Anti-Patterns (Smells) in Temporal Specifications

Dor Ma’ayan  
Tel Aviv University  
Israel

Shahar Maoz  
Tel Aviv University  
Israel

Jan Oliver Ringert  
Bauhaus University Weimar  
Germany

Abstract—Temporal specifications are essential inputs for verification and synthesis. Despite their importance, temporal specifications are challenging to write, which might limit their use by software engineers. To this day, almost no quality attributes of temporal specifications have been defined and investigated. Our work takes a first step toward exploring and improving the quality of temporal specifications by proposing a preliminary catalog of anti-patterns (a.k.a. smells). We base the catalog on our experience in developing and teaching temporal specifications for verification and synthesis. In addition, we examined publicly available specification repositories and relevant literature. Finally, we outline our future plans for a better understanding of what constitutes high-quality temporal specifications and the development of tools that will help engineers write them.

I. INTRODUCTION

Temporal specifications are essential inputs for verification and synthesis. They serve as the primary medium to formally describe the expected behavior of a system. In practice, however, developing and using specifications is considered a challenging task reserved only for experts. One potential reason for the very limited adoption of temporal specifications outside academia and industry niches is the lack of knowledge on how to write high-quality specifications, the lack of tools that help in writing high-quality specifications, and fundamentally, to start with, an answer to the question: what constitutes a high-quality temporal specification?

One may consider two perspectives on the quality of specifications: (1) External quality, i.e., to what extent does a specification correctly or completely express the system’s requirements or the engineer’s intent? and (2) Internal quality of a specification that views the specification as a stand-alone document and looks for means to measure its readability, error-proneness, and maintainability. While the first perspective has been investigated, to some extent, in works that deal with the translation of requirements into formal specifications, e.g. [4], [20], [35], to our knowledge, no works have investigated the internal quality of temporal specifications. This is the context of our present and planned work.

Anti-patterns, a.k.a. smells, are issues that impair software quality. They have been extensively studied in code [17], [49], [51], as well as in tests [18], [50], in UML design [34], in continuous integration [12], [52], and in the context of energy consumption of Android applications [22], to give just a few examples from the relevant literature, see, e.g., a survey by Sharma and Spinellis [45].

In this work, we present a preliminary catalog of 9 anti-patterns in temporal specifications. We base the catalog on our experience in developing and teaching temporal specifications for use in verification and synthesis, examining publicly available specification repositories, and reading relevant literature. We believe a catalog of anti-patterns will serve as a starting point for the quality assessment of temporal specifications. Eventually, we expect this work to make specification developers aware of quality issues in their temporal specifications and lead to the development of better specification languages and supporting tools. Following the presentation of the anti-patterns catalog, we discuss our future research plan to fully evaluate and shape the catalog, develop automatic detection tools for the anti-patterns, and empirically explore the relationship between the anti-patterns and other potential quality measurements.

This work is a step in our broader vision toward better understanding of what constitutes high-quality temporal specifications and the development of tools that will help engineers write them.

II. BACKGROUND AND RELATED WORKS

A. Temporal Specifications

Temporal specifications use temporal logics [39] such as LTL and CTL, and fragments and variants of these, to express the expected behavior of systems. They are used as inputs for model checking, synthesis, test generation, and runtime verification tools, see e.g., [9], [30], [43], [47], [48].

We chose to focus our work on temporal specifications, as they are common to many synthesis and verification tools. In the future, one may consider looking for anti-patterns and quality attributes in other kinds of specifications, e.g., relational specifications, such as Alloy [24]. To date, most work on Alloy specifications has dealt with their satisfiability and repair, see, e.g., [8], not with internal quality aspects such as readability and maintainability that are related to the way they are written.

B. Research on Anti-Patterns

Code anti-patterns are quality issues in source code that can be refactored to improve the overall quality of the code [17]. Since their original introduction, anti-patterns have been extended and adapted as an indicator of deeper design problems affecting software quality in many subdomains of software systems [45]. These subdomains include tests [18], [50], UML design [34], and continuous integration [12], [52].
Past studies identified and explored some factors that influence the occurrence of code anti-patterns [45]. These causes include, for example, lack of skill or awareness, language constraints, and knowledge gaps.

The literature on anti-patterns includes works that define a set of anti-patterns [17], [50], works that discuss methodologies to detect and refactor (i.e., fix) anti-patterns [37], [38], [41], works that explore the relationship between anti-patterns and other quality measurements [6], [29], [42], [46], and works that explore the perception of the concept of anti-patterns by programmers [18], [51]. To the best of our knowledge, despite the extensive work on anti-patterns in software and the beneficial outcomes such works brought to the overall software development cycle, no work explored possible anti-patterns in temporal specifications.

C. Comprehension of Temporal Specifications

There are few works on the difficulties in understanding temporal specifications and on misconceptions involved in using them. Greenman et al. [19] identified a set of misconceptions of LTL users. Other works [10], [11] empirically checked the comprehension of temporal specifications, in particular w.r.t. the use of patterns [13]. These works, however, do not propose or discuss concrete anti-patterns. Moreover, in our present work, we look for anti-patterns whose impact is not limited to the comprehension of temporal specifications but also consider maintainability and error-proneness.

III. PRELIMINARY CATALOG OF ANTI-PATTERNS

The following preliminary catalog of anti-patterns is based on our experience developing and teaching temporal specifications, including reviewing many temporal specifications written by students and experts. In particular, to develop the following catalog of anti-patterns, we first studied relevant literature on temporal specification language constructs and analyses. We then inspected specifications in corpora such as the CRV [5] and MCC [27] competitions, many NuSMV [9] and Spin [21] specifications available on GitHub, the SYNTCOMP benchmark [25], and the SYNTech collection [30]. Finally, some of the suggested anti-patterns are inspired by relevant literature on well-known anti-patterns in code.

We classify the anti-patterns in the catalog in two dimensions: the scope in which the anti-pattern is defined and the nature of the impact it may have. Tbl. I presents the definitions. For each anti-pattern, we explain the rationale behind it, define it, and describe its scope and impact. We present the anti-patterns in no particular order.

A. Overusing Specification Patterns

Scope: ☰ Impact: ☠

Rationale. Specification patterns such as the Dwyer et al. patterns [13] and the Menghi et al. robotic mission patterns [33] were designed to hide the complexity of temporal specifications. However, overusing patterns may create specifications that are (1) too convoluted and (2) hide too much information from the developer, which may make specification maintenance and debugging harder and lead to unexpected behaviors. For example, consider the following instance of the response pattern to express that ready holds infinitely often:

Instead of using a specification pattern this could have been expressed by a shorter, equivalent LTL formula:

\[
\mathbf{GF}(\text{ready})
\]  

Definition. Using temporal specification patterns in places where they could be replaced with a simple temporal logic formula.

B. Misusing Specification Patterns

Scope: ☰ Impact: ☡

Rationale. While specification patterns such as the Dwyer et al. patterns [13] were shown to be easier to understand compared to pure LTL [11], studies show that pattern-based properties are still hard to understand by novice developers [10]. For example, in one of our classes, where students used Spectra [30] to write specifications for synthesis, we encountered participants using the following pattern instance:

At first sight, it seems like the developer wanted to state that the robot should visit the home point between any two visits to other locations. However, since the scopes of the patterns are closed to the left and open to the right, and she passed the same argument twice, this statement is trivially true, likely not expressing the desired meaning.

Definition. Using temporal specification patterns in ways not intended by their original developers.
C. Failure to Use Past Temporal Operators When Appropriate

**Scope:** 🌋 **Impact:** ⚠️ ⚠️ ⚠️

**Rationale.** Past temporal operators (PastLTL) are supported in several model-checking, synthesis, and runtime verification tools, and have been proven to be more succinct compared to standard future temporal operators in many cases [32]. For example, while the following expression of a property using the standard future temporal logic operators might be confusing and, therefore, error-prone

\[ \neg((\neg \text{request}) \ U (grant \land \neg \text{request})) \]  

(2)

describing the same property with past temporal operators leads to a shorter, more elegant formula:

\[ G(grant \implies F^{-1} \text{request}) \]  

(3)

Therefore, not using past temporal operators when appropriate may lead to confusing formulas, which are error-prone and harder to maintain. One reason to failing to use past operators may be engineers’ unawareness of these language constructs or lack of knowledge about their precise semantics.

**Definition.** Using only future temporal operators when using past temporal operators will lead to a shorter and more intuitive specification.

D. Boolean Formulas Misuse

**Scope:** 🌋 **Impact:** ⚠️ ⚠️ ⚠️

**Rationale.** Boolean expressions play a significant role in intuitive specification. There is evidence in the literature that Boolean expressions written in complex form (for example, using double negation) [1] are harder to understand and maintain. For example, the following property from a NuSMV [9] specification available on GitHub could be simplified by De Morgan’s laws to

\[ \begin{align*}
LTLSPEC & \ G \ (\neg (\text{northLight} \land \text{eastLight})) \\
\end{align*} \]

Complex logical Boolean expressions may also be more error-prone due to mistakes made by their developers.

**Definition.** Writing Boolean expressions in a nontrivial form, for example, using double negation.

E. Clones: Duplication of Expressions

**Scope:** 🌋 **Impact:** ⚠️ ⚠️ ⚠️

**Rationale.** Fowler [17] suggested that code duplication, or cloning, is one of the major indicators of poor code maintainability. Much research has empirically evaluated the relation between clones and code quality and examined means to detect them, e.g., [26], [40], [44]. Similarly, duplicated temporal expressions may also appear in specifications. For example, consider the following NuSMV specification, which describes the behavior of the inner and outer doors of a ship, which share many identical behaviors.

```plaintext
-- If the door opens (transitions from closed to open), the button must restart
LTLSPEC G ( Y (inner_door.status = closed) \& inner_door.status = open -> airlock. reset_inner )
LTLSPEC G ( Y (outer_door.status = closed) \& outer_door.status = open -> airlock. reset_outer )
```

Such duplication may make the specification harder to maintain in case of changes, and also harder to comprehend. Clones may occur due to copy-paste action of specification fragments and to unawareness or misuse of language constructs that allow the reuse of expressions. Therefore, if the specification language has a reuse mechanism that allows one to avoid such duplication, such as defines (available, e.g., in NuSMV and in Spectra) and parametric predicates (available, e.g., in Spectra), it is better to use it than to write many clones of the same expression. We have seen many such clones in many specifications in the SYNTech [30] benchmarks and in many NuSMV specifications available on GitHub.

**Definition.** Temporal expressions or sub-expressions that appear multiple times in the specification.

F. Local Inherent Vacuity

**Scope:** 🌋 **Impact:** ⚠️ ⚠️ ⚠️

**Rationale.** Some expressions in the specification may be trivially true or false [16], [31]. Such expressions may make the specification unnecessary long and harder to understand. They also hint at potential errors in the specification or the understanding of the domain.

**Definition.** Expressions that are trivially true or false, regardless of the actual values of their variables.

G. Global Inherent Vacuity

**Scope:** 🌋 **Impact:** ⚠️ ⚠️ ⚠️

**Rationale.** Parts of the specification may not affect its semantics [16], [31], e.g., if they are logically implied by other properties in the specification. For example, consider the following simple inherent vacuity:

```plaintext
LTLSPEC GF(safe)  
LTLSPEC G(safe)
```

Here, the first statement is redundant since it is implied by the second statement. Past studies have shown that such inherent vacuities are very common in specifications written by students and experts alike [31]. These redundant properties in the specification may make it harder to understand and maintain. Moreover, they may affect the performance of various analyses.

One potential reason for the existence of inherent vacuities in specifications is the overlapping semantics

1https://github.com/Ackuq/dd2460-nusmv-advanced/blob/b04e483311145c499c4e299c9ce9770b8d497069b/shp3.3.smv#L268
between some requirements in the specification. Moreover, inherent vacuities are difficult and computationally expensive to detect, and other than Spectra [31], there are no tools that support their automatic detection.

**Definition.** Fragments of the specification that are implied by the rest of the specification and therefore have no effect on its semantics.

### H. Long Expressions

**Scope:** ☑  **Impact:** ⚖  ⚖  ⚖

**Rationale.** Temporal expressions that are too long may be harder to comprehend and maintain. In such a case, one may consider dividing the expression into several sub-expressions, each representing individual and thus simpler logical units of the more complex property.

Long and complex expressions are frequent, for example, see the following statement taken from SYNTECH15:

```plaintext
G ((spec_state_return=S0 & (!((spec_prevBotMotReturn & ack_bot = MOVE)) | (spec_prevBotMotReturn & ack_bot = SLEEP) & (spec_prevBotMotReturn & ack_bot = MOVE))) & next(spec_state_return=S0) & (spec_state_return=S0 & ((spec_prevBotMotReturn & ack_bot = SLEEP) & (spec_prevBotMotReturn & ack_bot = MOVE) & next(spec_state_return=S1)) & (spec_state_return=S0 & (!spec_prevBotMotReturn & ack_bot = SLEEP) & (spec_prevBotMotReturn & ack_bot = MOVE))) & onlybotmoves -> (botMot = RETURN) & (spec_prevBotMotReturn & ack_bot = SLEEP) & next(spec_state_return=S1) & (spec_state_return=S1 & (!spec_prevBotMotReturn & ack_bot = SLEEP) & onlybotmoves -> (botMot = RETURN) & next(spec_state_return=S1)) & (spec_state_return=S1 & (!spec_prevBotMotReturn & ack_bot = SLEEP) & (spec_prevBotMotReturn & ack_bot = MOVE)) & next(spec_state_return=S3) & (spec_state_return=S2 & next(spec_state_return=S2)) & (spec_state_return=S2) & (spec_state_return=S3 & (!spec_prevBotMotReturn & ack_bot = SLEEP)) & onlybotmoves -> (botMot = RETURN) & (spec_prevBotMotReturn & ack_bot = SLEEP) & next(spec_state_return=S1));
```

This long statement can potentially be split into many shorter statements.

**Definition** Long temporal expressions that can be split into several independent logical units.

### I. Bad Naming

**Scope:** ☑  **Impact:** ☑  ☑

**Rationale.** The use of names and their effect on comprehension has been studied in code, in particular names that are too short, too long, or otherwise miscommunicating or not reflecting the intended function. There is evidence that bad names are correlated with other attributes of poor code quality [3], [7], [15], [28]. Similarly, bad names of variables in specifications may have a negative effect on comprehension and maintenance.

**Definition.** Variable names that are too short, too long, or otherwise not reflecting their correct function.

### IV. Future Plans

We presented a preliminary catalog of anti-patterns in temporal specifications. We expect this catalog to evolve as our knowledge about the quality of temporal specifications grows.

We now describe our future research plans for addressing the goal of high-quality temporal specifications. The plan is inspired by the large body of literature on code quality and works on anti-patterns in different sub-domains of software engineering.

Note that in the proposed plans below, we will consider and potentially distinguish different target groups, e.g., formal methods experts, students, or representative software engineers. While all are target users of temporal specifications, they may have different needs, characteristics of use patterns, and different comprehension cognitive processes.

**A Qualitative Study with Experts.** To evaluate the validity of the anti-patterns catalog and enhance and refine it, we plan to conduct a qualitative interview study in which we will introduce experts to our preliminary catalog of anti-patterns and ask for their feedback. Based on their feedback and suggestions, we plan to refine the catalog.

**Specification Corpus** To explore the quality of temporal specifications, it is essential to have a large and representative collection of specifications. Unlike studies on code, which benefit from the millions of online repositories, temporal specifications are not readily available. Moreover, those that are available, e.g., in collections such as MCC [27], SYNTCOMP [25], and CRV [5], are mostly benchmarks that experts created for the purpose of tool’s performance evaluation, not for the purpose of writing high-quality specifications. We plan to invest efforts in curating a set that will serve as a baseline for future.
studies on the quality of temporal specifications. Tbl. II lists some potential sources for such a corpus.

**Automatic Detection of Anti-Patterns** Using the corpus we will create, we plan to develop automatic methods to detect anti-patterns. Automatic detection of anti-patterns will have two use cases: (1) Providing specification developers with hints about the quality of their specifications so they can consider fixing them, and (2) Extracting statistics about the occurrences of different anti-patterns in real-world specifications to assess their frequency and overall impact. The automatic detection may be packaged in a tool like SpotBugs\(^2\) or other code smell detection tools [36], targeting temporal specifications rather than code. Some anti-patterns may be specific for particular use cases of temporal specifications (for example, anti-patterns that are relevant to synthesis but not to model checking or runtime verification).

**Empirical Studies** With the ability to detect anti-patterns automatically, we plan to conduct a series of empirical studies to examine the influence of the anti-patterns on different aspects (e.g., comprehension, maintainability), the relationship between the occurrences of different anti-patterns, the question of whether results on code anti-patterns transfer to temporal specification anti-patterns, and the question of the relationship between anti-patterns and the correctness of specifications. This step will evaluate the catalog and allow us to refine it.

**Refactoring** After gaining more insights into the nature of different anti-patterns and refining the catalog, we plan to develop refactoring techniques to deal with the different anti-patterns, for example, simplify temporal expression (to remove long expressions), extract common expression (to remove duplicates), etc. Similarly to the case of code refactoring, some temporal anti-patterns may be hard to automatically or semi-automatically refactor.

**Specification Language Evaluation and Future Design** The different anti-patterns can be used by us and others to evaluate existing temporal specification languages, e.g., PSL [14], [23] or ForSpec [2]. These languages were created with the intention to be more suitable for practitioners, but we are not aware of studies that actually examined how well they meet this goal. The anti-patterns may guide future language design. We specifically plan to use the anti-patterns in future tool and language development design decisions of Spectra [30], e.g., to decide whether and how to add support for various language constructs if some are prone to misuse.

** Specification Languages Documentation and Teaching** Including examples of anti-patterns and means to avoid them in the documentation of specification languages may help users write higher quality specifications. Similarly, anti-patterns may be used as a pedagogical tool for effective teaching of specification languages.

---

\(^{2}\)https://spotbugs.github.io/

---

**ACKNOWLEDGEMENTS**

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon Europe research and innovation programme (grant No 101069165, SYNTACT).

**REFERENCES**


