Semi-Automatic Distributed Synthesis

Bernd Finkbeiner and Sven Schewe

Universität des Saarlandes, 66123 Saarbrücken, Germany
{finkbeiner|schewe}@cs.uni-sb.de

Abstract. We propose a sound and complete compositional proof rule for distributed synthesis. Applying our proof rule only requires the manual strengthening of the specification into a conjunction of formulas that can be guaranteed by individual black-box processes. All premises of the proof rule can be checked automatically.
For this purpose, we give an automata-theoretic synthesis algorithm for single processes in distributed architectures. The behavior of the local environment of a process is unknown in the process of synthesis and cannot be assumed to be maximal. We therefore consider reactive environments that have the power to disable some of their own actions, and provide methods for synthesis (and realizability checking) in this setting. We establish upper bounds for CTL (2EXPTIME) and CTL* (3EXPTIME) synthesis with incomplete information, matching the known lower bounds for these problems, and provide matching upper and lower bounds for μ-calculus synthesis (2EXPTIME) with complete or incomplete information. Synthesis in reactive environments is harder than synthesis in maximal environments, where CTL, CTL* and μ-calculus synthesis are EXPTIME, 2EXPTIME and EXPTIME complete, respectively.

1 Introduction

In the synthesis of distributed systems, we transform a given specification into a collection of finite-state programs that satisfy the specification when composed according to a given architecture. For some restricted architectures, such as pipelines and rings in which only one designated process communicates with the environment [1], synthesis can be done automatically. However, as soon as the architecture contains an *information fork*, i.e., a pair of processes that have an incomparable degree of information about the system state, the problem becomes undecidable [2].

In this paper, we investigate a *semi-automatic* approach where we synthesize one process at a time. It turns out that the synthesis of a single process can be done automatically and it is always possible to decompose a realizable specification into a conjunction of properties that can be guaranteed by single processes. This approach therefore works for all distributed architectures, including those with information forks.

The problem of synthesizing a single process has been studied in a number of variations. *Closed synthesis* excludes any interaction with the environment

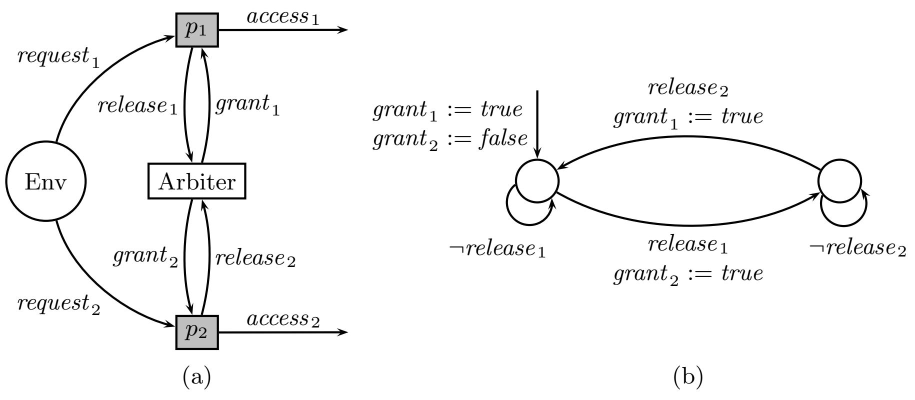

Fig. 1. A simple distributed shared-resource application. (a) The system architecture. An edge between two process nodes p and q labeled with variable v indicates that v is an output variable of process p and an input variable of process q. (b) The implementation of the white-box process Arbiter, represented as a finite-state automaton.

[3, 4]. *Open synthesis* finds implementations that satisfy a specification in any environment. For universal specifications (e.g., ACTL*), it suffices to consider the *maximal* environment, which shows all possible behaviors [5, 6]. In general, it is necessary to account for *reactive environments*, which may disable some of their responses [7].

We consider the problem of synthesizing a single black-box process in a given distributed *architecture*. An architecture consists of an external environment and a set of system processes, which we partition into subsets of *white-box* and *black-box* processes: each white-box process comes with a known and fixed implementation, while the implementation of the black-box processes is yet to be found.

A single black-box process may interact with the external environment, the white-box processes, and with other black-box processes. Like in open synthesis, we assume that the behavior of the external environment is maximal. The behavior of white-box processes is known beforehand, but may be nondeterministic. The other black-box processes show reactive [7] behavior: in each state, they may disable some (but not all) of their responses. An important difference between synthesizing systems that consist of a single process, and synthesizing a single process within a general architecture is that, while the process has *complete information* about the system state in the former case, it only sees a part of the state as defined by the architecture in the latter case.

Figure 1a shows the architecture of a simple distributed *shared-resource* application. The external environment Env can request *access* to the resource by setting the *request* variable of one of the two black-box processes p_1 and p_2. Mutual exclusion is accomplished using a white-box Arbiter process that alternates a *grant* between p_1 and p_2, such that each process retains the grant until the respective *release* variable is set, as shown in Figure 1b.

We can specify the expected behavior of the shared-resource system as a conjunction $\psi = \psi_1 \wedge \psi_2 \wedge \psi_3$ of three CTL* formulas, where the first two formulas specify that there is a way for both processes to use the resource infinitely often ($\psi_i = \text{EGF}\ \mathit{access}_i$ for $i \in \{0,1\}$) and the third formula specifies mutual exclusion ($\psi_3 = \text{AG}\ \neg(\mathit{access}_1 \wedge \mathit{access}_2)$).

Obviously, neither p_1 nor p_2 can guarantee ψ for *all* possible implementations of the other process (for example, if the other process constantly sets its *access* variable to *true*, mutual exclusion must be violated in some branch). We therefore strengthen ψ into two separate properties φ_{p_1} and φ_{p_2} that can be guaranteed by p_1 and p_2, respectively. A natural assumption to be made by process p_{3-i} about process p_i is that there is path, such that process p_i infinitely often releases the grant ($\alpha_1^{p_i} = \text{EGF}\ \mathit{release}_i$) and that, on every path, p_i only accesses the resource when permitted by Arbiter ($\alpha_2^{p_i} = \text{AG}\ \mathit{access}_i \rightarrow \mathit{grant}_i$). By adding these assumptions, we obtain a strengthened specification $\varphi = \varphi_{p_1} \wedge \varphi_{p_2}$ where

$$\varphi_{p_i} = \quad \alpha_1^{p_i} \wedge \alpha_2^{p_i} \quad \wedge \quad (\alpha_1^{p_{3-i}} \wedge \alpha_2^{p_{3-i}} \;\rightarrow\; \psi).$$

Once the auxiliary formulas φ_{p_1} and φ_{p_2} have been defined, an implementation can be found automatically. For example, process p_i can guarantee φ_{p_i} against any implementation of process p_{3-i}, by setting access_i after each $\mathit{request}_i$ as soon as grant_i becomes *true* and by setting $\mathit{release}_i$ in the immediately following state.

Contribution. We propose a sound and complete compositional proof rule for distributed synthesis. Applying our proof rule only requires the manual strengthening of the specification into a conjunction of formulas that can be guaranteed by individual black-box processes against the other black-box processes. All premises of the proof rule can be checked automatically.

For this purpose, we give an automata-theoretic synthesis algorithm for single processes in distributed architectures. Our environment model builds on open synthesis [5], but combines the maximal external environment with reactive black-box processes. Synthesis in reactive environments was studied before, but only under the assumption of complete information [7].

Our construction turns a specification into an alternating parity automaton accepting exactly the reactive models of a specification. For a specification φ with length $n = |\varphi|$ in CTL, CTL* and μ-calculus this automaton has $n^{O(n)}$, $2^{2^{O(n)}}$ and $n^{O(n^3)}$ states, respectively. We establish 2EXPTIME and 3EXPTIME upper bounds for synthesis with incomplete information in case of CTL and CTL* specifications, respectively. We defer a doubly exponential lower bound for μ-calculus specifications from the doubly exponential lower bound for CTL and establish a matching upper bound.

Overview. In the following section, we formally introduce the synthesis problem studied in this paper. We explain the compositional synthesis rule in Section 3. The synthesis algorithm is presented in Section 4.

2 Setting

In the *distributed synthesis* problem, we decide for a pair (A, φ), consisting of an architecture A and a specification φ, whether there exists a finite-state program (or *strategy*) for each black-box process in A, such that the joint behavior satisfies φ.

Architectures. An architecture

$$A = (B, W, \{I_p\}_{p \in B \uplus W \uplus \{env\}}, \{O_p\}_{p \in B \uplus W \uplus \{env\}}, \{s_w\}_{w \in W})$$

is given as a set of processes $P = B \uplus W \uplus \{env\}$ that is decomposed into a set B of black-box processes that have to be developed, a set W of white-box processes that already have an implementation $\{s_w\}_{w \in W}$, and the external environment *env*. The processes communicate through a set V of shared variables, which also serve as atomic propositions in the specification. Each process $p \in P$ has a fixed set of input and output variables $I_p, O_p \subseteq V$, such that the family of output variables $\{O_p\}_{p \in P}$ decomposes V. The environment is always omniscient ($I_{env} = V$).

Implementations. A process p is implemented by a (nondeterministic) *strategy*, i.e., a function $s_p : (2^{I_p})^* \to 2_{\emptyset}^{2^{O_p}}$ (where $2_{\emptyset}^{X} = 2^X \smallsetminus \{\emptyset\}$ denotes the nonempty subsets of a set X). A strategy is *finite-state* if it can be represented by a finite-state automaton. The implementations $\{s_w\}_{w \in W}$ of the white-box processes W are fixed for the architecture. An *implementation* of an architecture is a set of strategies $S = \{s_b\}_{b \in B}$ for the black-box processes.

We use trees as a representation for strategies and computations. As usual, an Υ-*tree* is given as a prefix-closed subset $Y \subseteq \Upsilon^*$ of all finite words over a given set of directions Υ. If the set of directions is not important or clear from the context, we call Y a tree. We define that every non-empty node $x \cdot \upsilon$, $x \in \Upsilon^*, \upsilon \in \Upsilon$, has the direction $dir(x \cdot \upsilon) = \upsilon$ and the empty word ε has some designated *root-direction* $dir(\varepsilon) = \upsilon_0 \in \Upsilon$. An Υ-tree Y is called *total*, if it contains the empty word $\varepsilon \in Y$ and every element $y \in Y$ of the tree has at least one successor $y \cdot \upsilon \in Y, \upsilon \in \Upsilon$. If $Y = \Upsilon^*$, the tree is called *full*.

For given finite sets Σ and Υ, a Σ-*labeled* Υ-*tree* is a pair $\langle Y, l \rangle$, consisting of a tree $Y \subseteq \Upsilon^*$ and a labeling function $l : Y \to \Sigma$ that maps every node of Y to a letter of Σ. The *successor-tree* $\langle Y, sucset \rangle$ of a tree Y is the 2^{Υ}-labeled Υ-tree, where every node is labeled with the set of its successors $sucset : Y \to 2^{\Upsilon}$, $sucset : y \mapsto \{\upsilon \in \Upsilon | y \cdot \upsilon \in Y\}$.

For a set $\Xi \times \Upsilon$ of directions and a node $x \in (\Xi \times \Upsilon)^*$, $hide_{\Upsilon}(x)$ denotes the node in Ξ^* obtained from x by replacing (ξ, υ) by ξ in each letter of x. For a Σ-labeled Ξ-tree $\langle \Xi^*, l \rangle$ we define the Υ-widening of $\langle \Xi^*, l \rangle$, denoted by $wide_{\Upsilon}(\langle \Xi^*, l \rangle)$, as the Σ-labeled $\Xi \times \Upsilon$-tree $\langle (\Xi \times \Upsilon)^*, l' \rangle$ with $l'(x) = l(hide_{\Upsilon}(x))$.

We consider specifications φ that are given as CTL, CTL*, or μ-calculus formulas. Such specifications define a set $\mathcal{M}_{\varphi}$ of total 2^{AP}-labeled Υ-trees, where $AP = V$ denotes the set of atomic propositions in φ.

Let $S_Q = \bigotimes_{p\in Q} 2_{\emptyset}^{2^{O_p}}$ denote the set of possible common outputs of a set of strategies for the processes in $Q \subseteq B \cup W$. The *composition* $\bigoplus_{p\in Q} s_p = s_Q : (2^V)^* \to S_Q$ of a set of strategies $\{s_p\}_{p\in Q}$ maps the global input history to the common output of the processes in Q: For $\langle (2^V)^*, s'_p\rangle = \mathit{wide}_{2^{V\smallsetminus I_p}}(\langle (2^V)^*, s'_p\rangle)$, $s_Q : y \mapsto \biguplus_{p\in Q} s'_p(y)$ naturally defines a S_Q-labeled 2^V-tree.

A *non-distributed implementation* of the processes B' is a function

$$s_{B'} : (2^{I_{B'}})^* \to S_{B'}, \text{ for } I_{B'} = \bigcup_{b\in B'} I_b.$$

A *(distributed) implementation* is a set of strategies $\{s_b\}_{b\in B'}$ whose composition is the widening of a non-distributed implementation: $\langle (2^V)^*, \bigoplus_{b\in B'} s_b\rangle = \mathit{wide}_{2^{V\smallsetminus I_{B'}}}(\langle (2^{I_{B'}})^*, s_{B'}\rangle)$ for some non-distributed implementation $s_{B'}$.

Realizability. An implementation $s_{B'}$ of a set $B' \subseteq B$ of black-box processes *guarantees φ against the remaining black-box processes*, if for all $S_{B\smallsetminus B'}$-labeled 2^V-trees $\langle (2^V)^*, s_{B\smallsetminus B'}\rangle$, the total 2^V-labeled 2^V-tree $\langle Y, \mathit{dir}\rangle$, whose branching restriction $\mathit{sucset}(y) = (s_{B'} \oplus \bigoplus_{w\in W} s_w \oplus s_{B\smallsetminus B'})(y) \times 2^{O_{env}}$ is defined by the strategies, is a model of φ.

We say that a specification is *realizable by the processes $B' \subseteq B$* for a given architecture $(B, W, \{I_p\}_{p\in B\uplus W}, \{O_p\}_{p\in B\uplus W}, O_{env}, \{s_w\}_{w\in W})$, $(A, B') \vDash \varphi$, if there is a distributed implementation of the processes B' that guarantees φ against $B \smallsetminus B'$. A specification is *realizable* if it is realizable by the entire set of black-box processes B.

3 A Compositional Synthesis Rule

The compositional synthesis rule reduces the realizability of a distributed system, $(A, B) \vDash \psi$, to the realizability of single processes, $(A, \{b\}) \vDash \varphi_b$, for each black-box process $b \in B$. The proof rule requires an auxiliary specification φ_b for each process $b \in B$. If each process b guarantees φ_b against the remaining black-box processes, the distributed system can be implemented to satisfy ψ.

For a distributed architecture A with a set of black-box processes $B = \{b_1, \cdots, b_n\}$, and CTL* or μ-calculus formulas $\psi, \varphi_{b_1}, \ldots \varphi_{b_n}$,

$$\begin{array}{lll}
\text{(R0)} & (A, \emptyset) & \vDash \bigwedge_{b\in B} \varphi_b \to \psi \\
\text{(R1)} & (A, \{b_1\}) & \vDash \varphi_{b_1} \\
\vdots & & \vdots \\
\text{(R}n\text{)} & (A, \{b_n\}) & \vDash \varphi_{b_n} \\
\hline
 & (A, B) & \vDash \psi
\end{array}$$

Premise (R0) shows that the auxiliary formulas $\varphi_{b_1}, \ldots, \varphi_{b_n}$ strengthen the original formula φ and hence any implementation that satisfies $\varphi_{b_1}, \ldots, \varphi_{b_n}$ must also satisfy φ. Premises (R1) through (Rn) prove that there are, for all b_i in B, strategies s_{b_i} that guarantee φ_{b_i} against the remaining black-box processes.

Theorem 1. *The proof rule is sound.*

Proof. Premises (R1) through (Rn) guarantee that, for each $b \in B$, there is an implementation s_b that guarantees φ_b against the remaining black-box processes $B \smallsetminus \{b\}$. Consequently, the strategies can be fixed independently; the distributed implementation thus obtained satisfies φ_b for all $b \in B$ and hence $\bigwedge_{b \in B} \varphi_b$. Premise (R0) guarantees that every non-distributed implementation of $\bigwedge_{b \in B} \varphi_b$ is also an implementation of ψ. As the distributed implementations form a subset of the non-distributed implementations, the claim holds true. $\square$

To show the completeness of the distribution rule, we derive the auxiliary formulas from a given implementation that realizes the specification: for a given architecture, we call a specification *strict*, if it completely determines its implementation. An implementation can be described by a strict LTL specification φ. A distributed implementation can be described by a strict specification φ_b for every black-box component $b \in B$, such that $\varphi = \bigwedge_{b \in B} \varphi_b$ is a strict specification for the implementation.

Theorem 2. *The proof rule is complete.*

Proof. Assume there is a distributed implementation for a specification ψ and $\varphi = \bigwedge_{b \in B} \varphi_b$ is a strict specification for this implementation. Then $(A, \{b\}) \vDash \varphi_b$ holds true for each $b \in B$. The implementation of φ is completely determined and $(A, \emptyset) \vDash \varphi \rightarrow \psi$ requires that every specification of φ is an implementation of ψ. As the unique implementation is by definition an implementation of ψ, $(A, \emptyset) \vDash \varphi \rightarrow \psi$ also holds true. $\square$

4 Single-Process Synthesis

We now develop a procedure that checks if a specification can be guaranteed by a single black-process b against the remaining black-box processes, $(A, \{b\}) \vDash \varphi$, as required for premises (R1) through (Rn), and a procedure that checks if a specification can be guaranteed by the empty set of black-processes against all black-box processes, $(A, \emptyset) \vDash \varphi$, as required for premise (R0).

Every formula of a temporal logic can be translated into an alternating tree automaton that accepts exactly its set of models. This automaton is the starting point for our construction, which consists of a series of tree automata transformations.

4.1 Tree automata

An *alternating parity tree automaton* is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, \alpha)$, where Q denotes a finite set of states, $q_0 \in Q$ denotes a designated initial state, δ denotes a transition function, and $\alpha : Q \to C \subset \mathbb{N}$ is a coloring function. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ maps a state and an input letter to a positive boolean combination of states and directions (for a predefined finite set Υ of directions).

An alternating automaton runs on full Σ-labeled Υ-trees. A *run tree* $\langle R, r\rangle$ on a given full Σ-labeled Υ-tree $\langle \Upsilon^*, l\rangle$ is a $Q \times \Upsilon^*$-labeled tree where the root is labeled with (q_0, ε) and where, for each node n with a label (q, y) with the set of labels of its successors $L = \{r(n \cdot \rho) | \rho \in \mathit{sucset}(n)\}$, there is a set $A \subseteq 2^{Q\times\Upsilon}$ which satisfies $\delta(q, l(y))$ such that $(q', \upsilon) \in A \Leftrightarrow (q', y \cdot \upsilon) \in L$.

An infinite path fulfills the *parity condition*, if the highest color of the states appearing infinitely often on the path is even. A run tree is *accepting* if all infinite paths fulfill the parity condition. A total Σ-labeled Υ-tree is accepted if it has an accepting run tree.

The set of trees accepted by an alternating automaton $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. $\overline{\mathcal{L}(\mathcal{A})}$ denotes the set of full Σ-labeled Υ-trees not accepted by $\mathcal{A}$. An automaton is empty, if its language is empty.

The acceptance of a tree can also be viewed as the outcome of a game, where player *accept* chooses, for a pair $(q, \sigma) \in Q \times \Sigma$, a set of atoms of $\delta(q, \sigma)$, satisfying $\delta(q, \sigma)$, and player *reject* chooses one of these atoms, which is executed. The input tree is accepted iff player *accept* has a strategy enforcing a path that fulfills the parity condition. One of the player has a memoryless winning strategy, i.e., a strategy where the moves only depend on the state of the automaton, the position in the tree and, for player *react*, on the choice of player *accept* in the same move.

A *nondeterministic* automaton is a special alternating automaton, where the image of δ consists only of such formulae that, when rewritten in disjunctive normal form, contain exactly one element of $Q \times \{\upsilon\}$ for all $\upsilon \in \Upsilon$ in every disjunct.

For nondeterministic automata, every node of a run tree corresponds to a node in the input tree. Emptiness can therefore be checked with an *emptiness game*, where player *accept* also chooses the letter of the input alphabet. A nondeterministic automaton is empty iff the emptiness game is won by *reject*.

Symmetric alternating automata are a variant of alternating automata that run on total Σ-labeled Υ-trees. For a symmetric alternating automaton $\mathcal{S} = (\Sigma, Q, q_0, \delta, \alpha)$, Q, q_0, and α are defined as before. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \{\Box, \Diamond\})$ now maps a state and an input letter to a positive boolean combination over atoms that refer to *some* ($\Diamond$) or *all* ($\Box$) successor states.

A *run tree* on a given Σ-labeled Υ-tree $\langle R, r\rangle$ is a $Q \times \Upsilon^*$-labeled tree where the root is labeled with (q_0, ε) and where, for a node n with a label (q, y) and a set of labels of its successors $L = \{r(n \cdot \rho) | \rho \in \mathit{sucset}(n)\}$, the following property holds: there is a set of atoms $A \subseteq 2^{Q\times\{\Box,\Diamond\}}$ satisfying $\delta(q, l(y))$ such that $\forall q' \in Q.((q', \Box) \in A \Rightarrow \forall \upsilon \in \mathit{sucset}(x).(q', y \cdot \upsilon) \in L) \wedge ((q', \Diamond) \in A \Rightarrow \exists \upsilon \in \mathit{sucset}(x).(q', y \cdot \upsilon) \in L)$.

We introduce a function $\mathit{suc} : (Q \times \Sigma \to \mathbb{B}^+(Q \times \{\Box, \Diamond\})) \to (Q \times \Sigma \times 2^{\Upsilon}_{\emptyset} \to \mathbb{B}^+(Q \times \Upsilon))$ that translates the transition function of a symmetric alternating automaton running on total Σ-labeled Υ-trees into the corresponding transition function of an alternating automaton running on full $\Sigma \times 2^{\Upsilon}_{\emptyset}$-labeled Υ-trees. For the set $2^{\Upsilon}_{\emptyset} = 2^{\Upsilon} \smallsetminus \{\emptyset\}$ of possible sets of successors, $\mathit{suc}(\delta) : Q \times \Sigma \times 2^{\Upsilon}_{\emptyset} \to \mathbb{B}^+(Q \times \Upsilon)$ maps a state, an input letter and a set of successors to a positive boolean combination of states and directions.

4.2 Overview

We represent the joint behavior of a system as a total 2^V-labeled 2^V-tree $\langle Y, \mathit{dir} \rangle$, where the label is completely determined by the direction. The process strategies determine the tree: By the proper widening of a strategy $s'_p : (2^V)^* \to 2^{2^{O_p}}_{\emptyset}$, each input history (or initial sequence of a path) is mapped to a nonempty subset of 2^{O_p}, restricting the set of successors. The nodes of Y consist of the root and all nodes $y \cdot \upsilon$ whose predecessor y is in Y, and whose direction agrees with the decisions of the processes: $y \cdot \upsilon \in Y \Leftrightarrow y \in Y \wedge \forall p \in B \uplus W. \upsilon \cap O_p \in s'_p(y)$.

We start our construction with a symmetric automaton $\mathcal{S}_\varphi$ that accepts the models of the specification φ. Automata transformations are simpler for automata running on full trees; we therefore represent total trees as full trees by decorating each node with its own set of successors. Considering a full $2^V \times 2^{2^V}_{\emptyset}$-labeled 2^V-tree $\langle (2^V)^*, l' \rangle$, where the nodes are additionally decorated with the sets of relevant successors, one can easily determine the original total 2^V-labeled 2^V-tree $\langle Y, l \rangle$, which we call its *characteristic tree*.

We continue with an automaton that accepts those full $2^V \times 2^{2^V}_{\emptyset}$-labeled 2^V-trees whose characteristic tree is a model of φ. The labeling of the nodes $(2^V)^* \smallsetminus Y$ of $\langle (2^V)^*, l' \rangle$ that are not on the characteristic tree has no influence on the acceptance of the tree. We restrict the language under consideration to $2^V \times S_b \times S_W \times S_{B'} \times S$-labeled 2^V-trees, where S_b, S_W, $S_{B'}$ and S describe the possible restrictions on the successor sets induced by the black-box process b, the set of white-box processes W, the remaining black-box processes $B' = B \smallsetminus \{b\}$, and the environment, respectively. By that, we obtain an automaton $\mathcal{A}_\varphi$ that accepts $2^V \times S_b \times S_W \times S_{B'} \times S$-labeled 2^V-trees. Since $S_p = 2^{2^{O_p}}_{\emptyset}$ for all processes[1] $p \in B \uplus W$, the sets of possible restrictions can be identified with $S_W = \bigotimes_{w \in W} S_w$, $S_{B'} = \bigotimes_{b \neq b' \in B} S_{b'}$ and $S = \{2^{O_{env}}\}$ (as we assume the environment to be maximal).

It remains to find a strategy s_b such that for its proper widening s'_b, $\langle (2^V)^*, \mathit{dir} \times s'_b \times (\bigoplus_{w \in W} s_w) \times s_{B'} \times \{O_{env}\} \rangle$ is accepted for all strategies $s_{B'} : (2^V)^* \to \bigotimes_{b \neq b' \in B} 2^{2^{O_{b'}}}_{\emptyset}$.

[1] For generality, we allow all processes to be nondeterministic. If a subset $D \subseteq B \uplus W$ of the processes is to be deterministic, one can simply choose, for all $p \in D$, the set of singleton subsets of 2^{O_p} instead of the set of non-empty subsets.

We first build an automaton $\mathcal{R}_\varphi$ that accepts a $2^V \times S_b \times S_W$-labeled 2^V-tree, if its complete cylinder is accepted by $\mathcal{A}_\varphi$, establishing independence from the decision of the black-box processes. Subsequently, we use the determination of the 2^V and S_W fraction of the label to defer an automaton $\mathcal{D}_\varphi$ that accepts all strategy trees $\langle (2^V)^*, s_b \rangle$ that would guarantee φ against the remaining black-box processes if process b were omniscient. This automaton is then transformed into an automaton $\mathcal{B}_\varphi$ accepting the strategies of b ($2^{2_\emptyset^{O_b}}$-labeled 2^{I_b}-trees).

Checking this automaton for emptiness answers the question of realizability. In case of realizability, the emptiness test can be extended to synthesize a finite-state strategy for b.

In summary, our construction consists of seven steps:

1. ***From formulas to automata:*** We construct a symmetric alternating automaton $\mathcal{S}_\varphi$ that accepts the models of φ.
2. ***Characteristic trees:*** The alternating automaton $\mathcal{A}_\varphi$ accepts a $2^V \times S_b \times S_W \times S_{B'} \times S$-labeled 2^V-tree if its characteristic tree is accepted by $\mathcal{S}_\varphi$.
3. ***Quantification:*** The alternating automaton $\mathcal{R}_\varphi$ accepts a $2^V \times S_b \times S_W$-labeled 2^V-tree if all $S_{B'} \times S$ extensions are accepted by $\mathcal{A}_\varphi$.
4. ***Adjusting for white-box processes:*** The alternating automaton $\mathcal{W}_\varphi$ accepts a $2^V \times S_b$-labeled 2^V-tree if the $2^V \times S_b \times S_W$-labeled 2^V-tree obtained by adding the decisions of the white-box processes is accepted by $\mathcal{R}_\varphi$.
5. ***Pruning directions from the labeling:*** The alternating automaton $\mathcal{D}_\varphi$ accepts a S_b-labeled 2^V-tree if the $2^V \times S_b$-labeled 2^V-tree obtained by adding the direction of a node to the label is accepted by $\mathcal{W}_\varphi$.
6. ***Narrowing:*** The alternating automaton $\mathcal{B}_\varphi$ accepts a S_b-labeled 2^{I_b}-tree if its proper widening is accepted by $\mathcal{D}_\varphi$.
7. ***Emptiness check:*** The realizability claim $(A, \{b\}) \vDash \varphi$ holds true iff $\mathcal{B}_\varphi$ is not empty. To perform an emptiness test, $\mathcal{B}_\varphi$ can be transformed into an equivalent nondeterministic automaton $\mathcal{C}_\varphi$, which can be checked for emptiness by solving the emptiness game. A winning strategy in the emptiness game implies an implementation for the process b.

In the following, we discuss the automata transformations in detail.

4.3 Automata Transformations

From formulas to automata. We use standard constructions to translate a temporal specification φ into a symmetric alternating automaton $\mathcal{S}_\varphi$ that accepts the models of the formula: $\mathcal{L}(\mathcal{S}_\varphi) = \mathcal{M}_\varphi$.

Theorem 3. *Given a CTL specification φ, we can construct a symmetric alternating automaton $\mathcal{S}_\varphi$ with $O(|\varphi|)$ states and two colors such that $\mathcal{L}(\mathcal{S}_\varphi) = \mathcal{M}_\varphi$ [8]. Given a CTL* specification φ, we can construct a symmetric alternating automaton $\mathcal{S}_\varphi$ with $2^{O(|\varphi|)}$ states and five colors such that $\mathcal{L}(\mathcal{S}_\varphi) = \mathcal{M}_\varphi$ [8]. Given a μ-calculus specification φ, we can construct a symmetric alternating automaton $\mathcal{S}_\varphi$ with $O(|\varphi|^2)$ states and $O(|\varphi|)$ colors such that $\mathcal{L}(\mathcal{S}_\varphi) = \mathcal{M}_\varphi$ [6].* □

Characteristic trees. For a $\Sigma \times \Xi$-labeled Υ-tree $\langle Y, l\rangle$, we denote the Σ-*projection* $proj_\Sigma : \langle Y, l\rangle \mapsto \langle (Y, l_\Sigma\rangle$ with $l(y) = (\sigma, \xi) \Rightarrow l_\Sigma : y \mapsto \sigma$ that maps $\Sigma \times \Xi$-labeled Υ-trees to Σ-labeled Υ-trees.

For a full $\Sigma \times 2^\Upsilon_\emptyset$-labeled Υ-tree $\langle \Upsilon^*, l\rangle$, we define the *characteristic tree* as the total Σ-labeled Υ-tree $\langle Y, l_c\rangle = char(\langle \Upsilon^*, l\rangle)$ to be the sub-tree of $proj_\Sigma(\langle \Upsilon^*, l\rangle)$ with $y \in Y \Rightarrow \forall \upsilon \in \Upsilon . y \cdot \upsilon \in \Upsilon \Leftrightarrow \upsilon \in proj_{2^\Upsilon_\emptyset}(\langle \Upsilon^*, l\rangle)$. Intuitively, the second argument in the label defines the set of successors of a node.

Lemma 1. *Given a symmetric alternating automaton $\mathcal{S} = (\Sigma, Q, q_0, \delta, \alpha)$, running on total Σ-labeled Υ-trees, we can construct an alternating automaton $\mathcal{A} = (\Sigma \times 2^\Upsilon_\emptyset, Q, q_0, suc(\delta), \alpha)$ that accepts a full $\Sigma \times 2^\Upsilon_\emptyset$ labeled Υ-tree $\langle \Upsilon^*, l\rangle$, iff $proj_\Sigma(char(\langle \Upsilon^*, l\rangle))$ is accepted by $\mathcal{S}$.*

Proof. Let $\langle T, l_T\rangle = char(\langle \Upsilon^*, l\rangle)$. Then the successor set of a node $x \in T$ is defined by the label: $sucset(x) = proj_{2^\Upsilon_\emptyset}(l_{T(x)}) = proj_{2^\Upsilon_\emptyset}(l(x))$. □

Quantification. To construct an alternating automaton $\mathcal{R}_\varphi$ that accepts a $2^V \times S_b \times S_W$-labeled 2^V-tree if all $S_{B'} \times S$ extensions are accepted by $\mathcal{A}_\varphi$, we

1. complement $\mathcal{A}_\varphi$, i.e., we compute an alternating automaton $\mathcal{I}_\varphi$ with $\mathcal{L}(\mathcal{I}_\varphi) = \overline{\mathcal{L}(\mathcal{A}_\varphi)}$,
2. build a nondeterministic automaton $\mathcal{N}_\varphi$ with the same language $\mathcal{L}(\mathcal{N}_\varphi) = \mathcal{L}(\mathcal{A}_\varphi)$,
3. compute a nondeterministic automaton $\mathcal{P}_\varphi$ that accepts a $2^V \times S_b \times S_W$-labeled 2^V-tree if it is the the $S_{B'} \times S$-projection of a tree accepted by $\mathcal{N}_\varphi$,
4. complement $\mathcal{P}_\varphi$, i.e., we compute an alternating automaton $\mathcal{R}_\varphi$ with $\mathcal{L}(\mathcal{R}_\varphi) = \overline{\mathcal{L}(\mathcal{P}_\varphi)}$.

Lemma 2. *[9] Given an alternating automaton $\mathcal{A} = (\Sigma, Q, q_0, \delta, \alpha)$ that runs on Σ-labeled Υ-trees, the dual automaton $\mathcal{I} = (\Sigma, Q, q_0, \overline{\delta}, \alpha + 1)$, where $\overline{\delta}$ is the function dual to δ, accepts a tree $\langle \Upsilon^*, l\rangle$ iff $\langle \Upsilon^*, l\rangle$ is not accepted by $\mathcal{S}$.* □

Lemma 3. *[2, 10] Given an alternating automaton $\mathcal{A}$ with n states and c colors, we can construct an equivalent nondeterministic automaton $\mathcal{N}$ with $n^{O(c \cdot n)}$ states and $O(c \cdot n)$ colors.* □

Lemma 4. *Given a nondeterministic automaton $\mathcal{N} = (\Sigma \times \Xi, Q, q_0, \delta, \alpha)$ that runs on $\Sigma \times \Xi$-labeled Υ-trees, we can construct a nondeterministic automaton $\mathcal{P} = (\Sigma, Q, q_0, \delta', \alpha)$ that accepts a Σ-labeled Υ-tree $\langle \Upsilon^*, l\rangle$ iff there is a $\Sigma \times \Xi$-labeled Υ-tree $\langle \Upsilon^*, l_\Xi\rangle$ accepted by $\mathcal{N}$ with $\langle \Upsilon^*, l\rangle = proj_\Sigma(\langle \Upsilon^*, l_\Xi\rangle)$.*

Proof. $\mathcal{P}$ can be constructed by using δ' to guess the correct tree: we set $\delta' : (q, \sigma) \mapsto \bigvee_{\xi \in \Xi} \delta(q, (\sigma, \xi))$. □

In the following two transformations, the decisions of the white-box processes and the labeling imposed by the directions are deleted from the label.

Adjusting for white-box processes. The S_W fraction of the label represents the decisions made by the white box processes. Consequently, we are only interested in those trees, where the label of every node is in accordance with these decisions. This information is then redundant and can be pruned. We assume that the composed strategy $\bigoplus_{w\in W} s_w$ of the white-box processes is represented as a finite-state automaton $\mathcal{O} = (2^V, O, o_0, d_W, o_W)$, where O is a set of states, o_0 the initial state, the transition function $d_W : 2^V \times O \to O$ is a mapping from the input alphabet and the set of states to the set of states, and the output function $o_W : O \to 2_{\emptyset}^{2^{O_W}}$ maps each state to a nonempty set of output letters. The following operation performs the pruning; the state-space of the resulting automaton is linear in the state-space of the original automaton and the number of states of $\mathcal{O}$, while the set of colors remains unchanged.

Lemma 5. *Given an alternating automaton $\mathcal{R} = (\Sigma \times \Xi, Q, q_0, \delta, \alpha)$ over $\Sigma \times \Xi$-labeled Υ-trees and a finite automaton $\mathcal{O} = (\Sigma, O, o_0, d_W, o_W)$ that produces a Ξ-labeled Υ-tree $\langle \Upsilon^*, l\rangle$, we can construct an alternating automaton $\mathcal{W} = (\Sigma, Q \times O, (q_0, o_0), \delta', \alpha')$ over Σ-labeled Υ-trees, such that $\mathcal{W}$ accepts $\langle \Upsilon^*, l'\rangle$ iff $\mathcal{R}$ accepts $\langle \Upsilon^*, l''\rangle$ with $l'' : y \mapsto (l'(y), l(y))$.*

Proof. If $\delta : (q, \sigma, \xi) \mapsto b_{(q,\sigma,\xi)}(\{q_i, \upsilon_i\}_{i\in I})$, we can set $\delta' : (q, o, \sigma) \mapsto b_{(q,\sigma,o_W(o))}(\{q_i, d_W(\sigma, o), \upsilon_i\}_{i\in I})$. The coloring function can simply be set to $\alpha' : (q, o) \mapsto \alpha(q)$. □

Pruning directions from the labeling. We are only interested in those trees where the label of every node is in accordance with its direction. This information then becomes redundant and can be pruned. The following operation performs this pruning; the state-space of the resulting automaton is linear in the state-space of the original automaton, while the set of colors remains unchanged.

For a Σ-labeled Υ-tree $\langle \Upsilon^*, l\rangle$, we define the function *xray* : $\langle \Upsilon^*, l\rangle \mapsto \langle \Upsilon^*, l'\rangle$ with $l'(x) = (dir(x), l(x))$ that maps Σ-labeled Υ-trees to $\Upsilon \times \Sigma$-labeled Υ-trees.

Lemma 6. *[8] Given an alternating automaton $\mathcal{W} = (\Upsilon \times \Sigma, Q, q_0, \delta, \alpha)$ over $\Upsilon \times \Sigma$-labeled Υ-trees, we can construct an alternating automaton $\mathcal{D} = (\Sigma, Q \times \Upsilon, (q_0, \upsilon_0), \delta', \alpha')$ over Σ-labeled Υ-trees, such that $\mathcal{D}$ accepts $\langle \Upsilon^*, l\rangle$ iff $\mathcal{R}$ accepts $xray(\langle \Upsilon^*, l\rangle)$.* □

The transition function $\delta' : Q \times \Upsilon \times \Sigma \to \mathcal{B}^+(Q \times \Upsilon \times \Upsilon)$ can be constructed from $\delta : Q \times \Upsilon \times \Sigma \to \mathcal{B}^+(Q \times \Upsilon)$ by replacing all occurrences of (q, υ) in each $\delta(q', \upsilon', \sigma')$ by (q, υ, υ), storing the direction as quasi-input. $\alpha' : (q, c) \mapsto \alpha(q)$ simply evaluates the first component of the new state-space.

Narrowing. The process b is in general not omniscient, and its output may only depend on the history of the input visible to b. The following transformation therefore accepts a 2^{O_p}-labeled 2^{I_p}-tree if its proper widening is accepted by $\mathcal{D}_\varphi$. The state-space and the set of colors remain unchanged.

Lemma 7. *[8] Given an alternating automaton $\mathcal{D} = (\Sigma, Q, q_0, \delta, \alpha)$ over Σ-labeled $\Xi \times \Upsilon$-trees, we can construct an alternating automaton $\mathcal{B} = (\Sigma, Q, q_0, \delta', \alpha)$ over Σ-labeled Ξ-trees, such that $\mathcal{B}$ accepts $\langle \Xi^*, l\rangle$ iff $\mathcal{W}$ accepts $wide_\Upsilon(\langle \Xi^*, l\rangle)$.* □

δ' can be constructed from δ by replacing all occurrences of $(q, (\xi, \upsilon))$ by (q, ξ) in $\delta(q', \sigma)$ for all $q, q' \in Q, \sigma \in \Sigma, \xi \in \Xi$ and $\upsilon \in \Upsilon$.

Emptiness check. To perform an emptiness test, $\mathcal{B}_\varphi$ can be transformed into an equivalent nondeterministic automaton $\mathcal{C}_\varphi$.

Theorem 4. *Given a symmetric alternating automaton $\mathcal{S}_\varphi$ that accepts the models of φ, an architecture $(B, W, \{I_p\}_{p\in B\uplus W\uplus\{env\}}, \{O_p\}_{p\in B\uplus W\uplus\{env\}}, \{s_w\}_{w\in W})$ and a designated black-box process $b \in B$, we can construct a nondeterministic automaton $\mathcal{C}_\varphi$ that accepts a full 2^{O_p}-labeled 2^{I_p}-tree $\langle (2^{I_p})^*, s_b\rangle$ iff s_b guarantees φ against $B \smallsetminus \{b\}$. If $\mathcal{S}$ has n states and c colors, $\mathcal{C}$ has $2^{n^{O(n\cdot c)}}$ states and $n^{O(n\cdot c)}$ colors.*

Proof. By applying the transformation steps in the order described in the overview of the algorithm, we obtain an alternating automaton $\mathcal{B}_\varphi$ with $n^{O(n\cdot c)}$ states and $O(n \cdot c)$ colors that accepts an implementation $\langle (2^{I_b})^*, s_b\rangle$ of a process b if it guarantees φ against $B \smallsetminus \{b\}$. A nondeterminisation of $\mathcal{B}_\varphi$ by the construction of Lemma 3 provides the required automaton. □

Theorem 5. *For a given architecture A and a black-box process b, we can check $(A, \{b\}) \vDash \varphi$ and, if the claim is true, provide an implementation for b guaranteeing φ, in 2EXPTIME in the length $|\varphi|$ if φ is a CTL or μ-calculus specification, and in 3EXPTIME in $|\varphi|$ if φ is a CTL* specification, respectively.*

Proof. By Theorem 3, we can turn a specifications φ in CTL, μ-calculus or CTL* with length $n = |\varphi|$ into a symmetric alternating automaton $\mathcal{S}$ with $O(n)$ states and two colors, $O(n^2)$ states and $O(n)$ colors or $2^{O(n)}$ states and five colors, respectively.

By Theorem 4, we can transform the symmetrical alternating automaton $\mathcal{S}$ into a nondeterministic automaton $\mathcal{C}$, accepting the strategies of b that guarantee φ against the remaining black-box processes. $\mathcal{C}$ has $2^{n^{O(n)}}$ states and $n^{O(n)}$ colors, $2^{n^{O(n^3)}}$ states and $n^{O(n^3)}$ colors or $2^{2^{2^{O(n)}}}$ states and $2^{2^{O(n)}}$ colors, respectively.

The actual emptiness test or the synthesis of a strategy for process n can be done in time polynomial in the state-space and exponential in the number of colors. More precisely, if $\mathcal{C}$ has m states and c colors, a strategy (or the proof of emptiness) can be found in $m^{O(c)}$ time [11]. The overall time complexity is hence $2^{n^{O(n)}}$, $2^{n^{O(n^3)}}$ and $2^{2^{2^{O(n)}}}$, respectively. □

Lower Bounds. To demonstrate that the upper bounds are sharp, we give a reduction from the synthesis problem in reactive environments with complete information, which is known to be 2EXPTIME and 3EXPTIME hard for CTL and CTL*, respectively [7]. In synthesis with reactive environments and complete information, we have only one process b, for which a (deterministic) strategy $s_b : (2^{O_{env}})^* \to S_b$ is sought (where S_b is the set of singleton subsets of $2^{O_{env}}$. The environment can react on the input by restricting its actions to a non-empty subset of its output variables O_e, which can be viewed as a non-deterministic strategy $s_e : (2^{O_{env} \cup O_b})^* \to 2^{2^{O_e}}_{\emptyset}$). In our terms, a strategy $s_b : (2^{O_{env}})^* \to S_b$ implements a specification φ if, for all strategies $s_e : (2^{O_{env} \cup O_b})^* \to 2^{2^{O_{env}}}_{\emptyset}$ of the environment, $s_b \times s_e$ is a model of φ.

We encode this synthesis problem as the realizability of φ by b against a black-box process e with output O_e and an environment without output. The second black-box process e plays the rôle of the reactive environment. Formally, we define the architecture $A = (\{b, e\}, \emptyset, \{I_b = O_e, I_e = I_{env} = V\}, \{O_b, O_e, O_{env} = \emptyset\}, \emptyset)$. The determinacy of s_b can be guaranteed by the construction (by setting S_b to the set of singleton subsets of 2^{O_b}). Alternatively, we can ensure the determinacy of s_b by strengthening the specification φ such that only deterministic strategies are allowed: For $\psi = \bigwedge_{o \in O_b} AG\ (EXo \ \to \ AXo)$, we can solve the realizability problem for $\varphi' = \varphi \wedge \psi$ (which is linear in φ).

Theorem 6. *The realizability problem $(A, \{b\}) \models \varphi$ is 3EXPTIME complete for CTL* and 2EXPTIME complete for CTL and μ-calculus specifications in the size $|\varphi|$ of the specification.*

Proof. The lower bounds for CTL and CTL* follow from the equal lower bounds for the synthesis problem with reactive environments. The lower bound for the μ-calculus is established by the lower bound for CTL. The upper bound is demonstrated by Theorem 5. ☐

Premise R0. The correctness of premise R0 can be checked along the same lines: we check whether the empty strategy guarantees $\bigwedge_{b \in B} \varphi_b \to \psi$ against all black-box processes. Since $S_b = \{\emptyset\}$ and $I_b = \emptyset$, the automaton $\mathcal{B}_\varphi$ (with n states and c colors) is an alternating word automaton over the single-letter alphabet, whose emptiness can be checked in $n^{O(c)}$ time. Checking (R0) is therefore in EXPTIME for CTL and μ-calculus specifications and in 2EXPTIME for CTL* specifications, respectively, in $|\bigwedge_{b \in B} \varphi_b \to \psi|$.

5 Conclusions

In open synthesis, where we synthesize a system that consists of a single process, it is safe to assume that the environment behavior is *maximal*. For the synthesis of a black-box process in the architecture of a general distributed system, the environment model needs two extensions: (1) The other black-box processes add a *reactive* component to the environment and (2) the process only has *incomplete information* about the environment behavior.

Extension (1) turns out to be expensive. Adding the reactive component increases the complexity for CTL specifications from EXPTIME [8] to 2EXPTIME [7], and for CTL* specifications from 2EXPTIME [8] to 3EXPTIME [7]. As shown in Section 4, extension (2) has no extra cost. This settles an open question of [7]: The complexity of synthesizing a single process in a distributed architecture is still 2EXPTIME and 3EXPTIME, respectively.

The complexity of single-process synthesis is especially convincing in comparison to the cost of distributed synthesis: in the rare cases where distributed synthesis is decidable, the cost of synthesizing a distributed system with n processes (with distinguishable degree of information about the system state) is n-exponential in the size of the specification [1, 2].

Dividing the synthesis problem into several synthesis problems for single processes therefore appears as a promising approach to cope with the complexity and general undecidability of distributed synthesis. The situation is similar to the *verification* of distributed systems, where the compositional approach is well-established [12]. Our proof rule in Section 3 is a first example of a compositional synthesis technique. The rule is complete and therefore sufficient to decompose any realizable specification. The rule may, however, be less convenient to use than some compositional verification rules that, for example, apply circular assume-guarantee reasoning [13]. Defining such rules for the synthesis problem is an interesting topic of future research.

References

1. Kupferman, O., Vardi, M.Y.: Synthesizing distributed systems. In: IEEE Symposium on Logic in Computer Science. (2001)
2. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: IEEE Symposium on Logic in Computer Science. (2005)
3. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proc. IBM Workshop on Logics of Programs. Volume 131 of LNCS., Springer-Verlag (1981) 52–71
4. Wolper, P.: Synthesis of Communicating Processes from Temporal-Logic Specifications. PhD thesis, Stanford University (1982)
5. Kupferman, O., Vardi, M.Y.: Synthesis with incomplete informatio. In: Proc. 2nd International Conference on Temporal Logic (ICTL'97). (1997)
6. Kupferman, O., Vardi, M.Y.: μ-calculus synthesis. In: Proc. 25th International Symposium on Mathematical Foundations of Computer Science. Volume 1893 of LNCS., Springer-Verlag (2000) 497–507
7. Kupferman, O., Madhusudan, P., Thiagarajan, P., Vardi, M.Y.: Open systems in reactive environments: Control and synthesis. In: Proc. 11th Int. Conf. on Concurrency Theory. Volume 1877 of LNCS., Springer-Verlag (2000) 92–107
8. Kupferman, O., Vardi, M.Y.: Church's problem revisited. The bulletin of Symbolic Logic **5** (1999) 245–263
9. Muller, D.E., Schupp, P.E.: Alternating automata on infinite trees. Theor. Comput. Sci. **54** (1987) 267–276
10. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of rabin, mcnaughton and safra. Theor. Comput. Sci. **141** (1995) 69–107

11. Jurdziński, M.: Small progress measures for solving parity games. In: 17th Annual Symposium on Theoretical Aspects of Computer Science. Volume 1770 of LNCS., Springer-Verlag (2000) 290–301
12. de Roever, W.P., Langmaack, H., Pnueli, A., eds.: Compositionality: The Significant Difference. COMPOS'97. Volume 1536 of LNCS., Springer Verlag (1998)
13. Maier, P.: A Lattice-Theoretic Framework For Circular Assume-Guarantee Reasoning. PhD thesis, Universität des Saarlandes, Saarbrücken (2003)