On the Understandability of Semantic Constraints for Behavioral Software Architecture Compliance: A Controlled Experiment

Christoph Czepa*, Huy Tran*, Uwe Zdun*, Thanh Tran Thi Kim†, Erhard Weiss† and Christoph Ruhsam†
*Faculty of Computer Science
University of Vienna, Vienna, Austria
Email: {christoph.czepa, huy.tran, uwe.zdun}@univie.ac.at
†Isis Papyrus Europe AG, Maria Enzersdorf, Austria
Email: {thanh.tran, erhard.weiss, christoph.ruhsam}@isis-papyrus.com

***Abstract*—Software architecture compliance is concerned with the alignment of implementation with its desired architecture and detecting potential inconsistencies. The work presented in this paper is specifically concerned with behavioral architecture compliance. That is, the focus is on semantic alignment of implementation and architecture. In particular, this paper evaluates three representative approaches for describing semantic constraints in terms of their understandability, namely natural language descriptions as used in many architecture documentations today, a structured language based on specification patterns that abstract underlying temporal logic formulas, and a structured cause-effect language that is based on Complex Event Processing. We conducted a controlled experiment with 190 participants using a simple randomized design with one alternative per experimental unit. Overall all approaches support a high level of correct understanding, and the statistical inference suggests that all tested approaches are equally well suited for describing semantic constraints for behavioral architecture compliance in terms of understandability. In consequence this indicates that it is possible to benefit from the tested structured languages with underlying formal representations for automated verification without having to suffer from decreased understandability. Vice versa, the results suggest that the use of natural language can be a suitable way to document architecture semantics when reliable automated support for formal verification is of minor importance.**

I. Introduction

A key problem of architectural design is that the design and the implementation drift apart during development and system evolution. Inconsistencies between the designed and implemented architecture are commonly referred to by terms such as *architectural drift*, *architectural erosion* or *architectural decay* (cf. [1–3]). *Software architecture compliance* (also referred to as *software architecture conformance*) is concerned with the alignment of the architecture of software and its implementation to avoid architectural drift. Existing studies have a strong focus on either *structural architecture compliance* (e.g., Knodel & Popescu [4]) or *behavioral architecture compliance* (e.g., Ackermann et al. [5]). An example for structural non-compliance would be a missing relationship in the implementation that is part of architectural design. In contrast, behavioral architecture compliance investigates the semantic consistency of an implementation and its architecture. An example for behavioral non-compliance is that an architectural pattern like Model-View-Controller is used in the architecture description, but the order of the invocations between the Model, View, and Controller components in the implementation does not comply to the pattern's semantics. Architectural compliance has been a recurring research interest, and many contributions have been made in the domain of structural architecture compliance (e.g., Passos et al. [6] for design time checking and de Silva & Balasubramaniam [7] for runtime monitoring) and behavioral architecture compliance (e.g., Bucchiarone et al. [8] for design time checking and Nicolaescu & Lichter [9] for runtime checking).

Despite the progress in recent years, there are aspects that have not been studied sufficiently. One of the shortcomings of many existing studies is the lack of evaluation of the specification languages in terms of their understandability by potential stakeholders in the software development process. Obviously, understandability is an integral part of the potential practical success of an architecture compliance approach. Existing studies often propose their own specific representations for describing semantic or structural constraints, but so far they do not sufficiently evaluate the understandability of those representations (e.g., [9–11]). Some studies merely report the experience with a specific representation rather informally (e.g., [12]). As a consequence, it most often remains unclear, how understandable proposed representations for architecture compliance checking actually are.

In this study, we evaluate the understandability of two representations for semantic architecture constraints that are grounded on established runtime and design time verification techniques, plus natural language, which is commonly used for architecture documentation in practice today. In particular, the first of the structured languages we study is grounded on formal temporal logic (for verification by model checking [13] and automata-based runtime checking [14, 15]). The second language is based on Complex Event Processing (CEP) [16] which is used in many approaches for specifying (among others) behavioral architecture compliance aspects based on runtime events that can be automatically monitored while the system executes (e.g., [7, 17]). In these languages, following

Martin Fowler's advice on business-readable DSLs[1], we try to facilitate understanding structured languages by leveraging natural language elements and indentation-based structuring. We do not only evaluate the understandability of the representations, but also discuss the properties of their underlying checking approaches and how to transform the representations to formal languages for automated checking.

A. Problem Statement

It is possible to describe a semantic constraint in *natural language* (e.g., in English). This seems to have the advantage that the semantic constraint is accessible without the need for learning or understanding a specific notation, but natural language might leave room for interpretation, which could pose a problem for both humans and automated natural language processing.

To decrease the chance for ambiguity, it might make sense to limit the language elements. That is, an alternative is to use a controlled vocabulary and grammar which aims at reducing ambiguity. The downside of this approach might be that the resulting languages are not understood by users without additional explanations and learning of the syntax of the language. These *structured languages* have the advantage that they are directly transformable to technical representations. For example, Yu et al. [18] propose a property pattern-based specification language based on the property specification patterns by Dwyer et al. [19], named PROPOLS, for verifying BPEL service composition schemes. The patterns used in such structured languages are grounded on formalisms such as Linear Temporal Logic (LTL) [20]. That is, semantic constraints are automatically transformable from the structured language to LTL, and the LTL formulas can be used for model checking [21] or for runtime checking by transforming the LTL formula to a nondeterministic finite automaton [15].

Running systems (e.g., enterprise systems with many running processes) usually create a large quantity of events. *Complex Event Processing (CEP)* is an approach for processing a large quantity of events and for reasoning on the formed stream of events (cf. [16]). The Event Processing Language (EPL) [22] can be used to define semantic constraints consisting of temporal queries that are used for reasoning on the stream of events. In our previous work (cf. [23]) we propose to use a subset of EPL for describing semantic constraints, resulting in a reasonable set of constructs, which might qualify this approach as an approachable structured language for describing semantic constraints for runtime architecture compliance checking.

Currently, it is unknown which of the semantic constraint representations for architecture compliance is better understood by users. One could claim that natural language is the best approach as it is the language of everyday life and therefore often used for architecture documentation. An argument against this notion is its ambiguity, which leaves room for interpretation and error. Structured languages (i.e., abstracting formal and technically underlying languages such as LTL and EPL) leave less room for ambiguity, but might not be that intuitive and approachable as natural language. Each approach comes with assumed pros and cons. Scientific evidence is required to rank one approach below, above or equal to another in terms of understandability in the context of specifying and checking architecture compliance.

[1] http://www.martinfowler.com/bliki/BusinessReadableDSL.html

B. Research Objectives

This paper investigates three representative ways for describing semantic architecture constraints, namely in natural language and in two structured languages, one based on formal temporal logic patterns (cf. Dwyer et al. [19]), the other based on Complex Event Processing [16]. This investigation focuses on the perspective of the user (in the context of our evaluation, our focus is on novice software architects, designers or developers) who has to understand these semantic architecture constraints. In particular, this investigation is concerned with finding out whether there is a difference between the languages both in terms of the time needed and the correctness of understanding.

We state the experimental goal using the *GQM (Goal Question Metric) goal template* [24] as follows:

Analyze representations for describing semantic constraints for software architecture compliance
for the purpose of their evaluation
with respect to their understandability
from the viewpoint of a novice software architect, designer or developer
in the context (environment) of the Software Architectures Lab course at the University of Vienna.

C. Context

The study was carried out with 190 computer science students who enrolled in the course "Software Architectures Lab" at the University of Vienna in the end of the summer term 2016. This course is usually taken in the fourth semester. At that time, the students have been trained in software engineering, architecture and development in this and prior courses. Consequently, the students are used as proxies for novice software architects, designers or developers. The time of the experiment was limited to 105 minutes.

D. Guidelines

This work follows the guidelines for conducting and reporting experiments in software engineering by Jedlitschka & Ciolkowski [25], which integrate (among others) the preliminary guidelines by Wohlin et al. [26] and the guidelines by Kitchenham et al. [27]. Moreover, we consider the "Basics of Software Engineering Experimentation" by Juristo & Moreno [28] for the experiment design and the "Robust Statistical Methods for Empirical Software Engineering" by Kitchenham et al. [29] for the statistical evaluation of the acquired data.

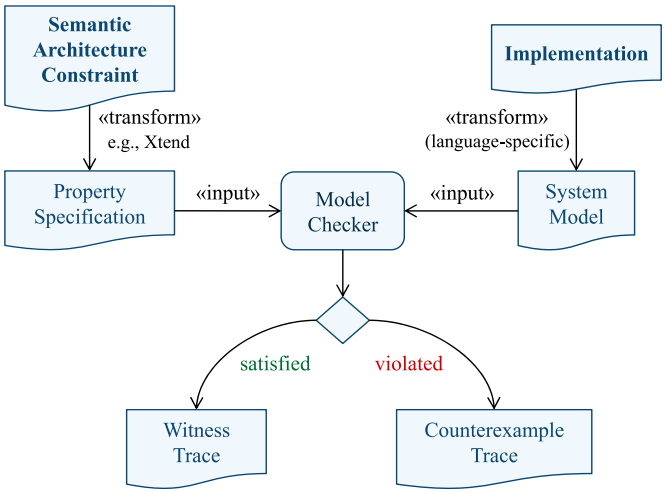


Fig. 1: Design time architecture compliance checking by model checking

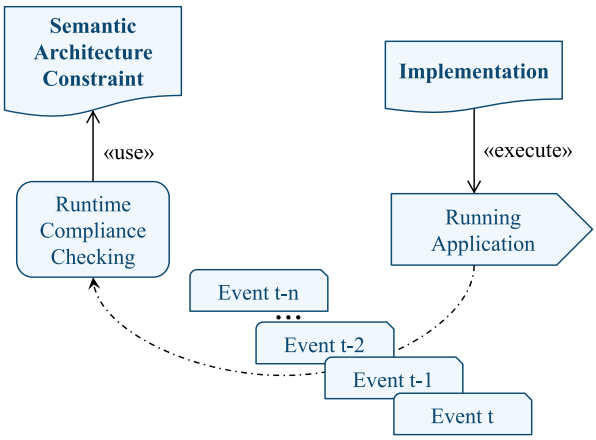


Fig. 2: Generic runtime architecture compliance checking



Fig. 3: Transformation for runtime checking by NFA

II. Background on Semantic Architecture Compliance Checking Techniques and Constraint Representations

Model checking is an established verification technique which evaluates the model of a system against its specification under the consideration of all possible execution traces [13, 30]. Figure 1 shows an overview of architecture compliance checking by model checking. Each *Semantic Architecture Constraint* is transformed (e.g., an automated DSL transformation by Xtend) to a *Property Specification*, a formal temporal logic formula in LTL (Linear Temporal Logic) or CTL (Computation Tree Logic) (cf. [21]). An *Implementation* can also be automatically transformed to a *System Model* (e.g., [31] for C++ and [32, 33] for Java). Some semantic constraints might depend on additional information (e.g., a layer number), so the code may have annotations in such cases that provide the required additional information. The *Model Checker* uses both the *Property Specification* and *System Model* as inputs, and it reports for each specification either a *Witness Trace* (in case of a positive checking result) or a *Counterexample Trace* (if the *Semantic Architecture Constraint* is not satisfied by the *Implementation*).

In contrast to design time checking on the basis of code, runtime checking observes the actual execution of software. Figure 2 provides a rather generic and abstract overview on architecture compliance checking at runtime. A *Running Application* emits events over time, and *Runtime Compliance Checking* evaluates each *Semantic Architecture Constraint* on the basis of received events. To become more concrete, we consider the extraction of events from Java programs at runtime by bytecode manipulation (e.g., de Silva & Balasubramaniam [7]). The built-in package `java.lang.instrument` can be used to enrich Java bytecode with additional code, such as console outputs or event emissions to architecture compliance checking approaches. There also exist additional bytecode manipulation libraries such as *OW2 ASM*[2], *Apache Commons BCEL*[3] and *JBoss Javassist*[4]. Especially Javassist provides a quite intuitive way to implant code before and after invocations by the methods `insertBefore` and `insertAfter` provided by the `CtMethod` class.

As shown in Figure 3, for the runtime verification by a *Nondeterministic Finite Automaton (NFA)*, it is necessary to transform the *Semantic Architecture Constraint* to an LTL formula in the first step. For example, if the original language is a structured DSL (e.g. in Xtext), then a generator (such as Xtend) can perform this transformation automatically. Having created the LTL formula, we make use of existing transformation approaches that automatically generate NFA from LTL such as `LTL2NFA` [15], and we enact the NFA as runtime checker [34].

For runtime checking by CEP, we make use of the open source version of *Esper*[5]. To create the *Complex Event Processing Instruction* for the runtime checking of a *Semantic Architecture Constraint*, a transformation (e.g., from an Xtext DSL to Xtend) must take place. The resulting instruction consists of the information about the initial truth value of the semantic constraint (i.e., some constraints are initially violated, others satisfied) and for setting up a root statement for observing the event stream. A *Statement* always has an EPL (Event Processing Language) expression. If the event pattern (described in EPL) is observed in the stream, the *Statement* invokes the specified *Listener*. A *Listener* can change the *Truth Value* of the constraint and lead to the creation of a new *Statement*.

[2]http://asm.ow2.org/
[3]https://commons.apache.org/proper/commons-bcel/
[4]http://www.javassist.org/
[5]http://www.espertech.com/products/esper.php

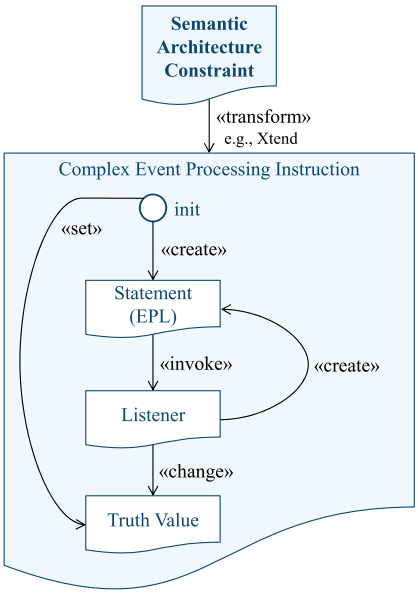


Fig. 4: Transformation for runtime checking by CEP

A. Natural Language Semantic Constraint Representation

Natural Language Constraints (NLC) are an approach that enables the description of semantic constraints by the full range of a natural language (e.g., English). Natural language processing (NLP) is concerned with analyzing the given information (cf. [35]). It should be mentioned that natural language understanding is considered to be AI-complete (cf. [36]). Consequently, it is considered to be difficult to create software that actually is able to understand each and every semantic constraint in natural language reliably. Nonetheless, according to a recent work by Ghosh et al. [37], the ARSENAL (Automatic Requirements Specification Extraction from Natural Language) approach supports the automated transformation of natural language to LTL formulas in specialized domains, and the authors even claim its generalizability to other domains. However, the authors also state that including the human in the transformation process usually leads to better results, so the approach must be considered as semi-automatic. Example 1 describes a strict layered architecture[6] in natural language.

Example 1:

```
A component A may only invoke interfaces of
components that are one layer beneath A.
```

B. Temporal Logic Pattern-Based Semantic Constraint Representation

Temporal Logic Constraints (TLC) are formally grounded on temporal logics such as Linear Temporal Logic (LTL) [20]. In particular, it uses established temporal logic patterns for abstraction of underlying formal logics as proposed by Dwyer et al. [19]. Frequently used patterns are the *Response*, *Precedence*, *Existence* and *Absence* patterns. Besides the *Global* scope, it is possible to define four other scopes, namely *Before*, *After*, *After Until* and *Between*. The approach further abstracts the patterns by providing a structured language for the definition of semantic constraints. Example 2 describes a layered architecture by this approach. It demands the *Absence* of an invocation of a component with a greater layer number and the *Absence* of an invocation of a component with a smaller layer number than the invoker's layer number minus one.

Example 2:

```
any of the following never occurs:

    'layer number' greater than
    'invoked by'.'layer number'

    'layer number' less than
    'invoked by'.'layer number' - 1
```

Due to the pattern-based approach, the structured natural language constraint given in Example 2 can be automatically transformed to its formal LTL representation (cf. [19]).

C. CEP-Based Cause-Effect Semantic Constraint Representation

Cause-Effect Constraints (CEC) have their roots in Complex Event Processing (CEP). In CEP, an event stream is observed for the occurrence of a specific pattern of events. This representation is based on a subset (cf. [23]) of EPL (Event Processing Language) [22]. In particular, the *Cause*-part may contain the structured natural language operators *every P*, *P does not happen until Q*, *P leads to Q*, *not P*, *P and Q*, and *P or Q*, which can be directly transformed to EPL. The *Effect*-part is indented to avoid structuring by brackets (as known from Python) and consists of a state value change and/or additional Cause-Effect statements. Example 3 describes a layered architecture using this approach. An invocation of a component with a greater layer number causes a violation of the constraint, and an invocation of a component with a smaller layer number than the invoker's layer number minus one causes a violation of the constraint as well.

Example 3:

```
is initially satisfied

'layer number' greater than
'invoked by'.'layer number'

    causes violation

'layer number' less than
'invoked by'.'layer number' - 1

    causes violation
```

III. Experiment Planning

A. Goals

The experiment's goal is to measure the construct *understandability* of the three representations, namely *Natural Language Semantic Constraints*, *Temporal Logic Pattern-Based*

[6]The layers example unites both static and behavioral aspects of software architecture compliance and is chosen to illustrate that the discussed behavioral checking approaches may also be used to verify static aspects. For strictly behavioral constraints, we refer the interested reader to the experimental material in Section III-C.

Semantic Constraints and *Cause-Effect Semantic Constraints*. The *quality focus* is on the *correctness* and the *response time* of the answers.

B. Experimental Units

The participants of the experiment are students of the Faculty of Computer Science at the University of Vienna, Austria, who enrolled in the "Software Architectures Lab" course in the summer term 2016.[7] This course, which is intended for students in the fourth semester, is concerned with teaching principles of software architecture, such as architectural views, styles and patterns, and domain-driven design. In previous courses, the students received training in programming, software engineering and modeling. We carried out the experiment just like a normal exercise, so the students were not informed that they take part in an experiment. The students earned credits in the course based on their performance in this exercise. In total, there are 190 participants. Due to random assignment of participants to groups, the final distribution resulted in $65 : 64 : 61$.

C. Experimental Material & Tasks

The experimental material is based on a selection of architectural software design patterns. These patterns are the basis for the semantic constraints used in the experiment. In particular, the *Broker*, *Layers*, *Pipes & Filters* and *Producer-Consumer* patterns (cf. [38]) are used. In total there are four tasks, each concerned with a semantic constraint related to an architectural pattern. Each task comprises a semantic constraint specification (for an example see Section II) and four questions. Each question consists of the sentence "*Is the following a legal trace of the system according to the specification above?*", an event log trace and an answer choice. Since in this paper the tasks cannot be described at a level of detail, so that a replication is possible, we make the questionnaire of each group available online.[8]

D. Hypotheses, Parameters, and Variables

We formulate the following hypotheses for this controlled experiment:

H_0 : There is no difference in terms of understandability between the approaches.
H_A : The approaches differ in terms of their understandability.

There are two dependent variables: (1) the *correctness* achieved in answering the questions, and (2) the *response time*, which is the time it took to answer the questions. These two dependent variables are commonly used to measure the construct *understandability* (cf. [39, 40]). The independent variable (also called factor) has three treatments, namely the three different representations for describing semantic constraints, which are the *Natural Language Semantic Constraint Representation*, the *Temporal Logic Pattern-Based Semantic Constraint Representation*, and the *CEP-Based Cause-Effect Semantic Constraint Representation*.

[7]https://ufind.univie.ac.at/en/course.html?lv=050054&semester=2016S
[8]https://swa.univie.ac.at/ICSA2017/

E. Experiment Design & Execution

We use a simple randomized design with one alternative per experimental unit. By this, we try to avoid learning effects and experimenter bias in the assignment to groups. Before the participants of the TLC and CEC group start working on the tasks, they read through a concise introduction of the notation. These introductions are provided at the beginning of the questionnaire. Aside from that information, no additional training is provided to the participants.

F. Procedure

The experiment has a total duration of 105 minutes. In this time, everything related to the experiment that has to be done by the participants has to be completed, which involves several activities to be performed in the handed-out document. Firstly, the participants have to fill out information on their background comprising the number of years of programming experience, the number of years working in the software industry, the number of years applying software design patterns, and whether or not there is any prior knowledge on logical formalisms or Complex Event Processing. Secondly, the document informs on how to track time correctly. Thirdly, the structure and meaning of event logs is quickly discussed. Fourthly, specific aspects of the provided semantic constraint approach are discussed concisely. Fifthly, the participants try to complete the given tasks, which includes tracking the time and answering the questions. The experiment was carried out following this plan without known deviations.

IV. Analysis

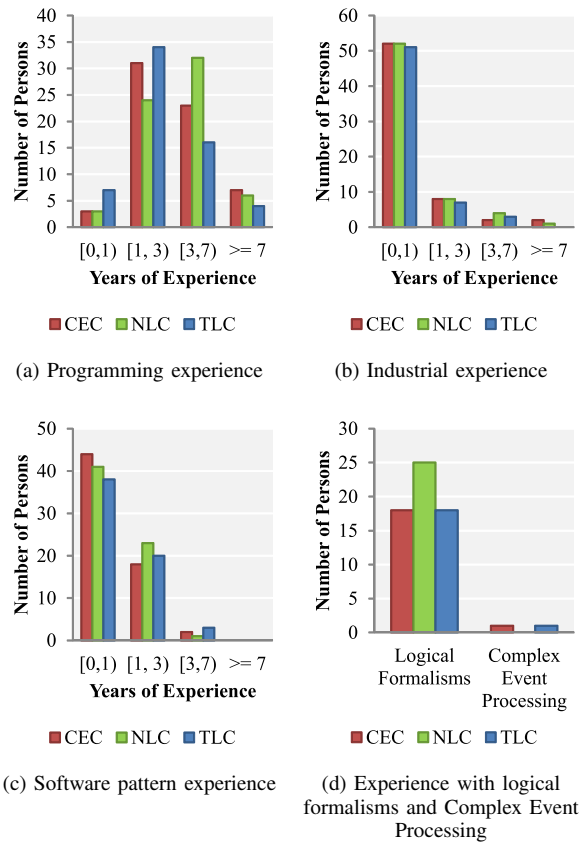
A. Descriptive Statistics & Data Set Preparation

In this subsection, we discuss the preparation of the data set, and we present the acquired data with the help of descriptive statistics.

The experience of the participants per group in different categories, namely programming experience, industrial experience, experience with software patterns and experience with logical formalisms and Complex Event Processing, is shown in Figure 5. Overall, the random distribution of participants to groups is almost balanced with few minor exceptions. The NLC group seems to be slightly more experienced in programming and logical formalisms while the TLC group seems to be slightly less experienced in programming.

Missing response times (12.1 percent) are substituted by the average response time of the group per task. There is not enough evidence to remove potential outliers from the data, because the response times are plausible and the correctness reflects the actual understanding of the participants, which might vary from person to person to a great degree. Consequently, a very good or very bad performance of a participant is considered to be a valuable piece of data that must not be removed.

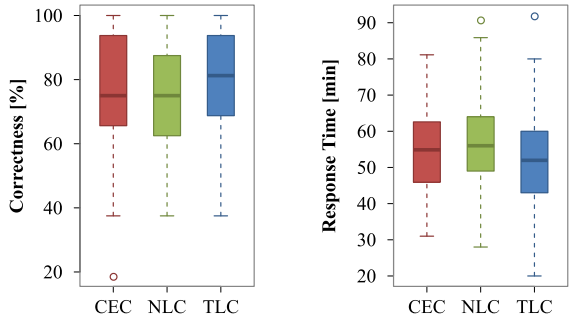
Table I contains the number of observations, central tendency measures and dispersion measures per notation. The acquired data is visualized by boxplots in Figure 6 and by kernel density plots Figure 7. The TLC group has its medians



(a) Programming experience (b) Industrial experience

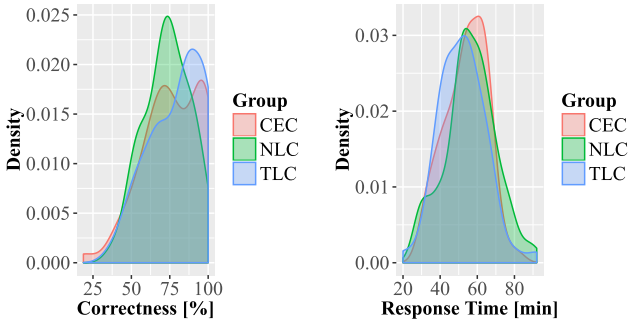
(c) Software pattern experience (d) Experience with logical formalisms and Complex Event Processing

Fig. 5: Experience of participants per group



(a) Correctness (b) Response time

Fig. 6: Boxplots of the dependent variables per group



(a) Correctness (b) Response time

Fig. 7: Kernel density plots of the dependent variables per group

TABLE I: Number of observations, central tendency and dispersion per group

	NLC	CEC	TLC
Number of observations	65	64	61
Mean response time [min]	56.23	53.98	51.97
Standard deviation [min]	13.64	11.23	12.53
Median response time [min]	56.00	54.87	51.97
Median absolute deviation [min]	10.38	11.90	13.29
Min. response time [min]	10.38	31.00	20.00
Max. response time [min]	90.88	81.13	92.00
Mean correctness [%]	73.27	76.95	79.30
Standard deviation [%]	14.97	19.02	17.40
Median correctness [%]	75.00	75.00	81.25
Median absolute deviation [%]	18.53	27.80	18.53
Min. correctness [%]	37.50	37.50	37.50
Max. correctness [%]	100	100	100

and quantiles above those of the other groups in terms of correctness. For the response time, this group has its medians and quantiles below those of the other groups. There is one outlier in the Cause-Effect group regarding the correctness variable, and there is a single response time outlier both in the TLC and NLC group. The kernel density plots enable a more detailed visualization of the distribution of the data than the previously discussed visualization by boxplots. According to the kernel density plots, the data does not appear to be normally distributed, and the distributions look different which implies unequal variances in the different groups. This holds especially for the correctness data. Moreover, it can be seen that the NLC group has its peak correctness at about 75%. In contrast to the other groups, the density drops rapidly thereafter, so less participants were able to achieve a high percentage of correctness. The TLC group has its peak at about 90%. In contrast to the two other groups, the CEC group has two peaks, one at about 70% and another at about 95%. With regards to response time, the peaks of the TLC and NLC groups are at about 50-55 minutes. However, there is a higher density in the NLC group in the range of 30 to 50 minutes. The peak density of the CEC group is at about 60 minutes.

The data acquired in this experiment is publicly available.[9]

B. Hypothesis Testing

The multivariate analysis of variance (MANOVA) is a suitable statistical procedure in the presence of two dependent variables. However, before applying MANOVA, necessary assumptions must be met. A graphical analysis by kernel density plots and normal Q-Q plots, and Shapiro-Wilk tests of the data suggest that the univariate normality assumption does not hold for the correctness of the CEC ($p = 0.00093$)

[9]http://doi.org/10.5281/zenodo.321632

TABLE II: Cliff's d for *correctness*

	CEC vs. TLC	CEC vs. NLC	TLC vs. NLC
$p_1 = P(X > Y)$	0.41291	0.52356	0.56242
$p_2 = P(X = Y)$	0.11347	0.10144	0.09987
$p_3 = P(X < Y)$	0.47362	0.375	0.3377
d	0.06071	−0.14856	−0.22472
s_d	0.10301	0.10119	0.10115
z	0.58933	−1.4681	−2.22158
$CI\ (low)$	−0.14151	−0.33905	−0.41194
$CI\ (high)$	0.25807	0.05364	−0.01929
p	0.55672	0.14455	0.02813
$FDR\ adjusted\ p$	0.55672	0.2891	0.16443

TABLE III: Cliff's d for *response time*

	CEC vs. TLC	CEC vs. NLC	TLC vs. NLC
$p_1 = P(X > Y)$	0.54995	0.45553	0.3947
$p_2 = P(X = Y)$	0.01409	0.01058	0.01412
$p_3 = P(X < Y)$	0.43596	0.53389	0.59117
d	−0.11399	0.07837	0.19647
s_d	0.10352	0.10214	0.10134
z	−1.10113	0.76723	1.93869
$CI\ (low)$	−0.31007	−0.12284	−0.00809
$CI\ (high)$	0.0914	0.27338	0.38525
p	0.27299	0.44437	0.05481
$FDR\ adjusted\ p$	0.40949	0.53324	0.16443

and TLC ($p = 0.00043$) groups. For checking the linearity assumption, we investigate Residuals vs. Fitted Graphs for each group. These plots indicate that the linearity assumption is not met by the data sufficiently. Consequently, the power of the test might be affected. Since multivariate and parametric testing might lead to unreliable results due to unsatisfied model assumptions, we fall back to univariate non-parametric testing. The Kruskal-Wallis test is strongly affected by unequal variances (cf. Kitchenham et al. [29]), so its outcome might also not be reliable, when we consider the properties of the acquired data. As a consequence, we use Cliff's δ [41], a robust non-parametric test. This test is unaffected by change in distribution, non-normal-data and possible non-stable variance. Table II and Table III show the results. Multiple testing ($n = 6$) requires us to lower the significance level in order to avoid type I errors (detecting an effect that is not present). As a classical and wide-spread method, the Bonferroni correction suggests to lower the alpha value to $\alpha = \frac{0.05}{6} = 0.008\dot{3}$, but the method is also known to skyrocket type II errors (failing to detect an effect that is present). As an alternative that is more robust against type II errors, we also look at FDR (False Discovery Rate) adjusted p-values [42]. Both approaches suggest that neither of the tests has a significant result. Additionally, the other mentioned tests (MANOVA and Kruskal-Wallis), which might however be affected by unsatisfied model assumption, do not have significant results. Consequently, there is evidence that H_0 ("There is no difference in terms of understandability between the approaches") cannot be rejected. We used the statistics software *R*[10] for all statistical analyses. In particular, we used the following libraries for the statistical analysis of the data: biotools, mvnormtest, usdm, car, psych, mvoutlier, ggplot2, orddom.

V. Discussion

A. Evaluation of Results and Implications

While the descriptive statistics slightly are in favor of the *Temporal Logic Pattern-Based Semantic Constraint Representation*, the inferential statistics do not indicate any significant difference between the three approaches. This might imply that it does not matter in terms of understanding by the user (who is moderately well trained in software development and design and with basic software architecture knowledge), which of the approaches is actually in use. Another implication is that the more technical approaches (*Temporal Logic Pattern-Based Semantic Constraint Representation* and *CEP-Based Cause-Effect Semantic Constraint Representation*), which support a straightforward transformation to formal languages for automated verification, can be applied without lowering the understanding of semantic constraints. This seems to be a benefit as opposed to the *Natural Language Semantic Constraint Representation*, which is harder to transform to formal representations. Overall all representations facilitate a high level of correct understanding of semantic architecture constraints.

B. Threats to Validity

1) Threats to Internal Validity: We are not aware of any disturbing history effects (environmental events). The short duration of the experiment (max. 105 minutes) limits the chance of maturation effects and experimental fatigue, and we did not observe any such. Since each person is only tested once, we can rule out learning effects. Instrumental bias is avoided by the design of the experiment, which enforces equal scoring of the different groups (i.e., it is ruled out that the researcher scores the groups differently). The chance of selection bias is limited due to the random assignment of participants to groups. A potential threat to the internal validity, which we cannot rule out, could be a cross-contamination between the groups, because the participants share the same social group as computer science students and interact outside of the research process as well. We did not observe compensatory rivalry or demoralization. All participants were compensated equally by gaining points for the fulfillment of the course based on their performance. The possibility of experimenter effects is marginal because the experiment sessions were carried out by (up to that time) uninvolved colleagues, and we are not aware of any personal bias towards a specific outcome. Subject effects are not expected since the experiment took place in the usual context of the course, and the students were not being made aware of taking part in an experiment explicitly.

2) Threats to External Validity: Carrying out the experiment with students as proxies for novice software architect, designer or developer must not necessarily be considered as a large thread to validity, but it reduces the ability to make generalizations to a wider population without additional evidence.

[10] https://www.r-project.org/

That is, it would be interesting to repeat the experiment with professionals. As the context of the current experiment are software architecture patterns, it would be interesting to find out, whether the results are generalizable to other settings, such as compliance in business process management. Consequently, repeating the study with business users in a business context could provide evidence for a potential generalizability to different contexts. The study investigates the understanding of already defined semantic constraints, but it does not consider the creation process of semantic constraints, so it is not generalizable in that regard.

3) Threats to Construct Validity: In this study, we focus on the specific construct *understandability of semantic constraints*. It is measured by the dependent variables *correctness* and *response time*, which are commonly used to measure this kind of construct (cf. [39, 40]). Consequently, we consider the measurement procedure to be suitable for measuring the given construct.

4) Threats to Content Validity: The given tasks comprise only a subset of existing semantic constraint patterns (cf. [19]), and other patterns are not investigated by the given experiment as they were not necessary to create the tasks. Each task of the experiment is related to an existing architectural pattern or style. Testing other scenarios (e.g., highly complex or defective constraints) and using a more extensive set of semantic constraint patterns would improve content validity.

5) Threats to Conclusion Validity: The statistical validity might be affected by the missing response time values, which were substituted by the mean response times, but the large number of observations is assumed to compensate that potential effect. Retaining outliers might be a threat to conclusion validity. However, the outliers appear to be valid measurements. Thus, deleting them would pose a threat to conclusion validity as well. We thoroughly investigated the model assumptions of statistic tests and selected the test with the greatest statistical power, which is beneficial to conclusion validity.

VI. Related Work

Graphically modeled semantic constraints are an alternative to the tested representations, which are not investigated by this experimental study. Dimech & Balasubramaniam [43] propose an approach for the automated extraction of structural and semantic constraints from UML diagrams. They state that the lack of support for choice and iteration in sequence diagrams limits their approach. Ganesan et al. [12] use colored Petri nets to specific constraints for behavioral architecture compliance checking at runtime. They report that constructing the Petri nets is an error-prone iterative process because it is difficult to precisely formalize the architecture through Petri nets. Moreover, although targeting runtime, the approach focuses on structural aspects (like other structural approaches, cf. [4, 44]) and neglects behavioral aspects. Awad et al. [45] propose a visual language called BPMN-Q for compliance checking of business processes with a transformation to temporal logic for formal verification. Van der Aalst & Pesic [46] propose a graphical language for defining service flows declaratively with underlying formal representations in Linear Temporal Logic (LTL). Both approaches are based on temporal logic patterns, and therefore they are related to the studied temporal logic pattern-based semantic architecture constraint representation. The presented study has a strong focus on different (structured) textual representations as textual notations appear to be a suitable approach for describing semantic architecture constraints (cf. Völter [47]), but future extensions may consider graphical or mixed representations as well.

The Grasp ADL [48] is proposed as a textual language for describing architectures, which can be transformed to a graph representation and mapped to the implementation by annotations for runtime monitoring [7] and static checking [49] of structural architecture erosion. Weinreich and Buchgeher [50] propose to specify a reference architecture by a set of roles, required as well as forbidden relationships, and a set of logical constraints on roles and relationships. Caracciolo et al. [11] propose a structured language consisting of natural language elements such as 'must' and 'cannot', and transformations to different evaluation tools. Miranda et al. [51] propose a YAML-based language for the specification of structural architecture constraints for Ruby called ArchRuby. In contrast, ArchJava [52] and Archface [53] directly combine and seamlessly unify architecture definitions and program code to prevent architecture drift. Mulo et al. [17] propose the behavioral runtime checking of service oriented architectures (SOA) by Complex Event Processing (CEP). The constraints are specified in a structured language focusing on execution order (i.e., parallel, in sequence, optionality), and they can be automatically transformed to EPL for compliance checking at runtime. Eichberg et al. use a logics-based language to specify structural constraints [54]. To the best of our knowledge, a dedicated evaluation of the understandability of the mentioned languages is not available.

There also exist related studies regarding the evaluation of constraint languages. Haisjackl et al. [55] conducted a user experiment on the understandability of temporal logic-based declarative models. They conclude that combinations of semantic constraints are harder to understand than single isolated constraints and that experience with imperative modeling is not transferable to declarative modeling. Their study does not compare different ways for defining semantic constraints as the work presented in this paper does. Fahland et al. [56] compare declarative and imperative process modeling languages in terms of understanding on basis of the cognitive dimensions framework (cf. Green & Petre [57]. They assume that understanding process models is similar to understanding programs (also cf. Gilmore & Green [58]). Hoisl et al. [40] conducted a controlled experiment on three notations for defining scenario based model tests. The result of the experiment indicates that a natural-language-based approach should be preferred for scenario-based model tests. However, the validity of the experiment suffers from small sample sizes and the lack of hypothesis testing. Kühne et al. [59] report a controlled experiment on the efficiency of communicating architecture design decisions in which neither textual nor graphical repre-

sentations had a significant advantage. We are not aware of any related studies that evaluate the understandability of semantic architecture constraint representations as our work does.

VII. Conclusion and Future Work

A. Summary

This paper reports a controlled experiment on the understandability of three representations for describing semantic constraints for behavioral software architecture compliance. In particular, the work focuses on architecture descriptions in natural language, as often used for architecture documentation today, and in two structured languages that are formally grounded on established verification techniques (i.e., model checking, automata-based runtime checking, and Complex Event Processing). The experiment took place as part of the "Software Architecture Lab" course in the summer term 2016 at the University of Vienna with 190 participants. The descriptive statistics are slightly in favor of the *Temporal Logic Pattern-Based Semantic Constraint Representation*, but the inferential statistic analyses suggest that there is no difference regarding understandability between the tested approaches. Overall, there is a high level of correctness measured for all of the tested representations.

B. Impact

The results of this controlled experiment suggest that all tested representations are similar in terms of their understandability. The more technical representations for describing semantic constraints (*Temporal Logic Pattern-Based Semantic Constraint Representation* and *CEP-Based Cause-Effect Semantic Constraint Representation*) enable an understandability similar to descriptions in natural language, while having at the same time the advantage of underlying formal representations that can be directly used for automated verification. The results of this study might impact the way semantic constraints are documented. In this regard, the proposed structured language representations, which enable a direct transformation to formal languages for automated verification, could be a viable alternative to natural language descriptions, because such natural language descriptions would require additional manual or semi-automatic processing to a formal/technical language such as LTL or EPL for automated verification.

C. Future Work

A potential limitation of this experiment is that mainly common semantic constraint patterns such as *Global Response*, *Global Precedence* were necessary to create the tested semantic constraints in the context of architectural design patterns. On the one hand, it is realistic that these constraint patterns occur predominantly, on the other hand, it would be interesting to measure the understandability construct with a larger set of semantic constraint patterns. Consequently, a controlled experiment that involves a broader set of semantic constraint patterns is an opportunity for future work. Moreover, not only the understanding of existing semantic constraints, but also the creation process must be evaluated. Other opportunities for future work are comparisons of graphical versus textual semantic constraint approaches, and the repetition of the experiment in different settings (e.g., with different stakeholders in the software development process or with professionals in a business process management context). Future experiments may also involve questionnaires that assess qualitative aspects.

Acknowledgment

We would like to thank all participants. The research leading to these results received public funding: FFG project CACAO, no. 843461; WWTF, Grant No. ICT12-001; Austrian Science Fund (FWF): I 2885-N33

References

[1] J. Rosik, A. L. Gear, J. Buckley, M. A. Babar, and D. Connolly, "Assessing architectural drift in commercial software development: a case study," *Softw., Pract. Exper.*, vol. 41, no. 1, pp. 63–86, 2011.

[2] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132 – 151, 2012, dynamic Analysis and Testing of Embedded Software.

[3] M. Riaz, M. Sulayman, and H. Naqvi, *Architectural Decay during Continuous Software Evolution and Impact of 'Design for Change' on Software Architecture*. Springer, 2009, pp. 119–126.

[4] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *WICSA '07*. IEEE Computer Society, 2007, pp. 12–.

[5] C. Ackermann, M. Lindvall, and R. Cleaveland, "Towards behavioral reflexion models," in *20th International Symposium on Software Reliability Engineering*, Nov 2009, pp. 175–184.

[6] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca, "Static architecture-conformance checking: An illustrative overview," *IEEE Softw.*, vol. 27, no. 5, pp. 82–89, Sep. 2010.

[7] L. de Silva and D. Balasubramaniam, "Pandarch: A pluggable automated non-intrusive dynamic architecture conformance checker," in *ECSA'13*. Springer, 2013, pp. 240–248.

[8] A. Bucchiarone, H. Muccini, P. Pelliccione, and P. Pierini, *Model-Checking Plus Testing: From Software Architecture Analysis to Code Testing*. Springer, 2004, pp. 351–365.

[9] A. Nicolaescu and H. Lichter, "Behavior-based architecture reconstruction and conformance checking," in *WICSA'16*, April 2016, pp. 152–157.

[10] P. Parizek and F. Plasil, "Modeling environment for component model checking from hierarchical architecture," *Electron. Notes Theor. Comput. Sci.*, vol. 182, pp. 139–153, Jun. 2007.

[11] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "A unified approach to architecture conformance checking," in *WICSA'15*, May 2015, pp. 41–50.

[12] D. Ganesan, T. Keuler, and Y. Nishimura, "Architecture compliance checking at runtime: An industry experience report," in *2008 The Eighth International Conference on Quality Software*, Aug 2008, pp. 347–356.

[13] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[14] M. Pesic and W. M. P. van der Aalst, *A Declarative Approach for Flexible Business Processes Management*. Springer, 2006, pp. 169–180.

[15] G. De Giacomo, R. De Masellis, and M. Montali, "Reasoning on ltl on finite traces: Insensitivity to infiniteness," in *AAAI*. AAAI Press, 2014, pp. 1027–1033.

[16] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *SIGMOD '06*. ACM, 2006, pp. 407–418.

[17] E. Mulo, U. Zdun, and S. Dustdar, "Domain-specific language for event-based compliance monitoring in process-driven soas,"

Serv. Oriented Comput. Appl., vol. 7, no. 1, pp. 59–73, Mar. 2013.

[18] J. Yu, T. P. Manh, J. Han, Y. Jin, Y. Han, and J. Wang, *Pattern Based Property Specification and Verification for Service Composition*. Springer, 2006, pp. 156–168.

[19] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE'99*. ACM, 1999, pp. 411–420.

[20] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77. IEEE, 1977, pp. 46–57.

[21] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, 2000.

[22] EsperTech Inc., "EPL Reference," http://www.espertech.com/esper/release-5.1.0/esper-reference/html/event_patterns.html, last accessed: October 4, 2017.

[23] C. Czepa, H. Tran, U. Zdun, T. Tran, E. Weiss, and C. Ruhsam, "Plausibility checking of formal business process specifications in linear temporal logic," in *CAiSE'16, Forum*, June 2016. [Online]. Available: http://eprints.cs.univie.ac.at/4692/

[24] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*. Wiley, 1994.

[25] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, *Reporting Experiments in Software Engineering*. Springer, 2008, pp. 201–228.

[26] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[27] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *ICSE'04*. IEEE, 2004, pp. 273–281.

[28] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*, 1st ed. Springer, 2010.

[29] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong, "Robust statistical methods for empirical software engineering," *Empirical Software Engineering*, pp. 1–52, 2016.

[30] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77. IEEE Computer Society, 1977, pp. 46–57.

[31] P. Rokai, J. Barnat, and L. Brim, "Model checking c++ programs with exceptions," *Science of Computer Programming*, vol. 128, pp. 68 – 85, 2016.

[32] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[33] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: Extracting finite-state models from java source code," in *ICSE '00*. ACM, 2000, pp. 439–448.

[34] F. M. Maggi, M. Westergaard, M. Montali, and W. M. P. van der Aalst, *Runtime Verification of LTL-Based Declarative Process Models*. Springer, 2012, pp. 131–146.

[35] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[36] R. V. Yampolskiy, *Turing Test as a Defining Feature of AI-Completeness*. Springer, 2013, pp. 3–17.

[37] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, *ARSENAL: Automatic Requirements Specification Extraction from Natural Language*. Springer, 2016, pp. 41–46.

[38] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture - Volume 4: A Pattern Language for Distributed Computing*. Wiley Publishing, 2007.

[39] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the #ifdef hell?" *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.

[40] B. Hoisl, S. Sobernig, and M. Strembeck, "Comparing three notations for defining scenario-based model tests: A controlled experiment," in *QUATIC'14*, Sept 2014, pp. 95–104.

[41] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychological Bulletin*, vol. 114, pp. 494–509, 1993.

[42] Y. Benjamini and Y. Hochberg, "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[43] C. Dimech and D. Balasubramaniam, *Maintaining Architectural Conformance during Software Development: A Practical Approach*. Springer, 2013, pp. 208–223.

[44] L. Pruijt, C. Kppe, and S. Brinkkemper, "Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support," in *International Conference on Software Maintenance*, Sept 2013, pp. 220–229.

[45] A. Awad, G. Decker, and M. Weske, "Efficient compliance checking using bpmn-q and temporal logic," in *6th International Conference on Business Process Management*, ser. BPM '08. Springer, 2008, pp. 326–341.

[46] W. M. P. van der Aalst and M. Pesic, *DecSerFlow: Towards a Truly Declarative Service Flow Language*. Springer, 2006, pp. 1–23.

[47] M. Völter, "Best practices for dsls and model-driven development," *Journal of Object Technology*, vol. 8, no. 6, pp. 79–102, 2009.

[48] L. de Silva and D. Balasubramaniam, *A Model for Specifying Rationale Using an Architecture Description Language*. Springer, 2011, pp. 319–327.

[49] L. de Silva and I. Perera, "Preventing software architecture erosion through static architecture conformance checking," in *ICIIS*, Dec 2015, pp. 43–48.

[50] R. Weinreich and G. Buchgeher, "Automatic reference architecture conformance checking for soa-based software systems," in *WICSA'14*, no. 14365363. IEEE, 4 2014, pp. 95–104.

[51] S. Miranda, E. R. Jr, M. T. Valente, and R. Terra, "Architecture conformance checking in dynamically typed languages," *Journal of Object Technology*, vol. 15, no. 3, pp. 1:1–34, Jun. 2016.

[52] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: Connecting software architecture to implementation," in *ICSE '02*. ACM, 2002, pp. 187–197.

[53] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: A contract place where architectural design and code meet together," in *ICSE '10*. ACM, 2010, pp. 75–84.

[54] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *ICSE '08*. ACM, 2008, pp. 391–400.

[55] C. Haisjackl, I. Barba, S. Zugal, P. Soffer, I. Hadar, M. Reichert, J. Pinggera, and B. Weber, "Understanding declare models: strategies, pitfalls, empirical results," *Software & Systems Modeling*, vol. 15, no. 2, pp. 325–352, 2016.

[56] D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal, *Declarative versus Imperative Process Modeling Languages: The Issue of Understandability*. Springer, 2009, pp. 353–366.

[57] T. Green and M. Petre, "Usability analysis of visual programming environments: A cognitive dimensions framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131 – 174, 1996.

[58] D. J. Gilmore and T. R. Green, "Comprehension and recall of miniature programs," *Int. J. Man-Mach. Stud.*, vol. 21, no. 1, pp. 31–48, Oct. 1984.

[59] T. Kühne, M. R. Chaudron, and W. Heijstek, "Experimental analysis of textual and graphical representations for software architecture design," *ESEM 2011*, pp. 167–176, 2011.