Detecting Unrealizable Specifications of Distributed Systems*

Bernd Finkbeiner and Leander Tentrup

Saarland University, Germany

Abstract. Writing formal specifications for distributed systems is difficult. Even simple consistency requirements often turn out to be unrealizable because of the complicated information flow in the distributed system: not every information is available in every component, and information transmitted from other components may arrive with a delay or not at all, especially in the presence of faults. The problem of checking the distributed realizability of a temporal specification is, in general, undecidable. Semi-algorithms for synthesis, such as bounded synthesis, are only useful in the positive case, where they construct an implementation for a realizable specification, but not in the negative case: if the specification is unrealizable, the search for the implementation never terminates. In this paper, we introduce *counterexamples to distributed realizability* and present a method for the detection of such counterexamples for specifications given in linear-time temporal logic (LTL). A counterexample consists of a set of paths, each representing a different sequence of inputs from the environment, such that, no matter how the components are implemented, the specification is violated on *at least one* of these paths. We present a method for finding such counterexamples both for the classic distributed realizability problem and for the distributed realizability problem with faulty nodes. Our method considers, incrementally, larger and larger sets of paths until a counterexample is found. While counterexamples for full LTL may consist of infinitely many paths, we give a semantic characterization such that the required number of paths can be bounded. For this fragment, we thus obtain a decision procedure. Experimental results, obtained with a QBF-based prototype implementation, show that our method finds simple errors very quickly, and even problems with high combinatorial complexity, like the Byzantine Generals' Problem, are tractable.

1 Introduction

The goal of program synthesis, and systems engineering in general, is to build systems that satisfy a given specification. Sometimes, however, this goal is unattainable, because the conditions of the specification are *impossible* to satisfy in an

* This work was partially supported by the German Research Foundation (DFG) as part of SFB/TR 14 AVACS and by the Saarbrücken Graduate School of Computer Science, which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 78–92, 2014.

implementation. A textbook example for such a case is the *Byzantine Generals' Problem*, introduced in the early 1980s by Lamport et al. [1]. Three generals of the Byzantine army, consisting of one commander and two lieutenants, need to agree on whether they should "attack" or "retreat." For this purpose, the commander sends an order to the lieutenants, and all generals then exchange messages with each other, reporting, for example, to one general which messages they have received from the other general. The problem is that one of the generals is a traitor and can therefore not be assumed to tell the truth: the tale of the Byzantine generals is, after all, just an illustration for the problem of achieving fault tolerance in distributed operating systems, where we would like to achieve consensus even if a certain subset of the nodes is faulty. Of course, we cannot expect the traitor to agree with the loyal generals, but we might still expect a loyal lieutenant to agree with the order issued by a loyal commander, and two loyal lieutenants to reach a consensus in case the commander is the traitor. This specification is, however, unrealizable in the setting of the three generals (and, more generally, in all settings where at least a third of the nodes are faulty).

Detecting unrealizable specifications is of great value because it avoids spending implementation effort on specifications that are impossible to satisfy. If the system consists of a single process, then unrealizable specifications can be detected with *synthesis* algorithms, which detect unrealizability as a byproduct of attempting to construct an implementation. For distributed systems, the problem is more complicated: in order to show that there is no way for the three generals to achieve consensus, we need to argue about the knowledge of each general. The key observation in the Byzantine Generals' Problem is that the loyal generals have no way of knowing who, among the other two generals, is the traitor and who is the second loyal general. For example, the situation where the commander is the traitor and orders one lieutenant to "attack" and the other to "retreat" is *indistinguishable*, from the point of view of the loyal lieutenant who is ordered to attack, from the situation where the commander is loyal and orders both lieutenants to attack, while the traitor claims to have received a "retreat" order. Since the specification requires the lieutenant to act differently (agree with the other lieutenant vs. agree with the commander) in the two indistinguishable situations, we reach a contradiction.

Since realizability for distributed systems is in general an undecidable problem [2], the only available decision procedures are limited to special cases, such as pipeline and ring architectures [3, 4]. There are semi-algorithms for distributed synthesis, such as *bounded synthesis* [5], but the focus is on the search for implementations rather than on the search for inconsistencies: if an implementation exists, the semi-algorithm terminates with such an implementation, otherwise it runs forever. In this paper, we take the opposite approach and study *counterexamples to realizability*. Intuitively, a counterexample collects a sufficient number of scenarios such that, no matter what the implementation does, an error will occur in *at least one* of the chosen scenarios. As specifications, we consider formulas of linear-time temporal logic (LTL). It is straightforward to encode the Byzantine Generals' Problem in LTL. Another interesting example is the famous

CAP Theorem, a fundamental result in the theory of distributed computation conjectured by Brewer [6]. The CAP Theorem states that it is impossible to design a distributed system that provides Consistency, Availability, and Partition tolerance (CAP) simultaneously. We assume there is a fixed number n of nodes, that every node implements the same service, and that there are direct communication links between all nodes. We use the variables req_i and out_i to denote input and output of node i, respectively. The consistency and availability requirements can then be encoded as the LTL formulas $\bigwedge_{1 \leq i < n} (\mathrm{out}_i \leftrightarrow \mathrm{out}_{i+1})$ and $(\bigvee_{1 \leq i \leq n} \mathrm{req}_i) \leftrightarrow (\bigcirc \bigvee_{1 \leq i \leq n} \mathrm{out}_i)$. The partition tolerance is modeled in a way that there is always at most one node partitioned from the rest of the system.

In both examples, a finite set of input sequences suffices to force the system into violating the specification on at least one of the input sequences. In this paper, we present an efficient method for finding such counterexamples. It turns out that searching for counterexamples is much easier than the classic synthesis approach of establishing unrealizability by the non-existence of strategy trees [2, 3, 4]. The difficulty in synthesis is to enforce the consistency condition that the strategy of a process must act the same way in all situations the process cannot distinguish. On the strategy trees, this consistency condition is not an ω-regular (or even decidable) property. When analyzing a counterexample, on the other hand, we only check consistency on a specific set of sequences, not on a full tree. This restricted consistency condition is an ω-regular property and can, in fact, simply be expressed in LTL as part of the temporal specification. Our QBF-based prototype implementation finds counterexamples for the Byzantine Generals' Problem and the CAP Theorem within just a few seconds.

Related Work. To the best of the authors' knowledge, there has been no attempt in the literature to characterize unrealizable specifications for distributed systems beyond the restricted class of architectures with decidable synthesis problems, such as pipelines and rings [3, 4]. By contrast, there is a rich literature concerning unrealizability for open systems, that is, single-process systems interacting with the environment [7, 8, 9]. In robotics, there have been recent attempts to analyze unrealizable specifications [10]. The results are also focused on the reason for unsatisfiability, while our approach tries to determine if a specification is unrealizable. Moreover, they only consider the simpler non-distributed synthesis of GR(1) specifications, which is a subset of LTL. There are other approaches concerning unrealizable specifications in the non-distributed setting that also use counterexamples [11, 12]. There, the system specifications are assumed to be correct and the information from the counterexamples are used to modify environment assumptions in order to make the specifications realizable. The Byzantine Generals' Problem is often used as an illustration for the knowledge-based reasoning in epistemic logics, see [13] for an early formalization. Concerning the synthesis of fault-tolerant distributed systems, there is an approach to synthesize fault-tolerant systems in the special case of strongly connected system architectures [14].

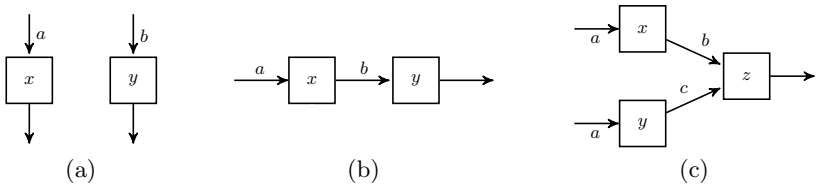


Fig. 1. Distributed architectures

2 Distributed Realizability

A specification is *realizable* if there exists an implementation that satisfies the specification. For distributed systems, the realizability problem is typically stated with respect to a specific system architecture. Figure 1 shows some typical example architectures: an architecture consisting of *independent* processes, a *pipeline* architecture, and a *join* architecture. The architecture describes the communication topology of the distributed system. For example, an edge from x to y labeled with b indicates that b is a shared variable between processes x and y, where x writes to b and y reads b. The classic *distributed realizability problem* is to decide whether there exists an implementation (or *strategy*) for each process in the architecture, such that the joint behavior satisfies the specification. In this paper, we are furthermore interested in the synthesis of fault-tolerant distributed systems, where the processes and the communication between processes may become faulty.

In order to have a uniform and precise definition for the various realizability problems of interest, we use a logical representation. Extended coordination logic (ECL) [15] is a game-based extension of linear-time temporal logic (LTL). ECL uses the *strategy quantifier* $\exists C \triangleright s$ to express the existence of an implementation for a process output s based on input variables C.

ECL Syntax. ECL formulas contain two types of variables: the set $\mathcal{C}$ of *input* (or *coordination*) variables, and the set $\mathcal{S}$ of *output* (or *strategy*) variables. In addition to the usual LTL operators Next $\bigcirc$, Until $\mathcal{U}$, and Release $\mathcal{R}$, ECL has the strategy quantifier $\exists C \triangleright s$, which introduces an output variable s whose values must be chosen based on the inputs in C. The syntax is given by the grammar

$$\varphi ::= x \mid \neg x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \varphi \,\mathcal{R}\, \varphi \mid \exists C \triangleright s.\, \varphi \mid \forall C \triangleright s.\, \varphi \ ,$$

where $x \in \mathcal{C} \dot{\cup} \mathcal{S}$, $C \subseteq \mathcal{C}$, and $s \in \mathcal{S}$. Beside the standard abbreviations true $\equiv x \vee \neg x$, false $= x \wedge \neg x$, $\Diamond \varphi \equiv \text{true}\,\mathcal{U}\,\varphi$, and $\Box \varphi \equiv \text{false}\,\mathcal{R}\,\varphi$, we use $\bigcirc_n \varphi$ as an abbreviation of n consecutive Next operators.

We denote by $\mathcal{Q}$ the (possibly empty) *quantification prefix* of a formula and call the remainder the *body*. For $Q \in \{\exists, \forall\}$, we use $\mathcal{Q}_Q$ if the prefix contains only Q-quantifiers. For the purposes of this paper, it suffices to consider the fragment $\text{ECL}_\exists$ that only contains existential quantifiers. We furthermore assume that the body is quantifier-free, i.e., that the formulas are in *prenex normal form (PNF)*.

Examples. We demonstrate how to express distributed realizability problems in $\text{ECL}_\exists$ with the example architectures from Fig. 1. The realizability of an LTL formula ψ_1 in the architecture from Fig. 1(a) is expressed by the $\text{ECL}_\exists$ formula

$$\exists\{a\} \triangleright x.\, \exists\{b\} \triangleright y.\, \psi_1 \ . \tag{1}$$

Interprocess communication via a shared variable b, as in the pipeline architecture from Fig. 1(b), is expressed by separating the information read from b from the output written to b. In the following $\text{ECL}_\exists$ formula we use output variable x to denote the output written to b:

$$\exists\{b\} \triangleright y.\, \exists\{a, b\} \triangleright x.\, \square\,(b = x) \rightarrow \psi_2 \tag{2}$$

The LTL specification ψ_2 is qualified by the input-output relation $\square\,(b = x)$, which expresses that ψ_2 is required to hold under the assumption that the information written to b by process x is also the information read from b by process y. This separation between sent and received information is useful to model faults that disturb the transmission. Failing processes can be specified by omitting the input-output relations that refer to the failing processes. As an example, consider the architecture in Fig. 1(c). The $\text{ECL}_\exists$ formula

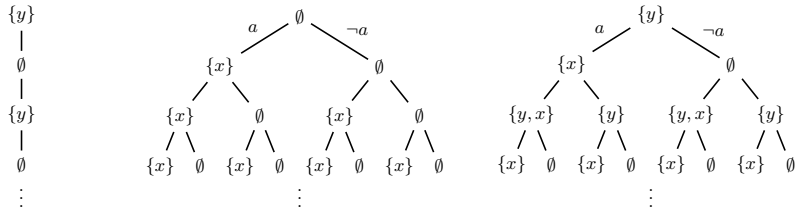
$$\exists\{a\} \triangleright x, y.\, \exists\{b, c\} \triangleright z.\, \big(\square\,(c = y) \rightarrow \psi_3\big) \wedge \big(\square\,(b = x) \rightarrow \psi_3\big) \tag{3}$$

specifies that there exists an implementation such that ψ_3 is guaranteed to hold even if process x or y (but not both) fails.

For a formula Φ, we differentiate two types of coordination variables, *external* and *internal*. A coordination variable $c \in \mathcal{C}$ is external iff it is a true input from the environment, i.e., not contained in any input-output relation of Φ. For example, the input a in (3) is external while b and c are internal.

ECL Semantics. We give a quick definition of the $\text{ECL}_\exists$ semantics for formulas in PNF and refer the reader to [15] for details and for the semantics of full ECL. The semantics is based on *trees* as a representation for strategies and computations. Given a finite set of directions Υ and a finite set of labels Σ, a (full) *Σ-labeled Υ-tree* $\mathcal{T}$ is a pair $\langle \Upsilon^*, l \rangle$, where $l : \Upsilon^* \rightarrow \Sigma$ assigns each *node* $v \in \Upsilon$ a label $l(v)$. For two trees $\mathcal{T}$ and $\mathcal{T}'$, we define the joint valuation $\mathcal{T} \oplus \mathcal{T}'$ to be the widened tree with the union of both labels. We refer to [15] for a formal definition. A path σ in a Σ-labeled Υ-tree $\mathcal{T}$ is an ω-word $\sigma_0\sigma_1\sigma_2 \ldots \in \Upsilon^\omega$ and the corresponding labeled path $\sigma^{\mathcal{T}}$ is $(l(\epsilon), \sigma_0)(l(\sigma_0), \sigma_1)(l(\sigma_0\sigma_1), \sigma_2)(l(\sigma_0\sigma_1\sigma_2), \sigma_3) \ldots \in (\Upsilon \times \Sigma)^\omega$.

For a strategy variable s that is bound by some quantifier $QC \triangleright s.\, \varphi$, we refer to C as the *scope* of s, denoted by $\text{Scope}(s)$. The meaning of a strategy variable s is a *strategy* or *implementation* $f_s : (2^{\text{Scope}(s)})^* \rightarrow 2^{\{s\}}$, i.e., a function that maps a history of valuations of input variables to a valuation of the output variable s. We represent the computation of a strategy f_s as the tree $\langle (2^{\text{Scope}(s)})^*, f_s \rangle$ where f_s serves as the labeling function (cf. Fig. 2(a)–(b)). $\text{ECL}_\exists$ formulas are interpreted over *computation trees*, that are the joint valuations of the computations for



(a) Strategy for y (b) Strategy for x (c) Computation tree

Fig. 2. In (a) and (b) we sketch example strategies for y and x satisfying the ECL$_\exists$ formula $\exists \emptyset \triangleright y. \exists \{a\} \triangleright x. \square (\bigcirc x \leftrightarrow a) \wedge \square (y \leftrightarrow \bigcirc \neg y)$. In (c) we visualize the resulting computation tree on which the body (LTL) formula is evaluated.

strategies belonging to the strategy variables in $\mathcal{S}$, i.e., $\bigoplus_{s \in S} \langle (2^{\text{Scope}(s)})^*, f_s \rangle$ (cf. Fig. 2(c)). Given an ECL$_\exists$ formula $\mathcal{Q}_\exists. \varphi$ in prenex normal form over strategy variables $\mathcal{S}$ and coordination variables $\mathcal{C}$, the formula is satisfied if there exists a computation tree $\mathcal{T}$ (over $\mathcal{S}$), such that all paths in $\mathcal{T}$ satisfy the LTL formula φ, i.e., $\forall \sigma \in (2^{\mathcal{C}})^\omega. \sigma^{\mathcal{T}}, 0 \vDash \varphi$ where the satisfaction of an LTL formula on a labeled path $\sigma^{\mathcal{T}}$ on position $i \geq 0$ is defined as usual.

3 Counterexamples to Distributed Realizability

We now introduce *counterexamples to realizability*, which correspond to *counterexamples to satisfiability* for the ECL$_\exists$ formula that represents the realizability problem. The satisfiability problem for an ECL$_\exists$ formula in prenex form asks for an implementation for all strategy variables in the quantification prefix of the formula such that the temporal specification in the body is satisfied.

Let $\Phi = \mathcal{Q}_\exists. \varphi$ be an ECL$_\exists$ formula in prenex form over coordination variables $\mathcal{C}$ and strategy variables $\mathcal{S}$, where the body of the formula is the LTL formula φ. A *counterexample to satisfiability* for Φ is a (possibly infinite) set of paths $\mathcal{P} \subseteq (2^{\mathcal{C}})^\omega$, such that, no matter what strategies are chosen for the strategy variables in $\mathcal{S}$, there exists a path $\sigma \in \mathcal{P}$ that violates the body φ. Formally, $\mathcal{P} \subseteq (2^{\mathcal{C}})^\omega$ is a counterexample to satisfiability iff, for all strategies $f_s : (2^{\text{Scope}(s)})^* \to 2^{\{s\}}$ for each $s \in \mathcal{S}$, it holds that there exists a path $\sigma \in \mathcal{P}$ such that $\sigma^{\mathcal{T}}, 0 \vDash \neg\varphi$ where $\mathcal{T} = \bigoplus_{s \in \mathcal{S}} \langle (2^{\text{Scope}(s)})^*, f_s \rangle$.

Proposition 1. *An ECL$_\exists$ formula Φ over coordination variables $\mathcal{C}$ and strategy variables $\mathcal{S}$ is unsatisfiable if and only if there exists a counterexample to satisfiability $\mathcal{P} \subseteq (2^{\mathcal{C}})^\omega$.*

Proof. By the semantics of ECL$_\exists$ and $\mathcal{P} = (2^{\mathcal{C}})^\omega$. □

In the remainder of the paper, we focus on counterexamples to realizability problems. The distributed realizability problem *without* faults correspond to ECL$_\exists$ formulas of the form $\Phi = \mathcal{Q}_\exists. \varphi_{\text{path}} \to \varphi$, where the φ_{path} defines the system architecture $\mathcal{A}_\Phi$: there is an edge from one strategy variable to another if the

input-output relation occurs in φ_{path}. A *finite* counterexample to satisfiability of Φ is a finite set of paths $\mathcal{P} \subseteq (2^{\mathcal{C}_{\text{ext}}})^\omega$ corresponding to external coordination variables, such that for any implementation $\mathcal{T}$ there exists a path $\sigma \in \mathcal{P}$ such that an extension $\sigma' \in (2^{\mathcal{C}})^\omega$ of σ violates φ. Note that the extension of σ by the valuation of the internal coordination variables is uniquely specified by the input path σ and the system implementation $\mathcal{T}$.

Corollary 2. *If there exists a finite counterexample to satisfiability $\mathcal{P} \subseteq (2^{\mathcal{C}_{ext}})^\omega$ for an ECL$_\exists$ formula $\Phi = \mathcal{Q}_\exists.\, \varphi_{path} \rightarrow \varphi$ over coordination variables $\mathcal{C}$ and strategy variables $\mathcal{S}$, then Φ is unsatisfiable.*

As an example consider again the ECL$_\exists$ formula (1) $\exists \{a\} \triangleright x. \exists \{b\} \triangleright y.\, \psi_1$, corresponding to the architecture from Fig. 1(a) in the previous section. Let $\psi_1 \coloneqq \square\,(\bigcirc y \leftrightarrow a)$, i.e., y must output the input a with an one-step delay. A simple counterexample for this formula consists of two paths $\mathcal{P}_1 \coloneqq \{\, \emptyset^\omega, \{a\}^\omega\}$ that differ in the values of a, but not in the values of b. Since process x cannot distinguish the two paths, but must produce different outputs, we arrive at a contradiction. Consider the same formula for the pipeline architecture specified by (2) $\exists \{b\} \triangleright y. \exists \{a, b\} \triangleright x. \square\,(b = x) \rightarrow \psi_2$. Due to the delay when forwarding the input a over shared variable b, the formula becomes unsatisfiable. $\mathcal{P}_1$ is a finite counterexample in this case, too: Given an implementation of x and y, we extend both paths such that the input-output specification $\square\,(b = x)$ is satisfied.

The distributed realizability problem *with* faults correspond to ECL$_\exists$ formulas of the form $\Phi = \mathcal{Q}_\exists. \bigwedge_{1 \leq i \leq n} \left(\varphi_{\text{path}_i} \rightarrow \varphi_i\right)$. If $\varphi_i = \varphi$ for all i, the formula states that there exists an implementation such that the specification φ should hold in all architectures induced by the path specifications φ_{path_i}. Omitted channel specifications in one of these formulas represent an arbitrary error at this channel. In this case, a counterexample identifies for each implementation one of these architectures where a contradiction occurs. A *finite* counterexample to satisfiability of Φ are n finite sets of paths $\mathcal{P}_i \subseteq (2^{\mathcal{C}^i_{\text{ext}}})^\omega$ each corresponding to external coordination variables $\mathcal{C}^i_{\text{ext}}$ in the respective architecture i, such that for any implementation $\mathcal{T}$ there exists an architecture j and a path $\sigma \in \mathcal{P}_j$ such that an extension $\sigma' \in (2^{\mathcal{C}})^\omega$ of σ violates φ_j.

Corollary 3. *An ECL$_\exists$ formula $\Phi = \mathcal{Q}_\exists. \bigwedge_{1 \leq i \leq n} \left(\varphi_{path_i} \rightarrow \varphi_i\right)$ over coordination variables $\mathcal{C}$ and strategy variables $\mathcal{S}$ is unsatisfiable if there exists a finite counterexample to satisfiability of Φ.*

A counterexample for the ECL specification (3) introduces paths for inputs as well as for every faulty node by introducing paths that model the exact channel specification and additional paths that model the arbitrary node failures. The target node that reads from a shared variable can, in contrast to incomplete information, react differently on the given paths, but the reaction must be consistent regarding its observations on all paths. Consider for example the specification $\psi_3 \coloneqq (\bigcirc_2 z \leftrightarrow a)$ for the ECL formula in (3), that is, process z should output the input a of nodes x and y. In both architectures we introduce additional paths for the coordination variable that is omitted in the channel specification, i.e., b and c for the first and second conjunct, respectively. Process z cannot tell

which of its inputs come from a faulty node. Since z must produce the same output on two paths it cannot distinguish, the implementation of z contradicts the specification in either architecture.

4 From ECL$_\exists$ to QPTL

We encode the existence of finite counterexample to realizability as a formula of *quantified propositional temporal logic (QPTL)*. QPTL extends LTL with a *path quantifier* $\exists p$, where a path $\sigma \in 2^{\mathrm{AP}}$ satisfies $\exists p.\, \varphi$ at position $i \geq 0$, denoted by $\sigma, i \vDash \exists p.\, \varphi$, if there exists a path $\sigma' \in 2^{\mathrm{AP} \cup \{p\}}$ which coincides with σ except for the newly introduced atomic proposition p, such that $\sigma', i \vDash \varphi$. In the encoding, we use the path quantifier to explicitly name the paths in the counterexample.

Realizability without Faults. We consider first the distributed realizability problems *without* faults, represented by ECL$_\exists$ formula $\Phi = \mathcal{Q}_\exists.\, \varphi_{\text{path}} \rightarrow \varphi$. We assume, without loss of generality, that the architecture $\mathcal{A}_\Phi$ is acyclic. Finkbeiner and Schewe [4] gave a realizability-preserving transformation to acyclic architectures that removes *feedback edges*.

Lemma 4 ([4]). *Any ECL$_\exists$ formula $\Phi = \mathcal{Q}_\exists.\, \varphi_{path} \rightarrow \varphi$ can be transformed into an equisatisfiable formula $\Phi' = \mathcal{Q}'_\exists.\, \varphi'_{path} \rightarrow \varphi'$ such that the system architecture $\mathcal{A}_{\Phi'}$ is acyclic.*

We search for a finite counterexample of Φ by bounding the number of paths regarding the *external* coordination variables. The bound on the number of paths is given as a function $K : \mathcal{C} \rightarrow \mathbb{N}$ that maps each coordination variable to the number of branchings that should be considered for this variable. For example, for coordination variables a and b, and $K(a) = K(b) = 1$, we encode 4 different paths, one per possible combination for the two paths for each variable. We fix an arbitrary strict order $\prec \subseteq \mathcal{C} \times \mathcal{C}$ between the coordination variables. For a set $C \subseteq \mathcal{C}$, we identify $K(C)$ by the vector $\mathbb{N}^{|C|}$ where the position of the value $K(c)$ for a coordination variable $c \in C$ is determined by $\prec$. For our encoding in QPTL, we use the following helper functions:

- deps(v) returns the set of coordination variables that *influence* variable v. A coordination variable c influences variable v if c belongs to a directed path that leads to v in $\mathcal{A}_\Phi$. For example in the architecture of Fig. 1(c), b and x are influenced by a while z is influenced by a, b, and c. A coordination variable is influenced by itself.
- branches(C, K) returns the set of branches belonging to coordination variables C. A branch is referenced by a tuple $\mathbb{N}^{|C|}$ and the set of branches is $\{(n_{c_1}, \ldots, n_{c_k}) \mid \{c_1 \prec \cdots \prec c_k\} = C \text{ and } 1 \leq n_c \leq 2^{K(c)} \text{ for all } c \in C\}$
- paths(C, K) and strategies(S, K) create the (path) variables in the QPTL formula that belong to the variables of the ECL$_\exists$ formula. For a variable $v \in C \cup S$ it introduces for each branch $\pi \in \text{branches}(\text{deps}(v), K)$ a separate variable p^v_π that represents the variable v belonging to this branch π.

– $\mathrm{header}(S, K)$ creates the alternating introductions of strategies and paths according to the acyclic architecture $\mathcal{A}_\Phi$. For every strategy variable $s \in S$ we introduce all paths belonging to coordination variables $c \in \mathrm{Scope}(s)$ prior to s and avoid duplicate path introductions:

$$\begin{aligned}
&\exists\, \mathrm{paths}(\mathrm{Scope}(s_1), K)\ \forall\, \mathrm{strategies}(\{s_1\}, K)\\
&\exists\, \mathrm{paths}(\mathrm{Scope}(s_2) \setminus \mathrm{Scope}(s_1), K)\ \forall\, \mathrm{strategies}(\{s_2\}, K)\\
&\dots\\
&\exists\, \mathrm{paths}(\mathrm{Scope}(s_n) \setminus \Big(\bigcup_{i=1,\dots,n-1} \mathrm{Scope}(s_i) \Big), K)\ \forall\, \mathrm{strategies}(\{s_n\}, K)\ ,
\end{aligned}$$

where $s_1, \dots, s_n$ are sorted in ascending order according to their informedness, i.e., the subset relation on their scopes.

– $\mathrm{consistent}(S, K)$ specifies the consistency condition for the variables belonging to the strategy variables on the different branches. The variables $p^s_{\pi_1}, \dots, p^s_{\pi_k}$ belonging to a strategy variable $s \in S$ must be equal as long as the coordination variables in the scope of s on the branches $\pi_1, \dots, \pi_k$ are equal. This can be specified in LTL as there are only finitely many branches.

The QPTL encoding for $\mathrm{ECL}_\exists$ formula Φ and function $K : \mathcal{C} \to \mathbb{N}$ is

$$\begin{aligned}
&\mathrm{unsat}_{\mathrm{dist}}(\Phi, K) \coloneqq \mathrm{header}(\mathcal{S}, K).\, \mathrm{consistent}(\mathcal{S}, K) \rightarrow\\
&\Big(\bigwedge_{\pi \in \mathrm{branches}(\mathcal{C}, K)} \varphi_{\mathrm{path}}(\pi) \Big) \wedge \Big(\bigvee_{\pi \in \mathrm{branches}(\mathcal{C}, K)} \neg\varphi(\pi) \Big)\ ,
\end{aligned} \tag{4}$$

where $\varphi(\pi)$ is the initialization of LTL formula φ on the branch π, that is we exchange v by $p^v_{\pi'}$ for $v \in \mathcal{C} \cup \mathcal{S}$ where π' is the subvector of π that contains the values for coordination variables in $\mathrm{deps}(v)$.

Theorem 5 (Correctness). *Given an $\mathrm{ECL}_\exists$ formula $\Phi = \mathcal{Q}_\exists.\, \varphi_{path} \rightarrow \varphi$ over coordination variables $\mathcal{C}$ and strategy variables $\mathcal{S}$ with an acyclic system architecture $\mathcal{A}_\Phi$. Φ is unsatisfiable if there exists a function $K : \mathcal{C} \to \mathbb{N}$ such that the QPTL formula $unsat_{dist}(\Phi, K)$ is satisfiable.*

Realizability with Node Failures. In the case of possible failures, the $\mathrm{ECL}_\exists$ formulas Φ has the more general form $\mathcal{Q}_\exists.\, \bigwedge_{1 \le i \le n} \big(\varphi_{\mathrm{path}_i} \rightarrow \varphi_i\big)$. In this specific setting we cannot assume acyclic architectures in general. The architecture belonging to Φ is acyclic if the architecture belonging to the conjunction of all paths specifications $\bigwedge_{1 \le i \le n} \varphi_{\mathrm{path}_i}$ is acyclic. An edge is a *common feedback edge* if and only if it is a feedback edge in all architectures. As before, we can eliminate common feedback edges but this does not give us acyclic architectures in general as depicted in Fig. 3. In the following, we assume acyclic architectures after removing common feedback edges.

The QPTL encoding of $\mathrm{ECL}_\exists$ formula Φ and functions $K_1 \dots K_n : \mathcal{C} \to \mathbb{N}$ is

$$\begin{aligned}
&\mathrm{unsat}_{\mathrm{fault}}(\Phi, K_1, \dots, K_n) \coloneqq \mathrm{header}(\mathcal{S}, K).\, \mathrm{consistent}(\mathcal{S}, K) \rightarrow\\
&\bigvee_{1 \le i \le n} \Big(\bigwedge_{\pi \in \mathrm{branches}(\mathcal{C}, K_i)} \varphi_{\mathrm{path}_i}(\pi) \Big) \wedge \Big(\bigvee_{\pi \in \mathrm{branches}(\mathcal{C}, K_i)} \neg\varphi_i(\pi) \Big)\ ,
\end{aligned} \tag{5}$$

where $K : \mathcal{C} \to \mathbb{N}$ is defined as $K(c) \coloneqq \max_{1 \le i \le n} K_i(c)$ for every $c \in \mathcal{C}$.

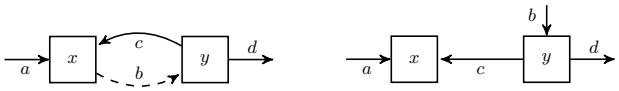


Fig. 3. Example illustrating common feedback edges: Edge c is a feedback edge in architecture (a), but not in architecture (b), thus it is also not a common feedback edge when considering both architectures

Theorem 6 (Correctness). *Given an ECL$_\exists$ formula $\Phi = \mathcal{Q}_\exists.\, \bigwedge_{1 \le i \le n} \left(\varphi_{path_i} \rightarrow \varphi_i\right)$ over coordination variables $\mathcal{C}$ and strategy variables $\mathcal{S}$ with an acyclic system architecture $\mathcal{A}_\Phi$ after removing common feedback edges. Φ is unsatisfiable if there exist functions $K_1 \dots K_n : \mathcal{C} \rightarrow \mathbb{N}$ such that the QPTL formula $unsat_{fault}(\Phi, K_1, \dots, K_n)$ is satisfiable.*

Example. We consider again the Byzantine Generals' Problem with three nodes g_1, g_2, and g_3. The first general is the commander who forwards the input v that states whether to attack the enemy or not. The encoding as ECL$_\exists$ formula is

$$\Phi_{\text{bgp}} \coloneqq \exists\{v\} \triangleright g_{12}, g_{13}.\, \exists\{c_{12}\} \triangleright g_{23}.\, \exists\{c_{13}\} \triangleright g_{32}.\, \exists\{c_{12}, c_{32}\} \triangleright g_2.\, \exists\{c_{13}, c_{23}\} \triangleright g_3.$$
$$(\text{operational}_{2,3} \rightarrow \text{consensus}_{2,3}) \wedge \bigwedge_{i \in \{2,3\}} (\text{operational}_{1,i} \rightarrow \text{correctval}_i)\ ,$$

where the quantification prefix introduces the strategies for the generals g_2 and g_3, as well as the communication between the three generals as depicted in the architecture in Fig. 4(a). Note that we omit the vote of the commander g_1 as it is not used in the specification. In the temporal part, we specify which failures can occur. The first conjunct, corresponding to Fig. 4(b), states that the commander is faulty ($\text{operational}_{2,3}$) which implies that the other two generals have to reach a consensus whether to attack or not ($\text{consensus}_{2,3}$). The other two cases, depicted in Fig. 4(c)–(d), are symmetric and state that whenever one general is faulty the other one should agree on the decision made by the commander. The QPTL encoding $\text{unsat}_{\text{fault}}(\Phi_{\text{bgp}}, K_1, K_2, K_3)$ is given as

$$\exists\, \text{paths}(\{v\}, K).\ \forall\, \text{strategies}(\{g_{12}, g_{13}\}, K).\ \exists\, \text{paths}(\{c_{12}, c_{13}\}, K).$$
$$\forall\, \text{strategies}(\{g_{23}, g_{32}\}, K).\ \exists\, \text{paths}(\{c_{23}, c_{32}\}, K).\ \forall\, \text{strategies}(\{g_2, g_3\}, K).$$
$$\text{consistent}(\{g_{12}, g_{13}, g_{23}, g_{32}, g_2, g_3\}, K) \rightarrow$$
$$\Big(\Big(\bigwedge_{\pi \in \text{branches}(\mathcal{C}, K_1)} \text{operational}_{2,3}(\pi) \wedge \bigvee_{\pi \in \text{branches}(\mathcal{C}, K_1)} \neg\text{consensus}_{2,3}(\pi) \Big) \vee$$
$$\Big(\bigwedge_{\pi \in \text{branches}(\mathcal{C}, K_2)} \text{operational}_{1,3}(\pi) \wedge \bigvee_{\pi \in \text{branches}(\mathcal{C}, K_2)} \neg\text{correctval}_3(\pi) \Big) \vee$$
$$\Big(\bigwedge_{\pi \in \text{branches}(\mathcal{C}, K_3)} \text{operational}_{1,2}(\pi) \wedge \bigvee_{\pi \in \text{branches}(\mathcal{C}, K_3)} \neg\text{correctval}_2(\pi) \Big) \Big)\ .$$

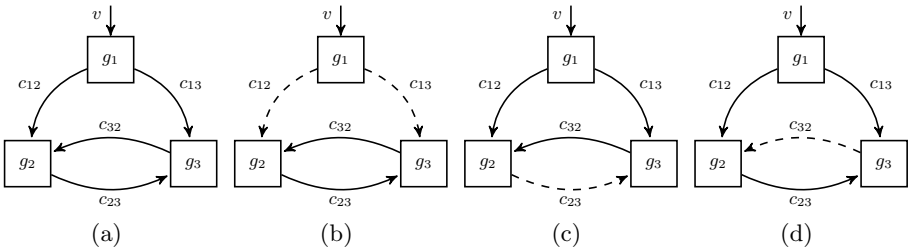


Fig. 4. The Byzantine Generals' architecture. Figure (a) shows the architecture in cases all generals are loyal. Figures (b)–(d) show the possible failures, indicated by the dashed communication links.

5 From QPTL to QBF

Presently available QPTL solver were unable to handle even small instances of our problem. We therefore simplify the problem using the following steps. Instead of checking the QPTL formula directly, we encode the formula as an equivalent *monadic second order logic of one successor (S1S)* formula using a straightforward translation. We then interpret the S1S formula with a WS1S formula, which can be checked using the WS1S solver Mona [16]. Some of our smaller instances were solved by Mona, but the Byzantine Generals' Problem failed due to memory constraints in the BDD library.

Taking the simplifications even further, we not only bound the *number* of paths but also the *length* of the paths by translating the problem to the satisfiability problem of *quantified Boolean formulas (QBF)*. The encoding translates a QPTL variable x to Boolean variables $x_0, \ldots, x_{k-1}$, each representing one step in the system where k is the length of the paths. We build the QBF formula by *unrolling* the QPTL formula for k-steps: Each variable in the quantification prefix of the QPTL formula is transformed into k Boolean variables in the QBF prefix, e.g., the 3-unrolling of $\exists x. \forall y. \varphi$ is $\exists x_0, x_1, x_2. \forall y_0, y_1, y_2. \varphi_{unroll}$. The unrolling of the remaining LTL formula is given by the expansion law for Until, $\varphi \,\mathcal{U}\, \psi \equiv \psi \vee (\varphi \wedge \bigcirc \varphi \,\mathcal{U}\, \psi)$. After the unrolling, the QBF formula is transformed into Conjunctive Normal Form (CNF) and encoded in the QDIMACS file format, that is the standard format for QBF solvers. Already with this encoding we could solve more examples than using the WS1S approach.

In this simple translation, one cause of high complexity is due to the consistency conditions between the strategy variables across different paths. However, most of these variables are not used for the counterexample itself but appear only in the consistency condition. One optimization removes these unnecessary variables from the encoding. Therefore, we collect all strategy variables and (when possible) their temporal occurrence from the LTL specification. For every used strategy variable we build the *dependency graph* that contains all variables which can influence the outcome of the strategy. In the last step, we remove all variables that are not contained in any dependency graph.

6 Completeness

Proposition 1 states that the characterization of unsatisfiable formulas with counterexamples is complete. Our method, however, searches for counterexamples involving only a bounded number of external paths and the following example shows this leads to incompleteness. Consider the $\text{ECL}_\exists$ formula $\Phi_{\text{inf}} := \exists \emptyset \triangleright y.\, \varphi_{\text{inf}}$ with temporal specification $\varphi_{\text{inf}} := \Diamond (y \neq x)$ where x is a free coordination variable. Φ_{inf} is unsatisfiable because for every strategy $f_y : \emptyset^* \to 2^{\{y\}}$ there exists a path $\sigma \in (2^{\{x\}})^\omega$ that simulates exactly the output of the strategy, as the formula is evaluated over the full binary x-tree. Assume for contradiction that a finite set of paths $P \subseteq (2^{\{x\}})^\omega$ suffices to satisfy $\neg\varphi_{\text{inf}}$ against any strategy f_y. Interpreting the outcome of the strategy as a path and considering all possible strategies gives us a full binary tree $\mathcal{T}$. Let ρ be a path from $\mathcal{T}$ that is not contained in P (after renaming y in ρ to x). Such a path must exists because there are infinite many different paths in $\mathcal{T}$. Choose the strategy f_y^ρ that belongs to ρ. For all paths in P it holds that $\Diamond (y \neq x)$ and thus no path satisfies $\neg\varphi_{\text{inf}}$.

However, in practice finite external counterexamples are sufficient to detect many errors in specifications. In this section we give a semantic characterization of the finite path satisfiability based only on the LTL specification.

Given an $\text{ECL}_\exists$ formula $\Phi = \mathcal{Q}_\exists.\, \varphi_{\text{path}} \to \varphi$. We assume w.l.o.g. that φ only contains coordination variables $\mathcal{C}_e \subseteq \mathcal{C}$ that are not used as a channel as otherwise one could replace a variable $c \in \mathcal{C} \setminus \mathcal{C}_e$ by the strategy variable corresponding to the channel. The semantics of the LTL formula $\neg\varphi$, denoted by $[\![\neg\varphi]\!]$, gives us a language $\mathcal{L} \subseteq (2^{\mathcal{S}} \times 2^{\mathcal{C}_e})^\omega$. From $\mathcal{L}$ we obtain the relation $\mathcal{R} \subseteq (2^{\mathcal{S}})^\omega \times (2^{\mathcal{C}_e})^\omega$ between paths of strategy variables and paths of coordination variables. We say that an LTL formula ψ over variables $\mathcal{S} \times \mathcal{C}_e$ admits finite external paths if there exists a function $r : (2^{\mathcal{S}})^\omega \to (2^{\mathcal{C}_e})^\omega$ such that (1) for all $\sigma \in (2^{\mathcal{S}})^\omega$ it holds that $r(\sigma) = \rho \Leftrightarrow \sigma\, \mathcal{R}\, \rho$, and (2) $\{r(\sigma) \mid \sigma \in (2^{\mathcal{S}})^\omega\}$ is finite.

Let $\mathcal{RA}_\psi$ be the deterministic Rabin word automata for LTL formula ψ. $\mathcal{RA}_\psi$ contains a *path split* if there exist a state q in the automaton where (1) there are two outgoing edges labeled with (s, p) and (s', p') where $s \neq s'$ and $p \neq p'$, and (2) from q we can build accepting runs visiting q infinitely often and containing exclusively the (s, p)-edge or (s', p')-edge.

Theorem 7. *An LTL formula ψ over variables $\mathcal{S} \times \mathcal{C}_e$ admits finite external paths if and only if the automaton $\mathcal{RA}_\psi$ has no path split.*

7 Experimental Results

We have carried out our experiments on a 2.6 GHz Opteron system. For solving the QBF instances, we used a combination of the QBF preprocessor Bloqqer [17] in version 031 and the QBF solver DepQBF [18] in version 1.0. For solving the WS1S instances, we used Mona [16] in version 1.4-15.

Table 1 demonstrates that the Byzantine Generals' Problem remains, despite the optimizations described above, a nontrivial combinatorial problem: we need

to find a suitable set of paths for every possible combination of the strategies of the generals. The bound given in the first column reads as follows: The first component is the number of branchings for the input variable v in all three architectures. The last three components state the number of branchings for the outputs of the faulty nodes in their respective architectures. For example, bound $(1, 1, 0, 0)$ means that we have two branches for v, c_{12}, and c_{13}, while we have only one branch for c_{23} and c_{32}. More precisely, starting from always zero functions K_1, K_2, K_3, the bound $(1, 1, 0, 0)$ sets $K_1(v) = K_2(v) = K_3(v) = K_1(c_{12}) = K_1(c_{13}) = 1$ and $K_2(c_{23}) = K_3(c_{32}) = 0$. To prove the unrealizability, we need one branching for the input v and one branching for every coordination variable that serves as a shared variable for a faulty node, i.e., the bound $(1, 1, 1, 1)$. The number of branches and thereby the formula size grows exponentially with the number of branchings for the input variables.

Table 1. Result of the *Byzantine Generals' Problem* example

Bound	Result	# Clauses	# Variables	Memory (MB)	Time (s)
$(0, 0, 0, 0)$	Unsatisfiable	57	44	5.06	0.00
$(1, 0, 0, 0)$	Unsatisfiable	228	143	5.71	0.05
$(1, 1, 0, 0)$	Unsatisfiable	2286	1095	17.83	2.16
$(1, 1, 1, 0)$	Unsatisfiable	2904	1375	18.41	2.42
$(1, 1, 1, 1)$	Satisfiable	3522	1655	28.88	11.95

The table shows the time and memory consumption of Bloqqer 031 and DepQBF 1.0 when solving the encoding of the Byzantine Generals' Problem in QBF with a fixed length of 3 unrollings.

The CAP Theorem for two nodes is encoded as the $\mathrm{ECL}_\exists$ formula

$$\exists\{\mathrm{req}_1\} \triangleright \mathrm{com}_1.\, \exists\{\mathrm{req}_1, \mathrm{chan}_2\} \triangleright \mathrm{out}_1.\, \exists\{\mathrm{req}_2\} \triangleright \mathrm{com}_2.\, \exists\{\mathrm{req}_2, \mathrm{chan}_1\} \triangleright \mathrm{out}_2.$$
$$(\square\,(\mathrm{chan}_1 = \mathrm{com}_1) \rightarrow \square\,((\mathrm{out}_1 = \mathrm{out}_2) \wedge ((\mathrm{req}_1 \vee \mathrm{req}_2) \leftrightarrow \bigcirc_2\,(\mathrm{out}_1 \vee \mathrm{out}_2)))) \wedge$$
$$(\square\,(\mathrm{chan}_2 = \mathrm{com}_2) \rightarrow \square\,((\mathrm{out}_1 = \mathrm{out}_2) \wedge ((\mathrm{req}_1 \vee \mathrm{req}_2) \leftrightarrow \bigcirc_2\,(\mathrm{out}_1 \vee \mathrm{out}_2)))) \ .$$

The architecture is similar to Fig. 1(a) with the difference that there is a direct communication channel between the two processes $(\mathrm{chan}_1, \mathrm{chan}_2)$. The formula states that the system should be available and consistent despite an failure of one process. Table 2 shows that our method is able to find conflicts in a specification with an architecture up to 50 nodes within reasonable time. When we drop either Consistency, Availability, or Partition tolerance, the corresponding instances (AP, CP, and CA) become satisfiable. Hence, our tool does not find counterexamples in these cases.

Discussion. We evaluate the different encodings that we have used in the following. There does not exist an algorithm that decides whether a given $\mathrm{ECL}_\exists$ formula is unsatisfiable. We used a sound approach where we bound the number of paths and encoded the problem in QPTL. The reason for incompleteness was

Table 2. Result of the *CAP Theorem* example

Instance	Result	# Clauses	# Variables	Memory (MB)	Time (s)
ap_2	Unsatisfiable	1232	619	9.22	0.29
ca_2	Unsatisfiable	1408	763	12.47	0.87
cp_2	Unsatisfiable	48	42	5.05	0.00
cap_2	Satisfiable	110	84	5.05	0.00
cap_5	Satisfiable	665	426	5.06	0.05
cap_10	Satisfiable	2590	1556	6.49	0.35
cap_25	Satisfiable	15865	9146	35.47	2.83
cap_50	Satisfiable	62990	35796	87.84	44.03

The table shows the time and memory consumption of Bloqqer 031 and DepQBF 1.0 when solving the encoding of the CAP Theorem in QBF with a fixed length of 2 unrollings.

shown in Sec. 6; in some cases one may need infinite many paths to show unsatisfiability. Our encoding in WS1S (Mona) loses the ability to find counterexample paths of infinite length, e.g., the $\mathrm{ECL}_\exists$ formula $\exists \emptyset \triangleright y. \Diamond\Box(\bigcirc y \leftrightarrow x)$ with free coordination variable x is unsatisfiable where two paths that are infinitely often different are sufficient to prove it. The QPTL encoding is capable of finding these paths while the WS1S encoding is not. However, Mona could not solve any satisfiable instance given in Tables 1 and 2. Lastly, for the translation in QBF we do not only restrict ourself to paths of finite length (WS1S), but we also bound the paths to length k where k is an additional parameter. With this encoding we approximate the reactive behavior of our system by a finite prefix. It turned out that despite of this restriction we could prove unsatisfiability for many interesting specifications. In practice, one would first use the QBF abstraction in order to find "cheap" counterexamples. After hitting the number of paths that the QBF solver can no longer handle within reasonable time, one proceeds with more costly abstractions like the WS1S encoding.

8 Conclusion

We introduced counterexamples for distributed realizability and showed how to automatically derive counterexamples from given specifications in $\mathrm{ECL}_\exists$. We used encodings in QPTL, WS1S, and QBF. Our experiments showed that the QBF encoding was the most efficient. Even problems with high combinatorial complexity, such as the Byzantine Generals' Problem, are handled automatically. Given that QBF solvers are likely to improve in the future, even larger instances should become tractable. Possible future directions include building a set of benchmarks, evaluating more solvers, and use the information about an counterexample given by QBF certification [19] to build counterexamples for the specification. As the bound given for the encoding is not uniform, i.e., there is a bound for each coordination variable, and the observation that the performance depend on the chosen bound, it is crucial to find suitable heuristics that rank the importance of the coordination variables. Also, more types of failures could be

incorporated into our model, e.g., variations of the failure duration like *transient*, or *intermittent*. Lastly, it would be also conceivable to use similar methods to derive a larger class of *infinite* counterexamples.

References

1. Lamport, L., Shostak, R.E., Pease, M.C.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)
2. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proc. FOCS 1990, pp. 746–757 (1990)
3. Kupferman, O., Vardi, M.Y.: Synthesizing distributed systems. In: LICS, pp. 389–398. IEEE Computer Society (2001)
4. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: LICS, pp. 321–330. IEEE Computer Society (2005)
5. Finkbeiner, B., Schewe, S.: Bounded synthesis. International Journal on Software Tools for Technology Transfer 15(5-6), 519–539 (2013)
6. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC, p. 7. ACM (2000)
7. Church, A.: Logic, arithmetic and automata. In: Proc. 1962 Intl. Congr. Math., Upsala, pp. 23–25 (1963)
8. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
9. Kupferman, O., Vardi, M.Y.: Synthesis with incomplete information. In: Proc. of ICTL (1997)
10. Raman, V., Kress-Gazit, H.: Analyzing unsynthesizable specifications for high-level robot behavior using LTLMoP. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 663–668. Springer, Heidelberg (2011)
11. Li, W., Dworkin, L., Seshia, S.A.: Mining assumptions for synthesis. In: MEMOCODE, pp. 43–50. IEEE (2011)
12. Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Environment assumptions for synthesis. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 147–161. Springer, Heidelberg (2008)
13. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. In: PODC, pp. 50–61. ACM (1984)
14. Dimitrova, R., Finkbeiner, B.: Synthesis of fault-tolerant distributed systems. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 321–336. Springer, Heidelberg (2009)
15. Finkbeiner, B., Schewe, S.: Coordination logic. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 305–319. Springer, Heidelberg (2010)
16. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)
17. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 101–115. Springer, Heidelberg (2011)
18. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. JSAT 7(2-3), 71–76 (2010)
19. Balabanov, V., Jiang, J.H.R.: Unified QBF certification and its applications. Formal Methods in System Design 41(1), 45–65 (2012)