Research Article

AND DATA SCIENCES
BULLETIN OF COMPUTER

From Unrealizable Scenario-Based Specifications to Environment Assumptions

Naveed Ahmad

University of Alaska, Alaska, United States

Abstract

Live Sequence Charts (LSCs) and related scenario-based notations enable requirements engineers to specify system behaviour in a way that is close to stakeholders' narratives. However, when such scenario collections are translated to reactive synthesis problems (e.g. GR(1) games), they may turn out to be unrealizable: no system strategy can satisfy all mandatory scenarios against an unrestricted environment. Prior work has mostly reported unrealizability but has given limited support for *repairing* the specification. This paper proposes a method for *assumption mining* for scenario-based specifications: given an unrealizable scenario set, we extract from the counter-strategy of the environment the minimum behavioural commitments that the environment must respect so that the original scenario set becomes realizable. These commitments are then projected back to scenario form as additional LSCs ("environment scenarios"). Our approach proceeds in four steps: (1) translate the LSCs to a GR(1)-like game structure; (2) detect unrealizability and compute a winning environment counter-strategy; (3) generalize the counter-strategy into a set of liveness/safety assumptions over environment-controlled messages; (4) render those assumptions as LSCs that requirements engineers can read and negotiate. We illustrate the approach on a variation of the elevator-with-vent example—a typical case where fairness of the environment is missing—and show that the synthesized assumptions are small, comprehensible, and sufficient to restore realizability. The contribution is a bridge between formal counterexample information and the scenario language used by stakeholders.

Keywords: live sequence charts,environment assumptions, mining, synthesis problems

1. Introduction

Scenario-based modelling, and in particular Live Sequence Charts (LSCs), has been shown to be an appealing way to capture early requirements, exceptional situations, and modal distinctions between mandatory and possible behaviour [1]. LSCs extend classical message sequence charts with the ability to distinguish between *hot* (must) and *cold* (may) behaviour, to activate scenarios conditionally through precharts, and to express forbidden interactions [1]. Because of this, they are especially suitable in domains where stakeholders think in terms of stories—"if the user presses the emergency button while the door is open, then the system must keep the door open"—rather than in terms of monolithic state machines. Scenario-based approaches also support incremental development: new charts can be added as new requirements are discovered, without rewriting the whole model.

Corresponding author: Naveed Ahmad (rainaveed444@gmail.com)

BCDS 2022, Vol. 3(2), 1-12. DOI: 10.71448/bcds2232-1

An important advancement in this line of work is that scenario collections are not only *descriptive* but can be made *executable*. Through the notions of *play-out* and, later, *reactive synthesis* from LSCs, we can move from "this is what the system should do" to "here is a controller that enforces it" [2]. In the reactive setting, the specification is compiled into a two-player game between the environment and the system, and a winning strategy for the system corresponds to an implementation that reacts to environment moves while ensuring that all hot parts of all active charts eventually succeed [2]. This provides a high degree of faithfulness: the behaviour that stakeholders specified in scenarios is the behaviour that is enforced at runtime.

However, a central difficulty appears precisely at this synthesis step: the specification can be *unrealizable*. Unrealizability means that, under the default assumption of a fully adversarial environment, there exists no system strategy that can guarantee the satisfaction of all mandatory scenario obligations [3]. Intuitively, the environment can always "play in a way" that prevents some hot condition from ever becoming true. This is not an exotic corner case—it arises very naturally in realistic scenario sets. In practice, stakeholders are very good at specifying what the system must do, but they are much less explicit about what the *environment* (users, sensors, other components) is allowed or required to do. They will write "the door must eventually close," but they will not write "the user will not hold the vent button down forever." When such a specification is given to a synthesis engine that, by design, considers *all* environment behaviours that are not forbidden, the engine rightfully reports that no winning strategy exists.

This observation reveals a structural mismatch: scenario languages are author-friendly and intentionally permissive about the environment, whereas synthesis engines are correctness-oriented and intentionally pessimistic about the environment. Bridging that gap requires us to make the environment side of the story more explicit. One well-known way to do this in game-based synthesis is to add *environment assumptions*: temporal clauses that constrain the environment to behave "fairly," for example by releasing buttons, eventually providing sensor updates, or not perpetually triggering mutually exclusive requests [3]. In classical GR(1) synthesis, specifications are often written as *assumptions* $\rightarrow$ *guarantees*. In scenario-based settings, though, engineers typically do not start with such a two-part structure, so when unrealizability is detected, they are left with the question: *what assumptions should I add?*

This motivates the research question of this paper: **when a scenario-based specification is unrealizable, can we automatically propose additional environment assumptions—in the same scenario language—that make it realizable again?** [3]. Our answer is yes. The key insight is that an unrealizable synthesis attempt already produces exactly the artifact we need: an *environment counter-strategy* [3]. This counter-strategy is a witness of failure; it describes, step by step, how the environment can defeat the system by keeping it from fulfilling some hot obligation. Rather than discarding this counter-strategy or showing it only as a low-level diagnostic, we can *mine* it for the essence of the environment's misbehaviour. Typically, the misbehaviour has a simple shape: the environment keeps some input high forever, or it keeps cycling among a set of inputs in a way that perpetually postpones the system's progress. If we can summarise this behaviour and turn it into a small number of temporal assumptions, we can present them to the requirements engineer as candidate "environment scenarios" [4].

The approach we develop in this paper therefore has four ingredients, which the rest of the paper will elaborate. First, we start from an existing LSC-to-game translation (in particular, a GR(1)-like structure) so that we can run a standard realizability check [2]. Second, when this check

fails, we extract and analyse the environment's counter-strategy to detect recurring environment-controlled cycles that block system liveness [3]. Third, we generalize these concrete cycles into abstract environment assumptions of standard forms such as $\square\lozenge\neg$`ventPressed` ("the environment will eventually release the vent button") or "every request is eventually cleared." Finally, because our users are scenario-oriented, we render these assumptions back into small LSCs that can be read, negotiated, and, if necessary, edited by the same stakeholders who wrote the original scenarios [1]. In this way, the repair stays at the same modelling level as the original specification.

By covering all these aspects—why unrealizability arises in scenario-based synthesis, how counter-strategies capture the cause, how to generalize them without over-constraining the environment, and how to project the result back to LSCs—the paper aims to turn unrealizable results from dead ends into actionable guidance. Instead of telling engineers "your scenarios cannot be implemented," we aim to tell them "your scenarios can be implemented provided the environment behaves according to these additional scenarios." The remainder of the paper formalizes this idea, illustrates it on a representative elevator-style example, and discusses limitations and possible extensions to symbolic or timed settings.

2. Background

We briefly recall the ingredients we need: (i) a scenario language, concretely Live Sequence Charts (LSCs), which is the front-end in which requirements engineers describe behaviour [1]; (ii) a translation from such scenarios to a formal *reactive synthesis* problem, for which we assume a GR(1)-like structure because it is expressive yet algorithmically manageable [2]; and (iii) the notion of *unrealizability*, which is the situation in which no system strategy can satisfy the specification against all admissible environment behaviours [3]. Explicating these elements is important because our contribution will later "sit" between the second and the third: we will take an unrealizable GR(1) instance that originates from LSCs and repair it by adding environment assumptions that are, again, rendered as LSCs.

Live Sequence Charts (LSCs) were introduced as a more expressive, modal extension of Message Sequence Charts (MSCs) [1]. While MSCs are good at describing exemplary interactions, LSCs add several features that make them suitable for prescriptive specifications: LSCs distinguish between *hot* and *cold* elements. A hot message, condition, or region must eventually occur once the chart is active; a cold one is optional. This allows the modeller to say not only "this can happen" but also "this must happen." An LSC typically has a prechart that specifies the activating situation. When the prechart is satisfied in a run of the system, the main chart becomes *live*, meaning its hot parts must be fulfilled. This is how scenario-based specifications achieve conditional obligations ("if the user presses X while Y holds, then do Z"). Each LSC is drawn over several lifelines (instances), each representing a component, object, or external actor. A system is usually described by a *collection* of LSCs, each capturing a particular requirement or use case; together, they form the global behaviour. Because different precharts can be triggered at different times, several LSCs can be active simultaneously. This is an essential point for synthesis: at runtime, the system must satisfy *all* hot obligations of *all* currently active charts, not just of one chart at a time. LSCs also allow the modeller to specify things that must *not* happen once a chart is active (e.g. a forbidden message). Violating these gives rise to consistency or realizability issues.

In earlier scenario-based work, these LSCs were executed by *play-out*, a kind of on-the-fly in-

terpreter that, given the current system state and possible environment moves, picks system moves that keep all active charts satisfied [1]. When we move from play-out to full reactive synthesis, we essentially want to compute *offline* a strategy that the play-out engine would have chosen *online*.

To reason formally about all possible interleavings of charts, LSC collections are compiled to a finite-state two-player game between the *environment* (inputs, external events, user actions) and the *system* (controller actions, outputs). This compilation follows the ideas already explored in scenario-to-automata translations [2]: for every LSC we build a small "cut automaton" that tracks where we are in that chart: which messages have already occurred, which hot condition is pending, whether the main chart is currently live, etc. A global game state is the product of all these chart-cut states. The game state also contains the current valuation of system-controlled and environment-controlled variables or messages (e.g. `doorOpen`, `ventPressed`, `callReq`). Because LSCs may require that something eventually happens, the game needs extra structure—often in the form of progress counters or ranking variables—to check that these liveness requirements are satisfied infinitely often.

Once in this form, the problem becomes a standard reactive synthesis problem. We assume a GR(1) structure because it strikes a balance between expressiveness and tractability: it allows multiple safety assumptions and guarantees, plus multiple liveness assumptions and guarantees, while admitting polynomial-time (in the size of the game graph) synthesis [2, 3].

Formally, a GR(1) specification is written as:

$$\Big(\bigwedge_i \Box\varphi_i^e \;\wedge\; \bigwedge_j \Box\Diamond\psi_j^e\Big) \;\rightarrow\; \Big(\bigwedge_k \Box\varphi_k^s \;\wedge\; \bigwedge_\ell \Box\Diamond\psi_\ell^s\Big),$$

where, $\Box\varphi_i^e$ are *environment safety assumptions* ("the environment never sends an ill-formed message," "the floor number is always within range"), $\Box\Diamond\psi_j^e$ are *environment liveness/fairness assumptions* ("the environment eventually releases the door sensor," "every call is eventually cleared"), $\Box\varphi_k^s$ are *system safety guarantees* (the system never opens the door while moving), $\Box\Diamond\psi_\ell^s$ are *system liveness guarantees* (the system eventually serves every call).

Realizability in this setting means: for every environment behaviour that satisfies all the environment assumptions, there exists a system strategy that yields a play satisfying all the system guarantees [2, 3, 5]. GR(1) algorithms compute such a strategy, and if none exists, they can explain why, typically by producing a losing set of states for the system or an explicit environment counter-strategy [3, 5].

The important observation for this paper is that when we translate LSCs to such a GR(1) game, we often start with *no* (or very few) explicit environment assumptions, because the original LSCs mostly talk about the system side [1]. Thus, the left-hand side of the implication above is too weak, and the environment is effectively unconstrained. As a result, the environment is permitted to exhibit behaviours that domain experts would consider unrealistic but that the synthesis engine must consider, leading directly to unrealizability.

Unrealizability is the situation in which the system has *no* winning strategy in the game derived from the LSCs. Equivalently, the environment has a *winning counter-strategy*. GR(1) synthesis procedures, when they fail, can construct such a counter-strategy: it is a description of how the environment can respond to any system move so as to eventually force violation of some system liveness guarantee (or to reach a bad state) [3, 4, 6]. This is analogous to counterexample generation in model checking, but here the counterexample is *strategic*—it tells us how the environment can keep winning forever.

Intuitively, in scenario-based settings, the most common way this happens is through **environmental blocking**: the environment repeatedly chooses an input that keeps some LSC's hot part from progressing. For instance, in the elevator-with-vent example, there might be one LSC that says "after servicing a floor, eventually close the door" and another that says "if the vent button is pressed, keep the door open" [1]. If the environment is allowed to press and hold the vent button forever, the system can never complete the "eventually close the door" obligation. The GR(1) checker will trace this back to an environment strategy that, whenever the system is about to progress, keeps the relevant input high. This is precisely the kind of pattern our mining procedure targets.

More generally, unrealizability can arise from (i) **missing fairness** on environment inputs (the environment may starve the system), (ii) **cyclic triggering** of scenarios that mutually postpone each other, or (iii) **true contradictions** among hot parts of concurrently active charts [4, 6, 7]. Missing fairness is the easiest to repair by adding environment assumptions. Cyclic triggering may require more structured assumptions (e.g. "if chart A is triggered infinitely often, chart B must eventually be allowed to complete"). True contradictions, on the other hand, typically indicate that the scenario set itself needs refactoring, not just additional assumptions.

In all these cases, the counter-strategy is extremely valuable: it tells us not just that "this set of LSCs cannot be implemented," but also *which* environment moves, repeated in *which* context, make it impossible. Our work will use exactly this counter-strategy as the starting point for mining additional environment assumptions that, once added to the left-hand side of the GR(1) implication, restore realizability and can then be rendered back as LSCs [4, 6].

3. Problem Statement

Let S be a scenario-based specification given as a finite set of LSCs, each capturing a particular requirement, use case, or exceptional situation [1]. Using the standard translation sketched in the background, we obtain from S a two-player game $G(S)$ between the environment and the system, typically of GR(1) shape [2, 3]. Now assume that, when we run a realizability check on $G(S)$, the answer is negative: the system does not have a winning strategy. In such a case, modern GR(1) procedures provide us not only with the verdict "unrealizable," but also with an *environment counter-strategy* $\mathcal{C}$, that is, a description of how the environment can play so as to make some hot obligation in S forever unachievable [3, 4, 6]. This is the exact situation we target: we already have a well-formed scenario model, we already have its game semantics, and we already know precisely how the environment can defeat it.

The problem we address is how to turn this information into a *repair* of the original scenario specification. Concretely, we want to compute a set of additional clauses—call them environment assumptions—denoted $A = \{a_1, \ldots, a_n\}$, that we can add to S so that the new specification $S \cup A$ becomes realizable. These assumptions should be derived from the way the environment misbehaves in $\mathcal{C}$, but they must satisfy several constraints. First, the repaired specification must truly restore realizability: once A is added, the GR(1) check on the corresponding game should succeed, meaning the system now has a winning strategy [3, 5]. Second, each assumption must talk *only* about environment-controlled messages or variables; we are not allowed to "fix" the problem by forcing the system to do something it currently cannot do—our goal is to constrain the environment, not to rewrite system guarantees [4]. Third, because our users write and read LSCs, every mined assumption must be expressible back in that same formalism, ideally as a small, comprehensible chart that

says things like "after the vent is pressed, it is eventually released" [1]. Finally, and importantly from a requirements-engineering point of view, the set A must be *weak* enough to be acceptable to stakeholders: it should not ban large classes of realistic environment behaviour, but rather capture the minimal fairness or release conditions that the environment must satisfy in order for the system obligations in S to make sense [4, 7]. In short, the task is to bridge an unrealizable game and a scenario-level explanation by synthesizing environment-side LSCs that (i) eliminate the specific counterplay and (ii) remain natural to the people who authored the original scenarios.

This is not a unique problem: several assumption sets can work. We therefore focus on computing a *locally weakest* set derived directly from $\mathcal{C}$ [6].

4. Assumption Mining Approach

The core idea of our approach is to take the information that the synthesis engine already computes in the unrealizable case—the environment's counter-strategy—and to systematically turn it into additional, human-readable environment scenarios [3, 4, 6, 8]. Conceptually, the pipeline has four stages: (1) build the usual game structure from the LSCs and run realizability; (2) inspect the losing explanation, i.e. the counter-strategy, to see *how* the environment is blocking progress; (3) abstract these concrete blocking behaviours into small, standard temporal assumptions such as fairness or eventual release; and (4) project those assumptions back into LSC form so that they can be added to the original specification without changing notations [1, 2, 4, 9].

4.1. Step 1: Translate to Game and Check Realizability

We begin exactly as existing LSC-based synthesis approaches do. The scenario set S is translated into a GR(1)-like game $G(S)$ by taking the product of all chart-cut automata and annotating the transitions with which player (environment or system) chooses the next move [2, 5]. Liveness conditions are introduced to capture the hot parts of the charts. We then invoke a standard GR(1) realizability check [3, 5]. If the game is realizable, nothing more needs to be done. If it is not realizable, the GR(1) engine also returns an environment counter-strategy $\mathcal{C}$, which is our starting point [3, 6].

4.2. Step 2: Analyse the Counter-Strategy

The counter-strategy $\mathcal{C}$ describes, possibly as a Mealy machine, how the environment can react to any system move in order to keep some system liveness goal from ever being met [6, 8]. We traverse this structure to locate *environment-controlled cycles*: runs in which the environment can remain indefinitely, while a specific system-progress proposition (for example, "door closed" or "service completed") never becomes true. Each such cycle witnesses a missing fairness condition: it shows that the environment is allowed to hold a certain input forever or to repeat a certain pattern forever, and that this is exactly what prevents realization [4, 6, 8].

4.3. Step 3: Generalize to Assumptions

Because these cycles are phrased in terms of a particular run of the game, we generalize them into small temporal formulas that capture the essential shape of the problem. The most common case is a *release* pattern: some input (e.g. `ventReq`) is kept high forever, so we propose the assumption $\square\lozenge\neg$`ventReq`, meaning the environment must release it infinitely often. This follows the standard

"add fairness to repair unrealizability" pattern in reactive synthesis [4, 8]. Another case is *alternation*: the environment keeps re-triggering the same request, starving another chart; here we propose obligations of the form "after a request, it is eventually cleared," which is a bounded-response or progress-style assumption [7, 9]. Collecting these formulas over all problematic cycles yields a candidate assumption set A.

4.4. Step 4: Render as Environment LSCs

To keep everything in the scenario world, we render each temporal formula back into a small LSC that lives on the environment lifeline [1]. Such a chart typically allows the environment to raise a signal (cold part) but also contains a hot condition requiring that the signal is eventually lowered. Since LTL clauses of the form $\square\lozenge p$ can be encoded as repeatedly activatable LSCs whose main chart demands p, this projection is straightforward [1, 9]. The result is a set of LSCs that can be shown to stakeholders and added to S, preserving the original modelling style while enforcing the additional environment commitments inferred from the counter-strategy.

4.5. Algorithm

Algorithm 1 summarizes these steps: translate, check, if unrealizable extract blocking cycles, generalize them to assumptions, and finally render those assumptions as LSCs that can be merged with the original scenario set.

Algorithm 1 Assumption Mining from Unrealizable Scenario Specs

Require: Scenario set S

$(G, \mathit{goals}) \leftarrow \textsc{TranslateToGame}(S)$
$(\mathit{realizable}, \mathcal{C}) \leftarrow \textsc{GR1Check}(G)$
if *realizable* **then**
    return $\emptyset$
$\mathcal{P} \leftarrow \textsc{ExtractEnvCycles}(\mathcal{C}, \mathit{goals})$
$A \leftarrow \emptyset$
for all $p \in \mathcal{P}$ **do**
    $a \leftarrow \textsc{GeneralizeToAssumption}(p)$
    $A \leftarrow A \cup \{a\}$
$A_{\text{LSC}} \leftarrow \textsc{RenderAsLSC}(A)$
return A_{LSC}

5. Running Example: Elevator with Vent

To make the approach concrete, we revisit a classic pedagogical scenario-based example: a small elevator controller. The system interacts with (i) an environment that issues floor requests and may press a "ventilation" button, and (ii) the elevator door mechanism. We specify the behaviour using two very natural LSCs.

- LSC_1 (Serve floor): "On floor request, the elevator moves to the requested floor, opens the door, and must *eventually* close it." In LSC terms, the prechart is the occurrence of a floor

request event, and the main chart contains a hot sequence: `moveTo(floor)`; `doorOpen := true`; eventually `doorOpen := false`. The key point is that the final close is *hot*—it must happen.

- LSC_2 (Ventilation): "When the vent button is pressed, keep the door open to allow ventilation." Here the prechart is the input `ventPressed := true`, and the main chart states a hot condition that the door remains open as long as ventilation is requested. Informally: "if the user wants ventilation, don't close the door."

Individually, each chart is reasonable. LSC_1 ensures basic serviceability: every floor request leads to a proper open–close cycle. LSC_2 ensures a safety/comfort feature: ventilation must not be disrupted by door closure. The problem arises when we let the environment be fully adversarial, as reactive synthesis requires. Nothing in the two LSCs says that the environment *must* ever release the ventilation button. Thus, in the combined scenario set $S = \{\mathrm{LSC}_1, \mathrm{LSC}_2\}$, there is a simple environment strategy: issue a floor request so that LSC_1 becomes active, and then press the ventilation button and keep it pressed forever. Because LSC_2 is now active and hotly requires the door to stay open, the system can never complete the "eventually close door" obligation of LSC_1. When we translate S to a GR(1)-like game and run realizability, the tool detects exactly this and reports that the system cannot win.

From the perspective of our method, this is an ideal case. The counter-strategy $\mathcal{C}$ returned by the GR(1) engine is extremely simple: in every environment move where `ventPressed` could be released, the environment instead chooses to keep `ventPressed` = true. If we trace the run, we see an *environment-controlled cycle* in which the system waits to close the door, but the environment never gives the opportunity. Our cycle analysis therefore extracts the following missing fairness statement:

> "If the vent is pressed, it cannot be held forever; it must sometimes be released."

We generalize this into the LTL-style assumption

$$\square\lozenge(\neg\texttt{ventPressed}),$$

which says that at infinitely many points along the execution, the ventilation button is *not* pressed. This is a standard and intuitively acceptable fairness requirement: real users do not keep the vent button down forever.

To feed this back to the scenario modeller, we render the assumption as a small environment LSC. Its prechart is "vent is pressed," and its main chart contains a hot condition that, eventually, "vent is not pressed." Graphically, such a chart can be drawn on the environment lifeline with a loop: the environment may press vent, but cannot stay in that state indefinitely. Because it is just another LSC, it can be added to S without changing the notation or tooling.

After adding this single assumption chart, we re-run the translation and realizability check on $S \cup \{\text{Vent-Release-LSC}\}$. This time the game is realizable: since the environment is no longer allowed to keep `ventPressed` true forever, there will eventually be a step in which the system is free to carry out the door-closing part of LSC_1. The synthesis engine can therefore construct a winning strategy for the system that reacts correctly both to ordinary floor requests and to temporary ventilation periods.

While this paper focuses on the method, it is useful to outline how one would evaluate it experimentally. First, we would collect a small set of tutorial LSC models that are known to produce

unrealizability under an adversarial environment: elevator-with-vent, train-gate where the sensor can stop reporting, or telephony scenarios where the caller never hangs up. Second, we would generate medium-sized synthetic LSC sets (5–10 charts) that intentionally leave some environment inputs unconstrained. On each benchmark we would run: (i) baseline realizability (expected to fail), (ii) our assumption mining, and (iii) realizability on the repaired set. The main metrics would be: the number of mined assumptions; the syntactic size of each assumption in both LTL and LSC form; whether the repaired set is realizable; and whether any of the assumptions is unnecessarily strong (checked by dropping them one by one). We expect that for the blocking patterns we target—inputs held high forever—very few, very small assumptions (often just one) are sufficient to restore realizability, confirming that the approach produces concise, stakeholder-friendly repairs.

6. Discussion

The proposed assumption-mining procedure should be seen as a bridge between two worlds that, in practice, are often kept separate: on the one hand, the world of reactive synthesis and GR(1) games, where unrealizability is a well-understood technical notion and counter-strategies can be computed automatically [3, 5, 10, 11]; on the other hand, the world of scenario-based requirements engineering, where stakeholders express behaviour in LSCs and expect feedback in the same language [1, 12]. Our results indicate that the counter-strategy produced in the synthesis world actually contains enough semantic information to produce meaningful feedback in the scenario world. This is significant because unrealizability is notoriously hard to explain to non-formal audiences: telling an analyst "the environment can starve your liveness" is much less helpful than telling them "you must ensure the vent button is eventually released," especially when the latter is phrased as an LSC they can read [4, 13].

A central issue in this bridge is the *strength* of the mined assumptions. If we take the environment counter-strategy at face value and negate it wholesale, we might end up with an assumption that says, effectively, "the environment shall not behave in the only way that made the spec unrealizable." While correct, such an assumption may ban more behaviours than necessary. Our generalization step therefore aims to capture the essence of the bad behaviour, not its exact path. For simple starvation patterns, the essence is almost always a fairness condition of the form "do not hold this input forever" or "after triggering this request, eventually clear it" [4, 8, 10]. These are assumptions that domain experts often implicitly have in mind but forgot to write down. In more complex counter-strategies—e.g. those exploiting alternation or repeatedly activating two charts that postpone each other—we may derive assumptions that mention several environment signals together. In practice, there is a design choice here: remain close to the counter-strategy and risk over-specification, or weaken the assumption and risk failing to restore realizability. Our framework supports either choice as a post-processing step [11, 14].

Another point worth discussing is human *acceptability*. Reactive synthesis treats the environment as adversarial because it must, not because real users behave that way [10]. Real users are often cooperative or at least benign: they press buttons for a while, then release them; they do not usually try to deadlock the system. This gap is precisely what produces unrealizability when we go from scenarios to games. By rendering our assumptions back as LSCs, we enable a negotiation step: the requirements engineer can show the mined environment chart to the domain expert and ask "Is it realistic to require that the vent is eventually released?" If the answer is yes, the assumption can be

committed to and the specification is repaired. If the answer is no—because, for example, a medical device must tolerate a sensor being stuck forever—then the engineer has learned that the problem is not missing fairness but rather a true conflict in the system-level scenarios, and another kind of repair is needed [12, 15]. Either way, the explanation is given in the modeller's own notation.

It is also important to note that our approach is *diagnostic*, not just constructive. Even when the mined assumptions are ultimately rejected, they tell us exactly where the specification relies on environment cooperation. This is valuable because LSC collections can become large and intertwined; manually spotting the root of unrealizability in a network of 10 interacting charts is non-trivial. The mined assumption points to a particular input and a particular situation (chart active, signal held) that is problematic. In that sense, assumption mining can also be used as a debugging tool for scenario models: every mined assumption is a hint of "here the environment can hurt you" [9, 13, 16].

There are, however, clear limitations. Not all unrealizability stems from missing environment fairness. Sometimes two hot obligations from two different charts are simply incompatible—one requires the door to be open, the other requires it to be closed, and neither is guarded by any environmental condition. In those cases, the counter-strategy may suggest assumptions that are unnatural, such as "the environment never triggers chart A" or "the environment never sets condition X that activates a conflicting chart." These may restore realizability mechanically, but they do so by disabling parts of the original requirement, which is often not what stakeholders want [7, 14, 17]. This is a signal that the specification itself must be refactored—e.g. by adding priorities between charts, by relaxing one of the hot requirements to cold, or by introducing explicit conflict-resolution charts. Our method cannot resolve genuine semantic contradictions; it can only expose them.

Scalability is another dimension. The running example is intentionally small, and in such examples the counter-strategy is easy to read and the corresponding assumption is short. In larger LSC sets, especially those with many concurrently active charts, the environment counter-strategy can become more entangled: it may rely on cycling through several states, exploiting different inputs at different times to keep different charts from progressing. When we extract environment-controlled cycles in such settings, we may obtain several candidate assumptions. Some of these may overlap or be implied by others. A practical implementation should therefore include a minimization step (e.g. SAT-based implication checking or GR(1)-level implication checking) to drop redundant assumptions and present a clean, small set to the user [11, 16, 18]. Even then, our expectation—supported by the structure of typical unrealizable traces in GR(1)—is that many real cases will be covered by 1–3 fairness-like assumptions.

From a methodological perspective, one could also integrate assumption mining into an *interactive* loop. Instead of generating all assumptions at once, the tool could propose the single weakest assumption that repairs the counter-strategy at hand, re-check realizability, and, if the specification is still unrealizable, repeat the process on the new, smaller counter-strategy. This produces a sequence of increasingly constrained environment behaviours, each of which can be validated with the domain expert. Such a loop would mirror existing "counterexample-guided abstraction refinement" (CEGAR) workflows, but for the problem of environment-specification refinement in LSCs [18, 19].

Finally, our discussion would be incomplete without mentioning extensions. We have deliberately worked in an untimed, non-symbolic GR(1) setting because that is where off-the-shelf realizability checkers are available. In practice, scenario models may include time bounds ("keep the door open for at least 10 seconds") or parametric instances ("any floor button"). In timed settings, the environment can violate system liveness by not only holding a variable forever, but also by *delaying* an action

beyond an admissible window. The same mining idea still applies—look at the environment part of the counterexample and ask what time or parameter commitment would remove it—but the projection back to LSCs must then use the timed extensions of LSCs [15, 19]. Exploring this systematically is a natural piece of future work.

7. Conclusion

We presented a method for repairing unrealizable scenario-based specifications by mining environment assumptions from the environment's counter-strategy. The key insight is that unrealizability in scenario-based synthesis is often due to missing fairness or release commitments on environment-controlled inputs. By analysing the counter-strategy and generalizing recurring environment behaviours to small temporal clauses, then re-expressing them as LSCs, we can offer requirements engineers actionable feedback in their own language. Future work includes integrating assumption-strength control, ranking assumptions by plausibility, and extending the mining process to timed or symbolic scenarios.

References

[1] Harel, D., & Marelly, R. (2003). *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine* (Vol. 1). Heidelberg: Springer.

[2] Piterman, N., Pnueli, A., & Sa'ar, Y. (2006, January). Synthesis of reactive (1) designs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation* (pp. 364-380). Berlin, Heidelberg: Springer Berlin Heidelberg.

[3] Chatterjee, K., Henzinger, T. A., & Jobstmann, B. (2008, August). Environment assumptions for synthesis. In *International Conference on Concurrency Theory* (pp. 147-161). Berlin, Heidelberg: Springer Berlin Heidelberg.

[4] Uchitel, S., Kramer, J., & Magee, J. (2003). Behaviour model elaboration using partial labelled transition systems. *ACM SIGSOFT Software Engineering Notes, 28*(5), 19-27.

[5] Piterman, N., Pnueli, A., & Sa'ar, Y. (2006, January). Synthesis of reactive (1) designs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation* (pp. 364-380). Berlin, Heidelberg: Springer Berlin Heidelberg.

[6] Jobstmann, B., & Bloem, R. (2006, November). Optimizations for LTL synthesis. In *2006 Formal Methods in Computer Aided Design* (pp. 117-124). IEEE.

[7] Damm, W., & Harel, D. (2001). LSCs: Breathing life into message sequence charts. *Formal methods in system design, 19*(1), 45-80.

[8] Clarke, E., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2000, July). Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification* (pp. 154-169). Berlin, Heidelberg: Springer Berlin Heidelberg.

[9] Harel, D., Marron, A., & Weiss, G. (2012). Behavioral programming. *Communications of the ACM, 55*(7), 90-100.

[10] Pnueli, A., & Rosner, R. (1989, January). On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 179-190).

[11] Albarghouthi, A., D'Antoni, L., & Drews, S. (2017, July). Repairing decision-making programs under uncertainty. In *International Conference on Computer Aided Verification* (pp. 181-200). Cham: Springer International Publishing.

[12] Liu, Z. L., Zhang, Z., & Chen, Y. (2012). A scenario-based approach for requirements management in engineering design. *Concurrent Engineering, 20*(2), 99-109.

[13] Ramezani, E., Fahland, D., & van der Aalst, W. M. (2013, August). Supporting domain experts to select and configure precise compliance rules. In *International Conference on Business Process Management* (pp. 498-512). Cham: Springer International Publishing.

[14] Kramer, J., & Magee, J. (1990). The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering, 16*(11), 1293-1306.

[15] Alur, R., & Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science, 126*(2), 183-235.

[16] Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model Checking the Mit Press*. Cambridge, Massachusetts, London, UK, 988.

[17] Ben-Abdallah, H., & Leue, S. (1997, April). Syntactic detection of process divergence and non-local choice in message sequence charts. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 259-274). Berlin, Heidelberg: Springer Berlin Heidelberg.

[18] Henzinger, T. A., Jhala, R., Majumdar, R., & McMillan, K. L. (2004, January). Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 232-244).

[19] Donzé, A., & Maler, O. (2010, September). Robust satisfaction of temporal logic over real-valued signals. In *International conference on formal modeling and analysis of timed systems* (pp. 92-106). Berlin, Heidelberg: Springer Berlin Heidelberg.

How to cite this article: Naveed Ahmad (2022). From Unrealizable Scenario-Based Specifications to Environment Assumptions. *Bulletin of Computer and Data Sciences*, 3(2), 1-12. DOI: 10.71448/bcds2232-1

Received: 19/04/2022 **Revised:** 13/08/2022 **Accepted:** 25/09/2022 **Publish:** 30/12/2022