Do you smell it too? Towards Bad Smells in IEC 61499 Applications

Lisa Sonnleithner*†, Michael Oberlehner*, Elene Kutsia*†,
Alois Zoitl*†, *Member, IEEE*
†*CDL VaSiCS*
**LIT CPS Lab, Johannes Kepler University Linz*
Linz, Austria
{lisa.sonnleithner, michael.oberlehner, elene.kutsia, alois.zoitl}@jku.at

Sándor Bácsi
Budapest University of Technology and Economics
Budapest, Hungary
bacsi.sandor@aut.bme.hu

***Abstract*—Bad Smells are certain suboptimal structures or patterns in software. They can cause maintenance issues and hinder understandability. Therefore, it is essential to avoid Bad Smells in software. While the topic is well researched in other fields, it is still an open issue in industrial automation. In this work, we are taking a step towards closing that gap and propose a catalog of IEC 61499 Bad Smells.**

***Index Terms*—IEC 61499, distributed control system, CPS, CPPS, design smell, code smell, bad smell, anti-pattern, OCL**

I. Introduction

In modern industrial systems, the share and complexity of software is increasing steadily [1]–[3]. To stay competitive, having high quality software that is easily understood and well maintainable is crucial. So called Bad smells [4] can negatively impact software quality. Bad smells are certain structures [4] or anti-patterns [5] in software that should be avoided. They can hinder understanding, maintaining, or extending a software system [6], [7].

Due to the importance of this topic, many programming languages and tools already support automatic smell detection of varying degree. In programming languages and tools used in the domain of cyber-physical production systems (CPPSs) such support is lagging behind. In this paper, we take a step towards closing that gap. The goal is to present a catalog of Bad Smells for IEC 61499 [8], a standard that describes domain specific modeling language used in industrial automation. To achieve this goal, we will discuss existing work on Bad Smells in general and related to IEC 61499 in Section II. In Section III and IV we will analyze how popular existing smells can be adapted to IEC 61499. Before concluding the paper in Section VI, we will discuss additional smells specific to IEC 61499 and present our Bad Smells Catalog in Section V.

II. Related Work

In this section, we will investigate related work about Bad Smells. We will first take a look at Bad Smells in general and then discuss Bad Smells in domains related to IEC 61499 before discussing related work about Bad Smells in IEC 61499.

978-1-7281-2989-1/21/$31.00 ©2021 IEEE

A. Bad Smells in general

Bad Smells have been studied extensively in the past decades. The term became popular mostly due to the work of Fowler et al. [9] and Fowler and Beck [4]. In [9] the authors proposed a catalog of over 20 Bad Smells and extended it in [4]. Many smells in the literature refer to these or variants of these smells. In their tertiary systematic literature review Lacerda et al. [10] discussed the main observations and challenges of smells and refactoring. They identified the top ten most cited code smells as 10 smells presented by Fowler and Beck [4] and they found that the top 5 design smells are smells introduced by Brown et al. [5]. They also identified several studies that discuss the negative impact of Bad Smells on software quality attributes and their relation to bugs.

B. Bad Smells in languages related to IEC 61499

By mapping concepts of object orientation to block based languages, Hermans et al. [11] propose a catalog of smells for two such block based languages. Most of the smells are based on Bad Smells by Fowler and Beck [4].

As UML is used for modeling structure, behaviour, and interaction there are some similarities to IEC 61499. In their literature review Misbhauddin et. al [12] analyzed refactoring and smells for UML. While they also identified UML-specific smells, the most frequent smells are based on Fowler and Beck [4] and Brown et al. [5].

MATLAB® / Simulink® is a tool that is widely used for the development of model-based software. A catalog of Simulink® smells and how to detect them is proposed in the work of Gerlitz et al. [13].

C. Bad Smells in IEC 61499

For languages used in the domain of CPPS (i.e., IEC 61131 and IEC 61499) research on Bad Smells is scarce. Vogel-Heuser et. al [14] mention code clones as one of the challenges in the software of automated production systems. Code clones or duplicated code is mostly the result of copy-paste-programming or of clone and own reuse, which is still the main technique of deriving new software versions in the domain.

Patil et al. [15] identified several Bad Smells in IEC 61499. They identified smells related to the size of a software part

(e.g., too many states or transitions in an Execution Control Chart (ECC), an Function Block (FB) having too many interface elements) and a smell that can be classified as duplicated code (repeating algorithms and patterns in ECCs).

Hagge and Wagner [16] identify terminal ECC states (i.e., a state that cannot be exited), dead ECC transitions (i.e., transitions that are never evaluated, possibly making the connected state unreachable) and absorbable input events (i.e., events that are always ignored).

D. Mapping Concepts from Object Orientation to IEC 61499

As most work on smells, and especially the work of Fowler and Beck, is based on object oriented languages, we will discuss the mapping of the principles of object orientation to IEC 61499 software components in this section to set a common basis for the following sections.

Cruz and Vogel-Heuser [17] analyze how IEC 61131 [18] and IEC 61499 address object orientation. We will use their work as a basis of the mapping.

Classes and Objects: In IEC 61499, FB types and their instances are the equivalent to classes and objects, respectively.

Methods and Variables: The IEC 61499 equivalent to methods are algorithms. Algorithms can be defined within Basic and Simple FBs. Their execution can be triggered by incoming event signals. They can utilize input data interface elements and internal variables of the FB for processing and change the values of output data elements. These data interface elements and the internal variables are the IEC 61499 equivalent to variables. Events and the data elements they are connected with, can be seen as method signatures. This becomes especially obvious when events and data elements are grouped to an adapter.

Inheritance: In [17] the authors argue, that inheritance can be achieved through IEC 61499 Composite Function Blocks (CFBs). We, however, have to disagree with that. A CFB is composed of the interconnection of other FBs, but just like a class that is using other classes does not inherit from these classes, a CFB does not inherit from its encapsulated FBs, but merely uses them. Therefore, we argue that there is no equivalent to inheritance in IEC 61499.

III. Fowler's Bad Smells

In this section, we will analyze Bad Smells introduced by Fowler and Beck [4]. They introduced almost 30 code structures to avoid. To stay within the scope of this paper, we limit our detailed analysis to their top ten Bad Smells identified in [10].

Duplicated Code: This smell is defined as having the same expression come up in more than 2 places of the software system. It is very often a result of copy-paste-programming. In general, there are four different types of clones, exact clones, similar clones, similar clones with minor syntactical changes and semantic clones. As copy-paste-programming and clone and own are commonly used for reuse in CPPS, this is a ubiquitous Bad Smell in IEC 61499 on all levels.

Long Method: Long methods are often hard to understand and maintain. Fowler and Beck argue that this is not an issue of size, but rather of semantic distance between what is done and how it is done. The IEC 61499 equivalent of this smell is a long algorithm.

Large Class: The Bad Smell *Large Class* refers to a class that is too big. Such a class usually has many responsibilities and contains many methods and variables. According to Fowler and Beck it is *"a prime breeding ground for duplicated code, chaos, and death."* [4]. The IEC 61499 equivalent of this smell is a large FB type.

Long Parameter List: A long parameter list is a Bad Smell as it often leads to confusion. The IEC 61499 equivalent of this smell is an interface of an FB with too many elements.

Divergent Change and Shotgun Surgery: Software should be structured in a way that change is easily possible. Ideally, when a change is necessary there should be a single point in the software to implement that change. Whenever that is not possible, it smells like *Divergent Change* or *Shotgun Surgery*. Fowler and Beck smell *Divergent Change* whenever a single change leads to many changes within one class and *Shotgun Surgery* whenever a single change leads to changes in many different classes. In IEC 61499, it smells of *Divergent Change* when a single change leads to many changes within an FB type and of *Shotgun Surgery* whenever a single change leads to many changes in different FB types.

Feature Envy: Software modularization is about organizing software components in a way that interactions within components are maximized and interactions between components are minimized. Whenever parts of a software component are more interested in another component's part than in parts of their own component, it stinks of *Feature Envy*. In IEC 61499, this smell can be sniffed on various levels. Whenever cohesion within an IEC 61499 software component should be high but is low and coupling to another is high, we smell *Feature Envy*. CFBs and Basic Function Blocks (BFBs) are IEC 61499 software components that should show high cohesion within and low coupling to other blocks, if that is violated it stinks of *Feature Envy*. The distribution of IEC 61499 applications to multiple physical devices is an essential aspect of the language and another example where it could smell of *Feature Envy*.

Data Clumps: It smells of *Data Clumps* whenever the same data elements occur together in different places. The same interface elements of one FB being connected to the same interface elements of another FB are a *Data Clump*.

Lazy Element: This smell can be seen as the opposite of *Long Method* or *Large Class*. It describes a software part that does not provide sufficient functionality, e.g., a method named as its body reads or a class that is nothing more than a simple function. A CFB encapsulating only one FB, or a BFB with just one ECC state would stink intensely of *Lazy Element*.

Refused Bequest: It smells of *Refused Bequest* whenever a subclass does not use the methods and data that it inherits. Fowler and Beck argue that this is a rather faint smell that is often not worth cleaning. As there is no inheritance in IEC 61499, we cannot find an equivalent for this smell.

IV. Brown's AntiPatterns

In this section, we will analyze anti-patterns introduced by Brown et al. [5]. They introduced around 40 patterns to avoid. To stay within the scope of this paper, we limit our detailed analysis to their top five anti-patterns identified in [10].

Blob: A *Blob* is a class, resulting from procedural thinking, that is solely responsible for all the processing, while other classes only encapsulate data. It describes a class with many responsibilities and is therefore similar to Fowler and Becks' *Large Class* smell.

Lava Flow: This anti-pattern is related to the issues arising from the appearance of dead code. Dead code is a term for lines of code that are not executed or whose results are not used after execution. The IEC 61499 equivalents of this smell are *Dead states* and *Dead transitions* in ECC and *Dead FBs* in Function Block Network (FBN).

Functional Decomposition: According to Brown et al., this anti-pattern is the result of procedural coding in object oriented languages. It often results from experienced non-object-oriented programmers switching to an object-oriented language. An alternative name of this anti-pattern is *No Object-Orientation*. In our case, we could encounter such an anti-pattern when an experienced IEC 61131 developer starts developing IEC 61499 software. Investigating how IEC 61131 code can be translated to IEC 61499 would exceed the scope of this paper.

Poltergeist: A *Poltergeist* is a class with a short effective life-cylce and limited roles. Such a class often starts a process for another object. In IEC 61499, the description of *Poltergeist* and also of *Blob* reminds us of the commonly used orchestrator FBs. Such blocks are usually very large, have responsibility over the whole process and start certain functionalities in their subordinate blocks. In previous work [19], [20], we already discussed the advantages and disadvantages of a hierarchical application design using orchestrators and alternative solutions. As it is still an open research question, whether alternative solutions are feasible, we will not include this smell in our catalog, but use this information as valuable feedback for our work on IEC 61499 software architecture and design.

Spaghetti Code: The anti-pattern *Spaghetti Code* describes software with little structure, that lacks clarity and might even be hard to understand for the original developer. Whenever not enough thought is put into the structure of a program one might eventually encounter this anti-pattern. As *Spaghetti Code* describes a general issue and mostly appears in combination with other anti-patterns and Bad Smells (or can be considered as the result of such), we will not include it in our catalog.

Cut-and-Paste Programming: This anti-pattern basically describes the origin of *Duplicated Code*. Brown et al. state that *"Cut-and-Paste Programming is a very common, but degenerate form of software reuse which creates maintenance nightmares."* [5]. Especially in the domain of CPPSs this is still the most common technique of reuse. Therefore, *Duplicated Code* is an especially important smell. As we already described *Duplicated Code* in Section III we will not go into further details.

Swiss Army Knife: A *Swiss Army Knife* is a class with an extremely complex interface. This can occur, when a developer tries to offer interface signatures for every possible use case of the class. The abundance of interface signatures can obscure the intended use of the class and cause problems maintaining or debugging it. For IEC 61499, events with their data elements and adapters can be seen as the equivalent to signatures. Therefore, add this anti-pattern to our catalog as part of the *Large Interface* smell.

V. Specific IEC 61499 Smells

Extending the Bad Smells and anti-patterns we identified as applicable to IEC 61499 in Section III and IV, we have also collected additional smells that are specific to IEC 61499. The smells *Dead state*, *Dead Transition*, *Dead Event* and *Terminal state* are inspired by [16] as discussed in Section II. We adapted the idea of *Dead Transition* and formulated a suitable IEC 61499 definition. Additionally, we identified *Unused event*, *Unused data* and *Mutable data*. In Table I, we summarized all identified smells. The table states the name of the smell, the level where the smell may be sniffed (FBN, FB, ECC, Algorithm (ALG)), followed by a short description and an optional reference if the smell was adapted to IEC 61499 from another smell or was already proposed for IEC 61499. For some of these smells we could already implement detection methods in Eclipse 4diac IDE [21], an IEC 61499 open source development tool. We used the Object Constraint Language (OCL) [22], a part of the UML standard, to implement the detection of smells as constraints on the Eclipse 4diac models. The last column indicates whether detection is already implemented. Due to lack of space, we will detail the OCL constraint only for one IEC 61499-specific smell. A detailed description of all via OCL constraints detectable smells can be found in [23].

Let us consider the following definition for the terminal state: *A terminal state is a state with no outgoing ECC transitions*. In order to detect terminal states, we formulated a rule as an OCL invariant (*inv*) in the context of ECC states (*ECState*), which checks that each ECC state has at least one outgoing ECC transition. If any ECC state violates this constraint, it is reported as an error in the 4diac IDE.

```
context ECState
inv TerminalState: outTransitions->size() >= 1
```

VI. Conclusion

In this work, we presented a catalog of IEC 61499 Bad Smells. After discussing related work, we analyzed the applicability of Bad Smells introduced by Fowler and Beck and anti-patterns described by Brown et al. to IEC 61499. Additionally, we discussed smells that are specific to IEC 61499. Finally, we combined our findings to an IEC 61499 Bad Smells Catalog. This catalog should not only serve as a reference for developers to improve their code quality, but also as a starting point for future research on the topic. Some of the smells are already detectable in Eclipse 4diac IDE, utilizing OCL.

TABLE I: Proposed IEC 61499 Bad Smells Catalog.

Smell Name	Level	Description	Ref.	Impl.
Duplicated Code	ALL	The same or similar code appearing more than once.	[4]	N
Long Algorithm	FB	An algorithm that is too long and complex.	[4]	N
Large Type	FB	An FB type that is too large and complex.	[4]	N
Large Interface	FB	Too many interface elements.	[4]	N
Divergent Change	FB	One change leading to many changes within the same FB type.	[4]	N
Shotgun Surgery	FB	One change leading to changes in many different FB types.	[4]	N
Feature Envy	ALL	An IEC 61499 component having high cohesion to another that should not be coupled tightly.	[4]	N
Data Clumps	FBN	A group of interface elements that always appear together.	[4]	N
Lazy Element	ALL	An IEC 61499 component without purpose (e.g., CFB only containing one FB).	[4]	N
Dead State	ECC	State (except start state) which does not have any input transitions or to which a path cannot be found from the EC initial state by following the directed links.	[5], [16]	Y
Dead Transition	ECC	Transition with lower priority than the 1 transition condition; Transition which has higher priority than the 1 transition condition and does not just have *pure* data-driven condition.		Y
Dead FB	FB	FB (except start FB) which does not have any input event connections.	[5]	Y
Terminal State	ECC	State that is reachable, but which does not have any outgoing EC transitions.	[16]	Y
Unused Event	FB	Event input/output of the FB type containing the ECC that is not used in any EC transitions.		Y
Unused Data	FB	When the particular input event is connected, the associated data input is unconnected or not configured.		Y
Mutable Data	ALG	The algorithm writes on a data input.	[4]	N
Dead Event	FB	An event that is not used in the transition condition of any stable ECC state and is thus always ignored.	[16]	N

In the future, we will not only expand the catalog, but also investigate which smells can be detected and how such a detection could be achieved. Additionally, we plan on combining the proposed Bad Smells Catalog with the refactoring catalog that we introduced in [24] to provide a way of resolving detected Bad Smells. Besides that, we will use the findings related to Brown et al.'s anti-patterns as input for our work on IEC 61499 architecture.

ACKNOWLEDGEMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and the Christian Doppler Research Association is gratefully acknowledged.

REFERENCES

[1] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Trans. on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, 2013.

[2] B. Vogel-Heuser, C. Diedrich, A. Fay, S. Jeschke *et al.*, "Challenges for software engineering in automation," *J. of Software Engineering and Applications*, vol. 07, no. 05, pp. 440–451, 2014.

[3] M. Törngren and P. Grogan, "How to deal with the complexity of future cyber-physical systems?" *Designs*, vol. 2, no. 4, p. 40, 2018.

[4] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[5] W. J. Brown, R. C. Malveau, H. W. McCormick III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. NY: John Wiley & Sons, Inc., 1998.

[6] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *2011 15th European Conf. on Software Maintenance and Reengineering*, 2011, pp. 181–190.

[7] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *2012 28th IEEE Int. Conf. on Software Maintenance (ICSM)*, 2012, pp. 306–315.

[8] IEC TC65/WG6, "IEC 61499-1, function blocks - part 1: architecture: Edition 2.0," Geneva, 2012. [Online]. Available: www.iec.ch

[9] M. Fowler, K. Beck, J. Brant, and W. Opdyke, "Refactoring: Improving the design of existing code," 1999.

[10] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *J. of Systems and Software*, vol. 167, p. 110610, 2020.

[11] F. Hermans, K. T. Stolee, and D. Hoepelman, "Smells in block-based programming languages," in *IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC)*, 2016, pp. 68–72.

[12] M. Misbhauddin and M. Alshayeb, "Uml model refactoring: a systematic literature review," *Empirical Software Engineering*, vol. 20, no. 1, pp. 206–251, 2015.

[13] T. Gerlitz, Q. M. Tran, and C. Dziobek, "Detection and handling of model smells for matlab/simulink models." in *MASE@ MoDELS*, 2015, pp. 13–22.

[14] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *J. of Systems and Software*, vol. 110, pp. 54–84, 2015.

[15] S. Patil, D. Drozdov, G. Zhabelova, and V. Vyatkin, "Refactoring of IEC 61499 function block application — A case study," in *IEEE Industrial Cyber-Physical Systems (ICPS)*. IEEE, 2018, pp. 726–733.

[16] N. Hagge and B. Wagner, "Analyzing the liveliness of IEC 61499 function blocks," in *2008 IEEE Int. Conf. on Emerging Technologies and Factory Automation*, 2008, pp. 377–382.

[17] L. A. Cruz Salazar and B. Vogel-Heuser, "Applying core features of the object-oriented programming paradigm by function blocks based on the iec 61131 and iec 61499 industrial automation norms," in *Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future*. Springer Int. Publishing, 2020, pp. 273–289.

[18] IEC, "IEC 61131 - programmable controllers, part 3: Programming languages: Edition 3.0," Geneva, 2013. [Online]. Available: www.iec.ch

[19] B. Wiesmayr, L. Sonnleithner, and A. Zoitl, "Structuring distributed control applications for adaptability," in *IEEE Conf. on Industrial Cyberphysical Systems (ICPS)*, vol. 1, 2020, pp. 236–241.

[20] L. Sonnleithner, B. Wiesmayr, V. Ashiwal, and A. Zoitl, "IEC 61499 distributed design patterns," in *26th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, 2021, in press.

[21] Eclipse 4diac, "Eclipse 4diac - the open source environment for distributed industrial automation and control systems," 2019. [Online]. Available: https://www.eclipse.org/4diac

[22] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Pub. Co., Inc., 2003.

[23] S. Bácsi, 2020. [Online]. Available: https://wiki.eclipse.org/Eclipse_4diacWiki/Development/Detecting_Model_Inconsistencies_in_4diac_Models_with_OCL

[24] M. Oberlehner, L. Sonnleithner, and A. Zoitl, "Catalog of refactoring operations for IEC 61499," in *26th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, 2021, in press.