Using Software Architecture Descriptions to Detect Architectural Smells at Design Time

Everton Cavalcante, Thais Batista

Federal University of Rio Grande do Norte
Natal, Brazil

everton.cavalcante@ufrn.br, thaisbatista@gmail.com

Abstract. *Architectural smells are decisions made at the software architecture level, whether intentional or not, that may negatively impact the quality of a software system. In the literature, architectural smells are identified mainly by relying on the source code or other implementation artifacts. However, architectural smells could be detected at design time, even before employing implementation efforts and preventing them from being reflected at the system implementation. This research investigates how software architecture descriptions realized through architecture description languages (ADLs) can be used to identify architectural smells at design time. This work focuses on how architectural smells manifest and can be detected in SysADL, an ADL that allows describing both structure and behavior of software architectures using standardized diagrams from the OMG's SysML language.*

1. Introduction

Software architectures are acknowledged as essential assets for the success of software systems, contributing to satisfying both functionalities and quality requirements and incorporating principles and decisions guiding their design and evolution [Bass et al. 2022]. Such architectures can be visualized at a high level in the form of components representing the functional elements of the system, connectors that promote communication between those components, and their behavior.

Even though a software system can work with a poor architecture resulting from inadequate or incorrect decisions, future problems will arise in its operation, maintenance, and evolution. One of these problems, which has been of growing interest in the Software Architecture field, refers to the so-called *architectural smells*. An architectural smell is an architectural-level decision, whether intentional or not, that may negatively impact a software system's quality, despite not representing an error that prevents the system from working [Garcia et al. 2009a]. A typical example of architectural smell is cyclic dependencies among software architecture components, which strongly impact the system's maintainability and harm its quality.

Architectural smells are identified mainly in the advanced stages of the software development life cycle. A recent literature review aimed to identify the existing techniques and tools for detecting architectural smells and their limitations concluded that most of these techniques and tools primarily analyze artifacts such as the source code [Mumtaz et al. 2021]. However, detecting architectural smells from a recovered architecture is undoubtedly problematic because the system is already implemented. This means that some effort has already been employed to implement the system, and additional effort will be required to fix existing architectural smells.

Although many works in the literature use the source code to identify the occurrence of architectural smells, there is evidence that not every architectural smell can be detected from it, and the existence of a problem in the code does not necessarily imply an architectural smell [Macia et al. 2012, Arcelli Fontana et al. 2019]. The available tools to detect architectural smells cannot be used at design time when the system implementation does not exist yet [Azadi et al. 2019, Mumtaz et al. 2021]. Moreover, almost all existing techniques and tools focus only on detecting architectural smells related to dependencies, and many others still need to be covered.

Detecting architectural smells at design time can prevent this problem from being reflected at the system implementation, hampering later development stages and negatively affecting software quality attributes [Stafford and Wolf 2001, Chondamrongkul et al. 2020]. *Architecture descriptions* can play a crucial role at this point as they constitute a first representation of the software system under development. However, the literature on using architectural descriptions to detect architectural smells is relatively scarce. The existing works (see Section 3) have limitations concerning the set of identified architectural smells and the considered architectural viewpoints.

This ongoing research investigates how software architecture descriptions realized through architecture description languages (ADLs) can be used to identify architectural smells at design time when these architectures are being conceived. This work focuses on how architectural smells manifest and can be detected in SysADL [Oquendo et al. 2016]. SysADL is an ADL complying with the ISO/IEC/IEEE 42010 International Standard for architectural descriptions and allows describing both structure and behavior of software architectures using standardized diagrams and elements from the OMG's SysML language. The ultimate result will be implementing a mechanism into the SysADL tool to automatically detect architectural smells from architecture descriptions and assist software architects in mitigating them at design time.

The remainder of this paper is organized as follows. Section 2 briefly introduces some architectural smells. Section 3 discusses related work. Section 4 describes the identification of architectural smells from the perspective of software architecture descriptions in the SysADL architectural language. Section 5 brings final remarks and directions for ongoing and future work.

2. Architectural Smells: A Bird's Eye View

Architectural smells mainly stem from the poor use of software architecture elements such as components, connectors, interfaces, etc., including how they handle the different system concerns and interact with each other. Despite the relevance of addressing architectural smells and the negative consequences on the quality of software systems, the literature still needs a uniform understanding of their characteristics and root causes. Garcia et al. [Garcia et al. 2009a, Garcia et al. 2009b] were the pioneers in defining the first catalog of architectural smells. This collection was revisited and expanded throughout the years [Azadi et al. 2019, Mumtaz et al. 2021].

[Le et al. 2017] present a framework to understand and classify architectural smells. They proposed five different classes of architectural smells and described only the ones that could be detected through architectural recovery. Table 1 lists those categories along with well-known architectural smell instances. It is worth mentioning that

Table 1. Architectural smell categories and some instances.

Categories	Causes	Architectural smell instances
Concern-based smells	Inappropriate or inadequate separation of concerns	*Concern Overload*, *Scattered Parasitic Functionality*, *Functionality Overload*
Dependency-based smells	Inappropriate interconnections and interactions among system elements	*Dependency Cycle*, *Link Overload*
Interface-based smells	Deficiency in defining component's interfaces	*Ambiguous Interface*, *Unused Interface*, *Ununsed Component*, *Sloppy Delegation*, *Lego Syndrome*
Coupling-based smells	Couplings among components	*Duplicate Function*, *Co-change Coupling*
Connector-based smells	Inappropriate definition or use of connectors	*Extraneous Adjacent Connector*, *Connector Envy*

the literature presents a high diversity of names and definitions, even though referring to the same architectural smell [Azadi et al. 2019, Mumtaz et al. 2021].

3. Related Work

The Stafford and Wolf's work is one of the first studies related to the software architecture analysis from architecture descriptions to specifically detect dependencies among architectural elements, which is the most common architectural smell [Stafford and Wolf 2001]. These authors argue that ADLs allow reflecting on software system properties at a high abstraction level, e.g., identifying if a given component affects or is affected by another, thus meaning that there is some dependency between these components.

[Sanchez et al. 2015] proposed an approach to verify the nonexistence of architectural smells from architecture descriptions in Archery, a formal ADL proposed by them. In their approach, the structure of software architectures is represented in Archery. An extension called Archery-Constraints allows formally specifying the nonexistence of architectural smells as constraints to be formally verified.

[Chondamrongkul et al. 2020] proposed and implemented an approach to detect architectural smells at design time from formalized software architecture descriptions. The structural viewpoint is represented through ontologies in the OWL language, and the behavioral viewpoint is represented through specifications in Wright#, a formal ADL. Relationships and rules in the ontology codify possible occurrences of architectural smells in the architecture's structure. Next, formal verification is used to detect architectural smells related to behavioral concerns.

In general, the works related to detecting architectural smells through architecture descriptions are limited to analyzing a small set of smells. They do not also consider standardized notations such as UML, SysML, and its derivations, which are preferred for use by practitioners in industry [Malavolta et al. 2013, Ozkaya 2018]. In the Sanchez et al.'s work, the used ADL does not offer an appropriate distinction between architectural

elements (components and connectors) as typically expected for an ADL, besides focusing only on the structural viewpoint and not providing mechanisms for the automatic detection of architectural smells. The Chondamrongkul et al.'s approach considers both structural and behavioral viewpoints in the identification of the occurrence of architectural smells. However, it uses different notations to describe architecture's structure and behavior, and the verification relies on computationally-limited model-checking.

4. Identifying Architectural Smells in SysADL Architecture Descriptions

SysADL [Oquendo et al. 2016] is an architectural language designed to support multiple-view modeling, cross-view checking, and execution of software architectures. It relies on the diagrammatic notation from the SysML Systems Modeling standard and is aligned with the ISO/IEC/IEEE 42010 International Standard. SysADL encompasses three viewpoints for software architectures, namely *structural*, *behavioral*, and *executable*. The structural viewpoint concerns the structure of the software architecture, specifying its elements (components, ports, connectors) and how they are interconnected. The behavioral viewpoint is related to the behavior of the architecture and its elements through activities, actions, constraints, and protocols. The executable viewpoint copes with implementing the behavioral viewpoint and allows for simulating the architecture behavior at runtime.

The structural viewpoint of SysADL primarily relies on two diagrams, namely (i) the structural *Block Definition Diagram* (BDD) for defining components, connectors, configurations, ports, and types, and (ii) the *Internal Block Definition Diagram* (IBD) to detail a configuration by instantiating the elements defined in the BDD. In the behavioral viewpoint, SysADL uses a *behavioral BDD* for defining actions, activities, and constraints. An *Activity Diagram* (AD) defines an activity in terms of actions. A *Parametric Diagram* allows relating constraints to activities and actions. At last, the executable viewpoint of SysADL allows for the early detection of architectural problems, thus saving valuable time and human resources for the final deliverable implementation.

Almost all categories of architectural smells (see Section 2) can be identified in SysADL architecture descriptions as SysADL comes with typical ADL structural constructs and allows describing the behavior of architectural elements. Architectural recovery techniques used to identify architectural smells fail to address connector-based smells, i.e., those caused by inappropriate definition or use of connectors. Consequently, these techniques cannot differentiate between components and connectors reliably [Le et al. 2017]. However, connector-based smells can be identified in SysADL architecture descriptions since connectors are first-class architectural elements in the language, explicitly distinct from components.

In general, the identification of architectural smells using architecture descriptions consists in first characterizing how a given architectural smell manifests through the constructs of the ADL used to describe the software architecture. This characterization leads to the specification of rules stating the occurrence of an architectural smell in an architecture description. If some automation is envisioned, then a mechanism can be implemented to concretize the previously specified rules and point out if the architecture description matches any of these rules, thereby indicating the occurrence of architectural smells. Once an architectural smell is detected in the architecture description, software architects can be assisted on how to mitigate the smell.

This section describes how architectural smells can show up in software architecture descriptions in SysADL by using an adapted version of the Automated Guided Vehicle (AGV) system [Araújo et al. 2021, Gomaa 2011] as an illustrative example. In the AGV system, a real-time system controls a vehicle to start/stop moving along a track and relies on an arrival sensor to detect when it has arrived at a station. The AGV system interacts with two other existing systems by (i) receiving commands from a Supervisory System and sending vehicle acknowledgments and (ii) sending the vehicle status to an external Display System.

For the sake of space, this section presents only some instances of the architectural smells listed in Table 1[1]. This paper focuses on the steps related to characterizing architectural smells in software architecture descriptions using SysADL as architectural language. The detection tool support and mitigation is an ongoing work.

***Dependency Cycle* (DC).** DC is a dependency-based smell that occurs when two or more components depend on each other in a circular chain [Azadi et al. 2019], causing high coupling among components. A DC can be identified in the structural viewpoint of SysADL through the IBD. In an IBD, a DC appears when two connectors link the components in opposite directions, from an output port to an input port and vice-versa. Figure 1 depicts part of an IBD describing the AGV system as a composite component. There is a DC between instances of the *RobotArm* and *VehicleControl* components as they are linked through instances of the *CommandArm* and *NotificationArm* connectors in a bidirectional flow between their input/output ports.

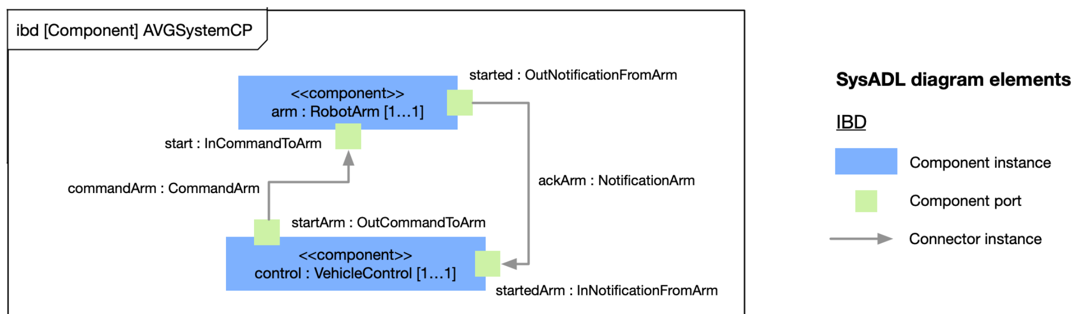

Figure 1. Part of a SysADL Internal Block Diagram with occurrences of *Dependency Cycle* and *Extraneous Adjacent Connector* in the AGV system.

***Extraneous Adjacent Connector* (EAC).** EAC is a connector-based architectural smell that occurs when two or more different connector types are used to link the same pair of components, even though it may be acceptable in certain cases [Garcia et al. 2009b]. An EAC can be identified in the structural viewpoint of SysADL through the IBD when instances of different connector types link the involved components. Figure 1 also presents a manifestation of EAC as the instances of the *RobotArm* and *VehicleControl* components are linked through instances of two different connectors, namely *NotificationArm* and *CommandArm*.

[1]The figures presented in this paper are excerpts of a complete architecture description of the AGV system in SysADL available at https://doi.org/10.5281/zenodo.6546959.

***Ambiguous Interface* (AI).** AI is an interface architectural smell that refers to a single general interface provided by a component that accepts requests and dispatches them internally. An AI can be identified in the structural viewpoint of SysADL through the IBD when a composite component has a proxy input port linked via a binding connector to multiple input ports of different internal components. In SysADL, ports of composite components are always proxy ports, so they need to be delegated to the ports of internal components [Oquendo et al. 2016]. Figure 2 depicts another part of an IBD describing the AGV system as a composite component. The AGV system interacts with the Supervisory System by receiving start/stop moving and loading/unloading commands. These commands are received via *commands*, a single input proxy port of the AGV system linked to two input ports of the *VehicleControl* component through binding connectors. The commands received via the proxy input port are dispatched to those internal input ports according to their nature. If the command is about start/stop moving the AGV, then the delegation is to the *move* input port. If the command is about loading/unloading onto and off the AGV, then the delegation is to the *load* input port.

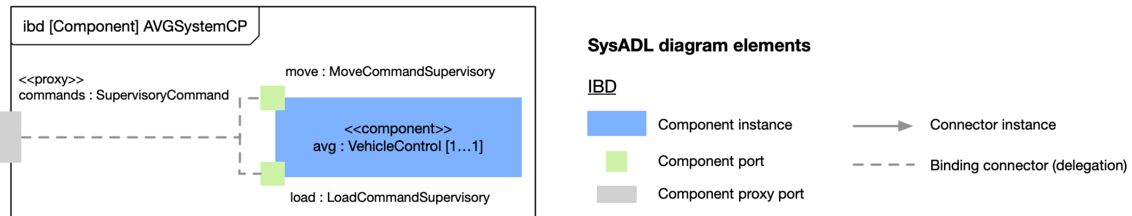

Figure 2. Part of a SysADL Internal Block Diagram with an occurrence of *Ambiguous Interface* in the AGV system.

***Functionality Overload* (FO).** FO is a concern-based smell that indicates that a component accumulates excessive functionalities and control, being an inappropriate form of modularity and violating the principle of separation of concerns. An FO can be identified in the behavioral viewpoint of SysADL through the behavioral BDD. A highly complex BDD with many actions composing an activity may indicate FO. Nevertheless, defining a certain threshold is required to judge a fair number of actions for describing the behavior of an architectural element. It is worth highlighting that existing tools that detect FO relying on architectural recovery adopt thresholds for this purpose [Azadi et al. 2019]. Figure 3 depicts a BDD specifying of the *MoveAC* activity performed by the *VehicleControl* component of the AGV System. This activity is defined as a composition of four actions, namely *ProcessCommandAN*, *SendDestinationAN*, *SendCommandAN*, and *SendStartMotorAN*. Suppose that a customizable threshold establishing the maximum number of actions to compose an activity is set to $ac_{max} = 3$. In this case, the *MoveAC* activity would be considered as possibly suffering from FO as its number of composing actions exceeds the established threshold ac_{max}.

5. Final Remarks

This paper aimed to present ongoing work on investigating how software architecture descriptions realized through ADLs can be used to identify architectural smells at design time before the system implementation. The main thrust behind these ideas is to avoid employing efforts to implement an architecture that may already manifest an architectural smell. This represents a different research direction from what has been done in the state

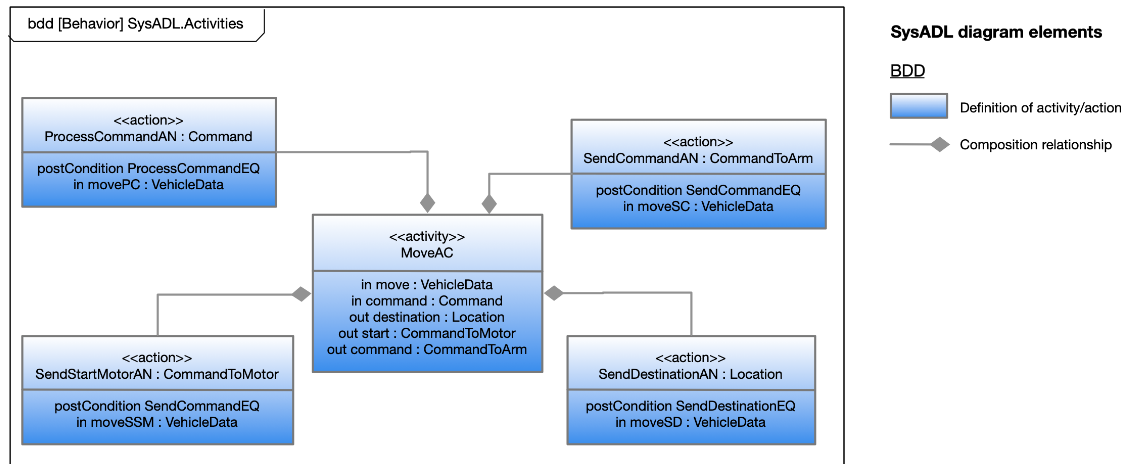

Figure 3. Part of a SysADL Block Definition Diagram with an occurrence of *Functionality Overload* in the AGV system.

of the art since almost all techniques currently used to detect architectural smells rely on architectural recovery or metrics from the source code of the system's implementation. The literature reports that the existing tools to detect architectural smells cannot be used at design time when the system implementation does not even exist. Architectural smells should be detected and eradicated before implementing the system since doing this at later stages through refactoring is costly.

This paper also showed how some architectural smells could be identified in SysADL architectural language. Ongoing work focuses on continuing the specification of rules that state the occurrence of architectural smells in SysADL diagrams. The following stages will be devoted to implementing a mechanism to automatically identify architectural smells in SysADL architecture descriptions. This mechanism will ultimately be integrated into the SysADL Studio tool [Leite et al. 2018], enabling software architects to analyze the occurrence of architectural smells when designing their software architectures. As SysADL complies with the ISO/IEC/IEEE 42010 International Standard and offers well-known, consolidated abstractions to describe software architectures, the same principles presented in this work for identifying architectural smells can be applied with other ADLs relying on the typical architectural abstractions.

References

Araújo, C., Batista, T., Cavalcante, E., and Oquendo, F. (2021). Generating formal software architecture descriptions from semi-formal SysML-based models: A model-driven approach. In Gervasi, O. et al., editors, *Computational Science and Its Applications – ICCSA 2021*, Lecture Notes in Computer Science, vol. 12951, pages 394–410. Springer Nature Switzerland AG, Switzerland.

Arcelli Fontana, F., Lenarduzzi, V., Roveda, R., and Taibi, D. (2019). Are architectural smells independent from code smells? An empirical study. *Journal of Systems and Software*, 154:139–156.

Azadi, U., Arcelli Fontana, F., and Taibi, D. (2019). Architectural smells detected by tools: A catalogue proposal. In *Proceedings of the 2019 IEEE/ACM International Conference on Technical Debt*, pages 88–97, USA. IEEE.

Bass, L., Clements, P., and Kazman, R. (2022). *Software Architecture in practice*. Addison-Wesley/Pearson Education, Inc., USA, 4th edition.

Chondamrongkul, N., Sun, J., Warren, I., and Lee, S. U.-J. (2020). Semantic-based architecture smell analysis. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, pages 109–118, USA. ACM.

Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009a). Identifying architectural bad smells. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, USA. IEEE.

Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009b). Toward a catalogue of architectural bad smells. In Mirandola, R., Gorton, I., and Hofmeister, C., editors, *Architectures for Adaptive Software Systems*, Lecture Notes in Computer Science, vol. 5581, pages 146–162. Springer-Verlag Berlin Heidelberg, Germany.

Gomaa, H. (2011). *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, USA.

Le, D., Link, D., Shahbazian, A., Zhao, Y., Mattmann, C., and Medvidovic, N. (2017). Toward a classification framework for software architectural smells. Technical report, Center for Systems and Software Engineering, University of Southern California, USA.

Leite, J., Batista, T., Oquendo, F., Silva, E., Santos, L., and Cortez, V. (2018). Designing and executing software architecture models using SysADL Studio. In *2018 IEEE International Conference on Software Architecture Companion*, pages 81–84, USA. IEEE.

Macia, I., Arcoverde, R., Garcia, A., Chavez, C., and von Staa, A. (2012). On the relevance of code anomalies for identifying architecture degradation symptoms. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pages 277–286, USA. IEEE.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2013). What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 39(6):869–891.

Mumtaz, H., Singh, P., and Blincoe, K. (2021). A systematic mapping study on architectural smells detection. *Journal of Systems and Software*, 173.

Oquendo, F., Leite, J., and Batista, T. (2016). *Software Architecture in Action: Designing and executing architectural models with SysADL grounded on the OMG SysML Standard*. Springer International Publishing, Swizerland.

Ozkaya, M. (2018). The analysis of architectural languages for the needs of practitioners. *Software: Practice and Experience*, 48(5):985–1018.

Sanchez, A., Barbosa, L. S., and Madeira, A. (2015). Modelling and verifying smell-free architectures with the Archery language. In Canal, C. and Idani, A., editors, *Software Engineering and Formal Methods*, Lecture Notes in Computer Science, vol. 8938, pages 147–163. Springer International Publishing, Switzerland.

Stafford, J. A. and Wolf, A. L. (2001). Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–451.