Behavior-based Architecture Reconstruction and Conformance Checking

Ana Nicolaescu, Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
Email: {nicolaescu, lichter}@swc.rwth-aachen.de

Abstract—**The reconstruction of software architectures and the evaluation of architecture conformance of software systems is a long-studied research topic. Although up-to-date architecture descriptions are necessary to understand and evolve systems, they are rarely available. Consequently, many software architecture reconstruction approaches and tools have been proposed. Despite this, software architects still do not extensively employ these tools and suffer from negative effects when relying only on outdated descriptions. In this paper we present ARAMIS, an approach and associated toolbox that aims to support the behavior-based reconstruction of up-to-date architecture descriptions based on the correction of possibly outdated prescriptive ones. Additionally, ARAMIS addresses the so-called meta-model incompatibility problem by allowing architects to use their own architecture description language instead of the one that the reconstruction tool requires. ARAMIS checks the behavior-based architecture conformance of a system based on pre-specified communication rules, derives up-to-date descriptions and enables their exploration. We have evaluated ARAMIS during several combined surveys and interviews with subjects from both the industry and academia and have obtained positive results.**

I. Introduction

"Understanding and updating a system's software architecture is arguably the most critical activity" in a system's life-cycle [1]. Therefore, a software can only be sustainably evolved if the architects know the system's currently implemented architecture (often called *as-implemented architecture*). However, due to architecture erosion and drift, the as-implemented architecture is rarely documented, i.e., a *descriptive architecture description* is missing, and, if it exists, it is often inconsistent with the initial architecture description (called *prescriptive architecture description*). The prescriptive architecture description is typically created very early in the software development process and usually consists of static views only. Even if outdated, it can still support the understanding of a system but also often proves misleading due to the differences introduced by architecture degeneration.

A large number of techniques have been developed and proposed both by academia and industry (e.g., [2]) to reconstruct the static views of descriptive architecture descriptions using respective prescriptive architecture descriptions as input. However, to fully understand a system, its behavior (i.e., its dynamic view) has to be understood as well. Considering the huge size of today's software systems (measured e.g. in number of code or architecture units as well as in their interactions), useful reconstruction techniques need to offer views on higher abstraction levels (e.g., layers or components) not only for the static view, but also for the dynamic one.

Furthermore, the languages used in an architecture reconstruction process to create the prescriptive architecture descriptions (called *ADLs*) are always tool-specific. For example, the prescriptive ADL defined by Sonargraph-Architect [3] allows to model layers, layer groups, vertical slices, vertical slices groups and subsystems. Hence, an architect must define the prescriptive architecture description using the reconstruction tool's ADL. According to our experience [4], the fact that architecture descriptions created in existing ADLs (e.g., boxes and lines or company-specific UML diagram types) have to be transformed to a given tool's ADL (called the *meta-model incompatibility problem*) is frustrating and leads to acceptance problems. In order to alleviate this problem, much more flexible architecture reconstruction approaches are needed, which allow the easy "docking" of new or existing ADLs.

The overall goals of our research and the proposed behavior-based approach to architecture reconstruction (called ARAMIS) are the following:

- Support architects to understand the behavior view of a system by automatically mapping the run-time traces on defined architecture units. To this end, ARAMIS recreates the descriptive architecture description by augmenting and correcting a given prescriptive architecture description.
- Enable arbitrary prescriptive architecture descriptions as input and creating respective output descriptions.
- Support architecture conformance checking by detecting communication integrity violations (Luckham et al., [5]) based on predefined communication rules.

ARAMIS assumes that (1) there is a prescriptive architecture description (even if outdated), (2) there is some knowledge to map the code to architecture units, and (3) the system can be executed in order to be monitored.

Compared to our previous publications (e.g., [6]), this paper introduces an extended meta-model with a richer rules taxonomy that enables the communication integrity validation of systems of systems, a refined solution to the meta-model incompatibility problem and an evaluation conducted with users from academia and industry.

The remainder of this paper is organized as follows. Section II describes the conceptual meta-model of ARAMIS. Section III presents our solution to the aforementioned meta-model incompatibility problem. Section IV discusses a first evaluation of our work. Section V offers an overview of the related work and Section VI concludes the paper and gives some insights regarding our future work.

II. ARAMIS Concept Meta-Model

The concept meta-model of ARAMIS can be divided into four areas of interest, as depicted in Figure 1. The *Architecture* area presents the high-level concepts concerning architecture descriptions, their structure and governing communication rules. The *Monitoring* area highlights the concepts specific to its run-time monitoring character. The *Communication Rules* area depicts a taxonomy of the rules that can be specified in ARAMIS. The parametrized rules, being more complex, are addressed in the *Parametrization* area of the meta-model.

A. Architecture

The root element of the meta-model is the analyzed *software system* which can be composed of further systems, thus constituting a system of systems. A software system has a *descriptive* and possibly a *prescriptive architecture description*. Each architecture description defines the *architecture units* comprising the system (e.g., layers, components, etc.) and the *communication rules* that should govern their communication.

B. Monitoring

Since ARAMIS analyzes a software system using run-time information, the system must be initially monitored during some *episodes* of interest. Currently, ARAMIS integrates two well-known monitoring systems: *Kieker* [7], to collect run-time data from Java- and JavaEE-based systems and *Dynatrace* [8], to monitor heterogeneous systems of systems implemented using different languages and technologies. An episode can be created for example by running a set of test cases or by interacting with the system's GUI to achieve a desired functionality. Each episode produces some run-time *traces* consisting of so-called *execution record pairs*, which are ordered pairs of *code building blocks* that accessed each other.

In order to abstract from programming-language specific code building blocks (e.g., Java classes or C# namespaces) the meta-model offers untyped *code units* which are linked to code building blocks via regular expression-based *filters*.

Code units are included in untyped architecture units (e.g., a set of packages form a "layer" architecture unit). This grants independence from ADLs and allows modeling architectures based on informal architecture descriptions. To document the designed purpose expressed in the language used by the architects (e.g., layer, subsystem, component etc.) the architecture units have an optional *role* attribute. As architecture units are coarser grained structural units they can build hierarchies of other *units*, be them code units (a layer consists of a set of packages) or architectural units (a component is further structured in several layers). The *identifier* attribute is used to uniquely identify architecture units.

Because real-life systems are heterogeneous using various kinds of interaction mechanisms, ARAMIS not only allows to analyze the direct communication between code building blocks and map it on architectural units [6] but also allows the analysis of more "complex" communication, occurring, e.g., through web service calls. To characterize this communication, execution record pairs contain a list of *communication parameters*. These are simple but flexible name-value tuples, as they depend on the infrastructure employed to collect data from the monitored system and the level of detail to which the communication is documented. The most common communication parameters are: *protocol* (values are e.g., REST, SOAP, AMPQ messaging), *webservice-endpoint*, and *queue* (values are the queue names used for transmitting messages).

C. Communication Rules

The communication between architecture units is governed by communication rules. The ARAMIS meta-model applies view inheritance to classify these rules according to 4 criteria.

First, according to its *permission type*, a rule can *allow* or *deny* the communication between architecture units.

Second, according to its *emergence type*, a rule can be explicitly *specified* (e.g. "layer A should not access layer B") or implicit and thus named *derived* (e.g., "component C1 should not access component C2" because they are included in further architecture units that are explicitly not allowed to access each other according to a specified rule). The derived rules are consequently automatically computed by ARAMIS based on the specified ones, as explained in [6].

Third, based on the *communication type* we distinguish between three flavors of communication rules: *caller rule*: concerns the communication to all other architecture units emerging from a given caller architecture unit (e.g., "utility layer" is not allowed to issue calls towards any other architecture units); *callee rule*: concerns the communication emerging from all other units towards a given callee architecture unit (e.g., "facade layer" can be called from all other architecture units); *caller-callee* rule: concerns the communication between a pair of specified caller and callee architecture units (e.g., "layer A" must not access "layer B").

Forth, according to their *parametrization type*, the rules can be either non-parameterized, referring to direct communication only, or parameterized, referring to more complex communication mechanisms, specified using an *expression*-based language.

D. Parameterization

The ARAMIS communication rule language, implemented by a XML-based DSL, enables the specification of more complex communication rules based on expressions. Obviously this adds complexity, but these complex communication rules are needed to specify and finally validate inter-systems communication rather than just direct one.

To this end, ARAMIS offers three types of expressions.

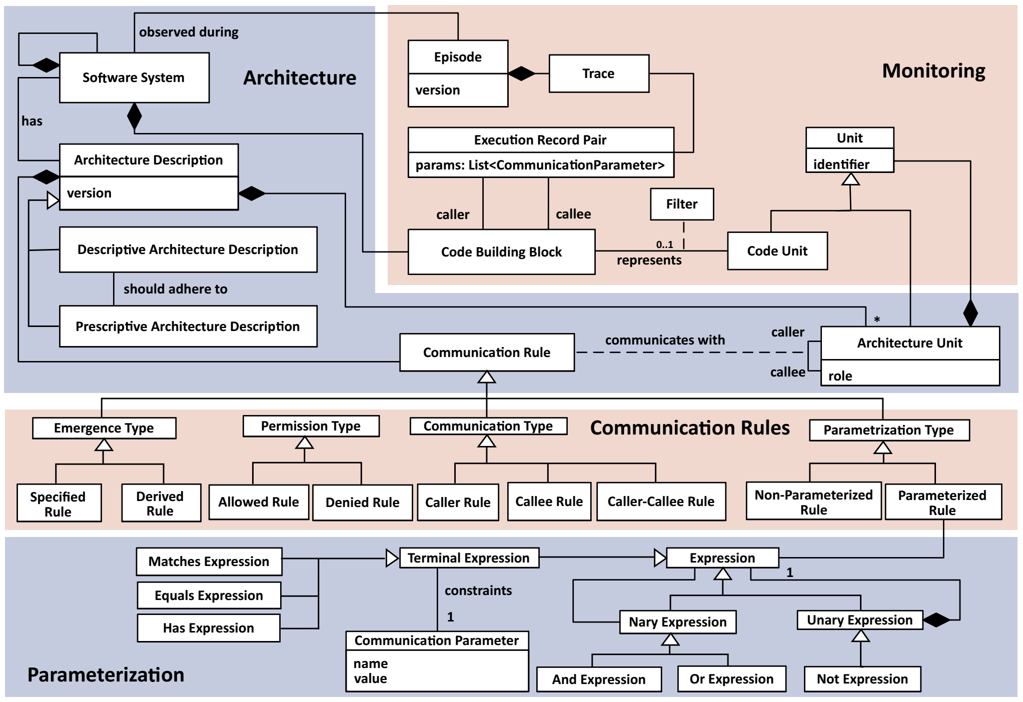

Fig. 1. ARAMIS Meta-Model

Terminal expressions are constraining the communication based on the communication parameters of the corresponding execution record pair. The *matches* and *equals* expressions can be used to allow/deny a certain communication if the value of a parameter specified using its name matches a given regular expression or has a precise value, respectively. The *has* expression is used to allow/deny a communication based on whether the corresponding execution record pair exhibits a parameter with a given name (e.g., allow only communication that has a documented communication "protocol" in its list of communication parameters).

N-ary and *unary* expressions can be further used to express more complex conditions. The unary *not expression* can be applied to an expression to reverse its boolean value. E.g., the not expression used in combination with an equals expression can be used to identify communication in which the value of a parameter is different than a specified one (e.g., communication whose "protocol" is not "REST"). Furthermore, using

```
<calleeCommunicationRule permission="ALLOWED"
                          name="FacadeAccess">
  <callee name="Facade" />
  <expression>
     <and>
        <equals  parameter="protocol" value="REST" />
        <matches parameter="webservice-endpoint" value="*" />
     </and>
  </expression>
</calleeCommunicationRule>
```

Fig. 2. Example of a Communication Rule

an n-ary expression, one can constrain a communication if several expression-based conditions apply simultaneously (*and expressions*) or only if some of them hold (*or expressions*). By combining these different types of expressions, complex rules can be specified for constraining the communication within a system (of systems).

An example of using the ARAMIS communication rule language is depicted in Figure 2. The callee rule `FacadeAccess` specifies that the `Facade` architecture unit can be accessed by any other unit, as long as the communication `protocol` is `REST`. All `REST endpoints` of the `Facade` are freely accessible.

III. Addressing External Prescriptive Architecture Descriptions

As briefly introduced in Section I, architects employing software architecture reconstruction tools are faced with the so-called meta-model incompatibility problem: the need to redefine the prescriptive architecture descriptions using the ADL (meta-model) of the considered reconstruction tool.

To lessen this problem, we have developed a model-based solution to accept prescriptive architecture descriptions as input adhering to other meta-models than the ARAMIS one. Additionally, the applied model-based solution enables to create an output that augments the given input with information resulting from the analysis performed by ARAMIS. The designed model-transformation chain is depicted in Figure 3. It relies on the following two assumptions:

First, prior to requesting ARAMIS to process a given **P**rescriptive **A**rchitecture **M**odel (PAM) that adheres to a non-

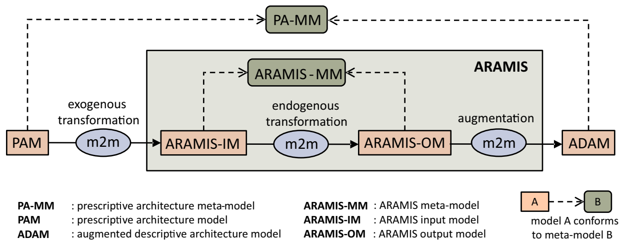
PA-MM
ARAMIS
ARAMIS - MM
exogenous transformation
endogenous transformation
augmentation
PAM
m2m
ARAMIS-IM
m2m
ARAMIS-OM
m2m
ADAM
PA-MM : prescriptive architecture meta-model
PAM : prescriptive architecture model
ADAM : augmented descriptive architecture model
ARAMIS-MM : ARAMIS meta-model
ARAMIS-IM : ARAMIS input model
ARAMIS-OM : ARAMIS output model
A
B
model A conforms to meta-model B

Fig. 3. ARAMIS Transformation Chain (based on [9])

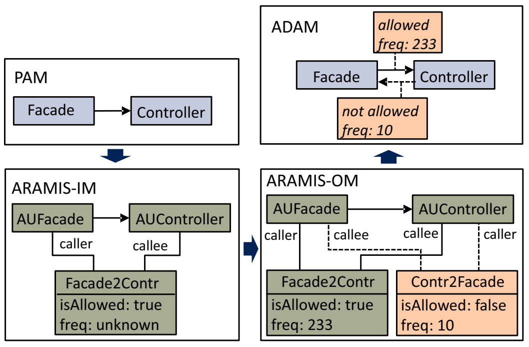

Fig. 4. Example of the Transformation Chain

ARAMIS specific **P**rescriptive **A**rchitecture **M**eta **M**odel (PA-MM), an exogenous model to model (m2m) transformation from the PA-MM to the ARAMIS-MM has to be defined. This transformation applied on a PAM creates an ARAMIS **I**nput **M**odel (ARAMIS-IM) containing the corresponding code and architecture units and the architectural rules governing their communication. Currently, we have implemented such transformations using the Epsilon model-engineering environment for simple *boxes and lines* and *component diagram* PA-MMs.

Second, as we want to present the output model as an augmentation of the input model (called **A**ugmented **D**escriptive **A**rchitecture **M**odel, ADAM), the respective augmentation has to be specified. Currently this has to be done programmatically: for any considered PA-MM, a specific *augmenter* must be implemented by inheriting from a generic one that employs the template design pattern. The specific process steps specified in the template must then be implemented in the specialized augmenter. To implement the augmentation, questions such as the following must be conceptually answered in advance: how should a newly discovered communication, not specified in the PAM, be depicted (e.g., in the boxes and lines example, one can opt for creating dashed/curved/thicker lines for newly discovered communication)? how should violations be marked (e.g., drawing red lines)? etc.

The transformation chain itself is consequently built as follows: giving a non-ARAMIS PAM as input, first an exogenous m2m transformation is applied to create a corresponding ARAMIS-IM conforming to the ARAMIS-MM. The resulted transformation links are also recorded for further use. Then, ARAMIS analyzes the system based on information collected at run-time. The result of this analysis is the behavior-based descriptive architecture description of the system, also named the ARAMIS **O**utput **M**odel (ARAMIS-OM). Next, based on the ARAMIS-OM and the previously documented transformation links, ARAMIS augments (thus performing a second m2m transformation) the input PAM and consequently creates an ADAM which complies to the same, input meta-model (PA-MM). Figure 4 illustrates the model transformation process for the lines and boxes example.

We have chosen to implement an augmentation-based solution instead of a bidirectional model transformation, because of simplicity reasons. Due to the generic nature of the ARAMIS-MM, there is a high probability for several, different elements of a PA-MM (e.g., layers, subsystems) to be mapped on the same element type (e.g., architecture unit) from the ARAMIS-MM. This would typically hinder straight-forward definitions of bidirectional transformations. Instead, we have opted for the augmentation-based solution in which we reuse the transformation links resulting from the m2m transformation between the PAM and the ARAMIS-IM.

IV. Evaluation

Evaluating an approach like the presented ARAMIS one is challenging, as evaluation techniques like experiments are hard to be performed.

We have chosen to apply a combination of interviews and surveys to obtain feedback from the subjects involved. We conducted 21 combined interviews and surveys. Thirteen subjects were professionals from three different companies (twelve architects and one developer), the others were Ph.D or master students (see Table I).

TABLE I
Distribution of the Evaluation Subjects

Sector	Company Size	Role	# Subjects
Academia	-	Master Student	3
		Ph.D Student	5
Insurance	2 companies >1000 employees	Architect	9
Energy Management	1 company 500-1000 employees	Architect	3
		Developer	1

Each evaluation session lasted for approximately one hour. At first we introduced ARAMIS and its toolbox. Then the subjects completed the first part of the survey, containing questions regarding the employed architecting process (e.g., the types of architecture descriptions that they create) and their first impressions concerning the relevance and usefulness of ARAMIS. In the next step the subjects were introduced to two test systems: (1) a users and events manager developed especially for evaluation purposes and (2) the open-source JHotDraw framework. For the users and events manager we defined an architecture containing 10 architecture units that communicated in the recorded episodes with each other over 14128 interactions. The architecture was supposed to adhere to a well-known "client → facade → controllers → db manager" pattern. For evaluation proposes we have introduced 1516 violations to simulate architecture degradation. For the JHotDraw system (529 classes and 38 packages) we have recorded and analyzed 23902 interactions, defined 18 architecture units and discovered 5 architecture violations not conforming to the available prescriptive architecture description of JHotDraw. For each of these systems, we presented the respective prescriptive architecture descriptions and the analysis results, i.e., the behavior-based descriptive architecture descriptions. Next, the subjects were asked to perform eight tasks using ARAMIS on the two introduced systems, in order to ensure that they acquire some in-depth familiarity with the system. Then, the subjects were asked to rate again the quality of ARAMIS, based on the usage experience they have gained in the meantime. Finally, an open discussion took place to gather further feedback and improvement proposals.

The defined tasks to be performed by the subjects mainly focused on the ARAMIS goals related to understanding the behavior view and the identification of violations such as: determining how many violations occurred in the considered system, determining which architecture units caused most of the violations, identifying the order in which the various units interacted to achieve a given functionality, analyzing which architecture units are too coupled or not internally cohesive, deciding which (if any) architecture unit could be a facade of the considered system, etc. The tasks were solved using a series of ARAMIS visualizers, which were presented in more detail in [10] and were performed under our supervision.

A. Some Evaluation Results

We evaluated the suitability of ARAMIS by comparing the results obtained by the subjects with the results that we have previously assessed to be correct. All subjects managed to finish the tasks in time and produced correct results.

Furthermore, all subjects from the industry insisted on the importance of the ability to monitor systems of systems. This supports our decision to allow the specification of complex communication rules based on communication parameters as well as the integration of Dynatrace as a monitoring tool.

Last, a wide range of questions were asked to evaluate the suitability and usefulness of the different ARAMIS visualizers to explore the results and to answer tasks-related questions, as exemplified before. The user-friendliness of the ARAMIS tools, although mostly positively evaluated (2, 5, 11 and 3 subjects have evaluated it to be low, neutral, high and very high respectively), has obvious improvement potentials. However, the usefulness of ARAMIS and its toolbox was clearly validated by the results, as all of the participants rated it as useful (15) or very useful (6).

B. Threats to Validity

Given that no other reconstruction technique besides ARAMIS were used to analyze the evaluated systems, a comparison baseline with other approaches is still missing. All subjects agreed that, if applying ARAMIS on their systems, they would prefer the results to be augmented on their existing boxes and lines or component diagrams. However, in the actual evaluation, we didn't consider external meta-models for the prescriptive architecture. Internally, after having created a m2m transformation and augmenter for the boxes and lines ADL, we needed one and a half person days to re-implement the process for a component diagram ADL. Given our previous experience with creating ARAMIS visualizations (scattered over several months of work) we estimate that the effort spared by applying model engineering techniques is considerable but we do not have exact qualifications thereof.

V. Related Work

Several tools that support the enforcement of architecture description and/or the reconstruction of an up-to-date variant of it have been proposed over time. The majority of developed reconstruction tools focus on recovering the structure of the analyzed system (e.g., [11], [2]).

Concepts and tools that leverage the run-time monitoring of software systems have been proposed but are less numerous than structural ones. The Kieker tool [7] focuses on application performance monitoring and architecture discovery using aspect oriented programming. It displays the information using, e.g., sequence diagrams, dependency graphs and call-trees. ARAMIS also leverages Kieker as a monitoring tool for extracting low-level interactions from J2EE-based systems but then aggregates and validates the data on higher level architectural units. ExplorViz [12] focuses on the run-time of large systems of systems to provide scalable visualizations that enable architecture conformance checking. Unlike ARAMIS, with ExplorViz the prescriptive architecture description cannot be pre-specified and checked against using architectural-relevant rules. The identification of violations is left to the architect, based on his exploration of the provided visualizations. Similarly, an experiment regarding the extraction and monitoring of communication within systems of systems was also presented by Vierhauser et al. [13]. Furthermore, in [14] Arias et al. describe the extraction of a set of predefined architecture views corresponding to an execution viewpoint based on log data and run-time measurements of an industrial software-intensive system. Unlike in the case of ARAMIS, these approaches focus only on the extraction and documentation of communication, while not considering the communication integrity aspect. By exploring the reconstructed dynamic architecture, the architect should investigate by himself if this complies to the prescriptive architecture description. Support for the specification of rules, as in the case of ARAMIS is missing. Dynatrace [8] and Nagios [15] are powerful, commercial tools for application performance management that consider the run-time of heterogeneous large-scale systems of systems. These can trace transactions occurring over multiple system borders and identify aspects such as CPU consumption, performance bottlenecks and other system health-related issues. ARAMIS can easily integrate such tools and already uses Dynatrace as a monitoring tool, enhancing it with architecture conformance checking features.

Yan et al. used the Java Platform Debugger Architecture to develop DiscoTect [16], a system that exploits patterns in the implementation of architectural styles to enable architecture discovery. The approach is based on the construction of architecturally-relevant state-machines that enable the mapping of low-level interactions on architecturally relevant events, such as the creation of components and connectors. With ARAMIS, one does not need to create complex and error-prone state machines for mapping implementation and architectural events and the focus is set mostly on automatic architecture conformance checking. Similarly to ARAMIS, SoftArch [17] augments created static views using its extensible editor with information collected during run-time. The SoftArch modeling editor has been criticized during user evaluations to be too complex and cumbersome, thus reinforcing our observation that architects are more comfortable with employing their own ADLs, as possible with ARAMIS. In [18], Saadatmand et al. present an example of checking low-level behavior consistency based on pre-specified expected state transitions for monitoring software in the automotive domain. Contrastingly, ARAMIS has a richer rules taxonomy that enables conformance assessments on higher, more abstract architectural levels.

Solutions for interchanging architecture description languages to solve the meta-model incompatibility problem were proposed (e.g., ACME [19], the DUALLY framework [20]) but remained mostly a topic of research rather than being adopted in the industry. ARAMIS offers a pragmatic solution, dealing with the complexity of transforming an incompatible meta-model only when evidence requires it and without employing any intermediate format that would further introduce complexity to understand and adopt.

VI. Conclusions and Future Work

In this paper we presented our approach to behavior-based architectural understanding and conformance checking, called ARAMIS. ARAMIS supports the creation of descriptive architecture descriptions based on the augmentation of pre-specified prescriptive ones. Leveraging its integration with the Dynatrace monitoring tool, ARAMIS can check the behavior conformance of systems of systems by enabling architects to flexibly specify simple or complex, expression-based communication rules that place constraints on the captured communication parameters. Moreover, using model engineering techniques, ARAMIS also approaches the meta-model incompatibility problem that most reconstruction tools are faced with. We have evaluated our approach within a series of combined interviews and surveys and obtained positive results.

As future work, we intend to extend ARAMIS to support the simulation and comparison of evolution scenarios. Furthermore, to reduce the effort needed to specify prescriptive descriptions, we will investigate the integration with static architecture recovery and/or clustering-based approaches.

References

[1] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE Press, May 2013, pp. 901–910.

[2] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[3] "Sonargraph-architect," https://www.hello2morrow.com/products/sonargraph/architect, accessed on December, 14th, 2015.

[4] A. Dragomir, M. F. Harun, and H. Lichter, "On bridging the gap between practice and vision for software architecture reconstruction and evolution: A toolbox perspective," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA) 2014 Companion Volume*. ACM, April 2014, pp. 10:1–10:4.

[5] D. C. Luckham, J. Vera, and S. Meldal, "Three concepts of system architecture," in *Technical Report, Stanford University*, 1995.

[6] A. Dragomir, H. Lichter, J. Dohmen, and H. Chen, "Run-time monitoring-based evaluation and communication integrity validation of software architectures," in *the 21st Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1. IEEE, December 2014, pp. 191–198.

[7] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, April 2012, pp. 247–248.

[8] "The dynatrace tool," http://www.dynatrace.com/en/index.html, accessed on December, 8th, 2015.

[9] D. T. Le, A. Nicolaescu, and H. Lichter, "Adapting heterogeneous ADLs for software architecture reconstruction tools," in *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA)*. IARIA XPS Press, November 2015.

[10] A. Nicolaescu, H. Lichter, A. Göringer, P. Alexander, and D. Le, "The ARAMIS Workbench for Monitoring, Analysis and Visualization of Architectures Based on Run-time Interactions," in *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW)*. ACM, September 2015, pp. 57:1–57:7.

[11] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *Sixth Working IEEE / IFIP Conference on Software Architecture (WICSA)*, January 2007, p. 12.

[12] F. Fittkau, P. Stelzer, and W. Hasselbring, "Live visualization of large software landscapes for ensuring architecture conformance," in *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ser. ECSAW '14. New York, NY, USA: ACM, 2014, pp. 28:1–28:4. [Online]. Available: http://doi.acm.org/10.1145/2642803.2642831

[13] M. Vierhauser, R. Rabiser, P. Grünbacher, C. Danner, S. Wallner, and H. Zeisel, "A flexible framework for runtime monitoring of system-of-systems architectures," in *Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, April 2014.

[14] T. B. C. Arias, P. America, and P. Avgeriou, "A top-down approach to construct execution views of a large software-intensive system." *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 233–260, 2013.

[15] "The nagios tool," https://www.nagios.org/, accessed on February, 19th, 2016.

[16] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "Discotect: A system for discovering architectures from running systems," in *The 26th International Conference on Software Engineering (ICSE)*, May 2004, pp. 470–479.

[17] J. Grundy and J. Hosking, "Softarch: Tool support for integrated software architecture development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 13, pp. 125–152, 2003.

[18] M. Saadatmand, D. Scholle, C. W. Leung, S. Ullström, and J. F. Larsson, "Runtime verification of state machines and defect localization applying model-based testing," in *Proceedings of the Working IEEE / IFIP Conference on Software Architecture (WICSA) 2014 Companion Volume*. New York, NY, USA: ACM, April 2014, pp. 6:1–6:8.

[19] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM Press, November 1997, pp. 7–22.

[20] I. Malavolta, H. Muccini, P. Pelliccione, and D. Tamburri, "Providing architectural languages and tools interoperability through model transformation technologies," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 119–140, Jan. 2010.