, note that the number of its iterations is the minimum of the two numbers of iterations to compute the fixpoints $\mu x.z \vee \overleftarrow{\Diamond} x$ and $\mu x.z \vee \Diamond x$. Now, consider the two cases according to (2) above:
   (a) Assume $|\mathcal{S}_1| \leq \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$, i.e., that $|\mathit{Converged} \setminus C| \leq \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$ holds. It directly follows from this case assumption that $|\mathit{Converged}| \leq |C| + \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$ holds, and since $|\mathit{Converged}|$ is an upper bound of the iteration of the first while loop, the proposition follows.
   (b) If $|\mathcal{S}_1| \leq \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$ does not hold, we know by (2) that $|\mathcal{S}_2| \leq \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$, and thus that $|\mathcal{S}| - |C| \leq \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$ must hold. The non-converged set (either $\mathcal{F}$ or $\mathcal{B}$) is then a subset of $C \cup \mathcal{S}_2$, and for this reason, it has at most $|C| + |\mathcal{S}_2| \leq |C| + \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$ elements. Hence, even the fixpoint iteration of the non-converged set requires at most $|C| + \left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$ iterations, and thus the number of iterations of the while loop can not be more than that.

(4) *The second executed while-loop has at most* $|C|$ *iterations:* The result of this iteration is the SCC C, and the worst case is that we start with an approximation of cardinality 1, and add one state in each iteration. Hence, $|C|$ is an upper bound for the number of iterations of this loop.

(5) *Before the recursive calls,* LockStep *performs at most* $3|C| + 2\left\lfloor \frac{1}{2} |\mathcal{S}| \right\rfloor$ *predecessor and successor computations:* There are two such computations in each iteration of the first while loop, and one in the iteration of the second one,

hence, at most $2\left(|C| + \left\lfloor \frac{1}{2}\,|\mathcal{S}| \right\rfloor\right) + |C|$ predecessor and successor computations.

For convenience, define $\mathcal{S}_{\mathsf{small}} := \mathcal{S}_1$ if $|\mathcal{S}_1| \leq \left\lfloor \frac{1}{2}\,|\mathcal{S}| \right\rfloor$ holds, otherwise define $\mathcal{S}_{\mathsf{small}} := \mathcal{S}_2$. According to (2), we thus have (6) $|\mathcal{S}_{\mathsf{small}}| \leq \left\lfloor \frac{1}{2}\,|\mathcal{S}| \right\rfloor$. Thus, we can update the numbers *Charge*$[s]$ as follows to estimate the upper bound of predecessor and successor operations:

- for all states $s \in C$, we update *Charge*$[s] :=$ *Charge*$[s] + 3$
- for all states $s \in \mathcal{S}_{\mathsf{small}}$, we update *Charge*$[s] :=$ *Charge*$[s] + 2$
- for all states $s \in \mathcal{S} \setminus (C \cup \mathcal{S}_{\mathsf{small}})$, we retain the current value *Charge*$[s]$

Consider now how often one of the above two possible incrementations of *Charge*$[s']$ can take place for a particular state s': If s' belongs to the SCC C that is computed in the current recursion step of LockStep, then there is only one update, namely the above mentioned increase by 3. This is due to the fact that the SCC C is not considered in the further recursion steps where $\mathcal{S}_1$ and $\mathcal{S}_2$ are explored. If $s' \notin C$ holds, s' is passed to one of the two recursive calls. According to (2) above, there is always a call to $\mathcal{S}_{\mathsf{small}}$ and another one to $\mathcal{S} \setminus (C \cup \mathcal{S}_{\mathsf{small}})$. If $s' \in \mathcal{S} \setminus (C \cup \mathcal{S}_{\mathsf{small}})$ holds, there is no increase, and if $s' \in \mathcal{S}_{\mathsf{small}}$ holds, there is an increase by 2. However, as by definition of $\mathcal{S}_{\mathsf{small}}$ (6) $|\mathcal{S}_{\mathsf{small}}| \leq \left\lfloor \frac{1}{2}\,|\mathcal{S}| \right\rfloor$ holds, the latter can happen at most $\log_2 |\mathcal{S}|$ many times. Thus, we end up with a maximal charge of *Charge*$[s'] \leq 2\log_2 |\mathcal{S}| + 3$. Taking the sum over all states then yields an upper bound of $O(|\mathcal{S}| \log_2 |\mathcal{S}|)$ predecessor and successor operations.

The above estimation of the complexity counted the number of predecessor and successor operations. It is however easily seen that also the number of all symbolic operations is limited by $O(|\mathcal{S}| \log_2 |\mathcal{S}|)$. As we can implement every symbolic operation in time $O(|\mathcal{K}|)$, we obtain the following result:

Theorem 3.53 (Correctness and Complexity of LockStep**).** *Algorithm* LockStep *given in Figure 3.33 computes all strongly connected components of a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ with $O(|\mathcal{S}| \log_2 |\mathcal{S}|)$ symbolic operations. Hence, it runs in time $O(|\mathcal{K}|\,|\mathcal{S}| \log_2 |\mathcal{S}|)$.*

The above algorithm has been further improved by Gentilini, Piazza, and Policriti [212] in 2003 (see Figure 3.36). Similar to XieBeerel and LockStep, their algorithm computes $\mathsf{SCC}(s)$ for a randomly chosen state s by computing the intersection of the set of states $\mathcal{F}$ that are reachable from s (the forward set of s) and the set of states $\mathcal{B}$ that can be reached from s (the backward set of s). After this, the algorithm is recursively applied to $\mathcal{F} \setminus \mathsf{SCC}(s)$ and $\mathcal{S} \setminus \mathcal{F}$, similar to XieBeerel and LockStep. The additional improvement is due to the observation that the iterative computation of the forward set $\mathcal{F}$ induces an *ordering of the states* in $\mathcal{F}$ according to the *frontiers*, i.e., the state sets that are added to $\mathcal{F}$ in the fixpoint iteration. This order is exploited in that the recursive calls do not select arbitrary states unless states of a previously computed order are

available. Hence, the computed order drives the recursive calls, and this allows the algorithm to run in only time $O(|\mathcal{K}|\,|\mathcal{S}|)$, where only $O(|\mathcal{S}|)$ symbolic operations are necessary.

The function SkeletonSCC given in Figure 3.35 therefore not only computes the forward set $\mathcal{F}$ and the SCC of a state s_1, but additionally a so-called *skeleton* $(\mathcal{S}_{\mathsf{skeleton}}, s_n)$ of $\mathcal{F}$. Intuitively, such a skeleton of a state s_1 corresponds with a *shortest path of maximal length from s_1 to s_n in $\mathcal{F}$*. Hence, it is obtained by selecting one state of each frontier such that the selected states form a path in the forward set $\mathcal{F}$. Clearly, the path has maximal length, and it is a shortest path, since it has no cycles. However, the path is not really required; instead, it sufficies to have the start and end state together with the *set of states $\mathcal{S}_{\mathsf{skeleton}}$ on the path.* This is sufficient to drive the recursive calls in a way that previous symbolic operations are amortized such that a linear upper bound is achieved. Without the relationship to forward sets, pairs like $(\mathcal{S}_{\mathsf{skeleton}}, s_n)$ are called *spines* in [212], and are formally defined as follows:

Definition 3.54 (Spines). *Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, a pair $(\mathcal{S}_{\mathsf{spine}}, s)$ with $\mathcal{S}_{\mathsf{spine}} \subseteq \mathcal{S}$ and $s \in \mathcal{S}_{\mathsf{spine}}$ is called a spine if there is a bijection $f : \mathcal{S}_{\mathsf{spine}} \to \{1, \ldots, |\mathcal{S}_{\mathsf{spine}}|\}$ with the following properties:*

Spine1*: $f(s) = |\mathcal{S}_{\mathsf{spine}}|$*
Spine2*: for all $(s, s') \in \mathcal{R}$, we have either $f(s') = f(s) + 1$, $s' = s$, or $f(s') < f(s)$*
Spine3*: for all $s, s' \in \mathcal{S}_{\mathsf{spine}}$, $f(s') = f(s) + 1$ implies $(s, s') \in \mathcal{R}$*

f is called the certifying function of the spine $(\mathcal{S}_{\mathsf{spine}}, s)$.

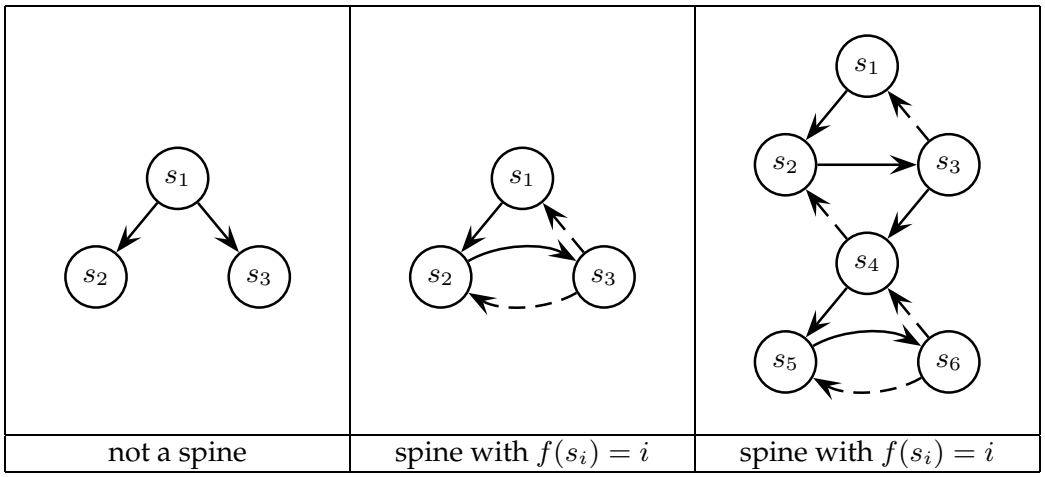

Fig. 3.34. Examples of spines

Note that for arbitrary sets of states $\mathcal{S}_\varphi$, we can not always find a state $s \in \mathcal{S}_\varphi$ such that $(\mathcal{S}_\varphi, s)$ would be a spine (see Figure 3.34). A necessary requirement

is that the spine must have a total order that is compatible with the transition relation: for a spine $(\{s_1, \ldots, s_n\}, s_n)$ with certifying function $f(s_i) := i$, we have the natural total order $s_i \prec_f s_j :\Leftrightarrow f(s_i) < f(s_j)$. Hence, every spine is totally ordered $s_1 \prec_f s_2 \prec_f \ldots \prec_f s_n$ with $(s_i, s_{i+1}) \in \mathcal{R}$ due to Spine3. Conversely, a path through pairwise distinct states s_1,...,s_n satisfies properties Spine1 and Spine3 with the function $f(s_i) := i$. However, Spine2 additionally forbids that there are forward transitions, i.e., transitions (s_i, s_j) with $i < j$. Self-loops, i.e., (s_i, s_i) and backward transitions (s_i, s_j) with $i > j$ are possible. The following lemma lists some important properties of spines:

Lemma 3.55 (Spines). *Given a spine $(\mathcal{S}_{\mathsf{spine}}, s_n)$ of a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$. Let $\mathcal{S}_{\mathsf{spine}} = \{s_1, \ldots, s_n\}$ and let $f(s_i) = i$ be the certifying function. Then, the following holds:*

(1) $(\{s_1, \ldots, s_k\}, s_k)$ is a spine with certifying function f restricted to $\{s_1, \ldots, s_k\}$
(2) $(\{s_k, \ldots, s_n\}, s_n)$ is a spine with certifying function $f(s_j) := j-k+1$ restricted to $\{s_k, \ldots, s_n\}$
(3) the certifying function f is uniquely determined
(4) if it exists, a shortest path s_1,...,s_n from s_1 to s_n is a spine
(5) $\mathsf{pre}^{\mathcal{R}}_{\exists}(\{s_n\}) \cap \{s_1, \ldots, s_{n-1}\} = \{s_{n-1}\}$
(6) $\mathcal{S}_{\mathsf{spine}} \cap \mathsf{SCC}(s_1) = \{s_1, \ldots, s_k\}$ for some k
(7) $\mathcal{S}_{\mathsf{spine}} \cap \mathsf{SCC}(s_n) = \{s_\ell, \ldots, s_n\}$ for some ℓ
(8) either $s_n \in \mathsf{SCC}(s_1)$ or $(\mathcal{S}_{\mathsf{spine}} \setminus \mathsf{SCC}(s_1), s_n)$ is a spine
(9) either $s_1 \in \mathsf{SCC}(s_n)$ or $(\mathcal{S}_{\mathsf{spine}} \setminus \mathsf{SCC}(s_n), s_\ell)$ is a spine, where $\mathsf{pre}^{\mathcal{R}}_{\exists}(\mathcal{S}_{\mathsf{spine}} \cap \mathsf{SCC}(s_n)) \cap (\mathcal{S}_{\mathsf{spine}} \setminus \mathsf{SCC}(s_n)) = \{s_\ell\}$ holds
(10) $\mathsf{SCC}(s_n) = \mathsf{FW}(\{s_n\}) \cap \bigcup_{i=1}^{n} \mathsf{SCC}(s_i)$, where $\mathsf{FW}(\{s_n\})$ is the set of states that can be reached from s_n

Proof. We prove the above properties one after the other:

(1) and (2): Points (1) and (2) are easily seen by verifying the properties Spine1, Spine2, and Spine3.

(3) The uniqueness of the certifying function is proved as follows: Assume g would be another certifying function. Hence, there must be a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ with $g(s_i) := \pi(f(s_i))$. We have to prove that π is the identity. Assume that this is not the case, hence, there is a $i \in \{1, \ldots, n\}$ with (3.1) $\pi(i) \neq i$ and (3.2) $\forall j > i . \pi(j) = j$. Hence, we have (3.3) $\pi(i) < i$, since the numbers $i+1, \ldots, n$ are already used by $\pi(i+1), \ldots, \pi(n)$ according to (3.2). Due to Spine2, we then should have for $(s_i, s_{i+1}) \in \mathcal{R}$ one of the following:
   - $g(s_{i+1}) = g(s_i) + 1$: This means $\pi(i+1) = \pi(i) + 1$, but $\pi(i+1) = i+1$ and $\pi(i) \neq i$ according to (3.2) and (3.1), respectively.
   - $s_{i+1} = s_i$: This can also not hold, since it would imply $f(s_{i+1}) = i$, but we have $f(s_{i+1}) = i + 1$.
   - $g(s_{i+1}) < g(s_i) + 1$: This means $\pi(i+1) < \pi(i) + 1$, which is due to (3.2) equivalent to $i < \pi(i)$. However, this contradicts (3.3).

   Hence, g can not exist, and therefore, f is the only certifying function

(4) Let $s_1, \ldots, s_n$ be a shortest path from s_1 to s_n, thus we have $s_i \neq s_j$ for $i \neq j$. We define $f(s_i) := i$, which is well-defined since the states are pairwise distinct, and easily verify that f is a certifying function:
Spine1: $f(s_n) = n = |\mathcal{S}_{\mathsf{spine}}|$
Spine2: Assuming $(s_i, s_j) \in \mathcal{R}$ with $i < j$ contradicts the assumption that the path $s_1, \ldots, s_n$ is a shortest one. Hence, there are no forward transitions, and thus $(s_i, s_j) \in \mathcal{R}$ implies $i \geq j$.
Spine3: By definition, $f(s_j) = f(s_i) + 1$ means $j = i + 1$, and the implication $(s_i, s_j) \in \mathcal{R}$ therefore means $(s_i, s_{i+1}) \in \mathcal{R}$, which holds due to our assumption.

(5) Due to Spine3, we have $(s_{n-1}, s_n) \in \mathcal{R}$, and thus $s_{n-1} \in \mathsf{pre}_\exists^{\mathcal{R}}(\{s_n\})$, which implies $\{s_{n-1}\} \subseteq \mathsf{pre}_\exists^{\mathcal{R}}(\{s_n\}) \cap \{s_1, \ldots, s_{n-1}\}$. For the other direction, assume $s_j \in \mathsf{pre}_\exists^{\mathcal{R}}(\{s_n\}) \cap \{s_1, \ldots, s_{n-1}\}$ holds with $j \neq n-1$. In particular, it follows that $(s_j, s_n) \in \mathcal{R}$ holds, and thus by Spine2 one of the following should hold:
- $f(s_n) = f(s_j) + 1$: However, this is equivalent to $n = j + 1$, and thus $j = n - 1$, which is a contradition to our assumption $j \neq n - 1$.
- $s_n = s_j$: This can not hold, since we assumed $s_j \in \{s_1, \ldots, s_{n-1}\}$.
- $f(s_n) < f(s_j)$: By definition of f, this means $n < j$, which is obviously not possible.

Hence, it follows that $(s_j, s_n) \in \mathcal{R}$ can not hold unless $j = n - 1$ holds.

(6) and (7) If $s_j \in \mathsf{SCC}(s_1)$, then $s_j \rightsquigarrow s_1$ and $s_1 \rightsquigarrow s_j$ follows. As $s_1 \to s_2 \to \ldots \to s_j$ holds due to the spine properties, it follows that $\{s_1, \ldots, s_j\} \subseteq \mathsf{SCC}(s_1)$. Hence, the intersection $\mathcal{S}_{\mathsf{spine}} \cap \mathsf{SCC}(s_1)$ is a 'prefix' of $\mathcal{S}_{\mathsf{spine}}$. (7) is proved analogously for a suffix of $\mathcal{S}_{\mathsf{spine}}$.

(8) According to fact (6), $\mathcal{S}_{\mathsf{spine}} \setminus \mathsf{SCC}(s_1) = \{s_k, \ldots, s_n\}$ holds for some k. $s_n \in \mathsf{SCC}(s_1)$ holds iff this set is empty, and the rest follows by fact (2).

(9) According to fact (7), $\mathcal{S}_{\mathsf{spine}} \setminus \mathsf{SCC}(s_n) = \{s_1, \ldots, s_\ell\}$ and $\mathcal{S}_{\mathsf{spine}} \cap \mathsf{SCC}(s_n) = \{s_{\ell+1}, \ldots, s_n\}$ hold for some ℓ. If $s_1 \notin \mathsf{SCC}(s_n)$ holds, we have $\ell \geq 1$, and by (1) that $(\{s_1, \ldots, s_\ell\}, s_\ell)$ is a spine. Due to Spine2, we have no forward transitions in a spine, hence, for $s_i \in \mathcal{S}_{\mathsf{spine}}$ with $i < \ell$, we can not have $(s_i, s_j) \in \mathcal{R}$ with $j > \ell$. Hence, $\mathsf{pre}_\exists^{\mathcal{R}}(\{s_{\ell+1}, \ldots, s_n\}) \cap \mathcal{S}_{\mathsf{spine}} \subseteq \{s_\ell, \ldots, s_n\}$ holds. The additional removal of states $\mathsf{SCC}(s_n)$ yields $\{s_\ell\}$.

(10) $\mathsf{SCC}(s_n) \subseteq \mathsf{FW}(\{s_n\}) \cap \bigcup_{i=1}^n \mathsf{SCC}(s_i)$ clearly holds, since $\mathsf{SCC}(s_n) \subseteq \mathsf{FW}(\{s_n\})$ and $\mathsf{SCC}(s_n) \subseteq \bigcup_{i=1}^n \mathsf{SCC}(s_i)$ holds. To prove the converse, recall (Lemma 3.45) that forward sets like $\mathsf{FW}(\{s_n\})$ are SCC-closed, and that unions and intersections of SCC-closed sets are SCC-closed. In particular, $\mathsf{FW}(\{s_n\}) \cap \bigcup_{i=1}^n \mathsf{SCC}(s_i)$ is SCC-closed. Now, let s be a state with (10.1) $\mathsf{SCC}(s) \subseteq \mathsf{FW}(\{s_n\}) \cap \bigcup_{i=1}^n \mathsf{SCC}(s_i)$, hence, (10.2) $\mathsf{SCC}(s) \subseteq \mathsf{FW}(\{s_n\})$ and (10.3) $\mathsf{SCC}(s) \subseteq \bigcup_{i=1}^n \mathsf{SCC}(s_i)$ holds. By (10.2), we have (10.4) $s_n \rightsquigarrow s$. By (10.3) we know that there is a s_i in the spine with (10.5) $\mathsf{SCC}(s) = \mathsf{SCC}(s_i)$, and thus (10.5) $s \rightsquigarrow s_i$ holds. By the spine properties, we moreover have (10.6) $s_i \rightsquigarrow s_n$. By (10.4), (10.5), and (10.6), we conclude (10.7) $\mathsf{SCC}(s) = \mathsf{SCC}(s_n)$, so that the proposition follows.

□

```
function SkeletonSCC(S_φ, s_1)
  Stack := [ ];
  F_front := {s_1};
  F := {};
  while F_front ≠ {} do
    Stack := (F_front ▷ Stack);
    F := F ∪ F_front;
    F_front := (S_φ ∩ suc∃R(F_front)) \ F
  end;
  F_front := HD (Stack); Stack := TL (Stack);
  s_n := selectfrom(F_front);
  S_skeleton := {s_n};
  while Stack ≠ [ ] do
    F_front := HD (Stack); Stack := TL (Stack);
    s := selectfrom(pre∃R(S_skeleton) ∩ F_front);
    S_skeleton := S_skeleton ∪ {s}
  end;
  C_front := {s_1}; C := {};
  while (C_front ≠ {} do
    C := C ∪ C_front
    C_front := (F ∩ pre∃R(C_front)) \ C
  end;
  return (F, C, S_skeleton, s_n)
end
```

Fig. 3.35. Computation of forward sets together with a skeleton and the root SCC

Hence, spines are shortest paths of maximal length. We will make use of the above facts in the algorithms given in Figure 3.36. To that end, we need one more definition to relate spines to forward sets (recall fact (4) of the previous lemma):

Definition 3.56 (Skeleton of Forwards Sets). *Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, an arbitrary state $s_1 \in \mathcal{S}$, and the states $\mathsf{FW}(\{s_1\})$ that can be reached from s_1. Moreover, assume the states $s_1, \ldots, s_n \in \mathsf{FW}(\{s_1\})$ form a shortest path of maximal length in $\mathsf{FW}(\{s_1\})$, then the spine $(\mathcal{S}_{\mathsf{spine}}, s_n)$ is a skeleton of $\mathsf{FW}(\{s_1\})$.*

Function SkeletonSCC given in Figure 3.35 is the heart of the overall algorithm given in Figure 3.36. The functions of Figure 3.36 use spines to select 'good' states s_1 for computing the next SCC. The computation of the SCC is done by function SkeletonSCC given in Figure 3.35. To this end, the state set of the Kripke structure is restricted to a SCC-closed set $\mathcal{S}_\varphi$. Function SkeletonSCC then computes the forward set $\mathcal{F}$ of a state s_1 together with a skeleton $(\mathcal{S}_{\mathsf{spine}}, s_n)$ of $\mathcal{F}$, and also the SCC C of s_1. To this end, it computes the forward set $\mathcal{F}$ according to the usual fixpoint characterization $\mu x.\varphi_{s_1} \vee \overleftarrow{\Diamond}\, x$, where additionally the frontiers $\mathcal{F}_{\mathsf{front}}$, i.e., the differences of the

fixpoint approximations are stored on a stack. After the forward set $\mathcal{F}$ has been computed, single states are selected from each frontier $\mathcal{F}_{\mathsf{front}}$, so that the selected states of the frontiers form a path. Since the frontiers are pairwise disjoint, it follows that the path is free of cycles, and thus, it is a shortest path from s_1 to the end of the spine s_n. Moreover, as it starts in s_1 and ends in a state s_n of the last frontier, it is a path of maximal length. Hence, by fact (4) of the previous lemma, the set $(\mathcal{S}_{\mathsf{spine}}, s_n)$ is a spine, and thus, a skeleton of $\mathcal{F}$. Concerning the complexity, it is easily seen that the first two loops in function Skeleton require $|\mathcal{S}_{\mathsf{skeleton}}|$ iterations, and the third one requires at most $|C|$ iterations. Moreover, each iteration requires a constant number of symbolic operations.

Lemma 3.57 (Skeleton of Forwards Sets). *Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, an arbitrary state $s_1 \in \mathcal{S}$, and a SCC-closed set $\mathcal{S}_\varphi \subseteq \mathcal{S}$ that is used to restrict the Kripke structure to the structure $\mathcal{K}_{|\mathcal{S}_\varphi}$ where only the states of $\mathcal{S}_\varphi$ are considered. The function* SkeletonSCC *given in Figure 3.35 called with arguments $(\mathcal{S}_\varphi, s_1)$ computes a tuple $(\mathcal{F}, C, \mathcal{S}_{\mathsf{skeleton}}, s_n)$ such that $\mathcal{F}$ is the forward set of s_1 (in $\mathcal{K}_{|\mathcal{S}_\varphi}$), $C = \mathsf{SCC}(s_1)$ (in $\mathcal{K}_{|\mathcal{S}_\varphi}$ or $\mathcal{K}$), and $(\mathcal{S}_{\mathsf{skeleton}}, s_n)$ is a skeleton of $\mathcal{F}$. These computations require $2\,|\mathcal{S}_{\mathsf{skeleton}}| + |C|$ symbolic operations, and hence, time $O(|\mathcal{S}_{\mathsf{skeleton}}| + |C|)$.*

The properties of spines are now exploited by the functions $\mathsf{SpineSCC}_i$ given in Figure 3.36 to reduce the number of symbolic steps to a linear upper bound. As already mentioned, the basis of these functions is still XieBeerel: Given a SCC-closed set $\mathcal{S}_\varphi \subseteq \mathcal{S}$, the forward set $\mathcal{F}$ (w.r.t. $\mathcal{S}_\varphi$) of some state $s_1 \in \mathcal{S}_\varphi$ is computed, and the SCC C of s_1 is then obtained by a backward traveral from s_1 in $\mathcal{F}$. Then, the algorithm is recursively called for the SCC-closed sets $\mathcal{S}_\varphi \setminus \mathcal{F}$ and $\mathcal{F} \setminus C$. However, in contrast to XieBeerel, a skeleton is additionally computed with the forward set $\mathcal{F}$, and this skeleton is used to select the next state to compute the next SCC.

In the original publication [212], the functions $\mathsf{SpineSCC}_0$ and $\mathsf{SpineSCC}_1$ given in Figure 3.36 were merged in a single function. To simplify the explanations, we split them into two functions that are called *if either a spine of previous computations is still available or not.*

If no spine is available, as in the initial situation, function $\mathsf{SpineSCC}_0$ is called. This function proceeds as described above: an arbitrary state s_1 is selected, function SkeletonSCC is used to compute its forward set $\mathcal{F}$, its SCC C, and a skeleton $(\mathcal{S}_{\mathsf{spine}}, s_2)$ of $\mathcal{F}$. Then the first recursive call is prepared for $\mathcal{S}_\varphi \setminus \mathcal{F}$: as $\mathcal{S}_{\mathsf{spine}} \subseteq \mathcal{F}$ holds, we have no spine for $\mathcal{S}_\varphi \setminus \mathcal{F}$, and hence, call again function $\mathsf{SpineSCC}_0$. For the second recursive call, recall fact (8) of Lemma 3.55: either $s_2 \in C$ holds or $(\mathcal{S}_{\mathsf{spine}} \setminus C, s_2)$ is a spine. If the first case holds, we have no further spine and call $\mathsf{SpineSCC}_0$ to compute the SCCs of $\mathcal{F} \setminus C$, otherwise we call $\mathsf{SpineSCC}_1$ with the spine $(\mathcal{S}_{\mathsf{spine}} \setminus C, s_2)$. Note further that $\mathcal{S}_{\mathsf{spine}} \setminus C \subseteq \mathcal{F} \setminus C$, since $\mathcal{S}_{\mathsf{spine}} \subseteq \mathcal{F}$ holds.

If a spine $(\mathcal{S}_{\mathsf{spine1}}, s_1)$ is already available with $\mathcal{S}_{\mathsf{spine1}} \subseteq \mathcal{S}_\varphi$ to partition $\mathcal{S}_\varphi$ into its SCCs, we call $\mathsf{SpineSCC}_1$: In contrast to $\mathsf{SpineSCC}_1$, no state has

```
function SpineSCC_0(S_φ)
  if S_φ = {} then return {} end;
  s_1 := selectfrom(S_φ);
  (F, C, S_spine, s_2) := SkeletonSCC(S_φ, s_1);
  C_1 := SpineSCC_0(S_φ \ F);
  if s_2 ∈ C then
    C_2 := SpineSCC_0(F \ C)
  else
    C_2 := SpineSCC_1(F \ C, S_spine \ C, s_2)
  end;
  return {C} ∪ C_1 ∪ C_2
end

function SpineSCC_1(S_φ, S_spine1, s_1)
  if S_φ = {} then return {} end;
  (F, C, S_spine2, s_2) := Skeleton(S_φ, s_1);
  if S_spine1 ⊆ C then
    C_1 := SpineSCC_0(S_φ \ F)
  else
    {s'_1} :≡ pre^R_∃(C ∩ S_spine1) ∩ (S_spine1 \ C);
    C_1 := SpineSCC_1(S_φ \ F, S_spine1 \ C, s'_1)
  end;
  if s_2 ∈ C then
    C_2 := SpineSCC_0(F \ C)
  else
    C_2 := SpineSCC_1(F \ C, S_spine2 \ C, s_2)
  end;
  return {C} ∪ C_1 ∪ C_2
end
```

Fig. 3.36. Symbolic computation of SCCs by spines

to be selected to compute the next SCC. Instead, the end of the available spine, i.e. s_1, is used. Hence, function SkeletonSCC is used to compute the forward set $\mathcal{F}$ of s_1, its SCC $C = \mathsf{SCC}(s_1)$, and a skeleton $(\mathcal{S}_{\mathsf{spine2}}, s_2)$ of $\mathcal{F}$. Then the first recursive call is prepared for $\mathcal{S}_\varphi \setminus \mathcal{F}$: as $\mathcal{S}_{\mathsf{spine2}} \subseteq \mathcal{F}$ holds, we can not use $\mathcal{S}_{\mathsf{spine2}}$ as a spine for this call. However, $\mathcal{S}_{\mathsf{spine1}}$ has a nonempty intersection with $\mathcal{S}_\varphi \setminus \mathcal{F}$ iff $\mathcal{S}_{\mathsf{spine1}} \not\subseteq C$ holds. Hence, if $\mathcal{S}_{\mathsf{spine1}} \subseteq C$ holds, we have no spine and call $\mathsf{SpineSCC}_0$ to compute the SCCs of $\mathcal{S}_\varphi \setminus \mathcal{F}$. Otherwise, we can compute a remaining spine of $\mathcal{S}_{\mathsf{spine1}}$ by fact (9) of Lemma 3.55 and call $\mathsf{SpineSCC}_1$. The explanations for the second recursive call are identical as explained above for function $\mathsf{SpineSCC}_0$.

The correctness of the algorithm is clear: The only difference to XieBeerel is that the choice of a state to compute the next SCC is driven in $\mathsf{SpineSCC}_1$ by the available spine. This does obviously not affect the correctness, however, it has an impact on the complexity: The computations that were invested in the

computation of the forward sets are amortized by better selections of states to compute the SCCs.

To estimate the complexity in detail, we proceed similarly as with algorithm LockStep: We virtually introduce numbers *Charge*$[s]$ for each state s to count the number of symbolic operations before the recursive call of $\mathsf{SpineSCC}_i$. Initially, we have *Charge*$[s] = 0$ for all states $s \in \mathcal{S}$. It is easily seen that each call of $\mathsf{SpineSCC}_i$ computes one SCC, and invokes one call to SkeletonSCC. Hence, it requires exactly $2\,|\mathcal{S}_{\mathsf{spine}}| + |C|$ symbolic steps, where $\mathcal{S}_{\mathsf{spine}}$ belongs to the skeleton that is computed in the call to SkeletonSCC. Hence, we may update the numbers *Charge*$[s]$ as described below:

Lemma 3.58 (Complexity of $\mathsf{SpineSCC}_i$). *Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ and numbers Charge$[s]$ for $s \in \mathcal{S}$ where initially Charge$[s] = 0$ holds. Assume, these numbers are updated before the recursive calls in functions $\mathsf{SpineSCC}_i$ of Figure 3.36 as follows, where $(\mathcal{F}, C, \mathcal{S}_{\mathsf{spine}}, s_2)$ is the result of the call to function* SkeletonSCC*:*

- *Charge*$[s] :=$ *Charge*$[s] + 3$ *for* $s \in \mathcal{S}_{\mathsf{spine}} \cap C$
- *Charge*$[s] :=$ *Charge*$[s] + 2$ *for* $s \in \mathcal{S}_{\mathsf{spine}} \setminus C$
- *Charge*$[s] :=$ *Charge*$[s] + 1$ *for* $s \in C \setminus \mathcal{S}_{\mathsf{spine}}$
- *Charge*$[s] :=$ *Charge*$[s]$ *for* $s \in \mathcal{S}_\varphi \setminus (\mathcal{S}_{\mathsf{spine}} \cup C)$

Then, before each recursive call to $\mathsf{SpineSCC}_i$, the following holds:

- *If $\mathsf{SpineSCC}_0(\mathcal{S}_\varphi)$ is called, then Charge$[s] = 0$ for all $s \in \mathcal{S}_\varphi$.*
- *If $\mathsf{SpineSCC}_1(\mathcal{S}_\varphi, \mathcal{S}_{\mathsf{spine1}}, s_1)$ is called, then Charge$[s] = 2$ for all $s \in \mathcal{S}_{\mathsf{spine1}}$ and Charge$[s] = 0$ for all $s \in \mathcal{S}_\varphi \setminus \mathcal{S}_{\mathsf{spine1}}$.*

Proof. We prove the above lemma by induction on the number of calls. Initially, the proposition obviously holds, since the first call is made with $\mathsf{SpineSCC}_0$ without a spine, and since initially *Charge*$[s] = 0$ holds. Hence, consider now any subsequent call to $\mathsf{SpineSCC}_i$.

First, consider a call $\mathsf{SpineSCC}_0(\mathcal{S}_\varphi)$. By induction hypothesis, we have *Charge*$[s] = 0$ for all $s \in \mathcal{S}_\varphi$. According to Lemma 3.57, SkeletonSCC computes the tuple $(\mathcal{F}, C, \mathcal{S}_{\mathsf{spine}}, s_2)$ with $2\,|\mathcal{S}_{\mathsf{spine}}| + |C|$ symbolic operations, and hence, the charges are incremented by this number. Hence, no state in $\mathcal{S}_\varphi \setminus \mathcal{F}$ is charged, and hence, we still have *Charge*$[s] = 0$ for all $s \in \mathcal{S}_\varphi \setminus \mathcal{F}$ before the first recursive call, which proves the induction step for this case. For the second recursive call, we first assume $s_2 \in C$. Then, $\mathcal{F} \setminus C$ has no intersection with $\mathcal{S}_{\mathsf{spine}}$, and therefore all states in $\mathcal{F} \setminus C$ have *Charge*$[s] = 0$. Hence, the precondition stated for the next recursive call to $\mathsf{SpineSCC}_0$ holds. It remains to consider the case $s_2 \notin C$. Then, a suffix of $\mathcal{S}_{\mathsf{spine}}$ is outside C and is charged with *Charge*$[s] = 2$, and the other states in $\mathcal{F} \setminus C$ have *Charge*$[s] = 0$. For this reason, the precondition for the next call to $\mathsf{SpineSCC}_1$ holds.

Finally, consider a call to $\mathsf{SpineSCC}_1(\mathcal{S}_\varphi, \mathcal{S}_{\mathsf{spine1}}, s_1)$. By induction hypothesis, we have *Charge*$[s] = 2$ for all $s \in \mathcal{S}_{\mathsf{spine1}}$ and *Charge*$[s] = 0$ for all

$s \in \mathcal{S}_\varphi \setminus \mathcal{S}_{\mathsf{spine1}}$. According to Lemma 3.57, SkeletonSCC computes the tuple $(\mathcal{F}, C, \mathcal{S}_{\mathsf{spine2}}, s_2)$ with $2\,|\mathcal{S}_{\mathsf{spine2}}| + |C|$ symbolic operations, and hence, the charges are incremented by this number. Clearly, we then have (1) $\mathcal{S}_{\mathsf{spine2}} \subseteq \mathcal{F}$ and (2) $C \subseteq \mathcal{F}$. Moreover, the intersection of the spines $\mathcal{S}_{\mathsf{spine1}} \cap \mathcal{S}_{\mathsf{spine2}}$ is nonempty: it contains at least the state s_1. Moreover, we have (3) $\mathcal{S}_{\mathsf{spine1}} \cap \mathcal{S}_{\mathsf{spine2}} \subseteq C = \mathsf{SCC}(s_1)$: assume $s \in \mathcal{S}_{\mathsf{spine1}} \cap \mathcal{S}_{\mathsf{spine2}}$ holds. As s_1 is the end of $\mathcal{S}_{\mathsf{spine1}}$, we have (4) $s \rightsquigarrow s_1$, and as s_1 is the root of $\mathcal{S}_{\mathsf{spine2}}$, we have (5) $s_1 \rightsquigarrow s$, so that $s_1 \in \mathsf{SCC}(s_1) = C$ follows. Hence, (3) holds.

As usual, $\mathcal{S}_\varphi$ is partitioned into $\mathcal{S}_\varphi \setminus \mathcal{F}$, C, and $\mathcal{F} \setminus C$. These partitions are further partitioned as follows:

- $\mathcal{S}_\varphi \setminus \mathcal{F}$ is partitioned by $(\mathcal{S}_\varphi \setminus \mathcal{F}) \setminus \mathcal{S}_{\mathsf{spine1}}$ and $(\mathcal{S}_\varphi \setminus \mathcal{F}) \cap \mathcal{S}_{\mathsf{spine1}}$.
- $C = \mathsf{SCC}(s_1)$ is partitioned by $C \setminus \mathcal{S}_{\mathsf{spine2}}$, $(C \cap \mathcal{S}_{\mathsf{spine2}}) \setminus \mathcal{S}_{\mathsf{spine1}}$, and $(C \cap \mathcal{S}_{\mathsf{spine1}}) \cap \mathcal{S}_{\mathsf{spine2}}$.
- $\mathcal{F} \setminus C$ is partitioned by $(\mathcal{F} \setminus C) \setminus \mathcal{S}_{\mathsf{spine2}}$ and $(\mathcal{F} \setminus C) \cap \mathcal{S}_{\mathsf{spine2}}$

The states in these partitions have the following charges according the the updating of charges:

- *Charge*$[s] = 0$ for $s \in (\mathcal{S}_\varphi \setminus \mathcal{F}) \setminus \mathcal{S}_{\mathsf{spine1}}$
- *Charge*$[s] = 2$ for $s \in (\mathcal{S}_\varphi \setminus \mathcal{F}) \cap \mathcal{S}_{\mathsf{spine1}}$
- *Charge*$[s] = 1$ for $s \in C \setminus \mathcal{S}_{\mathsf{spine2}}$
- *Charge*$[s] = 3$ for $s \in (C \cap \mathcal{S}_{\mathsf{spine2}}) \setminus \mathcal{S}_{\mathsf{spine1}}$
- *Charge*$[s] = 5$ for $s \in (C \cap \mathcal{S}_{\mathsf{spine1}}) \cap \mathcal{S}_{\mathsf{spine2}}$
- *Charge*$[s] = 0$ for $s \in (\mathcal{F} \setminus C) \setminus \mathcal{S}_{\mathsf{spine2}}$
- *Charge*$[s] = 2$ for $s \in (\mathcal{F} \setminus C) \cap \mathcal{S}_{\mathsf{spine2}}$

Note that the partitions of C are not further considered by the recursive calls. It is now easily seen that the preconditions hold for the recursive calls: If $\mathcal{S}_{\mathsf{spine1}} \subseteq C$ holds, it follows that $\mathcal{S}_{\mathsf{spine1}} \subseteq \mathcal{F}$ holds, and that therefore $\mathcal{S}_\varphi \setminus \mathcal{F}$ consists of only one partition, since $(\mathcal{S}_\varphi \setminus \mathcal{F}) \cap \mathcal{S}_{\mathsf{spine1}} = \{\}$ holds. As there is no spine left of $\mathcal{S}_{\mathsf{spine1}}$, we call $\mathsf{SpineSCC}_0$ with the correct preconditions If $\mathcal{S}_{\mathsf{spine1}} \subseteq C$ does not hold, then according to fact (9) of Lemma 3.55, we extract the suffix spine of $\mathcal{S}_{\mathsf{spine1}}$ and see that the preconditions hold for the subsequent call of $\mathsf{SpineSCC}_1$.

Finally, for the second call, where $\mathcal{F} \setminus C$ is to be partitioned, assume first that $s_2 \in C$ holds. In this case, we have $\mathcal{S}_{\mathsf{spine2}} \subseteq C$ so that $(\mathcal{F} \setminus C) \cap \mathcal{S}_{\mathsf{spine2}} = \{\}$ holds. Thus, we call $\mathsf{SpineSCC}_0$ with the first partition $(\mathcal{F} \setminus C) \setminus \mathcal{S}_{\mathsf{spine2}}$ of $\mathcal{F} \setminus C$, and see that the precondition holds. In case that $s_2 \notin C$ holds, a suffix of $\mathcal{S}_{\mathsf{spine2}}$ is contained in $\mathcal{F} \setminus C$, and thus $\mathcal{F} \setminus C$ has the above mentioned two partitions, where $(\mathcal{F} \setminus C) \cap \mathcal{S}_{\mathsf{spine2}} = \mathcal{S}_{\mathsf{spine2}} \setminus C$ is a spine set. Again, the preconditions for the call to $\mathsf{SpineSCC}_1$ hold.

□

Due to the above lemma, we see that each state is charged of only a constant number. In particular, we finally have *Charge*$[s] \leq 5$ for all states, so that we conclude that at most $5\,|\mathcal{S}|$ symbolic operations are required for the entire SCC partition of a Kripke structure.

Theorem 3.59 (Correctness and Complexity of $\mathsf{SpineSCC}_i$).
The call $\mathsf{SpineSCC}_0(\mathcal{S})$ for the function $\mathsf{SpineSCC}_0$ given in Figure 3.36 computes all strongly connected components of a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ with $O(|\mathcal{S}|)$ symbolic operations. Hence, this computation runs in time $O(|\mathcal{K}|\,|\mathcal{S}|)$.

As can be seen, the usage of spines to control the selection of the next SCC to be computed allows one to reduce the $O(|\mathcal{S}|^2)$ symbolic operations as required by XieBeerel to only $O(|\mathcal{S}|)$ symbolic operations. Nevertheless, as far as asymptotic complexity is concerned, the approaches that are based on depth first traversal like Tarjan's and Kosaraju's algorithms are still better, since they run in time $O(|\mathcal{K}|)$. However, for large structures, the explicit state set representation as required by depth first traversals is impossible.

3.7.4 Fully Symbolic Computation of Fair States

Considering weak fairness, as introduced in the previous section, we have to compute the set of states $\mathcal{S}_{\mathsf{fair}}$ that satisfy the formula $\mathsf{E}\bigwedge_{i=1}^{f}\mathsf{GF}\alpha_i$ for given propositional formulas α_i. As proposed in the previous sections, this can be done by decomposing the state set $\mathcal{S}$ into its SCCs and checking the fairness of these SCCs one after the other. After this, we have to compute the union of all SCCs that can reach the detected fair SCCs. However, if the number of SCCs is large, this algorithm would perform bad, even if a symbolic representation of the state sets is used, since it explicitly enumerates the SCCS (which are stored by implicit set representation). This is often the case with systems that terminate for all inputs, reactive systems that do not terminate may contain even only one large SCC.

To handle systems with prohibitively large numbers of SCCs, one can compute the set $\mathcal{S}_{\mathsf{fair}}$ in a fully symbolic manner. To this end, we consider in this section some μ-calculus formulas that hold exactly on $\mathcal{S}_{\mathsf{fair}}$. The most popular one[9], which goes back to Emerson and Lei [177, 178], is described in the following lemma:

Lemma 3.60 (Computing $\mathcal{S}_{\mathsf{fair}}$ by the Emerson-Lei Formula). *Given a structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ over V_Σ with $\mathcal{F} = \{\alpha_1, \ldots, \alpha_f\}$. Then, the following holds:*

- $$\mathsf{E}\left[(\mathsf{G}\beta) \wedge \bigwedge_{i=1}^{f}\mathsf{GF}\alpha_i\right] = \mathsf{fix}\begin{bmatrix} x_1 \overset{\mu}{=} y \wedge \alpha_1 \vee \beta \wedge \Diamond x_1 \\ \vdots \\ x_f \overset{\mu}{=} y \wedge \alpha_f \vee \beta \wedge \Diamond x_f \\ y \overset{\nu}{=} \beta \wedge \Diamond \bigwedge_{i=1}^{f} x_i \end{bmatrix} \mathsf{in}\ y\ \mathsf{end}$$
- $$\mathsf{E}\left[(\mathsf{G}\beta) \wedge \bigwedge_{i=1}^{f}\mathsf{GF}\alpha_i\right] = \nu y.\beta \wedge \Diamond \bigwedge_{i=1}^{f}[\mu x_i.y \wedge \alpha_i \vee \beta \wedge \Diamond x_i]$$

[9] There are many variants of this formula that we discuss later on.

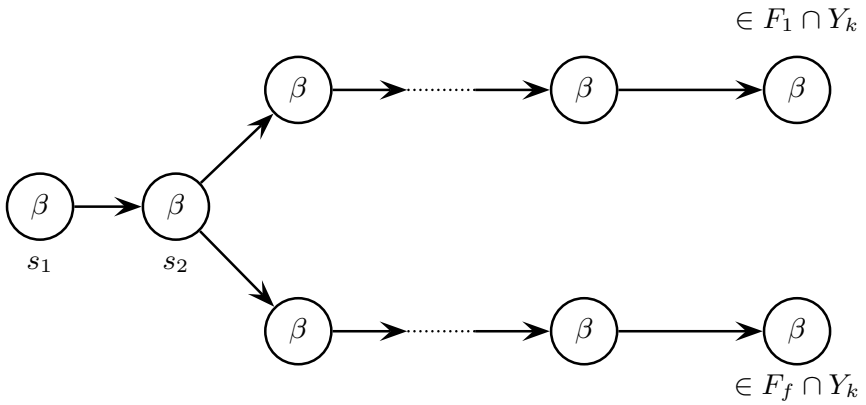

Fig. 3.37. Illustrating the computation of fair states by Emerson-Lei

Proof. Abbreviate $F_i := [\![\alpha_i]\!]_{\mathcal{K}}$, $\mathcal{F} := \{F_1, \ldots, F_f\}$, and $B := [\![\beta]\!]_{\mathcal{K}}$. It is easily seen that $[\![\mu x_i.y \wedge \alpha_i \vee \beta \wedge \Diamond x_i]\!]_{\mathcal{K}}$ is the set of states that have a possibly finite path that runs through states of B to finally reach a state in $[\![y \wedge \alpha_i]\!]_{\mathcal{K}}$. This is written as $\mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]$ in CTL. We use this notation in the following to make the proof more readable. Thus, we consider the Tarski/Knaster iteration for $\nu y.\beta \wedge \Diamond \bigwedge_{i=1}^{f} \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]$: Starting from $Y_0 := \mathcal{S}$, we iterate $Y_{k+1} := B \cap \mathsf{pre}_{\exists}^{\mathcal{R}}(\bigcap_{i=1}^{f} [\![\mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]]\!]_{\mathcal{K}_y^{Y_k}})$.

By the Tarski-Knaster theorem, we know that the sequence Y_k converges to a greatest fixpoint Y_∞. Note that we have $Y_k \subseteq B$ for $k > 0$, hence, also our fixpoint Y_∞ is a subset of B. We also know that the sequence is monotonically decreasing, i.e., that $Y_{k+1} \subseteq Y_k$ holds. To see which states are eliminated in an iteration of y, consider Figure 3.37.

Having already computed the approximation Y_k, we compute for all $i \in \{1, \ldots, f\}$ a backward traversal from $F_i \cap Y_k$ through B, and finally intersect the obtained state sets to obtain $\bigcap_{i=1}^{f} [\![\mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]]\!]_{\mathcal{K}_y^{Y_k}}$. For example, states s_1 and s_2 of Figure 3.37 are the only ones that belong to this set. After this, we apply a predecessor operation that eliminates s_2, so that only s_1 retains. s_1 also survives the intersection with B.

Hence, it is seen by induction on k that for $k > 0$, *Y_k is the set of states in B that have a successor in B that has for every k-tuple $\tau = (i_1, \ldots, i_k)$ with $i_j \in \{1, \ldots, k\}$ a possibly finite path π_τ through B that visits the state sets F_{i_1}, $\ldots$, F_{i_k} in the listed order. Hence, there are positions $t_1, \ldots t_k \in \mathbb{N}$ on π_τ such that $\pi_\tau^{(t_j)} \in F_{i_j}$ holds. In particular, the states that have such paths π_τ can visit all state sets F_i at least k times. Note that the predecessor step after the conjunction of the until's is necessary, since s_2 in Figure 3.37 may already belong to all $F_i \cap Y_k$, but may not be able to reach all of these sets $k + 1$ times.*

As a consequence, Y_∞ is the set of states that have an infinite path that visits every state set F_i. The equivalence with the vectorized fixpoint expression is trivial since this is exactly what is obtained by the algorithm of Figure 3.10 from the non-vectorized μ-calculus formula. □

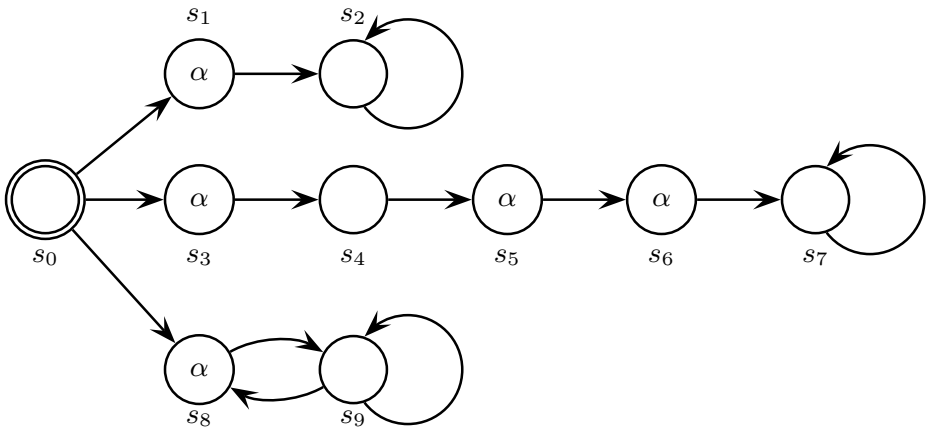

Fig. 3.38. Example Structure to Compute $\mathcal{S}_{\mathsf{fair}}$

The above fixpoint formula does more than just computing $\mathcal{S}_{\mathsf{fair}}$: additionally, it checks that another formula β must invariantly hold on the fair path. We will need that extension in the temporal logic chapter, and may omit β by instantiating $\beta := 1$ to compute $\mathcal{S}_{\mathsf{fair}}$.

As an example to see how the formula is evaluated consider the Kripke structure of Figure 3.38, where we have just one fairness constraint that is given by the states labeled with α. We evaluate the μ-calculus formula $\nu y.\Diamond[\mu x.y \wedge \alpha \vee \Diamond x]$ by the Tarski/Knaster iterations:

- $Y_0 := \{s_0, \ldots, s_9\}$ yields
  - $X_{0,0} := \{\}$
  - $X_{0,1} := \{s_1, s_3, s_5, s_6, s_8\}$
  - $X_{0,2} := \{s_0, s_1, s_3, s_4, s_5, s_6, s_8, s_9\}$
  - $X_{0,3} := X_{0,2}$
- $Y_1 := \mathsf{pre}^{\mathcal{R}}_{\exists}(X_{0,3}) = \{s_0, s_3, s_4, s_5, s_8, s_9\}$ yields
  - $X_{1,0} := \{\}$
  - $X_{1,1} := \{s_3, s_5, s_8\}$
  - $X_{1,2} := \{s_0, s_3, s_4, s_5, s_8, s_9\}$
  - $X_{1,3} := X_{1,2}$
- $Y_2 := \mathsf{pre}^{\mathcal{R}}_{\exists}(X_{1,3}) = \{s_0, s_3, s_4, s_8, s_9\}$ yields
  - $X_{2,0} := \{\}$
  - $X_{2,1} := \{s_3, s_8\}$
  - $X_{2,2} := \{s_0, s_3, s_8, s_9\}$
  - $X_{2,3} := X_{2,2}$
- $Y_3 := \mathsf{pre}^{\mathcal{R}}_{\exists}(X_{2,3}) = \{s_0, s_8, s_9\}$ yields
  - $X_{3,0} := \{\}$
  - $X_{3,1} := \{s_8\}$
  - $X_{3,2} := \{s_0, s_8, s_9\}$
  - $X_{3,3} := X_{3,2}$
- $Y_4 := Y_3 = \mathcal{S}_{\mathsf{fair}}$

As can be seen, Y_k contains the set of states that has a path where α holds at least k times. The condition $Y_k = Y_{k+1} = Y_{k+2} = \ldots$ is therefore satisfied by the set of states that can visit α infinitely often.

Establishing upper bounds for the time required to evaluate the Emerson-Lei formula is often too pessimistic: Using the upper bound of Theorem 3.40, we are forced to insert the alternation depth $\ell = 2$, $|E| \sim O(f)$, and would thus obtain $O(f^2 \, |\mathcal{K}| \, |\mathcal{S}|)$. However, this is not tight: Looking at the proof of Theorem 3.40, we estimated the runtime $T_{\mathsf{CleaStef}}(\ell)$ for an equation system with ℓ blocks as shown below. This can be instantiated for the Emerson-Lei formula to derive an upper bound $O(f \, |\mathcal{K}| \, |\mathcal{S}|)$ for its time complexity (note that $\ell = 2$, $n_1 = f$, and $n_2 = 1$ holds):

$$\begin{aligned} T_{\mathsf{CleaStef}}(\ell) &= \textstyle\sum_{i=1}^{\ell} \underbrace{\left(T_{\mathsf{CSInitBlock}}(B_i) + T_{\mathsf{CSUpdate}}(B_i)\right)}_{\in O(n_i |\mathcal{K}|)} \prod_{j=i+1}^{\ell} \left(n_j \, |\mathcal{S}| + 1\right) \\ &\in O\left(n_1 \, |\mathcal{K}| \, (n_2 \, |\mathcal{S}| + 1) + n_2 \, |\mathcal{K}|\right) \\ &\in O\left(f \, |\mathcal{K}| \, (|\mathcal{S}| + 1) + |\mathcal{K}|\right) \\ &\in O\left(f \, |\mathcal{K}| \, |\mathcal{S}|\right) \end{aligned}$$

In a similar way, a time complexity $O(f \, |\mathcal{K}| \, |\mathcal{S}|^2)$ can be proved for the algorithm Fixpoint given in Figure 3.8 on page 135.

In [431], the complexity to evaluate this formula is studied in the more general context of *SCC-hull algorithms*. Note that reachability, and hence, the existence of a fair path is an SCC-property: if a state s can reach some other state s'' at least a finite number of times k, then all states $s' \in \mathsf{SCC}(s)$ can do the same. In particular, $\mathcal{S}_{\mathsf{fair}}$ is SCC-closed. To relate this property to the above version of the Emerson-Lei formula, we have to consider the restriction $\mathcal{K}_{|\beta}$ of a Kripke structure $\mathcal{K}$ to those states $B = [\![\beta]\!]_{\mathcal{K}}$ that satisfy β. In particular, we consider the SCCs of $\mathcal{K}_{|\beta}$. Then, it is easily seen that the approximations Y_k of the Tarski-Knaster fixpoint iteration for the Emerson-Lei formula are SCC-closed and backward closed:

Lemma 3.61 (SCC- and Backward Closure of Emerson-Lei Approximants). *Given a structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ over the variables V_Σ with $\mathcal{F} = \{\alpha_1, \ldots, \alpha_f\}$. Moreover, consider the approximations of the Tarski-Knaster iteration for the evaluation of the formula given in Lemma 3.60, i.e., the sets*

- $Y_0 := \mathcal{S}$
- $Y_{k+1} := \left[\!\!\left[\beta \wedge \Diamond \bigwedge_{i=1}^{f} [\mu x_i . y \wedge \alpha_i \vee \beta \wedge \Diamond x_i]\right]\!\!\right]_{\mathcal{K}_y^{Y_k}}$.

Then, the following holds:

- *$s \in Y_k$ implies $\mathsf{SCC}_\beta(s) \subseteq Y_k$, where $\mathsf{SCC}_\beta(s) \subseteq [\![\beta]\!]_{\mathcal{K}}$ is the SCC of s in $\mathcal{K}_{|\beta}$*
- *$(s, s') \in \mathcal{R}$, $s \in [\![\beta]\!]_{\mathcal{K}}$, and $s' \in Y_k$ implies $s \in Y_k$, hence, $[\![\beta \wedge \Diamond y]\!]_{\mathcal{K}_y^{Y_k}} \subseteq Y_k$.*

The proof is obvious, just imagine the $\mathsf{SCC}_\beta(s_1)$ in Figure 3.37 and states that can reach this SCC. It is obvious that also these states have a successor that can reach each k-tuple of F_i's in the listed order, hence, also these states belong to Y_k.

For this reason, it has been observed in [431] that the approximations Y_k work SCC-wise. To explain this in more detail, consider the SCC-quotient of $\mathcal{K}$ and $\mathcal{K}_{|Y_k}$. Both graphs are acyclic, and the second one is a subgraph of the first one. SCCs that do not have successor SCCs are called *terminal* SCCs. Due to the backward closure of Y_k, it follows that $\mathcal{K}_{|Y_k}$ is obtained from $\mathcal{K}$ by a cut through SCCs. *The fixpoint iteration of the Emerson-Lei formula deletes unfair terminal SCCs of $\mathcal{K}_{|Y_k}$ until all terminal SCCs are fair.*

Hence, there are at most h iterations for Y_k when h is the length of the longest path through SCCs of the Kripke structure. For this reason, the complexity is bounded by $O(f\,|\mathcal{K}|\,h)$. This follows from the above estimations, since we can replace the number of iterations of the outer fixpoint variable y by h instead of $|\mathcal{S}|$.

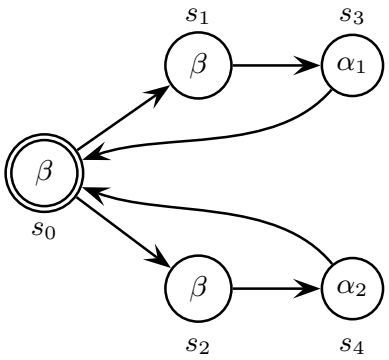

Fig. 3.39. Structure to Compare Variants of the Emerson-Lei Formula

The above computation to determine $\mathcal{S}_{\mathsf{fair}}$ goes essentially back to Emerson and Lei [177–179]. This formula can be rewritten to obtain equivalent variants [431]:

EmLe$_1$: $\nu y.\beta \wedge \Diamond \bigwedge_{i=1}^{f}[\mu x_i.y \wedge \alpha_i \vee \beta \wedge \Diamond x_i]$, i.e. $\nu y.\beta \wedge \Diamond \bigwedge_{i=1}^{f} \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]$
EmLe$_2$: $\nu y.\beta \wedge \bigwedge_{i=1}^{f} \Diamond[\mu x_i.y \wedge \alpha_i \vee \beta \wedge \Diamond x_i]$, i.e. $\nu y.\beta \wedge \bigwedge_{i=1}^{f} \Diamond\mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]$
EmLe$_3$: $\nu y.\beta \wedge \Diamond \bigwedge_{i=1}^{f}[\mu x_i.y \wedge (\alpha_i \vee \beta \wedge \Diamond x_i)]$, i.e. $\nu y.\beta \wedge \Diamond \bigwedge_{i=1}^{f} \mathsf{E}[(y \wedge \beta)\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]$
EmLe$_4$: $\nu y.\beta \wedge \bigwedge_{i=1}^{f} \Diamond[\mu x_i.y \wedge (\alpha_i \vee \beta \wedge \Diamond x_i)]$, i.e. $\nu y.\beta \wedge \bigwedge_{i=1}^{f} \Diamond\mathsf{E}[(y \wedge \beta)\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]$

All of the above formulas evaluate to the same set of states, hence, the formulas are all equivalent to each other. However, the evaluation of these formulas is not equivalent in each case, i.e., the approximants Y_k that appear during the Tarski-Knaster iteration can be different. Nevertheless, the previous lemma also holds for the approximants of the above formulas, i.e., also these are all SCC- and backward-closed.

EmLe$_1$ and *EmLe*$_2$ differ in that the modal operator $\Diamond$ is shifted over the conjunction of the fairness constraints, which makes the first variant more efficient: not only the number of predecessor operations is decreased, but also the sets Y_k converge faster: The 'inclusion' of the Y_k of *EmLe*$_1$ in the Y_k computed by *EmLe*$_2$ is easily seen by the general implication $\Diamond(\varphi \wedge \psi) \rightarrow \Diamond\varphi \wedge \Diamond\psi$. A counterexample for the reverse inclusion is given in Figure 3.39, where β does neither hold on s_3 nor on s_4. Using *EmLe*$_1$, we obtain $Y_0^{(1)} = \{s_0, \ldots, s_4\}$, $Y_1^{(1)} = \{s_0, s_1, s_2\} \cap \mathsf{pre}_\exists^{\mathcal{R}}(\{s_0, s_1, s_3\} \cap \{s_0, s_2, s_4\}) = \{s_0, s_1, s_2\} \cap \{s_3, s_4\} = \{\}$, and using *EmLe*$_2$, we obtain $Y_0^{(2)} = \{s_0, \ldots, s_4\}$, $Y_1^{(2)} = \{s_0, s_1, s_2\} \cap \mathsf{pre}_\exists^{\mathcal{R}}(\{s_0, s_1, s_3\}) \cap \mathsf{pre}_\exists^{\mathcal{R}}(\{s_0, s_2, s_4\}) = \{s_0, s_1, s_2\} \cap \{s_0, s_1, s_3, s_4\} \cap \{s_0, s_2, s_3, s_4\} = \{s_0\}$, and $Y_2^{(2)} = \{\}$. Note that s_0 can reach a_1 and a_2 at least once, but it does not have a successor with this property. For this reason, we have $s_0 \in Y_1^{(2)}$, but not $s_0 \in Y_1^{(1)}$. *Hence, EmLe*$_1$ *is more efficient than EmLe*$_2$*, both in the number of predecessor operations per iteration and in the number of iterations to compute* y*.*

EmLe$_3$ and *EmLe*$_4$ differ from *EmLe*$_1$ and *EmLe*$_2$, respectively, in that that backward traversal is not only made through β but even in the intersection $y \wedge \beta$. Again, it is easily seen that the Y_k of *EmLe*$_3$ and *EmLe*$_4$ are contained in the Y_k as computed by *EmLe*$_1$ and *EmLe*$_2$, respectively, since $[\![\mathsf{E}[(y \wedge \beta)\ \underline{\mathsf{U}}\ \varphi]]\!]_{\mathcal{K}} \subseteq [\![\mathsf{E}[\beta\ \underline{\mathsf{U}}\ \varphi]]\!]_{\mathcal{K}}$ holds. The converse also holds, since the approximants are backward closed: If $\mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]$ is computed, we start with $X_{j,0} = \{\}$, and obtain then $X_{j,1} = [\![y \wedge \alpha_i]\!]_{\mathcal{K}_y^{Y_k}}$. After this, predecessors are added that belong to $[\![\beta]\!]_{\mathcal{K}}$. By the backward closure, we know that these predecessors do also belong to Y_k, so that an additional intersection does not change the state set. Hence, *EmLe*$_1$ and *EmLe*$_3$ compute exactly the same state sets during the iterations, and also *EmLe*$_2$ and *EmLe*$_4$ are the same in each iteration.

Hence, EmLe$_1$ *and EmLe*$_3$ *are the same and are the best of the four variants (that are essentially two variants) listed above.* For a single fairness constraint, we have *EmLe*$_1 \equiv \nu y.\beta \wedge \Diamond[\mu x.y \wedge \alpha \vee \beta \wedge \Diamond x]$, and *EmLe*$_3 \equiv \nu y.\beta \wedge \Diamond[\mu x.y \wedge \alpha \vee y \wedge \beta \wedge \Diamond x]$. For this case, there are even further variants:

- $\nu y.\mu x.\beta \wedge \Diamond[y \wedge \alpha \vee x]$ [163]
- $\nu y.\mu x.\beta \wedge \Diamond y \wedge \alpha \vee \beta \wedge \Diamond x$ [523]

The evaluation of the Emerson-Lei formula has been considered in detail by many authors [52, 197, 254, 288, 431]. As the iteration of the outer fixpoint deletes unfair terminal SCCs, it is clear that the algorithm behaves bad if the Kripke structure consists of a single finite path where all states belong to α, so that $|\mathcal{S}|$ iterations are required for y. The reason for this is that many trivial SCCs have to be eliminated, which is only a small change in the outer fixpoint that invokes completely new computations of the inner fixpoints.

For example, consider Figure 3.40, which has been taken from [197]. The evaluation of *EmLe*$_1$ is as follows: we start with $Y_0 = \{s_1, \ldots, s_{2n+1}\}$ and

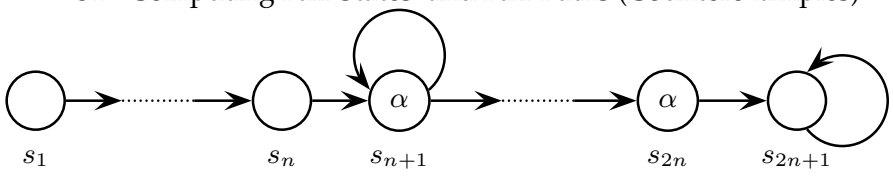

Fig. 3.40. Structure to Motivate the Hojati-Kesten Formula

compute then for $k \in \{1, \ldots, n-1\}$ the sets $Y_k = \{s_1, \ldots, s_{2n-k}\}$. Each computation of an Y_k requires $O(n)$ iterations, thus we end up with $O(n^2)$ iterations/symbolic operations. The idea to eliminate this bad behaviour is to eliminate deadend states after each iteration of the outermost fixpoint y. With this modification, only $O(n)$ symbolic steps are required to compute the fair states of the structure of Figure 3.40.

The idea to eliminate deadend states between the iterations of the outer fixpoint has already been proposed in [536, 537] in form of finite distance predecessors, in [254, 288] to eliminate trivial SCCs, and in [197] to make the overall algorithm more balanced, i.e., that the work that is spent for the inner and outer fixpoints should be more or less the same.

Lemma 3.62 (Deadend Elimination During Fixpoint Iteration). *Given a structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ over V_Σ with $\mathcal{F} = \{\alpha_1, \ldots, \alpha_f\}$. Then, the following holds:*

- $$\mathsf{E}\left[(\mathsf{G}\beta) \wedge \bigwedge_{i=1}^{f} \mathsf{GF}\alpha_i\right] = \mathsf{fix} \begin{bmatrix} x_1 \overset{\mu}{=} y \wedge \alpha_1 \vee \beta \wedge \Diamond x_1 \\ \vdots \\ x_f \overset{\mu}{=} y \wedge \alpha_f \vee \beta \wedge \Diamond x_f \\ z \overset{\nu}{=} y \wedge \Diamond z \\ y \overset{\nu}{=} \beta \wedge z \wedge \bigwedge_{i=1}^{f} x_i \end{bmatrix} \mathsf{in}\ y\ \mathsf{end}$$
- $$\mathsf{E}\left[(\mathsf{G}\beta) \wedge \bigwedge_{i=1}^{f} \mathsf{GF}\alpha_i\right] = \mathsf{fix} \begin{bmatrix} x_1 \overset{\mu}{=} y \wedge \alpha_1 \vee \beta \wedge \Diamond x_1 \\ \vdots \\ x_f \overset{\mu}{=} y \wedge \alpha_f \vee \beta \wedge \Diamond x_f \\ z \overset{\nu}{=} \Diamond z \wedge \beta \wedge \bigwedge_{i=1}^{f} x_i \\ y \overset{\nu}{=} z \end{bmatrix} \mathsf{in}\ y\ \mathsf{end}$$
- $$\mathsf{E}\left[(\mathsf{G}\beta) \wedge \bigwedge_{i=1}^{f} \mathsf{GF}\alpha_i\right] = \nu y.\Diamond y \wedge \beta \wedge \bigwedge_{i=1}^{f} [\mu x_i.y \wedge \alpha_i \vee \beta \wedge \Diamond x_i]$$
- $$\mathsf{E}\left[(\mathsf{G}\beta) \wedge \bigwedge_{i=1}^{f} \mathsf{GF}\alpha_i\right] = \nu y.\mathsf{EG}\left(\beta \wedge \bigwedge_{i=1}^{f} \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_i)]\right)$$

In [254], it is proved that replacing the $\Diamond$ operator in the Emerson-Lei formula by a EG operator does not change the final result. However, it gives the algorithm a better 'balance' between the work that is spent on the outer and the inner fixpoints, which has also been observed in [197].

As an example, reconsider Figure 3.38 on page 201. We have the following computations:

- $Y_0 := \{s_0, \ldots, s_9\}$ yields
  - $X_{0,0} := \{\}$
  - $X_{0,1} := \{s_1, s_3, s_5, s_6, s_8\}$
  - $X_{0,2} := \{s_0, s_1, s_3, s_4, s_5, s_6, s_8, s_9\}$
  - $X_{0,3} := X_{0,2}$
  - $Z_{0,0} := \{s_1, \ldots, s_9\}$
  - $Z_{0,1} := \{s_0, s_1, s_3, s_4, s_5, s_6, s_8, s_9\}$
  - $Z_{0,2} := \{s_0, s_3, s_4, s_5, s_8, s_9\}$
  - $Z_{0,3} := \{s_0, s_3, s_4, s_8, s_9\}$
  - $Z_{0,4} := \{s_0, s_3, s_8, s_9\}$
  - $Z_{0,5} := \{s_0, s_8, s_9\}$
  - $Z_{0,6} := Z_{0,5}$
- $Y_1 := Z_{0,6} = \{s_0, s_8, s_9\}$ yields
  - $X_{1,0} := \{\}$
  - $X_{1,1} := \{s_0, s_8, s_9\}$
  - $X_{1,2} := X_{1,1}$
  - $Z_{1,0} := \{s_1, \ldots, s_9\}$
  - $Z_{1,1} := \{s_0, s_8, s_9\}$
  - $Z_{1,2} := Z_{1,1}$
- $Y_2 := Y_1 = \mathcal{S}_{\mathsf{fair}}$

However, [197] also lists an example where *EmLe*$_1$ behaves better than *HoKo*. For this reason, it has been argued in [197] that model checkers should implement at least two symbolic algorithms to compute $\mathcal{S}_{\mathsf{fair}}$: in addition to *EmLe*$_1$, also *HoKo* should be implemented, since both algorithms behave incomparably.

Using the vectorized μ-calculus, we have even more freedom to construct formulas for computing $\mathcal{S}_{\mathsf{fair}}$. The one that is probably what many programmer would implement first is *EmLe*$_5$. For $f = 2$, we can flatten the vectorized expression as follows:

$$\nu y.\beta \wedge \Diamond \mathsf{E}[\beta \,\underline{\mathsf{U}}\, (\beta \wedge \alpha_2 \wedge \Diamond \mathsf{E}[\beta \,\underline{\mathsf{U}}\, (y \wedge \alpha_1)])]$$

Lemma 3.63 (Computing $\mathcal{S}_{\mathsf{fair}}$ by). *Given a structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ over V_Σ with $\mathcal{F} = \{\alpha_1, \ldots, \alpha_f\}$. Then, the following are all equivalent:*

$$\Phi_{\mathsf{fair}}\text{:}\ \mathsf{E}\left[(\mathsf{G}\beta) \wedge \bigwedge_{i=1}^{f} \mathsf{GF}\alpha_i\right]$$

$$\mathit{EmLe}_1\text{:}\ \mathsf{fix}\begin{bmatrix} x_1 = \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_1)] \\ \vdots \\ x_f = \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_f)] \\ y \overset{\nu}{=} \beta \wedge \Diamond \bigwedge_{i=1}^{f} x_i \end{bmatrix} \mathsf{in}\ y\ \mathsf{end}$$

$$\mathit{EmLe}_5\text{:}\ \mathsf{fix}\begin{bmatrix} x_1 \overset{\mu}{=} \beta \wedge \Diamond\mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_1)] \\ x_2 \overset{\mu}{=} \beta \wedge \Diamond\mathsf{E}[\beta\ \underline{\mathsf{U}}\ (x_1 \wedge \alpha_2)] \\ \vdots \\ x_f \overset{\mu}{=} \beta \wedge \Diamond\mathsf{E}[\beta\ \underline{\mathsf{U}}\ (x_{f-1} \wedge \alpha_f)] \\ y \overset{\nu}{=} x_f \end{bmatrix} \mathsf{in}\ y\ \mathsf{end}$$

$$\mathit{HoHa}\text{:}\ \mathsf{fix}\begin{bmatrix} x_1 \overset{\mu}{=} \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \alpha_1)] \wedge \mathsf{E}[\beta\ \underline{\overleftarrow{\mathsf{U}}}\ (y \wedge \alpha_1)] \\ x_2 \overset{\mu}{=} \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (x_1 \wedge \alpha_2)] \wedge \mathsf{E}[\beta\ \underline{\overleftarrow{\mathsf{U}}}\ (x_1 \wedge \alpha_2)] \\ \vdots \\ x_f \overset{\mu}{=} \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (x_{f-1} \wedge \alpha_f)] \wedge \mathsf{E}[\beta\ \underline{\overleftarrow{\mathsf{U}}}\ (x_{f-1} \wedge \alpha_f)] \\ z \overset{\nu}{=} z \wedge \beta \wedge \Diamond z \wedge \overleftarrow{\Diamond} z \\ y \overset{\nu}{=} z \end{bmatrix} \mathsf{in}\ y\ \mathsf{end}$$

$$\mathit{OneHoHa}\text{:}\ \mathsf{fix}\begin{bmatrix} x_1 \overset{\mu}{=} \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (y \wedge \Diamond(y \wedge \alpha_1)] \\ x_2 \overset{\mu}{=} \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (x_1 \wedge \Diamond(x_1 \wedge \alpha_2)] \\ \vdots \\ x_f \overset{\mu}{=} \mathsf{E}[\beta\ \underline{\mathsf{U}}\ (x_{f-1} \wedge \Diamond(x_{f-1} \wedge \alpha_f)] \\ z \overset{\nu}{=} x_f \wedge \Diamond z \\ y \overset{\nu}{=} z \end{bmatrix} \mathsf{in}\ y\ \mathsf{end}$$

[254] and [288]

In the meantime, other methods have been developed that have been compared by Somenzi et. al. in [431]. To describe these fixpoints in a more readable form, we abbreviate first the following macros:

- $\mathsf{BW}(\psi) := \mu y.\ \psi \vee \Diamond y$
- $\mathsf{FW}(\psi) := \mu y.\ \psi \vee \overleftarrow{\Diamond} y$
- $\mathsf{BW}_\varphi(\psi) := \mu y.\ \psi \vee \varphi \wedge \Diamond y$
- $\mathsf{FW}_\varphi(\psi) := \mu y.\ \psi \vee \varphi \wedge \overleftarrow{\Diamond} y$
- $\overleftarrow{\mathsf{EG}}(\varphi) := \nu y.\ \varphi \wedge \overleftarrow{\Diamond} y$
- $\mathsf{Cycle}(\varphi) := \nu y.\ \varphi \wedge y \wedge \overleftarrow{\Diamond} y \wedge \Diamond y$

Note that $\mathsf{BW}(\psi)$ holds on the set of states that have a possibly finite path that leads to a state where ψ holds. $\mathsf{BW}_\varphi(\psi)$ additionally requires that φ must

hold on every state of the (possibly finite) path up to the state where ψ holds. Similarly, $\mathsf{FW}(\psi)$ corresponds with the set of states that can be reached from a state where ψ holds, and $\mathsf{FW}_\varphi(\psi)$ additionally requires that φ must hold on every state of the path from the state where ψ held. $\overleftarrow{\mathsf{EG}}(\varphi)$ holds on the set of states where one can go infinitely long backwards through states where φ holds. Finally, $\mathsf{Cycle}(\varphi)$ is the union of all states that lie on a cycle where φ holds. Using these macros, some formulas to compute $\mathcal{S}_{\mathsf{fair}}$ are as follows (we added β to the original formulas):

Hojati-Kesten [254, 288]:

$$\mathsf{BW}(\nu y.\overleftarrow{\mathsf{EG}}\left(\beta \wedge \bigwedge_{i=1}^{f} \mathsf{FW}_{y\wedge\beta}(y \wedge \alpha_i)\right))$$

Hojati-Hardin [239, 254]:

$$\mathsf{BW}(\nu y.\mathsf{Cycle}\left(\beta \wedge \bigwedge_{i=1}^{f} y \wedge \mathsf{FW}_\beta(y \wedge \alpha_i) \wedge \mathsf{BW}_\beta(y \wedge \alpha_i)\right))$$

The differences of the above formulas are best explained by considering the strongly connected components (SCCs) of the Kripke structure. To this end, recall that the SCC-quotient is an acyclic graph. We call SCCs without predecessors in this graph initial SCCs, and SCCs without successors terminal SCCS.

We have already discussed the first Emerson-Lei formulas in detail. The second variant is a minor modification that restricts the backward search to the previously computed fixpoint approximation. In [116], this algorithm has been used to compute fair paths. To save expensive predecessor operations, Ravi, Bloem, and Somenzi's version add the fact that the predecessor operation $\Diamond$ can be shifted over the conjunction. Consider how the third formula works on the SCC-quotient graph of a Kripke structure (restricted to β): $\bigwedge_{i=1}^{f} \mathsf{BW}_{y\wedge\beta}(y \wedge \alpha_i)$ is the set of states that can reach all fairness constraints within the current set y. In each iteration of y, terminal SCCs that are not fair are eliminated. This is done in the same way as the deadend states are eliminated during the evaluation of $\Phi_{\mathsf{inf}} :\equiv \nu y.\Diamond x$ to compute $\mathcal{S}_{\mathsf{inf}}$. Hence, the result of y is the set $\mathcal{S}_{\mathsf{fair}}$.

In the Hojati-Kesten formula, the subformula $\bigwedge_{i=1}^{f} \mathsf{FW}_{y\wedge\beta}(y \wedge \alpha_i)$ yields the set of states that can be reached from all sets $y \wedge \alpha_i$. This set S is forward closed, i.e., if s is in this set, and $(s, s') \in \mathcal{R}$ holds, then also s' is in this set. It may however not be possible to go infinitely long backwards inside this state set. Thus, the outer iteration due to $\overleftarrow{\mathsf{EG}}$ removes the states that can not be reached from a cycle in S. The iterations due to $\overleftarrow{\mathsf{EG}}$ therefore remove initial unfair SCCs (instead of terminal unfair SCCs as by Emerson-Lei). Finally, a backward reachability has to compute $\mathcal{S}_{\mathsf{fair}}$, i.e., the states that can reach a fair SCC. The computation of counterexamples by this iteration is presented in [288].

The Hojati-Hardin formula [239, 254] is a mixture between the Hojati-Kesten and the Emerson-Lei formulas. The conjunction yields the set of states

that can reach and that can be reached from all sets $y \wedge \alpha_i$. The Cycle operator removes states that do not lie on a cycle in this set.

In [431], the runtimes of these algorithms have been compared by some experimental benchmarks. As a result, [431] states that none of the newer algorithms performs better than the old Emerson-Lei approach on a significant number of examples. In [197], however, it is argued that in addition to the Emerson-Lei formula, another 'balanced' variant of Hojati-Hardin should be implemented in a model checker.

3.8 Final Remarks on Completeness and Expressiveness

We conclude this chapter with final remarks on the μ-calculus. Fixpoints are manyfold in mathematics, and in particular, in computer science, and therefore there are a lot of applications for the μ-calculus. We only list some of them to show the role of the μ-calculus, not only as machinery for verifying reactive systems, but also for other fields in computer science.

The first remarkable point is that the model checking problem for the μ-calculus is in $\mathsf{NP} \cap \mathsf{coNP}$ [41, 173, 345]. This means that a nondeterministic machine is able to solve every μ-calculus model checking problem in polynomial time (see Appendix B.3). As for every formula φ, its negation $\neg\varphi$ is also a μ-calculus formula, it follows that the μ-calculus model checking problem is also in coNP. The interesting point here is that there are only a few problems in $\mathsf{NP} \cap \mathsf{coNP}$, where no deterministic polynomial algorithm is known. For examples, checking the primality of a given number is such a problem that has been recently shown [5] to have a polynomial decision procedure. Hence, there is still a hope that we can find more efficient model checking algorithms in the future. The latest improvement was that made by Jurdziński [277] in that he showed that the upper bound can be improved to $\mathsf{UP} \cap \mathsf{coUP}$ [277], which seems to be sharper than $\mathsf{NP} \cap \mathsf{coNP}$. UP is the class of problems that can be solved in polynomial time by an unambiguous Turing machine. An unambiguous Turing machine may have several runs for an input, but only one of these runs is accepting. Hence, this machine is not too nondeterministic.

Having seen that the μ-calculus is also of interest for complexity theory, we consider three further related fields in this section. In the next section, we consider the relationship of bisimulation relations and the μ-calculus, then the relationship to alternating tree automata and games, and finally the historical origins of the μ-calculus, namely dynamic logic and the Hoare calculus.

3.8.1 Bisimilarity and the Future Fragment

In this section, we derive two important results: First, we establish a relationship between bisimulation relations and the μ-calculus, and second we prove