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].

DOI 10.1109/ICSE-NIER58687.2023.00009

TABLE I: Classification of Anti-Patterns in Two Dimensions: Scope and Impact

Scope

Icon	Name	Description
	Single property	Defined in a single temporal property
	Entire specification	Defined in the entire specification

Impact

Icon	Name	Description
	Comprehension	Lead to a less readable specification
	Maintainability	Lead to specifications that are harder to change and maintain
	Error-proneness	May hint at hidden errors in the specification

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:

```
S_responds_to_P_globally(ready, true)
```

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

$$\mathbf{GF}(ready) \tag{1}$$

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:

```
P_becomes_true_between_Q_and_R(atHome,
    atAnyLocation, atAnyLocation)
```

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:**

Rational 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

$$\neg((\neg request)\ \mathbf{U}\ (grant \wedge \neg request)) \qquad (2)$$

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

$$\mathbf{G}(grant \implies \mathbf{F}^{-1}\ request) \qquad (3)$$

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 temporal formulas. 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

```
LTLSPEC G ! (northLight!=off & eastLight!=off)
```

could be simplified by De Morgan's laws to

```
LTLSPEC G (northLight=off | eastLight=off)
```

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[1] NuSMV specification, which describes the behavior of the inner and outer doors of a ship, which share many identical behaviors.

```
-- If the door opens (transitions from closed to
     open), the button must resetart

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:

```
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

[1]https://github.com/Ackuq/dd2460-nusmv-advanced/blob/b04e483311145c96ae4299c9ca9f770c8d497069/ship3.3.smv#L268

TABLE II: Potential Sources for a Corpus of Temporal Specifications

Name	Type	No. Files	Comments
CRV [5]	Runtime Verification	various	different specification formats
MCC [27]	Model Checking	>300,000	1617 model instances with >300,000 formulas (LTL, CTL, etc.)
NuSMV [9] specifications available on GitHub	Model Checking	>8,000	GitHub search: filename=`*.smv` text=`LTLSPEC`
Spin [21] specifications available on GitHub	Model Checking	>2,700	GitHub search: filename=`*.pml` text=`ltl`
SYNTCOMP [25]	Synthesis	>1,000	LTL (TLSF) specifications written by experts
SYNTECH [30]	Synthesis	>320	Spectra [30] specifications written by students

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:

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

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 to analyze 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

[1] S. Ajami, Y. Woodbridge, and D. G. Feitelson. Syntax, predicates, idioms - what really affects code complexity? *Empir. Softw. Eng.*, 24(1):287–328, 2019.

[2] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In J. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–211. Springer, 2002.

[3] V. Arnaoudova, M. D. Penta, and G. Antoniol. Linguistic antipatterns: what they are and how developers perceive them. *Empir. Softw. Eng.*, 21(1):104–158, 2016.

[4] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang. Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. *IEEE Trans. Software Eng.*, 41(7):620–638, 2015.

[5] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zalinescu, and Y. Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. *Int. J. Softw. Tools Technol. Transf.*, 21(1):31–70, 2019.

[6] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. W. Binkley. Are test smells really harmful? An empirical study. *Empir. Softw. Eng.*, 20(4):1052–1094, 2015.

[7] D. W. Binkley, M. Davis, D. J. Lawrie, and C. Morrell. To camelcase or under_score. In *The 17th IEEE Int. Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*, pages 158–167. IEEE Computer Society, 2009.

[8] S. G. Brida, G. Regis, G. Zheng, H. Bagheri, T. Nguyen, N. Aguirre, and M. F. Frias. Bounded Exhaustive Search of Alloy Specification Repairs. In *43rd IEEE/ACM Int. Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1135–1147. IEEE, 2021.

[9] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer Aided Verification, 14th Int. Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proc.*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[10] C. Czepa and U. Zdun. How understandable are pattern-based behavioral constraints for novice software designers? *ACM Trans. Softw. Eng. Methodol.*, 28(2):11:1–11:38, 2019.

[11] C. Czepa and U. Zdun. On the understandability of temporal properties formalized in linear temporal logic, property specification patterns and event processing language. *IEEE Trans. Software Eng.*, 46(1):100–112, 2020.

[12] P. M. Duvall and M. Olson. Continuous delivery: Patterns and antipatterns in the software life cycle. *DZone refcard*, 145, 2011.

[13] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of the 1999 Int. Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, pages 411–420. ACM, 1999.

[14] C. Eisner and D. Fisman. *A Practical Introduction to PSL.* Series on Integrated Circuits and Systems. Springer, 2006.

[15] D. G. Feitelson, A. Mizrahi, N. Noy, A. B. Shabat, O. Eliyahu, and R. Sheffer. How developers choose names. *IEEE Trans. Software Eng.*, 48(2):37–52, 2022.

[16] D. Fisman, O. Kupferman, S. Sheinvald-Faragy, and M. Y. Vardi. A framework for inherent vacuity. In *HVC*, volume 5394 of *LNCS*, pages 7–22. Springer, 2008.

[17] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999.

[18] V. Garousi and B. Küçük. Smells in software test code: A survey of knowledge in industry and academia. *J. Syst. Softw.*, 138:52–81, 2018.

[19] B. Greenman, S. Saarinen, T. Nelson, and S. Krishnamurthi. Little tricky logic: Misconceptions in the understanding of LTL, 2022. Talk at VardiFest: A workshop in honor of Moshe Y. Vardi, part of FLoC'22.

[20] J. He, E. Bartocci, D. Nickovic, H. Isakovic, and R. Grosu. DeepSTL - From English Requirements to Signal Temporal Logic. In *44th IEEE/ACM 44th Int. Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 610–622. ACM, 2022.

[21] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[22] E. Iannone, F. Pecorelli, D. D. Nucci, F. Palomba, and A. D. Lucia. Refactoring Android-specific Energy Smells: A Plugin for Android Studio. In *ICPC '20: 28th Int. Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 451–455. ACM, 2020.

[23] IEEE Standards. IEC 62531:2012(E) (IEEE Std 1850-2010): Standard for Property Specification Language (PSL). *IEC 62531:2012(E) (IEEE Std 1850-2010)*, pages 1–184, June 2012.

[24] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[25] S. Jacobs, N. Basset, R. Bloem, R. Brenguier, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, T. Michaud, G. A. Pérez, J. Raskin, O. Sankur, and L. Tentrup. The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results. In *Proc. Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017*, volume 260 of *EPTCS*, pages 116–143, 2017.

[26] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *31st Int. Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proc.*, pages 485–495. IEEE, 2009.

[27] F. Kordon, P. Bouvier, H. Garavel, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, D. Donatelli, S. Dal Zilio, P. Jensen, L. Jezequel, C. He, S. Li, E. Paviot-Adet, J. Srba, and Y. Thierry-Mieg. Complete Results for the 2022 Edition of the Model Checking Contest. http://mcc.lip6.fr/2022/results.php, June 2022.

[28] D. J. Lawrie, C. Morrell, H. Feild, and D. W. Binkley. What's in a name? A study of identifiers. In *14th Int. Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 3–12. IEEE Computer Society, 2006.

[29] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J. Syst. Softw.*, 80(7):1120–1128, 2007.

[30] S. Maoz and J. O. Ringert. Spectra: a specification language for reactive systems. *Softw. Syst. Model.*, 20(5):1553–1586, 2021.

[31] S. Maoz and R. Shalom. Inherent vacuity for GR(1) specifications. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 99–110. ACM, 2020.

[32] N. Markey. Temporal logic with past is exponentially more succinct. *Bull. EATCS*, 79:122–128, 2003.

[33] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger. Specification patterns for robotic missions. *IEEE Trans. Software Eng.*, 47(10):2208–2224, 2021.

[34] P. Mohagheghi, V. Dehlen, and T. Neple. Definitions and approaches to model quality in model-based software development - A review of literature. *Inf. Softw. Technol.*, 51(12):1646–1669, 2009.

[35] R. Nelken and N. Francez. Automatic translation of natural language system specifications. In *CAV*, volume 1102 of *LNCS*, pages 360–371. Springer, 1996.

[36] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna. On the evaluation of code smells and detection tools. *J. Softw. Eng. Res. Dev.*, 5:7, 2017.

[37] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman. A textual-based technique for smell detection. In *24th IEEE Int. Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016*, pages 1–10. IEEE Computer Society, 2016.

[38] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba. tsDetect: an open source test smells detection tool. In *ESEC/FSE*, pages 1650–1654. ACM, 2020.

[39] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[40] F. Rahman, C. Bird, and P. T. Devanbu. Clones: what is that smell? *Empir. Softw. Eng.*, 17(4-5):503–530, 2012.

[41] S. Reichhart, T. Gîrba, and S. Ducasse. Rule-based assessment of test quality. *J. Object Technol.*, 6(9):231–251, 2007.

[42] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol. An empirical study of code smells in javascript projects. In *IEEE 24th Int. Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 294–305. IEEE Computer Society, 2017.

[43] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstic, J. M. Lourenço, D. Nickovic, G. J. Pace, J. Rufino, J. Signoles, D. Traytel, and A. Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.*, 54(3):279–335, 2019.

[44] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou. Studying the impact of clones on software defects. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 13–21. IEEE Computer Society, 2010.

[45] T. Sharma and D. Spinellis. A survey on software smells. *J. Syst. Softw.*, 138:158–173, 2018.

[46] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. On the relation of test smells to software code quality. In *2018 IEEE Int. Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 1–12. IEEE Computer Society, 2018.

[47] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: towards flexible verification under fairness. In *Computer Aided Verification, 21st Int. Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proc.*, volume 5643 of *LNCS*, pages 709–714. Springer, 2009.

[48] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *Proc. of the 2004 IEEE Int. Conference on Information Reuse and Integration, IRI - 2004, November 8-10, 2004, Las Vegas Hilton, Las Vegas, NV, USA*, pages 493–498. IEEE Systems, Man, and Cybernetics Society, 2004.

[49] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Software Eng.*, 43(11):1063–1088, 2017.

[50] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok. Refactoring test code. In *XP*, pages 92–95. Citeseer, 2001.

[51] A. F. Yamashita and L. Moonen. Do developers care about code smells? An exploratory survey. In *20th Working Conference on Reverse Engineering, WCRE*, pages 242–251. IEEE Computer Society, 2013.

[52] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. C. Gall, and M. D. Penta. An empirical characterization of bad practices in continuous integration. *Empir. Softw. Eng.*, 25(2):1095–1135, 2020.