Detecting Architectural Erosion using Runtime Verification [⋆]

Diego Marmsoler✉ https://orcid.org/0000-0003-2859-7673 and
Ana Petrovska✉ https://orcid.org/0000-0001-6280-2461

Technische Universität München, Germany
diego.marmsoler@tum.de, ana.petrovska@tum.de

Abstract. The architecture of a system captures important design decisions for the system. Over time, changes in a system's implementation may lead to violations of specific design decisions. This problem is common in the industry and known as architectural erosion. Since it may have severe consequences on the quality of a system, research has focused on the development of tools and techniques to address the presented problem. As of today, most of the approaches to detect architectural erosion employ static analysis techniques. While these techniques are well-suited for the analysis of static architectures, they reach their limit when it comes to dynamic architectures. Thus, in this paper, we propose an alternative approach based on runtime verification. To this end, we propose a systematic way to translate a formal specification of architectural constraints to monitors, which can be used to detect violations of these constraints. The approach is implemented in Eclipse/EMF, demonstrated through a running example, and evaluated using two case studies.

Keywords: Architectural Erosion; Architectural Drift; Runtime Verification; FACTum;

1 Introduction

A system's architecture captures major design decisions made about the system, to address its requirements. However, changes in the implementation may sometimes change the original architecture, and some of the design decisions might become invalid over time. This situation, sometimes called architectural erosion [13], may have severe consequences on the quality of a system and is a common widespread problem in industry [11,14]. Thus, research has proposed approaches and tools to detect architectural erosion which, as of today, focus mainly on the analysis of static architectures and therefore employ static analysis techniques [9,10,15,17,31,32].

However, recent trends in computing, such as mobile and ubiquitous computing, require architectures to adapt dynamically: new components may join or leave the network and connections between them may change over time. Thereby, architectural changes can happen at runtime and depend on the state of the components. Thus, analysis of erosion for dynamic architectures requires the analysis of component behavior,

[⋆] This is a post-peer-review, pre-copyedit version of an article to be published in the proceedings of the 12th Interaction and Concurrency Experience, which will appear in Electronic Proceedings in Theoretical Computer Science.

and therefore it is only difficult, not to say impossible, to detect with static analysis techniques.

Consider, for example, the following scenario: An architect, to satisfy important memory requirements, decides to implement the Singleton pattern to restrict the number of active components of a specific type. Since the developers are not familiar with this memory requirement, they modify the code in a way that they create multiple instances of the corresponding type. Detecting the corresponding architectural violation with static analysis tools is difficult, not to say impossible.

Thus, traditional approaches to detect architectural erosion may reach their limits when it comes to dynamic architectures. To address this problem, we propose a novel approach to detect architectural erosion, based on runtime verification [18]:

– First, architectural assertions are formally specified in FACTUM [22], a language for the formal specification of dynamic architectural constraints.
– Second, code instrumentation and monitors are generated from the specification.
– Finally, the monitors are used to detect architectural violations at runtime.

To this end, we developed two algorithms to map a given FACTUM specification to corresponding events and LTL-formulæ over these events.

We evaluated the approach on two case studies from different domains. In the first case study, we applied the approach in a controlled environment for the analysis of an open source Java application in the domain of Business Information Systems. Accordingly, we implemented the algorithms for Java applications, specified architectural constraints in FACTUM and generated monitors and code instrumentation. Finally, we executed the system and observed it for violations.

The second case study was executed in a real, industrial setting, in which we applied the approach for the analysis of a proprietary C application in the automotive domain. To this end, we first implemented the algorithms for C applications. Again, we specified architectural constraints in FACTUM and generated corresponding monitors and code instrumentation. Lastly, we executed the system observing it for architectural violations.

With this paper, we provide two major contributions:

– We describe a systematic way to detect architectural erosion for dynamic architectures using runtime verification techniques and demonstrate its applicability using a running example. To this end, we also describe two algorithms for mapping a FACTUM specification to corresponding events and LTL-formulæ over these events.
– We show the feasibility of using runtime verification to detect architectural erosion through two case studies.

The paper is structured as follows: We first provide some background on FACTUM (Sect. 2) and runtime verification (Sect. 3). Then, we introduce our approach and demonstrate each step using a simple, running example (Sect. 4). In (Sect. 5) we present the two case studies. Finally, we discuss related work (Sect. 6) and conclude the paper with a summary and limitations that lead to potential future work (Sect. 7).

2 Specifying Dynamic Architectures in FACTum

FACTUM [22] is an approach for the formal specification of constraints for dynamic architectures. It consists of a formal system model for dynamic architectures and techniques to specify constraints over this model. FACTUM is also implemented in terms of an Eclipse/EMF application called FACTUM Studio [24], which supports a user in the development of specifications.

2.1 System Model

In our model [21,25], components communicate with each other by exchanging messages over ports. Thus, we assume the existence of set $\mathcal{M}$, containing all *messages*, and set $\mathcal{P}$, containing all *ports*, respectively. Moreover, we postulate the existence of a type function

$$\mathcal{T}\colon \mathcal{P} \to \wp(\mathcal{M}) \tag{1}$$

which assigns a set of messages to each port.

Port valuations. Ports are means to exchange messages between a component and its environment. This is achieved through the notion of port valuation. Roughly speaking, a valuation for a set of ports is an assignment of messages to each port.

Definition 1 (Port valuation). *For a set of ports $P \subseteq \mathcal{P}$, we denote with $\overline{P}$ the set of all possible, type-compatible* port valuation*, formally:*

$$\overline{P} \quad \overset{def}{=} \quad \Big\{\mu \in \big(P \to \wp(\mathcal{M})\big) \mid \forall p \in P\colon \mu(p) \subseteq \mathcal{T}(p)\Big\} \ .$$

Moreover, we denote by $[p_1, p_2, \ldots \mapsto M_1, M_2, \ldots]$ the valuation of ports $p_1, p_2, \ldots$ with sets M_1, M_2, …, respectively. For singleton sets we shall sometimes omit the set parentheses and simply write $[p_1, p_2, \ldots \mapsto m_1, m_2, \ldots]$.

In our model, ports may be valuated by *sets* of messages, meaning that a component can send/receive a set of messages via each of its ports at each point in time. A component may also send no message at all, in which case the corresponding port is valuated by the empty set.

Interfaces. The ports which a component may use to send and receive messages are grouped into so-called interfaces.

Definition 2 (Interface). *An* interface *is a pair (I, O), consisting of* disjoint *sets of* input ports $I \subseteq \mathcal{P}$ *and* output ports $O \subseteq \mathcal{P}$. *The set of all interfaces is denoted by $IF_{\mathcal{P}}$. For an interface $if = (I, O)$, we denote by*

- $\mathsf{in}(if) \quad \overset{def}{=} \quad I$ *the set of input ports,*
- $\mathsf{out}(if) \quad \overset{def}{=} \quad O$ *the set of output ports, and*
- $\mathsf{port}(if) \quad \overset{def}{=} \quad I \cup O$ *the set of all interface ports.*

Components. For the purpose of this paper, we assume the existence of a set of components $(\mathcal{C}_{if})_{if \in IF_{\mathcal{P}}}$. A *component port* is a port combined with the corresponding component identifier. Thus, for a family of components $(\mathcal{C}_{ct})_{if \in IF_{\mathcal{P}}}$ over a set of interfaces $IF_{\mathcal{P}}$, we denote by:

- $\mathsf{in}(\mathcal{C}) \quad \overset{\text{def}}{=} \quad \bigcup_{c \in \mathcal{C}} (\{c\} \times \mathsf{in}(c))$, the set of *component input ports*,
- $\mathsf{out}(\mathcal{C}) \quad \overset{\text{def}}{=} \quad \bigcup_{c \in \mathcal{C}} (\{c\} \times \mathsf{out}(c))$, the set of *component output ports*,
- $\mathsf{port}(\mathcal{C}) \quad \overset{\text{def}}{=} \quad \mathsf{in}(\mathcal{C}) \cup \mathsf{out}(\mathcal{C})$, the set of all *component ports*.

Moreover, we may lift the typing function (introduced for ports in Eq. 1), to corresponding component ports:

$$\mathcal{T}((c,p)) \overset{\text{def}}{=} \mathcal{T}(p) \ .$$

Finally, we can generalize our notion of port valuation (Def. 1) for *component ports* $CP \subseteq \mathcal{C} \times \mathcal{P}$ to a so-called *component port valuation*:

$$\overline{CP} \overset{\text{def}}{=} \Big\{\mu \in \big(CP \rightarrow \wp(\mathcal{M})\big) \mid \forall cp \in CP\colon \mu(cp) \subseteq \mathcal{T}(cp)\Big\} \ .$$

To better distinguish between ports and component ports, in the following, we shall use p, q, pi, po, $\ldots$; to denote ports and cp, cq, ci, co, $\ldots$; to denote component ports.

Architecture Snapshots. In our model, an architecture snapshot *connects* ports of *active* components.

Definition 3 (Architecture snapshot). *An* architecture snapshot *is a triple* (C', N, μ)*, consisting of:*

- *a set of active components* $C' \subseteq \mathcal{C}$*,*
- *a connection* $N\colon \mathsf{in}(C') \rightarrow \wp(\mathsf{out}(C'))$*, such that types of connected ports are compatible:*
$$\forall ci \in \mathsf{in}(C')\colon \bigcup_{co \in N(ci)} \mathcal{T}(co) \subseteq \mathcal{T}(ci) \ , \textit{ and} \tag{2}$$
- *a component port valuation* $\mu \in \overline{\mathsf{port}(C')}$.

We require connected ports to be consistent in their valuation, i.e., if a component provides messages at its output port, these messages are transferred to the corresponding, connected input ports:

$$\forall ci \in \mathsf{in}(C')\colon N(ci) \neq \emptyset \implies \mu(ci) = \bigcup_{co \in N(ci)} \mu(co) \ . \tag{3}$$

Note that Eq. (2) *guarantees that Eq.* (3) *does not violate type restrictions. The set of all possible architecture snapshots is denoted by* $AS^{\mathcal{C}}_{\mathcal{T}}$.

For an architecture snapshot $as = (C', N, \mu) \in AS^{\mathcal{C}}_{\mathcal{T}}$*, we denote by*

- $CMP_{as} \overset{def}{=} C'$ *the set of active components and with* $|c|_{as} \overset{def}{\iff} c \in C'$*, that a component* $c \in \mathcal{C}$ *is* active *in* as*,*
- $CN_{as} \overset{def}{=} N$*, its connection, and*
- $val_{as} \overset{def}{=} \mu$*, the port valuation.*

Moreover, given a component $c \in C'$*, we denote by*

$$\mathsf{cmp}^{c}_{as} \in \overline{\mathsf{port}(c)} \overset{def}{=} \Big(\lambda p \in \mathsf{port}(c)\colon \mu\big((c,p)\big)\Big) \tag{4}$$

the valuation of the component's ports.

Note that cmp^{c}_{as} is well-defined iff $|c|_{as}$.

Moreover, note that connection N is modeled as a set-valued function from component input ports to component output ports, meaning that:

1. input/output ports can be connected to several output/input ports, respectively, and

2. not every input/output port needs to be connected to an output/input port (in which case the connection returns the empty set).

Moreover, note that by Eq. (3), the valuation of an input port connected to many output ports is defined to be the *union* of all the valuations of the corresponding, connected output ports.

Example 1 (Architecture snapshot). Figure 1 shows a conceptual representation of an architecture snapshot (C', N, μ), consisting of:

- active components $C' = \{c_1, c_2, c_3\}$ with corresponding interfaces;
- connection N, defined as follows:
  - $N((c_2, i_1)) = \{(c_1, o_1)\}$,
  - $N((c_3, i_1)) = \{(c_1, o_2)\}$,
  - $N((c_2, i_2)) = \{(c_3, o_1)\}$, and
  - $N((c_1, i_0)) = N((c_2, i_0)) = N((c_3, i_0)) = \emptyset$; and
- component port valuation $[(c_1, o_0), (c_2, i_1), (c_3, o_1), \dots \mapsto \mathtt{M}_3, \mathtt{M}_5, \mathtt{M}_3, \dots]$.

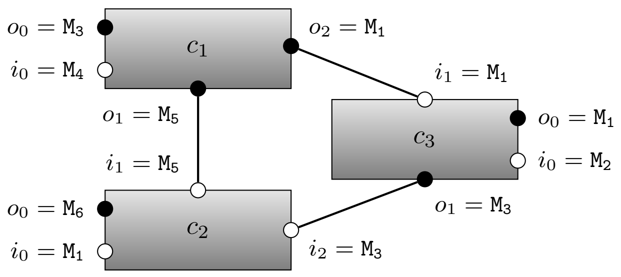


Fig. 1: architecture snapshot consisting of three components c_1, c_2, and c_3; a connection between ports (c_2, i_1) and (c_1, o_1), (c_2, i_2) and (c_3, o_1), and (c_3, i_1) and (c_1, o_2); and valuations of the component parameters and ports.

Architecture Traces. An *architecture trace* consists of a series of snapshots of an architecture during system execution. Thus, an architecture trace is modeled as a stream of architecture snapshots at certain points in time.

Definition 4 (Architecture trace). *An* architecture trace *is an infinite stream* $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$.

Example 2 (Architecture trace). Figure 2 shows an architecture trace $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$ with corresponding architecture snapshots $t(0) = k_0$, $t(1) = k_1$, and $t(2) = k_2$. architecture snapshot k_0, for example, is described in Ex. 1.

2.2 Specifying Constraints for Dynamic Architectures

FACTum provides several techniques to support the formal specification of constraints for dynamic architectures:

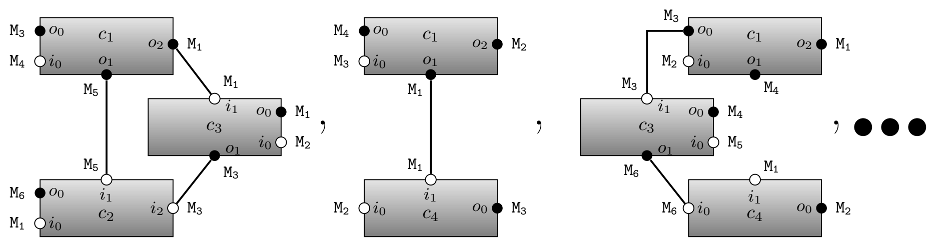


Fig. 2: The first three architecture snapshots of an architecture trace.

- First, the data types involved in an architecture are specified in terms of algebraic specifications [5,33].
- Then, a set of interfaces is specified graphically using architecture diagrams.
- Finally, a set of architectural assertions is added to specify constraints about component activation and deactivation as well as interconnection.

A FACTUM specification comes with a formal semantics in a denotational style. To this end, each specification is interpreted by a corresponding set of architecture traces (as introduced in Def. 4).

Architecture diagrams. Architecture diagrams [23] are a graphical formalism to specify interfaces. To this end, interfaces are represented by rectangles with their ports denoted by empty (input) and filled (output) circles. An example of an architecture diagram can be found in Fig. 4.

Specifying architectural constraints. Architectural constraints are specified in terms of *architecture trace assertions* [23]. These are a type of first order linear temporal logic formulæ, with variables denoting components and some special terms and predicates:

- With $c.p$, for example, we denote the valuation of port p of a component c.
- With $\widehat{c.p}$, we denote that port p of a component c is currently active.
- With $\wr c \wr$ we denote that component c is currently active.
- With $c.o \rightsquigarrow c'.i$ we denote that output port o of component c is connected to input port i of component c'.

An example of an architecture trace assertion can be found in Fig. 5. A formal semantics is provided in App. **??**.

3 Runtime Verification

Runtime verification (RV) is tightly related with model checking [8,12], and has its origins in model checking. RV is a dynamic analysis method aiming at checking whether a run of the system under scrutiny satisfies a given correctness property [20]. RV deals with observed executions as the system generates them, and consequently, it applies to black box systems for which no system model is available, or to systems where the system model changes during the execution.

In RV the correctness of a system is usually checked by a monitor. Therefore, through the literature, runtime verification is also referred to as runtime monitoring. "A monitor is a device that reads a finite trace and yields a certain verdict" [20]. In runtime verification, monitors are generated automatically from high-level specifications, and they need to be designed in a way that they consider system's executions incrementally. The specifications are usually formulated with temporal logic, for example, linear temporal logic (LTL) [27,2]. In the simplest form, a monitor decides if a program execution satisfies a particular correctness property or not. The system under analysis, as well as the generated monitor, are executed simultaneously [3]. Namely, the monitor observes the system's behavior. If the monitor detects that property is violated, then it returns a corresponding alarm signal. RV considers only the detection of violations of the correctness properties of a system. Even though RV does not affect the execution of a concrete program, yet it unfolds the possibility to take actions when incorrect or faulty behavior of the system is detected, simply by its nature of being performed at runtime. In RV, often it is distinguished between online and offline methods; in online, a data stream is directly fed into the monitor, whereas in offline monitoring, data is provided from a log file. In this paper, to generate monitors in the first and the second case study we used JavaMOP and LTL3 respectively. In the following subsections, we will briefly explain both of the tools.

JavaMOP. Monitoring-Oriented Programming (MOP) [7], is a formal software development and analysis framework for RV. In MOP the developer specifies desired properties, or generates monitors, using specification formalisms. The monitors are integrated with the user-defined code into the original system, and the code is executed whenever the properties are violated or validated at runtime. This allows the original system to check its dynamic behaviors during execution and it reduces the gap between formal specification and implementation by allowing them to form a system together. Once a violation is detected, user-defined actions are triggered. JavaMOP is MOP-based analysis and runtime verification system for Java, using AspectJ for instrumentation. Expressive requirements specification formalisms can be included in the framework via logic plug-ins, allowing not only to refer to the current state but also to both past and future states [29,6].

LTL3 tools. LTL_3 [3,4], is a 3-valued linear time temporal logic that can be interpreted over finite traces based on the standard semantics of LTL for infinite traces. LTL_3 shares the syntax with LTL but deviates in its semantics for finite traces. The readings of the finite traces and the creation of 3-valued LTL semantics can be automated, and accordingly directly deployed as a runtime verification. LTL_3 Tools are a collection of programs to generate Finite State Machine through LTL formula. LTL_3 Tools takes an LTL formula and outputs a 3-valued corresponding monitor [1].

4 Approach

Figure 3 depicts our approach for the verification of dynamic architecture constraints: As a first step, a set of architectural constraints is specified in FACTUM, consisting of a specification of component types CT and a specification of architectural constraints AS (see Sect. 2). The specification is then used to create two types of artifacts: a set of events that will be monitored and a set of LTL-formulæ based on these events. The events are then used to create corresponding instrumentation code for the system to notify the monitor about the occurrence of events. The LTL-fromulæ are used to generate corresponding monitors to supervise the system for violations of the constraints. While the first steps are mostly independent of the target platform of the system under test, the latter steps depend on the concrete platform of the system under test.

Algorithm 1 shows how to create a set of events from the specification of component types CT. For each type of component, we generate four types of events:

activation events denote activation and deactivation of components of a specific type.

execution events denote the execution of certain methods of a component with or without parameters.

call events denote method calls for a component with or without parameters.

connection events denote a method call with a concrete source and target component.

For each type of component, corresponding activation events are created (Line 2). Further, each input port results in the creation of corresponding execution events (Lines 4-7). Output ports, on the other hand, result in call events (Lines 8-11). Finally, each pair of input and output port (of the same name) results in the creation of a corresponding call event (Lines 12-20).

Creation of LTL-formulæ from a specification of architectural constraints AS is described by Alg. 2. The algorithm essentially modifies architectural assertions by replacing atomic FACTUM assertions with corresponding events created by Alg. 1: To this end, port activations are mapped to corresponding execution/call events without parameters (Lines 4-9), port valuations to corresponding execution/call events with parameters (Lines 10-15), component activations to corresponding activation events (Lines 16-17), and port connections to corresponding call events with source and target locations (Lines 18-19), respectively.

To demonstrate the approach, we use a running example from the domain of business information systems.

4.1 Running Example: Online Shop

In the following, we depict an excerpt of the implementation of a simple online shop in an object-oriented programming language. In this paper, we are only interested in two classes: baskets and items. The following listing sketches the implementation of class basket:

```
public class Basket {
```

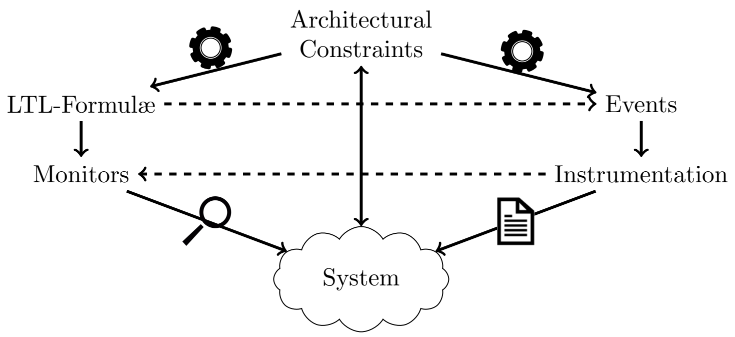


Fig. 3: Runtime Verification of Architectural Constraints

Algorithm 1 Mapping FACTUM to events for instrumentation

```
Input: a set $CT$ of component types
Output: a set of events
1: for all $ct \in CT$ do
2:   create activation event for $ct$
3:   for all ports $p$ of $ct$ do
4:     if $p$ is an input port then
5:       create execution event for $p$ without parameters
6:       create execution event for $p$ with parameters according to the type of $p$
7:     end if
8:     if $p$ is an output port then
9:       create call event for $p$ without parameters
10:      create call event for $p$ with parameters according to the type of $p$
11:    end if
12:    for all $ct' \in CT$ do
13:      if $ct' \neq ct$ then
14:        for all ports $p'$ of $ct'$ do
15:          if $p = p'$ and $p$ is an input port then
16:            create call event from $c.p$ to $c'.p'$ without parameters
17:          end if
18:        end for
19:      end if
20:    end for
21:  end for
22: end for
```

```
  private List items;

  public void addItem(String name, Integer price) {
    Item it = new Item();
    it.setName(price);
    it.setPrice(name);
    items.add(it);
  }
}
```

The basket contains a collection of items and a method to add items to the list by using their name and price.

Thereby, the class item is implemented as follows:

```
public class Item {
  private String name;
  private Integer price;

  public void setName(String nm) {
    this.name = nm;
  }

  public void setPrice(String pr) {
```

Algorithm 2 Mapping FACTUM to LTL-formulæ

```
Input: a set $AS$ of architectural constraints
Output: a set of LTL-formulæ
 1: for all $\varphi \in AS$ do
 2:   for all basic assertions $\psi$ in $\varphi$ do
 3:     switch ($\psi$)
 4:     case $\widehat{c.p}$:
 5:       if $p$ is an input port then
 6:         replace $\psi$ in $\varphi$ with corresponding execution event (without parameters)
 7:       else if $p$ is an output port then
 8:         replace $\psi$ in $\varphi$ with corresponding call event (without parameters)
 9:       end if
10:     case $c.p = M$:
11:       if $p$ is an input port then
12:         replace $\psi$ in $\varphi$ with corresponding execution event with parameters $M$
13:       else if $p$ is an output port then
14:         replace $\psi$ in $\varphi$ with corresponding call event with parameters $M$
15:       end if
16:     case $\wr c \wr$:
17:       replace $\psi$ in $\varphi$ with corresponding activation event
18:     case $c.o \rightsquigarrow c'.i$:
19:       replace $\psi$ in $\varphi$ with corresponding call event with source $c$ and target $c'$ in $\varphi$
20:     end switch
21:   end for
22: end for
```

```
    this.price = pr;
  }
}
```

Each item has a name and a price and methods to modify them.

4.2 Specifying the Property

As a first step, we need to identify architectural assertions for the system. For our webshop, one assertion could be the following:

> Whenever a user adds an item to its basket, a corresponding object is created and added to the list of items of the user's basket.

To formalize the property in FACTUM, we first need to create a corresponding architecture diagram. Figure 4 depicts the architecture diagram for our webshop example. It depicts the two types of components required to specify the property: a Basket and an Item. The *Basket* has one input port *addItem*, which can be used to trigger the addition of a new item. Moreover, it has two output ports *setName* and *setPrice* to set the name and price of an item, respectively. The *Item*, on the other hand, has two input ports *setName* and *setPrice* to set a name and price.

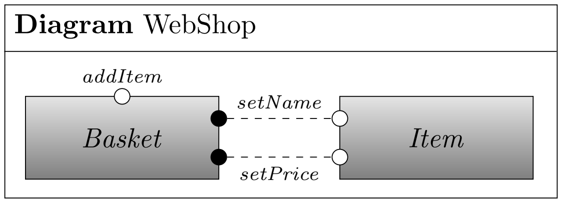


Fig. 4: Architecture diagram for WebShop

The architecture diagram depicts the interfaces for our architecture. The architecture constraint described above can now be formalized over these interfaces in terms of architectural assertions (Sect. 2). A corresponding formalization is depicted in Fig. 5. Roughly speaking, the specification requires that, whenever a component bs of type $Basket$ receives a message (n, p) on its input port $addItem$ (Eq. (5)), a component of type $Item$ is created (Eq. (6)) and initialized with name n (Eq. (7)) and price p (Eq. (8)). To prevent potential security issues, the connection constraints provided in Eq. (6) and Eq. (7), require that the initialization is indeed done through the basket component itself. Note that the specification imposes an ordering for the initialization of an item: first, the name has to be set and then the price.

ASpec AddItem		**for** WebShop
var	n:	$String$
	p:	$Integer$
	it:	$Item$
	bs:	$Basket$

$$\square\Big(bs.addItem = (n, p) \longrightarrow \tag{5}$$

$$\bigcirc(\wr it \wr) \tag{6}$$

$$\wedge \bigcirc\bigcirc\big(bs.setName = n \wedge bs.setName \rightsquigarrow it.setName\big) \tag{7}$$

$$\wedge \bigcirc\bigcirc\bigcirc\big(bs.setPrice = p \wedge bs.setPrice \rightsquigarrow it.setPrice\big)\Big) \tag{8}$$

Fig. 5: Architectural Constraints for Webshop.

4.3 Generating the Monitors

After specifying the property, we can apply Alg. 1 and Alg. 2 to create instrumentations and monitors, respectively.

Code instrumentation. Instrumenting our webshop example would require to monitor 16 types of events derived from the specification of component types depicted in Fig. 4: activation events for $Basket$ components and $Item$ components, respectively. Execution and call events (with and without parameters) for $addItem$, $setName$, and $setPrice$. Connection events for $setName$ and $setPrice$.

Monitors. Figure 6 depicts a possible monitor for our webshop system. The generated monitor is parameterized by a basket bs, an item it, name n, and price p, and starts in state S_1, whenever an $addItem$ event is observed with name n and price p. If the next observed event is the creation of a new item, it progresses to state S_2; otherwise it moves to an error state S_e, in which it remains forever. From state S_2 it either moves to state S_3 (for the case the next event is $setName$ with name n) or in the error state (if it observes any other event). From state S_3 it may again either move to S_4 (for the case the next event is $setPrice$ with name p) or the error state. State S_4, however, is a final state, which means that the monitor terminates. Note that the monitor has no state which signals the successful satisfaction of the property described by Fig. 5. This is because the satisfaction of formula Fig. 5 can only be determined when observing an infinite trace and never for any finite prefix.

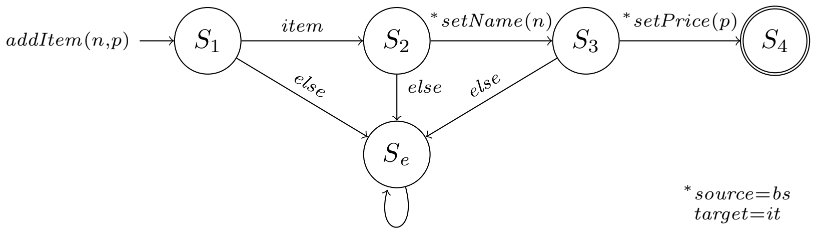


Fig. 6: Monitor for Webshop.

4.4 Performing the Verification

After installing the instrumentation code, we can start the monitor to detect architectural violations. Table 1 depicts a possible execution of the system in terms of method calls. It also lists the corresponding events as received by the monitor and the state of the monitor after receiving the event.

	Monitor	
System Event	Event	State
bs.addItem(book, 100)	$addItem(book, 100)$	S_1
it=new Item()	$item$	S_2
bs $\longrightarrow$ *it.setPrice(100)*	$setPrice(100)$	S_e
bs $\longrightarrow$ *it.setName(book)*	$setName(book)$	S_e
…	S_e	

Table 1: Possible execution and corresponding state of the monitor.

The occurrence of an event $addItem(book, 100)$ triggers the creation of a new monitor which is parameterized with name $book$ and price 100, and which starts in state S_1. Since the next observed event is the creation of a new item, the monitor moves on to state S_2. In state S_2, however, the monitor observes a price change event and thus, moves to the error state S_e in which it remains to signal violation of the architectural property.

5 Evaluation

We evaluated the proposed approach by applying it to two case studies. The first case study, described in Section 5.1 was done in the area of Business Application Systems. The second case study, described in Section 5.2 is performed on Embedded Systems in the automotive domain and involves industry practitioners in the validation efforts.

5.1 Case Study: Business Application Systems

Study context. As a study object in the first case study in the field of Business Application Systems, we choose JetUML [28], an *open source* java application to model UML diagrams. Its main features consist of creating new diagrams and adding graphical elements to it. The system's implementation consists of about $35,653$ lines of code, split into 242 classes.

Study setup. To implemented the approach, we implemented Alg. 1 to create corresponding specifications for AspectJ and Alg. 2 to generate monitors using JavaMOP [29]. The algorithm was implemented in XTend as a plugin for FACTUM Studio to generate corresponding code from a FACTUM specification[1].

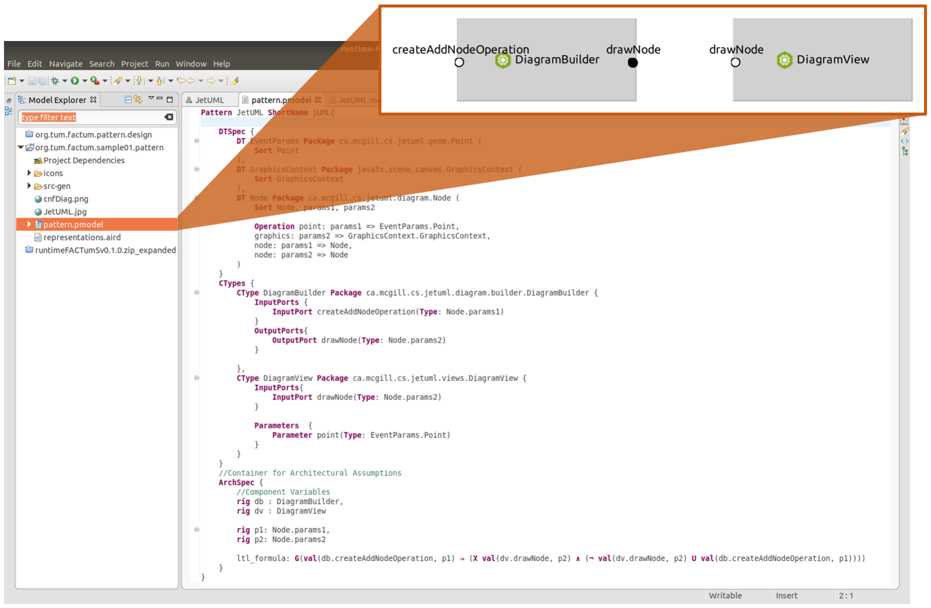
Study execution. We executed the study in a controlled setting. For each property, we first modeled the component types required for the property. Then, we formalized the properties involved. Figure 7 depicts the specification of one of the properties in FACTUM Studio. After formalizing the property in FACTUM Studio, we generated the corresponding code for JavaMOP. Then, we run the system, installed the instrumentation and started the monitors. Finally, we executed some steps of the use case and observed the monitor to detect violations.

Findings. In total, we executed two experiments for this case study: First, we specified 10 properties to check the creation of certain objects after some events. For example, we wanted to ensure that, whenever a user adds a new element to the drawing board, a corresponding object is created and drawn by executing the specific method. During the execution of these experiments, the monitors did not show any violations. For the second set of experiments, we strengthened the properties to ensure that the figure is drawn only once. While executing this experiment, the monitors signaled violations of properties. Namely, after creating the corresponding objects, they were drawn multiple times. While this is not a severe bug, it can indeed be considered as a design issue, since it involves unnecessary computation which might decrease performance.

5.2 Case Study: Embedded Systems

Study context. The second case study was executed in collaboration with an industrial partner from the automotive domain. In this use case we analyzed the architecture of the Service Disconnect (SD) software component in the Battery Management System (BMS) in the vehicles. The system is responsible for monitoring and regulating a car's battery.

[1] The plugin is now part of FACTUM Studio and can be downloaded from the corresponding website: `https://github.com/habtom/factum/tree/runtimeverification`



```
Pattern JetUML ShortName jUML{

    DTSpec {
        DT EventParams Package ca.mcgill.cs.jetuml.geom.Point (
            Sort Point
        ),
        DT GraphicsContext Package javafx.scene.canvas.GraphicsContext (
            Sort GraphicsContext
        ),
        DT Node Package ca.mcgill.cs.jetuml.diagram.Node (
            Sort Node, params1, params2

            Operation point: params1 => EventParams.Point,
            graphics: params2 => GraphicsContext.GraphicsContext,
            node: params1 => Node,
            node: params2 => Node
        )
    }
    CTypes {
        CType DiagramBuilder Package ca.mcgill.cs.jetuml.diagram.builder.DiagramBuilder {
            InputPorts {
                InputPort createAddNodeOperation(Type: Node.params1)
            }
            OutputPorts{
                OutputPort drawNode(Type: Node.params2)
            }

        },
        CType DiagramView Package ca.mcgill.cs.jetuml.views.DiagramView {
            InputPorts{
                InputPort drawNode(Type: Node.params2)
            }

            Parameters  {
                Parameter point(Type: EventParams.Point)
            }
        }
    }
    //Container for Architectural Assumptions
    ArchSpec {
        //Component Variables
        rig db : DiagramBuilder,
        rig dv : DiagramView

        rig p1: Node.params1,
        rig p2: Node.params2

        ltl_formula: G(val(db.createAddNodeOperation, p1) ⇒ (X val(dv.drawNode, p2) ∧ (¬ val(dv.drawNode, p2) U val(db.createAddNodeOperation, p1))))
    }
}
```

Fig. 7: Modeling Architectural Constraints for JetUML in FACTUM Studio

Study setup. Figure 8 depicts the simplified form of the BMC architecture provided by the partner and contains the following components:
Component *Vehicle* is an abstraction of the car itself.
Analog to digital converter (ADC) forwards the signal to component *BMC Master*. The ADC pin can additionally be used for checking the service disconnect status.
Battery disconnect unit (BDU) monitors the connection of the car to its battery. It communicates the status to *BMC Master* via the CAN bus.
BMC Master is the actual battery management component which provides a functionality "Service Disconnect" which communicates the connection state of the battery to the vehicle. To this end, it compares the values obtained from *ADC* and *BDU*. For the case that the values are the same, it forwards it to the car. If the two signals differ from each other, then *BMC Master* should send an error message. In *BMC Master* reside all battery management related software features. One of those features is SD whose dynamic properties we verify at runtime.

Study execution. One of the main objectives of this paper is to bridge the gap between the definition of dynamic properties written as LTL formula and monitor programs which can be run along with a software system during runtime. Also, the monitors can run along the log files generated by the system containing the time-stamps of signal values concerning the dynamic properties. Therefore together with the engineers, we formalized an architecture property in FACTUM. Accordingly to the approach depicted in Fig. 3, we instrumented the code and developed a corresponding monitor. Code instrumentation was done using CAPL script. The log communication from the components' signals was obtained through CANoe tool. The monitor was written in C# and created using LTL3 tools. Finally, the system was tested, and the collected log-files were inspected using the previously created monitor.

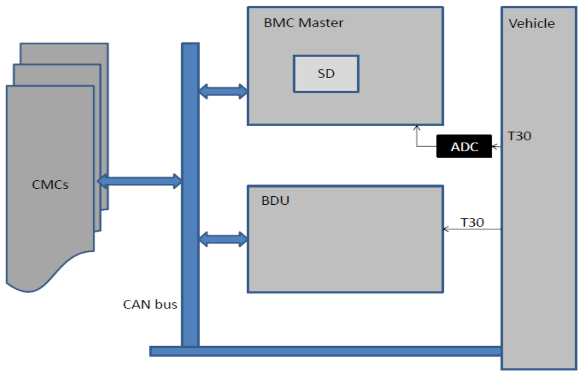


Fig. 8: Architecture of the Battery Management system

Findings. In this case study we specified 4 properties and our experiments revealed runs of the system in which the architecture property was indeed violated. After communicating our results to our industry partner, they confirmed that the architecture specification was wrong and they fixed it accordingly. The reason was a so-called architectural drift, i.e., the systems real architecture evolved over time; however, the architecture specification was not adapted accordingly.

6 Related Work

In this paper, we provide a systematic approach that detects architectural erosion or architectural drift based on runtime verification[2]. Although different techniques for controlling software architectural erosion have been proposed across the literature, previous work has mainly focused on static analysis methods. In this paper, instead of applying static analysis, we present a new approach for solving the architectural erosion problem by applying runtime verification. This allows us to go beyond detecting static violations of the systems, but rather focusing and checking dynamic violations that are emerging from the architectural dynamicity of the systems. Additionally, dynamic software analysis approaches or runtime verification is vastly used to solve various problems in many different fields. However, until now it has not been applied to ensure architecture consistency. We believe that our approach is the first one which combines and utilizes RV for the analysis of dynamic architectural drift. Therefore, in this section, we discuss related work on the field of architectural erosion and runtime verification.

Murphy at al. [26], Koschke and Simon [16] and Said et al. [30] propose reflection model techniques as architectural solutions for controlling the software erosion. The reflection model techniques compares a model of the implemented architecture and a hypothetical model of the intended architecture. The latter is created from a static analysis of the source code.

Lavery and Watanabe in [18] present a runtime monitoring method for actor-based programs and a scala-based asynchronous runtime-monitoring module that realizes the

[2] In this paper the difference between architectural erosion and architectural drift is not considered.

proposed method. They aim to provide failure recovery and mitigation mechanism for Scala applications by making use of the lightweight software to monitor the properties specified. The module does not require specialized languages for describing application properties that need to be monitored. The programmer specifies in Scala the property that needs to be verified, and the mitigation code that needs to be invoked when a particular property is violated.

To make dynamic reconfigurations more reliable [19] proposes an approach ensuring that system consistency and availability is maintained despite run-time failures and changes in the system. A reconfiguration is a modification of a system state during its execution, and it may potentially put this system in an inconsistent state. In the first step the authors provide a model of configurations and reconfigurations. They specify consistency by means of integrity constraints, i.e. configuration invariants and pre/post-conditions on reconfiguration operations. Alloy has been used as a specification language to model these constraints, which are later translated in FPath, a navigation language used as a constraint language in Fractal architectures to check the validity of integrity constraints on real systems at runtime.

7 Conclusion

In this paper, we present an approach that provides a solution to the problem of detecting architectural drift for dynamic architectures. To this end, architectural constraints are formally specified using FACTUM, a language for the specification of constraints for dynamic architectures, and then systematically transferred to corresponding monitors and code instrumentation which can be used to detect violations at runtime.

In the paper, we describe the approach and demonstrate its applicability through a running example. Next, we describe two algorithms which can be used to generate code instrumentations and monitors, which monitor the architectural violations from the FACTUM specification. Finally, we describe the outcome of two case studies on which we evaluated the proposed approach in the context of an open source Java application and a proprietary C application.

Our results suggest that runtime verification is indeed feasible to detect architectural erosion for different types of applications: from embedded C applications to object-oriented business information systems. Additionally, our evaluations of the approach that we propose in this paper show that it scales well and has the potential to uncover important architecture violations.

However, our results also expose some limitations of the approach. First, our approach can only be used to detect violations and not to guarantee the absence of architectural violations, nor to act whenever an incorrect behavior is detected. Second, it is not yet possible to analyze real-time requirements. This posed a serious limitation, particularly for the second use case, since many important architectural assertions require timed aspects.

The first limitation is a general limitation of runtime verification, and there is not much we can do about this. However, the second limitation can be addressed in the future, as future work. To this end, future work should focus to extend the approach with timing aspects.

Acknowledgements. We want to thank Ilias Gerostathopoulos for helpful discussions on architectural erosion. Parts of the work on which we report in this paper was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. 01Is16043A.

References

1. Bauer, A.: Ltl3 tools (2019), `http://ltl3tools.sourceforge.net/`
2. Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In: Software Engineering Conference, 2006. Australian. pp. 10–pp. IEEE (2006)
3. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: International Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 260–272. Springer (2006)
4. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for ltl and tltl. ACM Transactions on Software Engineering and Methodology (TOSEM) 20(4), 14 (2011)
5. Broy, M.: Algebraic specification of reactive systems. In: Algebraic Methodology and Software Technology. pp. 487–503. Springer, Springer Berlin Heidelberg (1996)
6. Chen, F., Roşu, G.: Java-mop: A monitoring oriented programming environment for java. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 546–550. Springer (2005)
7. Chen, F., Roşu, G.: Mop: an efficient and generic runtime verification framework. In: Acm Sigplan Notices. vol. 42, pp. 569–588. ACM (2007)
8. Clarke, E., Grumberg, O., Peled, D.A.: Model checking the mit press. Cambridge, Massachusetts, London, UK (1999)
9. Coverity: Architecture analysis (2019), `http://www.coverity.com/products/architectureanalysis`
10. Deissenboeck, F., Heinemann, L., Hummel, B., Jürgens, E.: Flexible architecture conformance assessment with conqat. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. pp. 247–250 (2010), `https://doi.org/10.1145/1810295.1810343`
11. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE Transactions on Software Engineering 27(1), 1–12 (Jan 2001)
12. Emerson, E.A.: The beginning of model checking: A personal perspective. In: 25 Years of Model Checking, pp. 27–45. Springer (2008)
13. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch: Why reuse is so hard. IEEE Softw. 12(6), 17–26 (Nov 1995), `https://doi.org/10.1109/52.469757`
14. Godfrey, M.W., Lee, E.H.S.: Secrets from the monster: Extracting mozilla's software architecture. In: In Proc. of 2000 Intl. Symposium on Constructing software engineering tools (CoSET 2000. pp. 15–23 (2000)
15. Klocwork: Klockwork architect (2019), `http://www.klocwork.com/products/insight/architect-code-visualization/`
16. Koschke, R., Simon, D.: Hierarchical reflexion models. In: null. p. 36. IEEE (2003)
17. Lattix: The lattix architecture management system (2019), `http://www.lattix.com/products`
18. Lavery, P., Watanabe, T.: An actor-based runtime monitoring system for web and desktop applications. In: 2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). pp. 385–390 (June 2017)

19. Léger, M., Ledoux, T., Coupaye, T.: Reliable dynamic reconfigurations in a reflective component model. In: International Symposium on Component-Based Software Engineering. pp. 74–92. Springer (2010)
20. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
21. Marmsoler, D., Gleirscher, M.: On activation, connection, and behavior in dynamic architectures. Scientific Annals of Computer Science 26(2), 187–248 (2016)
22. Marmsoler, D.: Hierarchical specication and verication of architecture design patterns. In: Fundamental Approaches to Software Engineering - 21th International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (2018)
23. Marmsoler, D.: Axiomatic Specification and Interactive Verification of Architectural Design Patterns in FACTum. Dissertation, Technische Universität München, München (2019)
24. Marmsoler, D., Gidey, H.K.: FACTUM Studio: A tool for the axiomatic specification and verification of architectural design patterns. In: Formal Aspects of Component Software - FACS 2018 - 15th International Conference, Proceedings (2018)
25. Marmsoler, D., Gleirscher, M.: Specifying properties of dynamic architectures using configuration traces. In: International Colloquium on Theoretical Aspects of Computing, pp. 235–254. Springer (2016)
26. Murphy, G.C., Notkin, D., Sullivan, K.: Software reflexion models: Bridging the gap between source and high-level models. ACM SIGSOFT Software Engineering Notes 20(4), 18–28 (1995)
27. Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science, 1977., 18th Annual Symposium on. pp. 46–57. IEEE (1977)
28. Robillard, M.: Jetuml (2019), https://github.com/prmr/JetUML
29. Runtimeverification: Javamop (2019), https://github.com/runtimeverification/javamop
30. Said, W., Quante, J., Koschke, R.: Reflexion models for state machine extraction and verification. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 149–159. IEEE (2018)
31. Sonargraph: Sonargraph-architect (2019), http://www.hello2morrow.com/products/sonargraph
32. structure101: (2019), https://structure101.com/
33. Wirsing, M.: Algebraic specification. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science (Vol. B), pp. 675–788. MIT Press, Cambridge, MA, USA (1990), http://dl.acm.org/citation.cfm?id=114891.114904