Mode-based Reactive Synthesis

Matías Brizzio[1,2], Felipe Gorostiaga[1], César Sánchez[1], Renzo Degiovanni[3]

[1] IMDEA Software Institute, Spain
[2] Universidad Politécnica de Madrid, Spain
[3] Luxembourg Institute of Science and Technology (LIST)
{first.last}@{imdea.org | list.lu}

Abstract. Reactive synthesis aims to automatically generate systems from high-level formal specifications, but its inherent complexity limits its scalability to real-world scenarios. This limitation can be addressed by decomposing the specification into independent parts for parallel synthesis, but the dependency between variables limits this approach.
At the same time, specifications used in Requirements Engineering (RE) often include high-level state machine descriptions, known as modes, which structure the specification.
This paper introduces a novel method for the sequential decomposition of reactive synthesis problems based on modes. Our approach automatically uses modes to break down a specification into smaller sub-specifications, synthesizes each independently, and then integrates the solutions into a cohesive global model. We present an algorithm that exploits mode transitions and ensures consistency across synthesized components leveraging off-the-self reactive synthesis tools.
We prove the correctness of our approach and show empirically that our method significantly improves scalability when decomposing real-world specifications, outperforming state-of-the-art monolithic tools. As the first sequential decomposition approach, our method offers a promising alternative for scalable reactive synthesis.

1 Introduction

Reactive systems [47], which continuously interact with their environment, are essential in domains like cyber-physical and embedded systems. These systems are crucial for tasks such as model checking [19], property monitoring [8], and model-based testing [30]. Linear-Time Temporal Logic (LTL) [56] is commonly used to specify properties of reactive systems, typically in an *assume-guarantee* format ($A \to G$). Here, Assumptions (A) describe the uncontrollable environment and Guarantees (G) define the desired system behavior. This separation into *environment-controlled* and *system-controlled* variables facilitates effective analysis and synthesis [11].

Reactive synthesis automates the construction of a *controller* from a specification, ensuring that for all valid environment inputs, the controller behaves as required. Despite progress in the field, synthesizing controllers for complex specifications remains computationally challenging. Even with efficient LTL fragments like GR(1) [55,10], deciding realizability and generating a controller can lead to exponential blow-ups [38]. Several decomposition techniques have been proposed [27,43,7,20,52] which try to perform synthesis independently for different variables, but these methods face difficulties when sub-specifications share *controllable* variables. Requirements Engineering (RE) methodologies provide useful mechanisms to modularize and specify the system behavior. A typical way to organize system requirements is through the use of high-level state machine descriptions in which the states, referred to as *modes*, encapsulate specific system behavior under particular situations and transitions represent how the system execution evolves and react to environmental events [37].

Mode-based synthesis leverages these modes to synthesize complex systems by decomposing global specifications into sub-specifications for individual modes. However, the existing approach [13] requires manual intervention, limiting their automation. In this work, we build upon these ideas to fully automate the mode-based synthesis process, eliminating the need for manual engineering input. Our approach takes as input the system's global specification, a description of its modes, and optionally, the transitions between them. Through a process we call *sequential decomposition*, the synthesis problem is addressed incrementally: each sub-problem is solved independently, focusing on one mode at a time. A key aspect of our method is the treatment of *initial conditions* for each mode. These conditions ensure coherence between sub-specifications and the global one, facilitating seamless connections across modes. By maintaining alignment with global requirements, our method guarantees consistency across mode transitions, minimizing potential conflicts and ensuring the system's correctness. We formalize *mode projection* and *mode-based synthesis*, proving consistency between modes and the global system. Crucially, our approach computes initial conditions ensuring that if each sub-specification is *realizable*, then the global specification is also *realizable*. The resulting structured controllers enhance transparency and interpretability while improving synthesis scalability, as shown empirically against state-of-the-art monolithic tools. The paper is organized as follows: Section 2 covers preliminaries, Section 3 details our approach, Section 4 presents empirical results, and Section 5 concludes.

Related Work. Reactive synthesis [57,10] aims to automatically generate correct-by-construction controllers from temporal logic specifications. LTL synthesis is 2EXPTIME-complete [57], tractable fragments like GR(1) enable polynomial-time synthesis [10]. However, challenges remain, particularly in constructing deterministic automata for large Safety-LTL formulas [70]. Compositional approaches improve scalability by decomposing synthesis tasks [60,57]. Dureja and Rozier [25] reduce model-checking tasks via dependency analysis, while Finkbeiner *et al.* [29] extend this idea to synthesis by focusing on *controllable* variables. However, these *simultaneous decomposition* methods, which address the synthesis problem in parallel, struggle when requirements share many controlled variables, limiting their applicability [39,29,51]. In RE, high-level state machines, or *modes*, are commonly used to structure system specifications [33,64,65,59,4]. Languages like EARS [50], SCR [36,35], TLSF [40], SPIDER [41], NASA's FRET [31], and Spectra [48] leverage modes for organizing computation and requirements. This aligns with IEEE standard 29148, which notes that *"some systems behave quite differently depending on the mode of operation. For example, a control system may have different features depending on its mode: training, normal, or emergency."* [1]. State-based techniques like Statecharts [34], Broy's hierarchical service modeling [15], and the SCR method further formalize mode-based specifications. Feature modeling [24,61] and safety analysis methods like FTA and FMEA [44,2] also utilize modes. However, translating these mode-based specifications into LTL can increase complexity, hindering *simultaneous methods* methods. For example, NASA's FRET, which heavily uses state variables (modes), poses challenges for decomposition tools, leading to issues like *false positives* [51] and *goal conflicts* [21,16,12,22,49,11]. This motivates the need for more efficient synthesis approaches in real-world applications. Recent RE research explores using Large Language Models (LLMs) for requirements specification [67,5,54,69,68,66]. LLMs, like GPT series [14,53], LaMDA [62], LLaMa [63], PaLM [18], and BERT [23], can assist in articulating mode-based requirements, potentially simplifying translation to LTL [45,3]. Related to us, Balkan et al. [6] use modes in a GR(1) subfragment for control design of *continuous* systems, focusing on quantitative performance. This contrasts with our focus on *discrete* systems and logical correctness in reactive synthesis. More directly, Brizzio et al. [13] proposed a mode-based decomposition for a fragment of safety, but require manual specification of initial conditions, which is tedious, error-prone, and deviates from standard RE practices. In contrast, our novel mode-based synthesis *automatically* generates initial conditions, ensuring consistency and eliminating manual intervention. Unlike simultaneous decomposition, we employ a sequential approach. By addressing one sub-problem at a time and leveraging natural mode transitions, our *sequential* method simplifies synthesis, reduces potential conflicts, and inherently ensures consistency. To the best of our knowledge, this is the first fully automated sequential mode-based synthesis method.

2 Preliminaries

LTL is a logical formalism widely used to specify reactive systems [56,46]. Given a set of propositional variables AP, LTL formulas are defined using standard logical connectives and the temporal operators $\bigcirc$ (next) and $\mathcal{U}$ (until) as follows:

$$\varphi ::= \mathtt{true} \mid a \in \mathsf{AP} \mid \varphi \vee \varphi \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

Other common operators, such as $\mathtt{false}$, $\wedge$ (and), $\square$ (always), and $\rightarrow$ (implies), can be derived: $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, $\square\phi \equiv \neg(\mathtt{true}\,\mathcal{U}\,\neg\phi)$, and $\phi \rightarrow \psi \equiv \neg\phi \vee \psi$. Given an LTL formula φ, $\mathit{Vars}(\varphi) \subseteq \mathsf{AP}$ denotes the set of atomic propositions used in φ. The semantics of LTL associate traces $\sigma \in \Sigma^\omega$ with formulae as follows (we omit the Boolean operators which are standard):

$$\begin{array}{lll} \sigma \models a & \text{iff} & a \in \sigma(0) \\ \sigma \models \bigcirc\varphi & \text{iff} & \sigma^1 \models \varphi \\ \sigma \models \varphi_1 \,\mathcal{U}\, \varphi_2 & \text{iff} & \text{for some } i \geq 0 \;\; \sigma^i \models \varphi_2, \text{ and for all } 0 \leq j < i, \sigma^j \models \varphi_1 \end{array}$$

where $\sigma(i)$ is the i-th letter of σ, and $\sigma^i \in \Sigma^\omega$ is its suffix starting at the i-th position. Given $L \subseteq \Sigma^\omega$ and a formula φ, we use $L \models \varphi$ if for all $\sigma \in L$, $\sigma \models \varphi$.

A Syntactic Fragment of Safety-LTL. We focus on a fragment of LTL (Safety-LTL) [70] commonly used in requirement engineering. LTL_G consists of formulas of the form $\psi \land \Box\varphi$, where ψ is propositional and $\varphi \in \text{LTL}_X$ (which uses only the $\bigcirc$ operator). LTL_G is widely used in industrial safety specifications [17,28,32]. Specifically, we work with GX_0 [13], a sub-fragment of Safety-LTL defined as $\alpha \rightarrow (\beta \land \Box\psi)$, where α, β, and ψ are conjunctions in LTL_X. This fragment extends LTL_G, still expresses safety properties, and is supported by tools like Strix [52]. A *reactive specification* $\varphi = (A, G)$ consists of $A = (\theta_e, \varphi_e)$ and $G = (\theta_s, \varphi_s)$, where $\theta_{\{e,s\}}$ represent initial conditions for the environment and system, respectively, and $\varphi_{\{e,s\}}$ are the assumptions and guarantees. Like previous works [29,51] that restrict their methods to LTL fragments for efficiency, we simplify our approach by using propositional formulas over $\mathit{Vars}(\varphi)$ for A to ensure a consistent environment during decomposition, avoiding interference that leads to *false-negatives* [51], and employ $G \in \text{LTL}_X$ for guarantees. Thus, the intended meaning of φ is the GX_0 formula: $(\theta_e \rightarrow (\theta_s \land \Box(\varphi_e \rightarrow \varphi_s)))$

Reactive Synthesis. Reactive LTL synthesis [57] is the problem of automatically constructing a system that reacts to the environment guaranteeing an LTL specification φ. The propositions $\mathit{Vars}(\varphi)$ are partitioned into $\mathcal{X} \cup \mathcal{Y}$, where $\mathcal{X}$ are *environment-controlled* variables and $\mathcal{Y}$ are *system-controlled* variables. A system strategy for φ is a function $\rho : (2^{\mathcal{X}})^+ \rightarrow 2^{\mathcal{Y}}$ mapping finite sequences of $\mathcal{X}$ valuations to $\mathcal{Y}$ valuations. Given an infinite sequence $X = X_1, X_2, \ldots \in (2^{\mathcal{X}})^\omega$, the play induced by strategy ρ is the infinite sequence $\sigma_{\rho,X} = (X_1 \cup \rho(X_1))(X_2 \cup \rho(X_1, X_2)) \ldots$ We use $\mathcal{L}(\rho) = \{\sigma_{\rho,X} \mid X \in (2^{\mathcal{X}})^\omega\}$ for the set of plays played according to ρ. A play σ is winning if $\sigma \models \varphi$. A strategy is winning if $\mathcal{L}(\rho) \models \varphi$. Realizability is the problem of deciding if a specification has a winning strategy, and synthesis is the problem of computing one.

3 Mode-Based Synthesis for GX_0

We now describe our mode-based reactive synthesis method for GX_0 specifications, beginning with some preliminary definitions. A mode m for a GX_0 formula φ is a predicate over $\mathit{Vars}(\varphi)$ describing a set of system states. A set of modes $M = \{m_0, \ldots, m_k\}$ is *valid* for φ if *(1)* modes are mutually exclusive ($m_i, m_j \in M$ with $i \neq j$, $m_i \land m_j$ is unsatisfiable) and *(2)* they cover all possible states ($\bigvee_{i=0}^{k} m_i$ is valid). A *mode-transition* from m to a *different* mode n in a trace σ occurs at index i if $\sigma(i) \vDash m$ and $\sigma(i+1) \vDash n$. A mode transition from m to n occurs under a strategy ρ if it occurs in some trace $\sigma \in \mathcal{L}(\rho)$.

Definition 1 (Mode-Graph). *A* mode-graph *is a directed graph* $\mathcal{G} = (M, \prec)$, *where* $M = \{m_0, m_1, \ldots, m_k\}$ *is a set of modes, and* $\prec \subseteq M \times M$ *is irreflexive.*

The intended meaning of $\prec$ is to restrict the search of strategies to those where mode-transitions are included in $\prec$. In a *complete mode graph*, $\prec = \{(m,n) \mid m,n \in M, m \neq n\}$ includes all possible strategies. In RE, it is common to specify modes and mode-transitions as part of the requirements, as discussed in Section 1 and 2.

3.1 Basic Mode-Based Synthesis with Initial Conditions

Given a mode $m \in M$ and a set of guarantees φ_s in a GX_0 specification φ, the following function reduce projects a new set of guarantees specific to mode m:

$$\mathsf{reduce}(\psi, m) = \begin{cases} \texttt{true}, & \text{if } (m \wedge \neg\psi) \vDash \texttt{false} \ (1) \\ \texttt{false}, & \text{if } (m \wedge \psi) \vDash \texttt{false} \ (2) \\ \mathsf{Simpl}(\mathsf{reduce}(\psi_1, m) \bullet \mathsf{reduce}(\psi_2, m)), & \text{if } \neg(1 \vee 2) \wedge \psi = \psi_1 \bullet \psi_2 \\ & \quad \text{with } \bullet \in \{\wedge, \vee, \rightarrow\} \\ \mathsf{Simpl}(\neg\mathsf{reduce}(\psi', m)), & \text{if } \psi = \neg\psi' \\ \Box\mathsf{Simpl}(\mathsf{reduce}(\psi', m)), & \text{if } \psi = \Box\psi' \\ \bigcirc\mathsf{Simpl}(\mathsf{reduce}(\psi', m)), & \text{if } \psi = \bigcirc\psi' \\ \psi, & \text{otherwise} \end{cases}$$

Given φ_s, reduce (φ_s, m) projects [1] the set of guarantees φ_s on mode m. We use $\mathsf{Simpl}(\psi)$ for a function that applies standard Boolean simplifications, like $(x \wedge \texttt{true}) \mapsto x$, etc. The reduced specification for mode m is denoted φ_m.

Example 1. Consider the following GX_0 specification φ and modes m_1 and m_2:

$$\boldsymbol{G_1} : \Box(e_1 \rightarrow (m_1 \rightarrow \bigcirc s_1)), \quad \boldsymbol{G_2} : \Box(m_2 \rightarrow \bigcirc(\neg s_3 \vee m_1)), \quad \boldsymbol{G_3} : \Box(\neg m_2 \rightarrow s_3)$$

Applying reduce (φ, m_1) results in:

$$\boldsymbol{G_1} : \Box(e_1 \rightarrow \bigcirc s_1), \quad \boldsymbol{G_2} : \Box\texttt{true}, \quad \boldsymbol{G_3} : \Box s_3$$

$\boldsymbol{G_2}$ simplifies to $\Box\texttt{true}$, and $\boldsymbol{G_3}$ simplifies to $\Box s_3$ after replacing m_2 with $\texttt{false}$.

The key-stone of mode-based synthesis is that one can focus on simplified specifications for each of the given modes independently—which improves the scalability of off-the-shelf reactive synthesis tools. However, during an execution, a system can transition between different modes. In these transitions, the system may leave the satisfaction of sub-formulas pending for the arriving modes. We call these sub-formulas *pending obligations*. The following definition captures a subset of them that a mode may leave pending for a successor mode.

Definition 2 (Pending Obligations). *The set of (potential) pending obligations for mode* m_i *is* $\mathcal{O}^p(m_i) = \{\psi \mid \bigcirc\psi \in \mathit{subformulas}(\varphi_{m_i})\}$.

[1] Throughout the paper, we use *"reduced spec."* and *"projection"* interchangebly.

Cumulative Obligations. Cumulative obligations, denoted $\mathcal{O}^c(m_j)$, represent the obligations a mode m_j inherits from its predecessors during transitions. They ensure that all pending requirements are satisfied throughout the system. A straightforward, though imprecise, method to compute them is by aggregating obligations from all predecessors. Formally, for a mode m_j: $\mathcal{O}^c(m_j) = \bigcup_{m_i \prec m_j} \mathcal{O}^p(m_i)$. The concept of cumulative obligations is illustrated in the following example.

Example 2. Consider a mode-graph with $M = \{m_1 m_2, m_3\}$ and $m_1 \prec m_2$, $m_2 \prec m_3$ and $m_3 \prec m_1$. Let e_1 and e_2 be environment-controlled variables, with the system controlling $\{s_1, s_2, s_3, s_4, m_1, m_2, m_3\}$. The specification is:

$$\boldsymbol{G_1} : \Box(e_1 \rightarrow (m_1 \rightarrow (\bigcirc m_2 \wedge \bigcirc\bigcirc s_2))) \qquad \boldsymbol{G_2} : \Box(e_2 \rightarrow (\neg m_3 \rightarrow \bigcirc\bigcirc(\neg s_3 \vee s_1)))$$
$$\boldsymbol{G_3} : \Box(\neg e_1 \rightarrow (m_2 \rightarrow \bigcirc(m_3 \wedge s_4))) \qquad \boldsymbol{G_4} : \Box(m_3 \rightarrow \bigcirc(m_1 \wedge s_1))$$

The reduced specifications for modes m_1, m_2, and m_3 are $\varphi_{m_1} = \boldsymbol{G_1} \wedge \boldsymbol{G_2}$, $\varphi_{m_2} = \boldsymbol{G_2} \wedge \boldsymbol{G_3}$, and $\varphi_{m_3} = \boldsymbol{G_4}$. When the system transitions from m_1 to m_2 while e_1 holds, mode m_2 must fulfill the pending obligation $\bigcirc s_2$. The exact obligations, however, depend on the order of events. If e_2 occurs before e_1, the system must satisfy both $\neg s_3 \vee s_1$ and $\bigcirc s_2$. On the other hand, if e_1 occurs before e_2, or if both hold simultaneously, the system must satisfy $\bigcirc(\neg s_3 \vee s_1)$ along with $\bigcirc s_2$. If the system is in mode m_2 and e_1 does not hold, the system transitions to m_3 leaving the pending obligation $m_3 \wedge s_4$. The pending obligations for m_1, m_2 and m_3 are:

$$\mathcal{O}^p(m_1) = \{m_2, \bigcirc s_2, s_2, \bigcirc(\neg s_3 \vee s_1), \neg s_3 \vee s_1\}$$
$$\mathcal{O}^p(m_2) = \{\bigcirc(\neg s_3 \vee s_1), \neg s_3 \vee s_1, m_3 \wedge s_4\} \qquad \mathcal{O}^p(m_3) = \{m_1 \wedge s_1\}$$

From this, the *cumulative obligations* are: $\mathcal{O}^c(m_1) = \mathcal{O}^p(m_3)$; $\mathcal{O}^c(m_2) = \mathcal{O}^p(m_1)$; $\mathcal{O}^c(m_3) = \mathcal{O}^p(m_2)$.

However, as mentioned before, this simple aggregation is imprecise and can lead to incorrect results. It fails to capture interactions between obligations arising from mode transitions in execution paths as demonstrated in the following example.

Example 3. Consider a specification with the following guarantees:

$$\boldsymbol{G_1} : \Box(m_1 \rightarrow \bigcirc m_2) \qquad \boldsymbol{G_2} : \Box(m_2 \rightarrow \bigcirc m_1)$$
$$\boldsymbol{G_3} : \Box(m_1 \rightarrow \bigcirc\bigcirc p) \qquad \boldsymbol{G_4} : \Box(m_1 \rightarrow p)$$

Here, $\mathcal{O}^p(m_1)$ includes $\{m_2, \bigcirc p, p\}$, while $\mathcal{O}^p(m_2)$ only contains $\{m_1\}$ because φ_{m_2} retains only $\boldsymbol{G_2}$. If we propagate obligations naively without accounting for mode interactions, $\mathcal{O}^c(m_1)$ might incorrectly focus on satisfying only m_1, neglecting the obligation to satisfy p. This error occurs because $\boldsymbol{G_2}$ forces m_2 to transition back to m_1 after just one step, leaving p—which originates from the obligation $\bigcirc p$ that m_1 transferred to m_2—unsatisfied. Such propagation would compromise the correctness of the specification leading to *false negatives*.

Algorithm 1: *Fixpoint Algorithm for Cumulative Obligations.*

Input: $\varphi = (A, (\theta_s, \varphi_s))$, $\mathcal{G} = (M, \prec)$
for $m_j \in M$ **do**
  $\mathcal{O}^c[m_j] \leftarrow \bigcup_{m_i \prec m_j} (\mathsf{obligations}(\mathsf{reduce}(\varphi_s, m_i)) \cup \mathsf{obligations}(\mathsf{reduce}(\theta_s, m_i)))$
changed $\leftarrow$ `true`
while *changed* **do**
  changed $\leftarrow$ `false`
  for $(m_i, m_j) \in \prec$ **do**
    for $u \in \bigcup_{\bigcirc^k p \in \mathcal{O}^c[m_i]} \{\bigcirc^{k-1} p, \bigcirc^{k-2} p, \ldots, p\}$ **do**
      if $u \notin \mathcal{O}^c[m_j]$ **then**
        $\mathcal{O}^c[m_j] \leftarrow \mathcal{O}^c[m_j] \cup \{u\}$; *changed* $\leftarrow$ `true`
return $\mathcal{O}^c$

To address these challenges, Alg. 1 systematically computes all *cumulative obligations* that a mode may need to satisfy, ensuring the global specification is correctly enforced across transitions. This algorithm iterates until a stable set of obligations is computed. Given a valid set of modes $M = \{m_0, \ldots, m_k\}$, the *cumulative obligations* $\mathcal{O}^c(m_i)$ for each $m_i \in M$ are the conditions that m_i may be forced to satisfy based on the pending obligations inherited from all its predecessors. Each $\mathcal{O}^c(m_i)$ is computed using Alg. 1. Revisiting Ex. 3, Alg.1 iteratively determines that when transitioning from m_2 to m_1, m_1 must satisfy both p and itself, resulting in $\mathcal{O}^c(m_1) = \{p, m_1\}$. We refer to the set $\mathcal{O} = \bigcup_{m_i \in M} \mathcal{O}^c(m_i)$ as the *universe of obligations* across different modes. To systematically explore the universe of obligations, we introduce *obligation variables* that encode each element within this universe, encoding whether the corresponding obligation is considered. We introduce a fresh variable $v_{\bigcirc^i \varphi}$ for each formula $\varphi \in \mathcal{O}$. We use $v(\varphi)$ for the variable corresponding to φ. While Alg. 1 performs the correct propagation of obligations, the set of obligations computed is a superset of the obligations that a mode may be requested to fulfill. Asking an instance of a mode to satisfy a larger subset of the cumulative obligations makes the instance more difficult to be realizable, while it helps predecessor instances to be realizable. The concept of *initial condition* captures this notion of combination of obligations.

Definition 3 (Initial Conditions). *Given a mode $m_i \in M$, the set of* initial conditions $\mathcal{I}(m_i)$ *is the set of all possible conjunctions of subsets of cumulative obligations:*

$$\mathcal{I}(m_i) = \left\{ \bigwedge_{\phi \in \mathcal{S} \cup \{\textit{true}\}} \phi \;\middle|\; \mathcal{S} \subseteq \mathcal{O}^c(m_i) \right\}$$

Example 4. Consider $\mathcal{O}^c(m_1) = \{\bigcirc q, \bigcirc r\}$ and $\mathcal{O}^c(m_2) = \{s\}$. The initial conditions are: $\mathcal{I}(m_1) = \{\bigcirc q \wedge \bigcirc r, \bigcirc q, \bigcirc r, \texttt{true}\}$ and $\mathcal{I}(m_2) = \{s, \texttt{true}\}$. [2]

[2] Each $x \in \mathcal{I}(m_i)$ is conjoined with m_i. For readability, it is omitted from the text.

Algorithm 2: *Sequential Decomposition.*

Input: $\varphi = (A, (\theta_s, \varphi_s))$, $\Theta = \{\theta_{s_0}, \dots, \theta_{s_n}\}$, $\mathcal{G} = (M, \prec)$, $\mathcal{O}$
$\mathcal{O}_{\text{sorted}} = \mathsf{sortBySize}(\mathcal{O}, \downarrow)$; $\theta'_{s_i} \leftarrow \texttt{true}$
for $\psi \in \theta_s$ **do**
  $\psi' \leftarrow \psi[f \backslash v(f)]$ for all $f \in \mathcal{O}_{\text{sorted}}$
  $\theta'_{s_i} \leftarrow \theta'_{s_i} \wedge \psi'$
for $m_i \in M$ **do**
  $\varphi'_s \leftarrow \mathsf{reduce}(\varphi_s, m_i)$; $\varphi^R_{m_i} \leftarrow \texttt{true}$
  for $\psi \in \varphi'_s$ **do**
    $\psi' \leftarrow \psi[f \backslash v(f)]$ for all $f \in \mathcal{O}_{\text{sorted}}$
    $\varphi^R_{m_i} \leftarrow \varphi^R_{m_i} \wedge (\neg done \rightarrow \psi')$
  for $\bigcirc^k p \in \mathcal{O}_{sorted}$ **do**
    if $k = 1$ **then** $\varphi^R_{m_i} \leftarrow \varphi^R_{m_i} \wedge ((\neg done \wedge v(\bigcirc p)) \rightarrow \bigcirc(\neg done \rightarrow p))$;
    else $\varphi^R_{m_i} \leftarrow \varphi^R_{m_i} \wedge ((\neg done \wedge v(\bigcirc^k p)) \rightarrow \bigcirc(\neg done \rightarrow v(\bigcirc^{k-1} p)))$;
  for $(m_i, m_j) \in \prec$, *and* $\bigcirc^k p \in \mathcal{O}_{sorted}$ *and* $\theta_{s_j} \in \Theta$ **do**
    if $(\theta_{s_j} \wedge \neg v(\bigcirc^{k-1} p))$ *is sat* **then** $\varphi^R_{m_i} \leftarrow \varphi^R_{m_i} \wedge (jump_j \rightarrow \neg v(\bigcirc^{k-1} p))$;
  $\varphi^R_{m_i} \leftarrow \varphi^R_{m_i} \wedge ((\bigvee_{(m_i, m_j) \in \prec} jump_j) \rightarrow \bigcirc done)$
  $\varphi^R_{m_i} \leftarrow \varphi^R_{m_i} \wedge ((\neg \bigvee_{(m_i, m_j) \in \prec} jump_j) \rightarrow (\neg done \rightarrow \bigcirc \neg done))$
  $\Pi[i] \leftarrow (A, (\theta'_{s_i}, \varphi^R_{m_i}))$
return Π

Given a mode-graph $\mathcal{G}$, a set of valid modes $M = \{m_1, \dots, m_k\}$, and a GX_0 specification $\varphi = (A, G)$, the mode-based synthesis method consists of generating a set of projections, $\Pi = \{(\theta_{e_1}, \theta_{s_1}, \varphi^R_{m_1}$ [3]$), \dots, (\theta_{e_k}, \theta_{s_k}, \varphi^R_{m_k})\}$ such that all are *realizable*, and can be composed into a strategy for the original specification.

This process must satisfy the following objectives:

I) Ensure that each projection $(\theta_{e_i}, \theta_{s_i}, \varphi^R_{m_i}) \in \Pi$ is realizable. Even though the local strategy generated is infinite, it could decide to *jump* into a successor node and move into a winning sink state.

II) For each $m_i \prec m_j$ the successor mode m_j must satisfy the obligations inherited from its predecessor m_i, denoted as $\mathcal{O}^p(m_i) \cap \mathcal{O}^c(m_j)$. The connection between projections is managed through the initial condition $\theta_{s_j} \in \mathcal{I}(m_j)$ of the successor m_j. Formally, for all $m_i \prec m_j$, $\theta_{s_j} \implies \bigwedge(\mathcal{O}^p(m_i) \cap \mathcal{O}^c(m_j))$. This ensures that a transition from m_i to m_j is valid only if m_j satisfies the pending obligations of mode m_i, ensuring the correct connection and composition of projections for sequential decomposition.

III) In the absence of a specific $\prec$ provided in $\mathcal{G}$, the complete graph is used.

Alg. 2 adapted from [13], outlines the mode-based synthesis approach. The algorithm generates a projection $\varphi^R_{m_i}$ for each mode m_i, ensuring consistency with **manually provided** initial conditions (as required in point (II)). To demonstrate the method, consider the specification ψ in Fig. 1. Due to its complexity and the number of variables, traditional synthesis tools often struggle with spec-

[3] We explain what the notation $\varphi^R_{m_i}$ means below

ifications like this. Assume that each mode corresponds to a *unique* counter value. For instance, in mode m_{20} (i.e., *counter=20*), the approach projects the global specification onto m_{20} relying on manually specified initial conditions. Alg. 2 ensures that the projection $\varphi_{m_{20}^R}$, along with others, is consistent with the global specification ψ. A *reduced* version of this projection is shown in Fig. 2. Transitions between modes are modeled using fresh variables *jump* (indicating a transition to a successor mode) and *done* (indicating that the game for the current mode is completed due to a transition). A transition is allowed only if the initial condition of the successor mode satisfies the obligations at the time of the jump. This guarantees that strategies for different modes are properly connected. Although Alg. 2 significantly improves synthesis speed, it introduces two limitations: (1) Each mode is associated with a single initial condition, and (2) these initial conditions must be manually specified. To address (1), Alg. 2 can be modified to iterate over a set of sets of possible initial conditions $\Theta = \{\{\theta^1_{s_i}, \ldots, \theta^k_{s_i}\}, \{\theta^1_{s_j}, \ldots, \theta^k_{s_j}\}, \ldots\}$, updating the jump variable to $jump^i_j$ for different initial conditions. However, (2) remains a significant challenge due to the considerable manual effort required to compute initial conditions. To address this, we propose here a method for *automatic mode-based synthesis*, which eliminates the need for manual specification by automatically computing initial conditions for each projection. Alg 2 then generates projections using these automatically computed initial conditions Θ, ensuring realizability and consistency.

```
env boolean reset, start;
sys Int(20) counter; sys Int(1) jump
sys boolean trigger, done, o_1;

asm G !(reset and start);
gar (counter=20 and !done);
gar G (!done -> trigger);
gar G (!done -> counter=20 -> o_1);
gar G (!done -> reset -> o_1);
gar G (!done and o_1 -> next(done))
gar G (jump=1 -> next(done));
gar G (done -> next(done));
```

Fig. 2: Reduced specification.

```
env boolean reset, start;
sys Int(1..20) counter; sys boolean trigger;
// Start and reset are not initially pressed
asm (!reset and !start);
// Only reset or start can be active at a time
asm G !(reset and start);
// Counter is initially at the lowest value
gar counter=1;
// Restart signal always reset the count
gar G (reset -> next(counter=1) );
// Always stay at the same number or increase it
gar G ( (counter=1 and start)  -> next(counter=2 or reset) );
gar G ( (counter=2 and !reset) -> next(counter=3 or reset) );
...
gar G ( (counter=19 and !reset) -> next(counter=20 or reset) );
// Only trigger a signal if the bound is reached.
gar G (counter=20 <-> trigger);
// Reach the limit and start again
gar G (counter=20 -> next(counter=1));
```

Fig. 1: Counter machine example written in `Spectra`.

3.2 A Fixpoint Search Method for Initial Conditions

We propose now an automatic mode-based synthesis method that removes the requirement for manual encoding of initial conditions. The challenge is to discover a collection of feasible initial conditions for each mode.

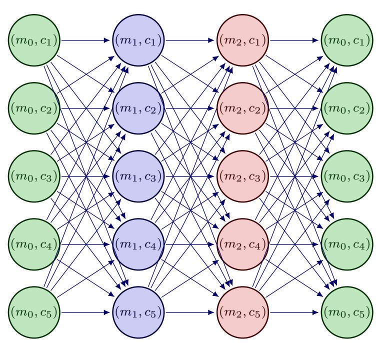
We consider instances of a mode m_j for each active initial condition θ_{s_j}, to support the realizability of predecessors m_i. Verifying the realizability of the pair $(\theta_{s_j}, \varphi^R_{m_j})$ is the main activity, which in turn depends on the active instances of successor modes. This complex circular dependencies between modes and initial conditions requires efficient exploration of the possible instantiations of the mode graph. We solve this problem with a fixpoint search technique that efficiently explores the space of instantiations.

Universe Representation. We start by capturing the representation of the set of possible instantiations.

Definition 4 (Mode Instance Graph). *Given a set of modes M, a universe of initial conditions $\mathcal{I}(m)$ for each mode m, and a mode graph $\mathcal{G} = (M, \prec)$, a Mode Instance Graph (MIG) is a graph $(V, \mapsto)$, where:*
- $V \subseteq \{(m, c) \mid m \in M \text{ and } c \in \mathcal{I}(m)\} \wedge V \neq \emptyset$
- $\mapsto = \{((m_i, c_j), (m_k, c_l)) \mid (m_i, c_j) \in V, (m_k, c_l) \in V \text{ and } m_i \prec m_k)\}$

In a mode instance graph, each mode is instantiated with possibly many initial conditions, which are connected to every instance of its successor nodes.

For example, consider a mode-graph with three modes m_0, m_1, and m_2 and $m_0 \prec m_1 \prec m_2 \prec m_0$. Assume each mode has five possible initial conditions, denoted as (m_i, c_k). The candidate initial conditions c_k for each mode m_i are determined by $\mathcal{I}(m_i)$, which includes all possible subsets of the cumulative obligations for that mode (see Def. 3). *The universe of exploration in the diagram on the left illustrates the MIG for this simplified example.* Our approach systematically evaluates each configuration to ensure that all projections meet the realizability requirements. Informally, our method iterates through the MIG to find a *proof of realizability* for the specification φ, pruning the search space as proofs of realizability are found. Formally:



Definition 5 (Realizability Proof). *Given an MIG $= (V, \mapsto)$, a proof of realizability is a subgraph $\mathcal{R} = (V', \mapsto')$, where $V' \subseteq V$ and $\mapsto' \subseteq \mapsto$, such that for every pair $(m, c) \in V'$, $(c \wedge \square(\varphi^R_m))$ is realizable.*

Alg. 3 presents our solution, which systematically searches for subsets of the initial conditions, one for each node, aiming to find a realizability proof. The search begins with a candidate proof by creating the MIG that contains all

Algorithm 3: *Search for Init. Cond.*

Input: $\varphi = (A, (\theta_s, \varphi_s))$, $\mathcal{G}$, $M = \{m_0, \ldots, m_n\}$
$\mathcal{O}^c \leftarrow$ Alg 1$(\varphi, M, \mathcal{G})$; $\mathcal{O} \leftarrow \bigcup_{m \in M} \mathcal{O}^c[m]$
for $m \in \{m_0..m_n\}$ **do** $\Theta[m] \leftarrow \bigwedge_{\phi \in \mathcal{S} \cup \{\texttt{true}\}} \phi \mid \mathcal{S} \subseteq \mathcal{O}^c[m]$;
MIG $\leftarrow$ createMIG$(\mathcal{G}, \mathcal{I})$
do
  initial $\leftarrow$ `false` ; *finished* $\leftarrow$ `true`
  for $m \in \{m_0..m_n\}$ **do**
    for *each candidate* $\theta \in \Theta[m]$ **do**
      $(A, (\theta, \varphi_m^R)) \leftarrow$ Alg 2$((A, (\theta, \varphi_s)), \Theta, M[m], \mathcal{G}, \mathcal{O})$
      if *realizable*$(A, (\theta, \varphi_m^R))$ **then**
        if $m = m_0$ **then** *initial* $\leftarrow$ `true`;
      else
        finished $\leftarrow$ `false`
        MIG $\leftarrow$ MIG $\setminus (m, \theta)$
        $\Theta[m] \leftarrow \Theta[m] \setminus \theta$
while $\neg$*finished* $\wedge \neg$*initial*;
return MIG

modes and their initial conditions. The algorithm proceeds in rounds, where at each round all remaining instance modes are checked for realizability. Every realizable instance is kept, otherwise (if no successor instantiated with some initial condition can support its realizability), the instance mode is deemed unrealizable and removed. The process iterates until a fixpoint is reached (i.e., a full round is passed with all remaining modes realizable). If all projections are realizable at the fixpoint, the resulting graph is a realizability proof $\mathcal{R}$; otherwise, it is empty.

A minimal modification of Alg. 3, allows us to bypass many realizability checks by inferring results based on previous checks in the same iteration, using *memoization* [9]. For instance, if a mode with candidate initial condition $p \wedge q$ is realizable, the same mode will also be realizable for initial conditions p, q and *true*. Conversely, if p is unrealizable, all initial conditions that imply p are also unrealizable. For example, in our case study (Fig. 1), m_1 spans from 1 to 10, while m_2 spans from 11 to 20. Our algorithm efficiently computes the initial conditions as follows: for m_1, it deduces the initial condition $\theta_1 =$ `counter=1`, and for m_2, it derives the initial condition $\theta_2 =$ `counter=11`. These results align with those manually derived in [13].

3.3 Correctness

This section establishes the soundness of the mode-based synthesis algorithm. We begin by stating a previously established result.

Theorem 1 (Soundness of Alg. 2 [13]). *Given a specification φ, a mode-graph $\mathcal{G}$, and a set of initial conditions $\mathcal{I}$, if all projections are realizable, then φ is realizable.*

The following theorem formalizes the soundness of Algorithm 3.

Theorem 2. *If Algorithm 3 returns a realizability proof $\mathcal{R}$, then φ is realizable.*

Proof (sketch). We proceed by induction on the length of the environment sequence $X_1, \ldots, X_i$. The algorithm constructs a Realizability Proof, where each mode m_i is associated with a set of possible initial conditions $\mathcal{I}(m_i)$ ensuring realizability of the projection $\varphi^R_{m_i}$. We construct a global winning strategy by composing local winning strategies for each instance.

- **Base case:** For the initial environment valuation X_1, the system starts in mode m_0. The algorithm checks the realizability of $\varphi^R_{m_0}$ for each initial condition in $\mathcal{I}(m_0)$. We can use a winning strategy for φ_{m_0} which provides a winning move in the first step.
- **Inductive step:** Assume that for the environment sequence $X_1, \ldots, X_{i-1}$, all projections $\varphi^R_{m_{i'}}$ for $i' < i$ are realizable for at least one initial condition in $\mathcal{I}(m_{i'})$. Now consider the next input X_i and the mode m_i. The algorithm has checked the realizability of $\varphi^R_{m_i}$ for each initial condition in $\mathcal{I}(m_i)$. There are two cases to consider:
  1. *The system remains in mode m_i.* In this case, following the strategy for the corresponding instance of mode m_i is winning for one step, so the specification is not violated at step $i + 1$.
  2. *The system transitions to a new mode m_j.* A transition from mode m_i to mode m_j is allowed only if mode m_j satisfies the cumulative obligations inherited from m_i. This is ensured by an initial condition $\theta_{s_j} \in \mathcal{I}(m_j)$ that satisfies the pending obligations from m_i (i.e., $\mathcal{O}^p(m_i) \cap \mathcal{O}^c(m_j)$). The combined strategy moves to such an initial state.

Therefore, for all steps, the combined strategy satisfies the specification. □

Thm. 2 establishes soundness but not completeness. Mode-based decomposition forces the system to choose the next mode based solely on the current history, disregarding the next environment input. This can lead to false negatives. Consider the specification $\Box(m_1 \to \bigcirc(e \vee m_2))$, where m_1, m_2 are modes and e is an environment input. This requires that if the system is in m_1, then in the next step, either e is true or the system transitions to m_2. Our approach must decide on the transition to m_2 *before* observing e. A winning strategy might depend on e: stay in m_1 if e is true, and transition to m_2 otherwise. For instance, if e alternates, a winning strategy is to stay in m_1 while e is true and move to m_2 when e is false. Our method, however, must choose between *always* staying in m_1 (violating the spec when e is false) or *always* transitioning to m_2 (unnecessarily when e is true), incorrectly reporting unrealizability. This demonstrates that premature decision-making can lead to false negatives. To mitigate incompleteness, we define a sufficient condition: *mode-determinism*, ensuring that the current variable valuation uniquely determines the next mode without violating the specification. For a GX_0 formula $\Box\psi$ [4] with variables $\overline{z} = \mathit{Vars}(\psi) = \mathcal{X} \cup \mathcal{Y}$, let $\varphi(\overline{z}, \overline{z}')$ be the relation between pre- and post-state variables satisfying φ.

[4] Nested $\bigcirc$ operators in ψ are handled by introducing fresh variables $v_{\bigcirc\alpha} \iff \bigcirc\alpha$.

Definition 6 (Mode-deterministic). *A specification* $\square\psi$ *with* $\overline{z} = Vars(\psi)$ *is* mode-deterministic *if there are no modes* m_1, m_2 *and* m_3, $m_2 \neq m_3$ *s.t.:*

$$\exists \overline{z}, \overline{z}_2, \overline{z}_3. \big(m_1(\overline{z}) \wedge (\psi(\overline{z}, \overline{z}_2) \wedge m_2(\overline{z}_2)) \wedge (\psi(\overline{z}, \overline{z}_3) \wedge m_3(\overline{z}_3))\big).$$

This condition ensures that no state in m_1 can transition simultaneously to m_2 and m_3. Mode-determinism can be verified using k^3 SAT queries, where k is the number of modes. For a mode-deterministic specification φ the *feasible mode jumps* $J(\varphi) \subseteq M \times M$ are defined as: $(m_i, m_j) \in J$ if: $\exists \overline{z}, \overline{z}'.\big(m_i(\overline{z}) \wedge \psi(\overline{z}, \overline{z}') \wedge m_j(\overline{z}')\big)$ This captures possible mode transitions within a φ trace.

Theorem 3. *Let* φ *be a mode-deterministic specification and* $(M, \prec)$ *a mode-graph such that* $J(\varphi) \subseteq \prec$*. Then, if the mode-based synthesis algorithm returns unrealizable,* φ *is unrealizable.*

It is always possible to choose a proper $\prec$ either by computing $J(\varphi)$ or by using the complete mode-graph.

4 Empirical Evaluation

We evaluate our approach around the following research questions:
RQ1: How effectively does our method compute initial conditions?
RQ2: Does our mode-based technique improve controller synthesis time compared to traditional methods?
RQ3: How well does our heuristic prune the search space?

Specifications. For our evaluation, we use benchmarks from [13] alongside new specifications from recent work [58,49], written in languages like `Spectra` and `FRET`. Additionally, we introduce *goal-conflicts* and adapt each specification to *mode-based determinism* [5] making them unrealizable to test whether our method correctly identifies those scenarios. Table 1 summarizes the specifications, including assumptions (#A), guarantees (#G), and modes (#M).

Spec.	$\#A - \#G$	$\#M$
counter(n)	2-(n+5)	2,4,7
sis-1500	2-7	3
thermostat(n)	3-4	3
cruise-fse	3-15	4
altlayer(n)	1-9	3
lift(n)	1-187	3
fret-lift	2-14	4
double-counter(n)	2-(2n + 5)	2,4,7

Table 1: Specs and Modes.

Setting and Evaluation. We implemented our approach in Java, leveraging the widely-used `OwL` library [42] for parsing and manipulating LTL formulas. For verifying realizability, we used `Strix` [52]. The experiments were conducted on a cluster featuring Intel Xeon processors clocked at 2.6GHz and equipped with 16GB of RAM, running GNU/Linux. Our tool, case studies, and instructions for replicating the experiments are in our replication package [6].

[5] using SCR methodology [35]
[6] https://sites.google.com/view/mode-decomposition

Specifications		Time (s)			#Strix		Detailed time (fp)	
Name.	$\#M$	Mon.	fp	fph	fp	fph	Proj.	Strix
counter-10	2	8	0.56	0.49	2	2	0.41	0.15
counter-14	7	`timeout`	1.23	1.31	7	7	0.82	0.41
counter-20	4	`timeout`	1.01	1.06	4	3	0.80	0.27
lift-15	3	`timeout`	4.03	4.00	5	4	3.44	0.59
sis-1500	3	`timeout`	46.40	43.00	3	2	42.84	3.92
double-counter-10	2	6	0.80	0.73	3	3	0.60	0.20
double-counter-14	7	125	3.23	2.96	13	10	2.23	1.00
double-counter-20	4	`timeout`	2.62	2.56	7	5	2.14	0.48
cruise-fse	4	`timeout`	65.37	63.89	11	9	45.86	19.51
altlayer	3	`timeout`	18.13	18.52	4	4	15.83	2.30
fret-lift	4	`timeout`	27.48	27.12	4	4	21.80	5.68
thermostat-80	3	60	29.64	29.66	3	2	27.40	2.24
thermostat-150	3	ERROR	60.53	55.41	3	2	52.68	7.85

Table 2: All experimental results (Realizable cases).

Effectiveness and Performance Evaluation. We evaluated three approaches: the monolithic method (*mon*), our fixpoint method (*fp*), and our memoization [9] fixpoint method (*fph*). Both *fp* and *fph* consistently outperformed the monolithic approach across various specifications (see Table 2). Moreover, the time distribution shows that *Strix* time is significantly lower than the projection time (*Proj*).

In our corpus, particularly in SCR specifications from requirements engineering, candidates are often mutually exclusive, limiting memoization's effectiveness. While it does not bypass many realizability checks, it still moderately reduces the total number of solver calls. For more complex specifications with larger search spaces and well-defined lattice structures, memoization becomes significantly more effective. By leveraging the fact that unrealizable formulas have unrealizable supersets, and realizable formulas have realizable subsets, our approach propagates results across the lattice, pruning the search space and minimizing solver calls. Tests on randomly generated formulas using `Spot` [26] show that our memoization approach can cut solver calls by up to 50%, leading to faster execution times. This demonstrates the potential of our approach for efficiently handling intricate specifications in mode-based synthesis. Additionally, in the context of unrealizable cases following *mode-based deterministic* specifications (see Table 3), we observed the same trends as in realizable cases. This consistency further underscores the robustness and reliability of our method across both realizable and unrealizable scenarios.

Impact on Synthesis Time. Our comparative analysis revealed significant differences in synthesis time between the approaches. The monolithic method consistently reached the *ten-minute* timeout on larger instances, with some cases failing due to the size of the formulas. In contrast, both *fp* and *fph* completed synthesis well within the time limits, reducing synthesis time by over 90%.

Specifications		Time (s)			#Strix		Detailed Time (fph)	
Name	#M	Mon	fp	fph	fp	fph	Proj.	Strix
counter-10	2	8	0.61	0.59	3	2	0.39	0.20
counter-14	7	timeout	4.33	2.66	21	16	1.69	0.98
counter-20	4	timeout	2.237	4.60	11	10	3.28	1.33
lift-15	3	timeout	6.92	10.93	9	7	9.50	1.44
sis-1500	3	timeout	44.72	58.19	3	3	53.58	4.62
double-counter-10	2	13	1.28	1.83	7	7	1.14	0.61
double-counter-14	7	timeout	2.52	3.18	13	10	2.32	0.84
double-counter-20	4	timeout	6.83	9.60	23	19	7.75	1.84
cru-fse	4	timeout	99.83	134.72	23	18	101.42	33.30
altlayer	3	timeout	65.40	65.45	11	10	57.77	7.68
fret-lift	4	timeout	79.80	81.80	10	10	64.18	17.61
thermostat-80	3	74	110.67	100.84	9	7	94.20	6.64
thermostat-150	3	ERROR	193.17	185.69	9	6	173.87	11.83

Table 3: All experimental results (Unrealizable cases). All cases executed using mode-determinism.

Moreover, our techniques complement state-of-the-art decomposition tools, none of which [39,29] could handle the specifications in our corpus. Frequent use of state variables or modes posed significant challenges for these tools, as noted by Mavridou *et al.* [51]. Our mode-based approaches, however, excelled in these environments, demonstrating adaptability and effectiveness where traditional methods fall short.

5 Conclusion and Future Work

This paper presents a novel, fully automated mode-based reactive synthesis method. Taking an LTL (GX_0) specification and a set of modes (with optional transitions) as input, our iterative algorithm efficiently searches for initial condition combinations that realize the overall specification or concludes unrealizability. This automatic search for suitable initial conditions is a key feature of our approach, simplifying the synthesis process for engineers.

A current limitation is that completeness for unrealizability requires mode-based determinism and a subsuming mode-graph, which we plan to address in future work. However, our method achieves significantly faster synthesis compared to monolithic methods, enabling more effective derivation of controllers for complex real-world specifications, as supported by a thorough empirical evaluation. Future work will also explore controller explainability and consider extensions to richer LTL fragments.

References

1. Iso/iec/ieee international standard - systems and software engineering – life cycle processes –requirements engineering. ISO/IEC/IEEE 29148:2011(E) pp. 1–94 (2011). https://doi.org/10.1109/IEEESTD.2011.6146379
2. Arabian-Hoseynabadi, H., Oraee, H., Tavner, P.: Failure modes and effects analysis (fmea) for wind turbines. International journal of electrical power & energy systems **32**(7), 817–824 (2010)
3. Arora, C., Grundy, J., Abdelrazek, M.: Advancing requirements engineering through generative AI: assessing the role of llms. CoRR **abs/2310.13976** (2023). https://doi.org/10.48550/ARXIV.2310.13976, https://doi.org/10.48550/arXiv.2310.13976
4. Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F.: Extracting domain models from natural-language requirements: approach and industrial evaluation. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. p. 250–260. MODELS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976767.2976769, https://doi.org/10.1145/2976767.2976769
5. Arvidsson, S., Axell, J.: Prompt engineering guidelines for llms in requirements engineering (2023)
6. Balkan, A., Vardi, M., Tabuada, P.: Mode-target games: Reactive synthesis for control applications. IEEE Transactions on Automatic Control **63**(1), 196–202 (2017)
7. Bansal, S., Li, Y., Tabajara, L.M., Vardi, M.Y.: Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In: AAAI'20
8. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011). https://doi.org/10.1145/2000799.2000800, https://doi.org/10.1145/2000799.2000800
9. Beame, P., Impagliazzo, R., Pitassi, T., Segerlind, N.: Memoization and dpll: formula caching proof systems. In: 18th IEEE Annual Conference on Computational Complexity, 2003. Proceedings. pp. 248–259 (2003). https://doi.org/10.1109/CCC.2003.1214425
10. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. JCSS **78**(3), 911–938 (2012)
11. Brizzio, M.: Resolving goal-conflicts and scaling synthesis through mode-based decomposition. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings. pp. 207–211. ICSE-Companion '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3639478.3639801, https://doi.org/10.1145/3639478.3639801
12. Brizzio, M., Cordy, M., Papadakis, M., Sánchez, C., Aguirre, N., Degiovanni, R.: Automated Repair of Unrealisable LTL Specifications Guided by Model Counting. In: Proc. of GECCO'23. pp. 1499–1507. ACM (2023), https://doi.acm.org/10.1145/3583131.3590454
13. Brizzio, M., Sánchez, C.: Efficient reactive synthesis using mode decomposition. In: Proc. of ICTAC 2023. LNCS, vol. 14446, pp. 256–275. Springer (2023), https://doi.org/10.1007/978-3-031-47963-2_16
14. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A.,

Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language Models are Few-Shot Learners. In: NeurIPS (2020)
15. Broy, M.: Multifunctional software systems: Structured modeling and specification of functional requirements. Science of Computer Programming **75**(12), 1193–1214 (2010). https://doi.org/https://doi.org/10.1016/j.scico.2010.06.007, https://www.sciencedirect.com/science/article/pii/S016764231000119X
16. Carvalho, L., Degiovanni, R., Brizzio, M., Cordy, M., Aguirre, N., Traon, Y.L., Papadakis, M.: ACoRe: Automated Goal-Conflict Resolution. In: Proc. of FASE 2023. LNCS, vol. 13991, pp. 3–25. Springer (2023), https://doi.org/10.1007/978-3-031-30826-0_1
17. Chatterjee, K., Henzinger, T.A., Otop, J., Pavlogiannis, A.: Distributed synthesis for ltl fragments. In: 2013 Formal Methods in Computer-Aided Design. pp. 18–25. IEEE (2013)
18. Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., others: Palm: Scaling language modeling with pathways. ArXiv **abs/2204.02311** (2022)
19. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The Cyber-Physical Systems Series, MIT Press (1999), https://books.google.es/books?id=Nmc4wEaLXFEC
20. De Giacomo, G., Favorito, M.: Compositional approach to translate LTLf/LDLf into deterministic finite automata. In: Proc. of ICAPS'21. pp. 122–130 (2021)
21. Degiovanni, R., Molina, F., Regis, G., Aguirre, N.: A genetic algorithm for goal-conflict identification. In: Proc. of ASE 2018. pp. 520–531 (2018), https://doi.org/10.1145/3238147.3238220
22. Degiovanni, R., Ricci, N., Alrajeh, D., Castro, P.F., Aguirre, N.: Goal-conflict detection based on temporal satisfiability checking. In: Proc. of ASE 2016. pp. 507–518 (2016), http://doi.acm.org/10.1145/2970276.2970349
23. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding (2019)
24. Dietrich, D., Atlee, J.M.: A mode-based pattern for feature requirements, and a generic feature interface. 2013 21st IEEE International Requirements Engineering Conference (RE) pp. 82–91 (2013), https://api.semanticscholar.org/CorpusID:29015370
25. Dureja, R., Rozier, K.Y.: More scalable LTL model checking via discovering design-space dependencies (D^3). In: Proc. of TACAS'18. pp. 309–327. Springer (2018)
26. Duret-Lutz, A.: Manipulating LTL formulas using Spot 1.0. In: Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13). Lecture Notes in Computer Science, vol. 8172, pp. 442–445. Springer, Hanoi, Vietnam (Oct 2013)
27. Esparza, J., Křetínskỳ, J.: From LTL to deterministic automata: A safraless compositional approach. In: Proc. of CAV'14. pp. 192–208. Springer (2014)
28. Filippidis, I., Murray, R.M.: Layering assume-guarantee contracts for hierarchical system design. Proceedings of the IEEE **106**(9), 1616–1654 (2018)
29. Finkbeiner, B., Geier, G., Passing, N.: Specification decomposition for reactive synthesis. ISSE (2022)
30. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Softw. Test., Verif. Reliab. **19**(3), 215–261 (2009). https://doi.org/10.1002/stvr.402, https://doi.org/10.1002/stvr.402

31. Giannakopoulou, D., Mavridou, A., Rhein, J., Pressburger, T., Schumann, J., Shi, N.: Formal requirements elicitation with fret. In: International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020). No. ARC-E-DAA-TN77785 (2020)
32. Golia, P., Roy, S., Meel, K.S.: Synthesis with explicit dependencies. In: 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2023)
33. Großer, K., Rukavitsyna, M., Jürjens, J.: A comparative evaluation of requirement template systems. In: Schneider, K., Dalpiaz, F., Horkoff, J. (eds.) 31st IEEE International Requirements Engineering Conference, RE 2023, Hannover, Germany, September 4-8, 2023. pp. 41–52. IEEE (2023). https://doi.org/10.1109/RE57278.2023.00014, https://doi.org/10.1109/RE57278.2023.00014
34. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3), 231–274 (1987). https://doi.org/https://doi.org/10.1016/0167-6423(87)90035-9, https://www.sciencedirect.com/science/article/pii/0167642387900359
35. Heitmeyer, C.: Requirements models for critical systems. In: Software and Systems Safety, pp. 158–181. IOS Press (2011)
36. Heitmeyer, C., Labaw, B., Kiskis, D.: Consistency checking of SCR-style requirements specifications. In: Proc. of RE'95. pp. 56–63. IEEE (1995)
37. Heitmeyer, C.L., Archer, M., Bharadwaj, R., Jeffords, R.D.: Tools for constructing requirements specifications: the SCR toolset at the age of nine. Comput. Syst. Sci. Eng. **20**(1) (2005)
38. Hermo, M., Lucio, P., Sánchez, C.: Tableaux for realizability of safety specifications. In: Proc. of the 25th International Symposium on Formal Methods (FM'23). LNCS, vol. 14000, pp. 495–513 (2023). https://doi.org/10.1007/978-3-031-27481-7_28, https://doi.org/10.1007/978-3-031-27481-7_28
39. Iannopollo, A., Tripakis, S., Vincentelli, A.: Specification decomposition for synthesis from libraries of LTL assume/guarantee contracts. In: DATE. IEEE (2018)
40. Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1.1. EPTCS **229**, 112–132 (11 2016)
41. Konrad, S., Cheng, B.: Facilitating the construction of specification pattern-based properties. In: 13th IEEE International Conference on Requirements Engineering (RE'05). pp. 329–338 (2005). https://doi.org/10.1109/RE.2005.29
42. Kretínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11138, pp. 543–550. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_34, https://doi.org/10.1007/978-3-030-01090-4_34
43. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Proc. of CAV'06. pp. 31–44. Springer (2006)
44. Leveson, N.G.: Safeware: system safety and computers. ACM (1995)
45. Liu, J.X., Yang, Z., Idrees, I., Liang, S., Schornstein, B., Tellex, S., Shah, A.: Lang2ltl: Translating natural language commands to temporal robot task specification. In: Conference on Robbot Learning (2023)
46. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag New York, Inc., New York, NY, USA (1992)
47. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer-Verlag, New York, NY, USA (1995). https://doi.org/10.1007/978-1-4612-4222-2

48. Maoz, S., Ringert, J.O.: Spectra: a specification language for reactive systems. Software and Systems Modeling **20**(5), 1553–1586 (2021). https://doi.org/10.1007/s10270-021-00868-z, https://doi.org/10.1007/s10270-021-00868-z
49. Maoz, S., Shalom, R.: Unrealizable cores for reactive systems specifications: artifact. In: Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings. p. 217–218. ICSE '21, IEEE Press (2021). https://doi.org/10.1109/ICSE-Companion52605.2021.00097, https://doi.org/10.1109/ICSE-Companion52605.2021.00097
50. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (ears). In: 2009 17th IEEE International Requirements Engineering Conference. pp. 317–322 (2009). https://doi.org/10.1109/RE.2009.9
51. Mavridou, A., Katis, A., Giannakopoulou, D., Kooi, D., Pressburger, T., Whalen, M.W.: From partial to global assume-guarantee contracts: Compositional realizability analysis in FRET. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13047, pp. 503–523. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_27, https://doi.org/10.1007/978-3-030-90870-6_27
52. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Proc. of CAV'18 (Part I). pp. 578–586. Springer (2018)
53. OpenAI: Gpt-4 technical report (2023)
54. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P.F., Leike, J., Lowe, R.: Training language models to follow instructions with human feedback. CoRR **abs/2203.02155** (2022)
55. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive (1) designs. In: Proc. of VMCAI'06. pp. 364–380. Springer (2006)
56. Pnueli, A.: The temporal logic of programs. In: SFCS'77. pp. 46–57. IEEE (1977)
57. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. of the 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'89). pp. 179–190. ACM, New York, NY, USA (1989). https://doi.org/10.1145/75277.75293, http://doi.acm.org/10.1145/75277.75293
58. Pressburger, T., Katis, A., Dutle, A., Mavridou, A.: Authoring, analyzing, and monitoring requirements for a lift-plus-cruise aircraft. In: Ferrari, A., Penzenstadler, B. (eds.) Requirements Engineering: Foundation for Software Quality - 29th International Working Conference, REFSQ 2023, Barcelona, Spain, April 17-20, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13975, pp. 295–308. Springer (2023). https://doi.org/10.1007/978-3-031-29786-1_21, https://doi.org/10.1007/978-3-031-29786-1_21
59. Pudlitz, F., Brokhausen, F., Vogelsang, A.: Extraction of system states from natural language requirements. In: 2019 IEEE 27th International Requirements Engineering Conference (RE). pp. 211–222 (2019). https://doi.org/10.1109/RE.2019.00031
60. de Roever, W.P., Langmaack, H., Pnueli, A. (eds.): Compositionality: The Significant Difference. Springer (1998). https://doi.org/10.1007/3-540-49213-5
61. Shaker, P., Atlee, J.M., Wang, S.: A feature-oriented requirements modelling language. In: 2012 20th IEEE International Requirements Engineering Conference (RE). pp. 151–160. IEEE (2012)

62. Thoppilan, R., Freitas, D.D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.T., Jin, A., Bos, T., Baker, L., Du, Y., Li, Y., Lee, H., Zheng, H.S., Ghafouri, A., Menegali, M., Huang, Y., Krikun, M., Lepikhin, D., Qin, J., Chen, D., Xu, Y., Chen, Z., Roberts, A., Bosma, M., Zhao, V., Zhou, Y., Chang, C.C., Krivokon, I., Rusch, W., Pickett, M., Srinivasan, P., Man, L., Meier-Hellstern, K., Morris, M.R., Doshi, T., Santos, R.D., Duke, T., Soraker, J., Zevenbergen, B., Prabhakaran, V., Diaz, M., Hutchinson, B., Olson, K., Molina, A., Hoffman-John, E., Lee, J., Aroyo, L., Rajakumar, R., Butryna, A., Lamm, M., Kuzmina, V., Fenton, J., Cohen, A., Bernstein, R., Kurzweil, R., Aguera-Arcas, B., Cui, C., Croak, M., Chi, E., Le, Q.: Lamda: Language models for dialog applications (2022)
63. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al.: Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023)
64. Vogelsang, A., Femmer, H., Winkler, C.: Systematic elicitation of mode models for multifunctional systems. In: 2015 IEEE 23rd International Requirements Engineering Conference (RE). pp. 305–314. IEEE (2015)
65. Vogelsang, A., Femmer, H., Winkler, C.: Take care of your modes! an investigation of defects in automotive requirements. In: Requirements Engineering: Foundation for Software Quality: 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings 22. pp. 161–167. Springer (2016)
66. Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E.H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., Fedus, W.: Emergent abilities of large language models. CoRR **abs/2206.07682** (2022)
67. Yang, J., Jin, H., Tang, R., Han, X., Feng, Q., Jiang, H., Yin, B., Hu, X.: Harnessing the power of llms in practice: A survey on chatgpt and beyond (2023)
68. Zeng, A., Liu, X., Du, Z., Wang, Z., Lai, H., Ding, M., Yang, Z., Xu, Y., Zheng, W., Xia, X., Tam, W.L., Ma, Z., Xue, Y., Zhai, J., Chen, W., Liu, Z., Zhang, P., Dong, Y., Tang, J.: Glm-130b: An Open Bilingual Pre-trained Model. ICLR 2023 poster (2023)
69. Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X.V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P.S., Sridhar, A., Wang, T., Zettlemoyer, L.: Opt: Open Pre-trained Transformer Language Models. ArXiv **abs/2205.01068** (2022)
70. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: A symbolic approach to safety LTL synthesis. In: Proc. of HVC. pp. 147–162. Springer (2017)