Electronic Communications of the EASST
Volume 28 (2010)

Proceedings of the Third International DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2010)

Architectural Constraints for Pervasive Adaptive Applications

Christian Straube, Andreas Schroeder

12 pages

Guest Editors: Sonia Ben Mokhtar, Romain Rouvoy, Michael Wagner
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Architectural Constraints for Pervasive Adaptive Applications

Christian Straube[1], **Andreas Schroeder**[1]

Institute for Computer Science, Ludwig-Maximilians-University Munich[1]

Abstract: To face the challenge in today's mobile applications, that software entities and devices enter and leave the application scope very frequently, component-based architectures are used more and more. With the flexibility of this concept and the ability to handle a huge amount of situations come inpredictability and less reliability of the application. This article presents a "safety net" weaved by architectural constraints and an internal DSL to ensure the integrity of the whole application even after multiple reconfigurations. With this integrated, not graph-oriented approach, software-systems can be much more flexible in combination with less code complexity, and the responsibility of architectural integrity is moved from the developer to the application.

Keywords: Component-Based Software-Engineering, System Architecture, Reconfiguration, Constraints, DSL

1 Introduction

System architectures of today's mobile applications are inherently volatile. Software entities and devices enter and leave the application scope as the users move through the physical environment, and new functionality is dynamically added or removed; as a consequence, connections among devices and relations among entities change frequently.

In order to keep such applications manageable, a simple but powerful model of software and system structure is needed. One approach followed is using component-based architectures. Essentially, these consist of two parts: the so-called components, which are responsible for implementing application behavior and encapsulate functionality [Szy98] and the so-called connectors, which bind components together and can be seen as mediators between components.

In the classical component-based approach, the advantages of reusability and minimized coupling of components are accompanied by an essential disadvantage: the model does not incorporate means to react to changes in the architecture, and as a consequence, the whole system cannot react to changes in its surroundings. The introduction of architectural reconfiguration [KM90, OMT98] dealt with this problem by introducing means and concepts for changing the structure of a system at run-time. Still, today's component systems lack the flexibility offered by these theories. One reason for this issue is that repeated reconfigurations may lead to systems with unclear structures, thereby losing the benefits of a clear system architecture.

Figure 1 gives an example of how the architecture of a software system may evolve over time without using any architectural constraints. In this example, the parameters in the surrounding change in an unpredictable order and combination, while at the same time, several rules must be satisfied to ensure intended operation of the software. It is easy to see that with hard-wired reconfiguration rules like nested if-then-else statements it is hardly possible to handle every pos-

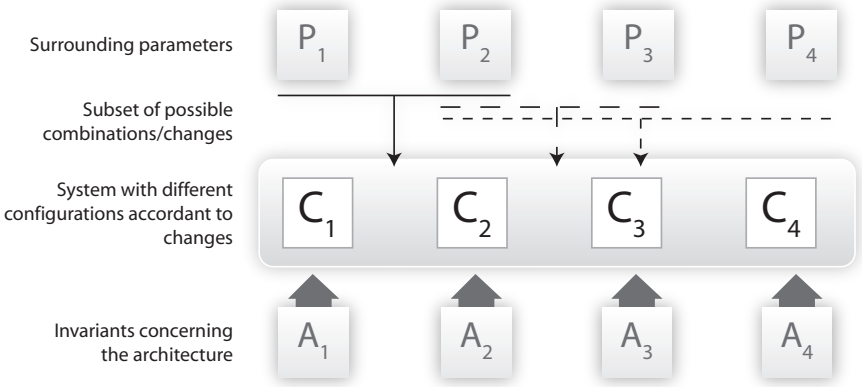

Figure 1: System evolution without architectural constraints. Every C depicts a configuration

sible combination. At this point, architectural constraints show their strength: with architectural constraints as safety net, it is easier to design reconfiguration rules which do not depend on a sequence of reconfigurations or changes in the surrounding.

Our contribution is to provide an architectural constraint framework at the level of code, allowing to check, introduce, and retract architectural constraints, and react to their violation at run-time. Due to the complexity of solving constraint violations, our approach only triggers methods which repair the architecture but does not implement those methods.

We provide this framework in a pervasive and adaptive way; it is pervasive because the approach underlays the whole software application as a safety net and not only at certain execution points. It is adaptive, as a hard-wired approach could not handle the extreme flexible environments of today's applications.

This article presents an approach to ensure application consistency through architectural constraints. We have implemented our approach to architectural constraints as an internal domain-specific language to Java on top of an OSGi-based component model [SZH08]. A DSL, or special purpose language (SPL) [Spi01], is a programming language focused on a very specific domain and lacks advanced control structures like loops [MHS05, Fow09b]. Typically, a DSL assists a General Purpose Language (GPL) like Java or C. An internal DSL uses its host language, encapsulates specific functionality and can be understand as a kind of library.

The remainder of this paper is organized as follows. Section 2 discusses the application area and background of our approach. In section 3 we present an overview of our approach and in section 4 we provide conceptual and implementation details. Finally, we discuss related work in section 5 and conclude in section 6.

2 Background

As we are moving on from desktop computers to a pervasive computing intelligence interwoven in the "fabric of everyday life", our interactions with computers are becoming more complex; new interaction schemes between humans and computers must be invented, as are interaction mechanisms between the devices making up our new ever-changing environment.

At the same time, these transformations also bring new opportunities for radically new ways of interactions between humans and computers. IT-systems are not only becoming capable of discovering our physical context in terms of location and time; as computer systems are getting closer to us, the possibility arises to measure and influence our physical, emotional and cognitive context by measuring responses of our body through sensors and cameras, and reacting to them [SZH08]. Through this interaction, a biocybernetic loop [SF09] can be created for the benefit of the user. A biocybernetic loop is characterized by continuous interaction and mutual influence between the user and the IT-system. By using wearable sensors and implicit channels, the IT-system can adapt the environment to the user's needs and create a more productive, more comfortable or more suitable environment for the user.

3 Idea

To create such a biocybernetic loop, a flexible software infrastructure is needed, as it has to handle virtually every condition of daily live. At the same time, the software has to be extremely stable and reliable, because of its deep and pervasive integration in everyone's business.

From the flexibility requirements, it can be deduced that it is much more harder to guarantee reliability and determinism. The software must react to an extremely wide range of constellations, some of which being even unknown at the point of design or implementation.

We believe that for a clean separation of concerns the architecture of the software must be brought into focus: verifying the system's integrity and repairing it should be done on the level of its architecture. As said before our approach does not cope with the repairing process, only with its invocation.

Thus, instead of making tools for verification of rules easier for the developer or optimizing existing case-related concepts, the aim of our approach is to find a new conceptual structure for the definition and verification of architectural rules. With this new concept, it should be possible to define generic rules with little code and to be independent from a special chain of events. Therefore, our approach shifts from a case-related treatment to a more declarative level, namely to the architecture of the whole IT-system, to the architectural surrounding of a component, and to its interactions with this surrounding.

In order to reduce the necessary amount of code complexity - for instance nested if-then-else blocks - and to meet the mentioned goals, our approach creates a security net at the level of architecture. We use the phrase "security net" because there are a lot of parallels to artists: the verification process happens outside of the main action (coding the application), the security net is the last instance before crashing and its effect does not depend on the concrete code. More details to the analogy can be found in section 4.1.

3.1 Automotive example

The aim of our approach is shown with a running example from the automotive domain. In this example, the noise (music, telephone call etc.) inside a car should be continuously adapted to the stress level of the driver, the endangerment of the situation, and sources of noise and distraction; incoming telephone calls and music should be managed by the pervasive adaptive system. This

system is an extension of the personalized affective music player as proposed in [JBW09]. The structure and architectural constraints of the example are shown in figure 2.

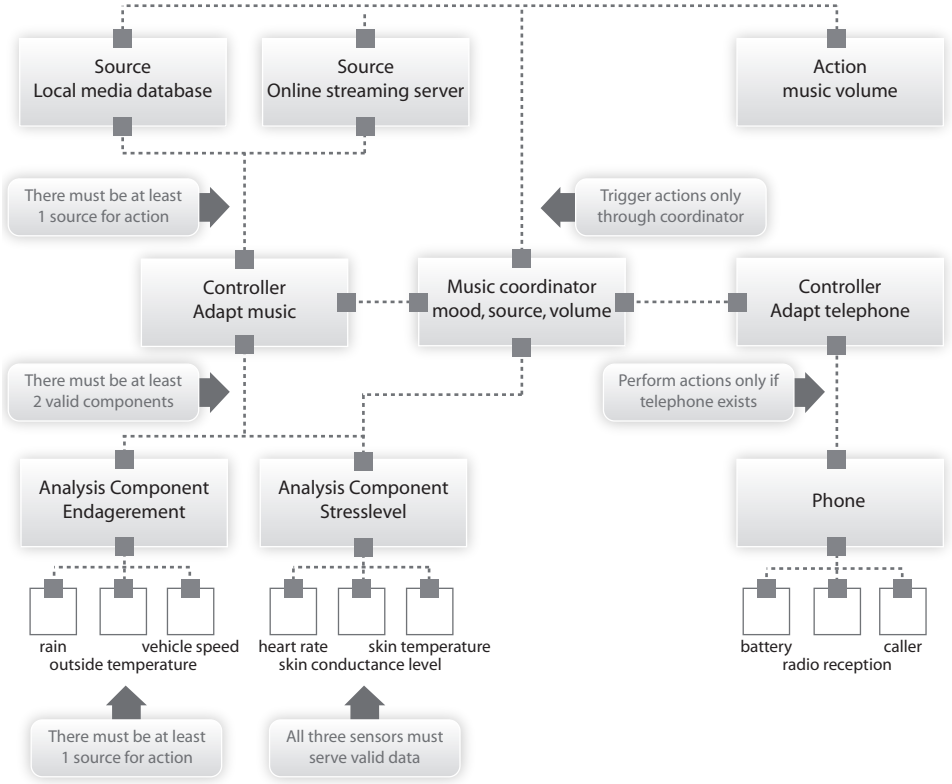

Figure 2: Adapt the noise to the stress level of the driver

In this example, three different types of constraints can be identified: `HasState`, `Connected` and `Exists`. The stress level of the driver is detected by an analysis component connected with physiological sensors like skin conductance and ECG producing physiological features like skin conductance level, skin conductance response and heart rate variability. The analysis component has the state `active` (`HasState`) and the status property set to `operational`, indicating that all sensors work properly and serve valid data. The endangerment of the situation is detected by an analysis component as well. If one of these analysis components does not run properly – the state is not `active` or the status property has not the value `operational` – the noise level in the car is not managed by the pervasive adaptive system. In this situation, both analysis components can trigger callback methods which ”repair” the architecture and perform needed adaptations.

The controller component `Adapt music` is connected to both analysis components and receives data from them (`Connected`). Only with at least one music source connected, the controller performs actions. If there are no music sources, the music cannot be adapted. With the help of an architectural constraint, the controller determines whether the current source is a local

media database or an online streaming server. This is done by the combination of two architectural constraints: `Exists` and `Connected` . If it is a local media database, it gets the value `canChangeToTargetMusicMood` for the mood `relax` from it. This value is `true` if there are songs in the database with a more relaxing influence than the current one, and the music mood has not been changed within the last five minutes. If so, the coordinator component requests the music coordinator to change the mood of the music. If the mood cannot be changed or the current source is an online streaming server, the music controller request the music coordinator to decrease the music volume.

The action of adapting the music can only be achieved through the component music coordinator (`Exists`, `Connected`). The music coordinator guarantees that the most important request is served first and that the driver experiences a coherent behavior of the system. If for instance, the controller `Adapt telephone` requests the coordinator to forward a call to the driver and at the same time decrease the music volume for this, it will only be carried out if the stress level of the driver is under a certain limit or if the calling person is member of a vip-group, e.g. family.

The example shows only one of many different scenarios. Another possible scenario would be a very tired or inattentive driver. In this case, the music should not relax the driver but keep him attentive.

4 Details

4.1 System-wide security net

With the help of architectural constraints, a system wide security net can be established that ensures the architectural integrity after every reconfiguration or when manually triggered. However, only detecting violations of the architectural integrity is often unsatisfactory. Therefore, our architectural constraint system allows to specify arbitrary reactions to the violation of constraints – allowing in particular the developer to specify repairing actions to be undertaken in-place with the architectural constraint.

Using a security net has several benefits. First and most obviously the software is less error-prone. Whenever a constraint is violated, the security net avoids (fatal) runtime errors as architectural constraints are checked. Second, the code is easier to maintain because is not blown up by lines over lines of code to validate the architecture: instead of a procedural way, where multiple lines of code for every single case are needed, our concept has a more declarative approach. This declarative approach supports thinking about the IT-system as an entity. As one constraint can handle several different cases, the code is more generic and errors are not concealed in a special case deep inside the code. Third, even conditions which are not considered at design-time and during implementation can be handled due to our generic approach. The generic treatment at architectural level comes into play as well. Instead of defining rules depending on a specific configuration, every component define its very own requirements on the architecture. Therefore, the rule does not depend on a specific case but it defines the requirements of a component on the architecture for every case. Of course, there can be conflicts between two or more constraints, for instance one constraint postulates component A to exist, another constraints postulates A not to exist. This issue is discussed as future work in section 6.

Altogether, allowing the developer to specify a system-wide security net increases the system's

reliability and leads to a more deterministic behavior of the system.

4.2 Support for reuse

Software components are intended to be reusable entities and help to reduce multiple and redundant code development. This can only be achieved if they can be re-used in multiple environments. Our generic approach allows just this: instead of focussing on correctly assembling a single system, the use of architectural constraints guides developers towards thinking in terms of the structural requirements of components and of architectural constraints that the system must satisfy for the software component to operate correctly. Hence, using architectural constraints – and in particular our approach – is a supporting factor for component reusability.

4.3 Constraint types and structure

Figure 3 shows the main structure of the definable constraints. As visible in the UML diagram, there are four main constraint types and their corresponding negations plus a preparing constraint.

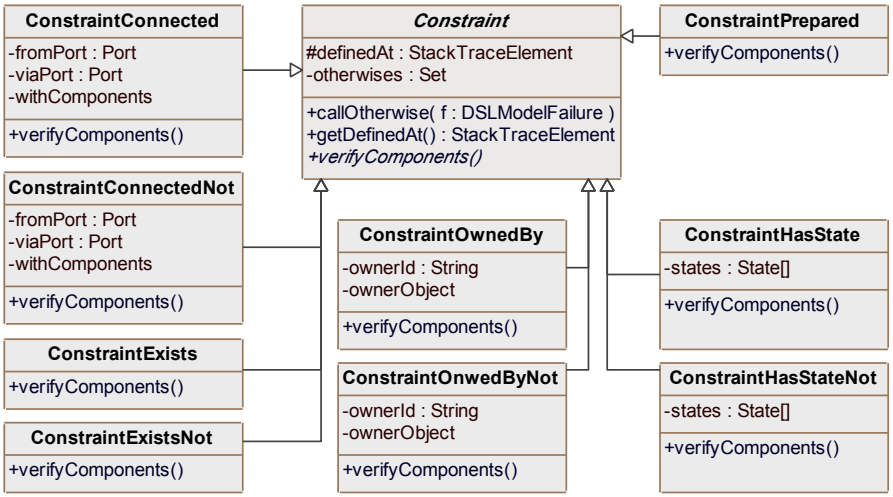

Figure 3: Constraint types

By default, every constraint tracks the line of code defining the constraint and the line of code where it failed. In this way, the developer gets the information needed to start debugging and fixing constraint violations.

Due to the fact that every constraint has its own Java-class and the encapsulation of the error information into a `DSLModelError`-object (section 4.5.1), the collected information of a violation can be extended very easily if needed and the line number can be seen as starting point for debugging. In addition, it is very easy to extend our approach with own constraint types. Other concepts, like automatic search and observe changing points, are less or even not flexible or extensible.

Details on callback method invocation and prepared constraints are given in section 4.5.1 and section 4.5.2, respectively.

4.4 Code example

This section contains a code example with every construct incorporated by our approach.

```
public class Manager {

    public void stresslevelWrongStatus(DSLModelFailure error) {
        System.out.println(error.getMessage());
        System.out.println("Defined " + error.getDefinedAt());
    }

    public void exceptionHandler(DSLModelException e) {
        System.out.println("Defined "
            + dslException.getFailureDetails().getDefinedAt());
    }

    public Manager() {

      check().get(id("stresslevel")).exists().otherwise(fail()).
          hasState(ACTIVE)).
            otherwise(call(this,"stresslevelWrongStatus"));

      check().let("controllerExists").get(id("musicController")).
          exists().otherwise(fail());

      check(this,"exceptionHandler").get(id("musicCoordinator")).
          exists().otherwise(fail()).check("controllerExists");
    }

    public void validateConstraints() { check().validate(); }
}
```

The method `stresslevelWrongStatus()` is an individual callback method that is invoked whenever the stress level constraint validation fails. The `DSLModelFailure`-parameter encapsulates information about the failure, e.g. where the constraint was defined. As mentioned above, the `DSLModelFailure`-object can be extended if needed.

In addition to callback methods, exception handlers can be defined, as seen in the definition of `exceptionHandler()`. This method handles exceptions thrown by the DSL, e.g. thrown on the usage of an undefined prepared constraint or when an unhandled constraint fails (`fail()` in the `otherwise()` call).

The constraints are defined in the class constructor `createConstraints()`. The first constraint gets all components with the id `stresslevel`, checks their existence, and throws

an exception (indicated by `fail()`) if none exists. As there is no component to detect the stress level of the driver, the noise cannot be adapted. If at least one stress level component exists, the constraint checks its state (done by `hasState()`). If one of the found components has a state different from `ACTIVE`, the callback method `stresslevelWrongStatus()` is invoked. Note that the state that is checked by a state constraint is abstract, i.e. it does not refer to the concrete data state, but instead to an abstract representation of state that can be inspected by external entities.

The next constraint definition prepares a constraint that can be referred to through its id `controllerExists`. It is only prepared because the existence of a music controller is only important in a certain context, for instance whenever the stress level of the driver and the endangerment of the situation can be detected or if a music coordinator exists. Hence, there's no need to validate the constraint immediately.

The last constraint states that the specified exception handler should be used instead of the default handler. It gets all components with the id `musicCoordinator`, fails if none exists and executes the prepared constraint `controllerExists` otherwise: the controller works as expected only if a coordinator exists.

In a productive IT-system, the callback methods and exception handlers would contain functionality to adapt the architecture instead of simple output invocations. Aware of the scope of our approach, we forsweared such functionality in our example.

4.5 Toolbox of our approach

4.5.1 Validation, error handling and individual method calls

Whenever the validation of an architectural constraint fails, it should be as easy as possible to define custom reaction schemes. Constraining the possibilities of reacting to violations would advance the use of workarounds, as there are many ways for reacting to violations. Of course, repairing the system in case of a violation is the most desired reaction, and already this can be performed in several ways: sending a special message to a software component, changing a property of a single component or reconfiguring a part of the system is only a subset of possible reparation activities. In addition to this, reporting the error and deactivating or restarting affected parts of the system may also be an option for reacting to architectural constraint violation. In that respect, constraint violation handling is very similar to exception handling: it needs the power of a general purpose language.

Therefore, we allow the developer to specify a set of violation handling methods to be called, and have implemented a default behavior for reacting to violations. The built-in default handling can be called with the method `failed()`. In this case, an exception is thrown and the following error message is written down in the console:

```
The component stresslevel does not exist
Defined at ...Manager.<constructor>(Manager.java:66)
Verified at ...Manager.validateConstraints(Manager.java:82)
```

In modern IDEs, `Manager.java:66` and `Manager.java:82` are linked to the respective lines of code, so the developer can easily jump with one click to the point of interest and is

assisted to fix the constraint validation failure.

The second possibility is to define one or more callback methods which are called when a constraint cannot be validated. Several callback methods can be specified for every single constraint as shown in the following example:

```
check().get(id("componentId")).exists().otherwise(
        call(this,"failureCallback"),
        call(demoInstance,"foo", "bar"));
```

The defined constraint fails if no component with the id `componentId` exists. In this case, all the methods referred in the `otherwise()` block are invoked. First, the method `failureCallback()` of the current object is invoked. Next, the method `foo("bar")` owned by the object `demoInstance` is invoked, receiving `"bar"` as parameter. The advantage of this implementation is that there are no container methods needed to collect different callback invocations. Instead, every method can be passed on its own, including all its parameters.

The last possibility is to define an own handler for exceptions thrown during the validation process. This handler gets the thrown exception as parameter. This exception contains a DSLModelError-object which contains all the important information about the error.

4.5.2 Prepared constraints and context-dependent verification

Sometimes a constraint should not be validated directly but either later or bound to a special event or context. It could be later if all constraints are defined in one central place and the validation is triggered later in a specific method. It could be bound to a specific context if a constraint should only be validated if another constraint was positive (or negative) validated.

For those situations, we added prepared constraint to our approach. A prepared constraint is defined like any other constraint with an additional id. The id is unique in the whole security net and can be referred from any place. Prepared constraints can be called via the `check(id)` method by specifying the constraint id.

4.5.3 Easy integration and usage

The best concept is useless if it is not reasonable to handle in practice. To support the usage of a new concept, there are two critical points: first, it must be easy to install. Our approach is implemented as internal DSL which means that it consists of a library without the need for additional parsers, compilers or runtime environments, since an internal DSL uses the host language itself.

Second, it should be easy to use. When new concepts are established, the main problem is that there is no time to learn how to use it and its new syntax, semantics and rules. To bring those demands to a minimum, our approach comes with a fluent interface, which means that all the architectural constraints can be defined as sentences, making them easy to read and understand.

4.5.4 Fluent interface

One advantage of our approach is the intuitive definition of architectural constraints through a fluent interface [Fow09a]. Instead of writing single lines of code, a whole sentence can be

created.

```
// common
Foo f = new Foo();
List<Object> list = f.getObjectsForPredicate("foo");
f.workWithList(list);
// fluentInterface
f.getList("foo").andHandle();
```

Compared to the common way of coding (commandQuery-API), where the meaning of a method must be inferred from its name without any context, in a fluent interface, the meaning of a method is defined by its position in the sentence. The meaning of a code block can be read as a sentence, which reduces the need to look up method documentation in a programming interface guide.

5 Related work

Applying architectural constraints to evolving component-based systems is everything but new; approaches to architectural constraints range back to the mid nineties [MK96, Bal96]. However, modern approaches to architectural constraints focus on more specific topics like model-driven development [KRG08], which propose using architecture models and variation points specified in UML for designing adaptive applications. Here, architectural constraints have the primary role to reduce the combinatorial explosion in the number of variants.

Other more recent component-based approaches similar to our approach are ArchJava [ACN02] and Rainbow [GSC09]. While ArchJava focuses on preventing architectural erosion on the implementation level, reconfigurations and the entailed need for run-time checking of system instances are not considered. The Rainbow framework is very similar to ours in technical details, and its capabilities goes even those of our constraint framework, but has a different focus. Rainbow provides means to realize adaptation logic, while we propose to use architectural constraints as safety net under such adaptive behavior; our methodological perspective is hence different.

Others, like [BLMT08] focus on formal analysis of software architectural constraints and system reconfigurations. In the former work a graphical and textual notation is used which is inspired by hypergraphs and term rewriting. The approach of Bruni et al. allows to statically verify that a component-based system under reconfiguration will always respect the architectural constraints imposed on the system.

All approaches so far introduced formal notations [BLMT08], UML profiles [KRG08, Fos09] or own programming languages [MK96, BJC05] for the specification of architectural constraints. Our approach differs from the existing ones in that respect. By using the host language that also contains the full application logic, architectural constraints can be specified and verified within the standard development environment, leveraging the full power of the Java programming language and Java IDEs. Hence, our approach takes a more pragmatic view on the use of architectural constraints, purposefully trading formal verification possibilities for usability.

6 Conclusion and future work

This paper presented an architectural constraint framework that is intended to be used to improve reliability and reusability in pervasive adaptive systems [SZH08]. In particular, it gives insights to the design rationale of our constraint framework based on a fluent-interface style internal DSL. The design of the domain specific language was driven by the idea of reducing code redundancy, promoting clear separation of concerns and allowing a concise specification of architectural constraints without the use of boilerplate code.

The goal of our approach is twofold, namely a) to shift the focus from the development single systems to the development of single reusable components in multiple scenarios. This is achieved by allowing the developer specifying architectural requirements for the use of components. Furthermore our goal is b) to increase the flexibility and adaptivity of component-based software systems by allowing to dynamically react to constraint violations.

The architectural constraint framework presented lives at the level of code, allowing to check, introduce, and retract architectural constraints, and react to their violation at run-time.

During the implementation and use of the constraint framework, future work on architectural constraints has been identified. Firstly, allowing every component to specify architectural constraints may lead to an inconsistent constraint base; repair actions trying to re-establish a subset of constraints would immediately invalidate other constraints. The detection of inconsistent constraint sets is therefore one extension for our framework. Additionally, it is worthwhile to investigate how autonomous self-repairing actions based on constraint-solving techniques can be of use in the domain of architectural constraints, and how to control their interference with developer-specified repairing actions. A solution to both inconsistent constraints and interactions between autonomous self-repair and explicit repair actions could be the use of soft constraints, in which a priority can be specified to architectural constraints.

Acknowledgements: This work has been partially sponsored by the EC project REFLECT, IST-2007-215893.

Bibliography

[ACN02] J. Aldrich, C. Chambers, D. Notkin. ArchJava: connecting software architecture to implementation. In *ICSE*. Pp. 187–197. ACM, 2002.

[Bal96] R. Balzer. Enforcing architecture constraints. In *2nd Int. Soft. Architecture Wshp. and Int. Wshp. on Multiple Perspectives in Soft. Dev.* Pp. 80–82. ACM, 1996.

[BJC05] T. V. Batista, A. Joolia, G. Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. In *EWSA*. LNCS 3527, pp. 1–17. Springer, 2005.

[BLMT08] R. Bruni, A. L. Lafuente, U. Montanari, E. Tuosto. Style-Based Architectural Reconfigurations. *Bulletin of the EATCS* 94, 2008.

[Fos09] H. Foster. Architecture and behaviour analysis for engineering Service Modes. In *Wshp. on Principles of Engineering Service Oriented Systems*. Pp. 1–8. IEEE, 2009.

[Fow09a] M. Fowler. Domain Specific Languages - Expression Builder. *Domain Specific Languages*, 2009.
http://www.martinfowler.com/dslwip/ExpressionBuilder.html

[Fow09b] M. Fowler. Domain Specific Languages - Intro. *Domain Specific Languages*, 2009.
http://martinfowler.com/dslwip/Intro.html#ProgrammingMissGrantsController

[GSC09] D. Garlan, B. Schmerl, S.-W. Cheng. *Software Architecture-Based Self-Adaptation*. Volume ISBN 978-0-387-89827-8, pp. 31–55. Springer, 2009.

[JBW09] J. H. Janssen, E. L. v. d. Broek, J. H. Westerink. Personalized affective music player. In *2009 Int. Conf. on Affective Comp. and Intelligent Interaction*. Pp. 472–477. IEEE, 2009.

[KM90] J. Kramer, J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Soft. Eng.* 16:1293–1306, 1990.

[KRG08] M. U. Khan, R. Reichle, K. Geihs. Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications. *IEEE Distributed Systems Online* 9(7):1, 2008.

[MHS05] M. Mernik, J. Heering, M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Comp. Surveys* 37:316–344, 2005.

[MK96] J. Magee, J. Kramer. Dynamic Structure in Software Architectures. In *SIGSOFT FSE*. Pp. 3–14. ACM, 1996.

[OMT98] P. Oreizy, N. Medvidovic, R. N. Taylor. Architecture-Based Runtime Software Evolution. In *ICSE*. Pp. 177–186. IEEE, 1998.

[SF09] N. B. Serbedzija, S. H. Fairclough. Biocybernetic loop: from awareness to evolution. In *11th Congress on Evolutionary Comp.* Pp. 2063–2069. IEEE Press, 2009.

[Spi01] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software* 56(1):91–99, 2001.

[SZH08] A. Schroeder, M. v. d. Zwaag, M. Hammer. A Middleware Architecture for Human-Centred Pervasive Adaptive Applications. In *2008 2nd IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems Wshp.* Pp. 138–143. IEEE, 2008.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.