Dynamic Architecture-Implementation Mapping for Architecture-Based Runtime Software Adaptation

Cuong Cu[1], Rachel Culver[2], and Yongjie Zheng[2]
[1]CyberSource Corporation, Austin, Texas, USA
[2]Department of Computer Science and Information Systems, California State University San Marcos, USA
csc823@gmail.com, culve005@cougars.csusm.edu, yzheng@csusm.edu

***Abstract*—Architecture-based software adaptation is a promising method that adapts a software system by evolving its architectural model, which is generally easier to understand and manipulate than source code. The wide-scale practice of the method requires an approach to automatically mapping adaptive changes that are planned and deployed in the architecture to modifications of running code. This involves two main challenges: maintaining architecture-implementation conformance and dynamic software updating. Existing approaches fail to address them simultaneously to enable architecture-based adaptation. This paper presents a novel approach combining an architectural variability implementation mechanism with an architecture framework. The approach automatically updates both source code and running code during architectural evolution. As an initial assessment, we applied the approach to the adaptation of a chat application.**

Keywords—software architecture, architecture-implementation conformance, software evolution

I. INTRODUCTION

A self-adaptive software [15] modifies its own behavior in response to changes in its operating environment, such as end-user input, external hardware devices and sensors, or program instrumentation. *Architecture-based adaptation* [5, 12, 17] is an important result from software architecture research. A software system's architecture is the set of principal design decisions made about the system [19]. It is commonly modeled as a configuration of components connected via interfaces, using an architecture description language (ADL). Architectural models do not contain implementation details and generally are easier to understand and manipulate than source code.

Figure 1 shows an existing infrastructure of architecture-based software adaptation that this research aims to support. It separates adaptation activities into two simultaneous processes: *adaptation management* (the upper half) and *evolution management* (the lower half). Adaptation management monitors and evaluates the application and its operating environment, plans adaptation, and deploys change descriptions in architectural terms (e.g., replacing a component) to the running application. Evolution management is responsible for evolving the application by mapping the deployed architectural changes to modifications of the application's running implementation, while ensuring runtime conformance between the architecture and implementation. This is our focus in this project, and it primarily involves the following two challenges.

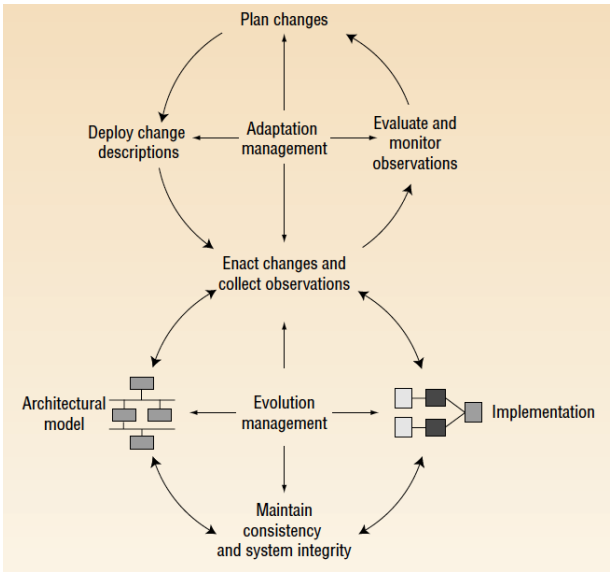

Fig. 1. An infrastructure of architecture-based runtime adaptation [17].

- *Mapping changes in the architecture to automatic updating of source code*. Software architecture may be frequently changed during architecture-based adaptation: a component may be replaced by another component, and a new interface may be added to a component. These changes affect the code in various ways and at different degrees of granularity [4]. Existing architecture-centric approaches often solely rely on code generation [6] to automatically update the code. This is not sufficient because the manually developed code (i.e., user-defined code) also exists and is often mixed with generated code. It is difficult under this circumstance either to protect user-defined code from being overwritten or to update user-defined code accordingly.
- *Dynamically modifying running code after source code is updated*. This is essentially a problem of dynamic software updating [13] that involves several issues, such as swapping code and transferring application state (e.g., the current values of program variables) to new code. None of the existing architecture-implementation mapping approaches [11, 14, 16, 22] addresses these issues. Existing dynamic software updating techniques [1, 10, 13] are mainly for

DOI reference number: 10.18293/SEKE2020-114

language-level program adaptation. They typically require extension of existing programming languages or the use of a third-party middleware. These requirements may add to the complexity of implementing architectures and thus are not appropriate for architecture-based runtime adaptation.

In this paper, we present an architecture-implementation mapping approach to supporting architecture-based runtime adaptation shown in Figure 1. An important insight that we have in this research is that the architectural elements that are planned/anticipated to be changed (i.e., *variable*) should be implemented differently from the stable or core elements. Their implementations should be either loosely bound to the rest of the system (e.g., in a separate module) or can be easily identified and updated (e.g., via code annotations). This opens up the opportunity of automatic updating of source code (including user-defined code) when the variable architectural elements are changed as planned during architectural adaptation.

The first contribution of our approach is a novel source code model that implements different kinds of variable architectural elements in specific ways. It extends an existing implementation model that we developed for product line architecture [21, 22], which also involves architectural variability. The original model decouples the generated code and user-defined code of each architecture component into independent code modules (e.g., classes). Our approach further divides the user-defined code into modules that implement the component's main logic and variable interfaces respectively. Moreover, our approach uses an annotative technique in the user-defined code to indicate code fragments (e.g., a single line of code) corresponding to variable architectural elements. When the architecture is changed, our approach automatically updates both generated code (via code regeneration) and user-defined code (via annotation processing) to maintain architecture-implementation conformance.

Additionally, our approach includes a novel software framework named *DynaMyx* that automatically updates running code of a component when its source code is changed (e.g., as a result of mapping architectural changes to source code described above). DynaMyx extends an existing architecture framework, *Myx* [9], which provides built-in implementations (e.g., APIs, abstract classes) for implementing architectures. On top of that, DynaMyx includes modules that encapsulate the logic (e.g., transfer state, swap code) of dynamic software updating from the overlying application. This allows the developer to focus on application-specific logic, while the DynaMyx framework automatically monitors source code, detects its changes, reloads the changed code, and migrates the runtime state to the new code without stopping the running code.

We implemented a prototype of the approach in ArchStudio [2], an Eclipse-based architecture development platform. As an initial assessment, we applied the approach to the adaptation of a chat application that has an explicit architectural model. We changed its architecture while the application was running, and observed that the running code was dynamically updated with our approach. After that, we inspected both the source code and the application's behavior (e.g., functions, runtime data) to assess whether the application functions appropriately and whether its architecture and running code are consistent. We created a video demo [7] to illustrate this process.

II. Approach

Figure 2 provides an overview of our approach. The rectangle at the top represents different kinds of variable architectural elements that the approach supports, including replacement of a component, replacement of an interface, and addition of an interface. We use an existing architectural modeling approach and tool called *ArchFeature* [3] that we developed in a prior project. ArchFeature supports modeling and evolution of architectural variations using an existing XML-based ADL, xADL [8]. Our focus in this project is on mapping of architectural changes to both source code and running code. The gray boxes in Figure 2 represent two main contributions of the approach: (1) a source code model (supported by a code generator and an annotation processor) that regulates the implementation of an architecture component to enable automatic modifications of source code; (2) the DynaMyx framework that extends the Myx framework as mentioned in Section I with the capability of automatic updating of running code. Each is introduced in the following subsections.

A. Architectural Variability Implementation

Our approach includes a novel source code model combining code generation, code separation, and an annotative technique in the implementation of an architecture component. As shown in Figure 2, the model divides a component's source code into the following three independent modules and uses a program composition mechanism (e.g., method delegation) to integrate the separated code. This enables a separation of decision space within the implementation of each component and offers a novel way to implement different variations in the architecture.

Generated Code: a module that is generated from the component's architectural specification. It contains routine implementation of the externally visible information (e.g.,

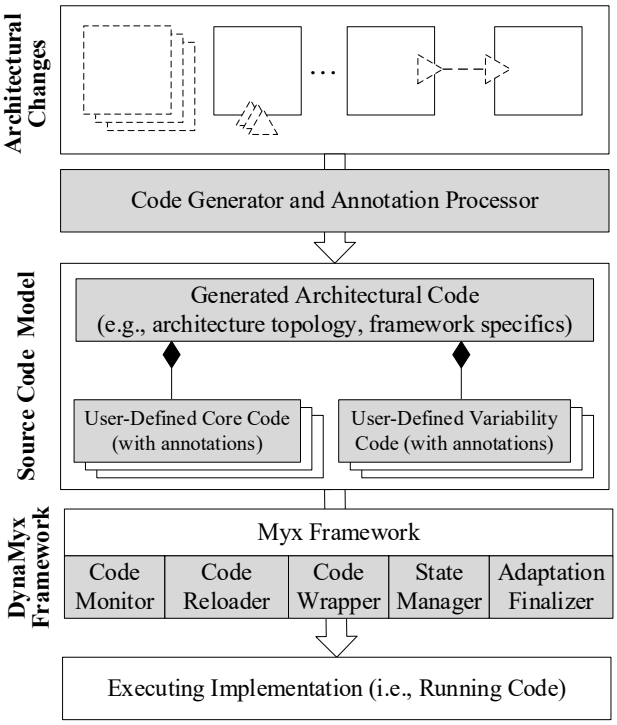

Fig. 2. Approach overview.

interfaces) of the component. The generated code encapsulates knowledge about architecture topology and related variations (e.g., implementation of an optional interface). It does not need or allow manual modification, and implements the application-specific methods (e.g., methods defined in the component's interfaces) by redirecting request to a separate module (i.e., user-defined code modules explained below), where the methods are manually implemented.

User-Defined Core Code: a module that contains manually developed implementation details of the component's main logic. It implements a program interface including the methods that generated code needs the programmer to develop. The user-defined core code represents the internal implementation of the component and encapsulates implementation-specific concerns (e.g., use of code libraries and algorithms). It addresses related architectural variations (e.g., replacement of a component for a different implementation) by switching between alternative user-defined modules, which are represented by the overlapping boxes in Figure 2.

User-Defined Variability Code: a manual module that is separated from the user-defined core code above. It contains implementation details of a construct (e.g., an optional interface) that can vary independently of the component. This reduces the impact of the variation (e.g., inclusion/exclusion of the construct) on the rest of the component's user-defined code. Similar to the core module, the variability module implements a program interface, which only includes methods specific to the construct. A library of variability modules containing different implementation mechanisms may also exist.

Our approach also includes an *architecture-based code annotation* technique that is used in the user-defined code modules described above to indicate optional *fine-grained* code fragments (e.g., a method, a line of code), which may be added or removed corresponding to the adaptive changes made to the architecture. An annotation is defined as a Java annotation (i.e., @Optional) wrapped by a block comment (i.e., /*…*/) as shown below. It contains the name(s) of the feature(s) that the annotated code is related to. Each feature is represented as a predefined value of a Java enum named *Feature* (i.e., *Feature.{feature-name}*). In particular, the *Feature* enum is generated from the architecture (hence architecture-based) and includes the names of all the features that are related to the corresponding component in the architecture. Only the included names can appear in an annotation used in the component's code. In this way, the programmer does not need to manually type in a feature name. When the architecture is changed, the related code fragments and annotations are automatically updated by the annotation processor shown in Figure 2.

```
/*@Optional(Feature.{feature-name}, …)*/
```

Figure 3a shows an architecture example of a text-based chat application. The architecture has four components (i.e., rectangles) that are connected via interfaces (i.e., triangles). All the elements drawn using dashed lines are variable for three features: sending system messages (e.g., a smiley face), saving chat history using different mechanisms (e.g., file system, database), and sharing files. Figure 3a also shows an example of variability specification in the xADL language. It defines an optional interface of Component *Server* for *FileSharing*.

Fig. 3. (a) Architecture example of a chat application; (b) Code example of Component *Server*.

Figure 3b shows the code example of Component *Server* implemented using our approach. The generated code (Class *ServerArch*) includes references to user-defined modules (Lines 2-3), references to connected components (Lines 4-5), lifecycle methods and APIs required by the DynaMyx framework (introduced in the following subsection), and application-specific methods (Lines 12-17) that are implemented by calling the corresponding user-defined module. The user-defined core code (Class *ServerImp*) and variability code (Class *FSImp*) each implements a program interface (*IServer* and *IFileSharing*) that is also generated and includes the methods to be manually developed. The user-defined code modules contain architecture-based annotations that are attached to optional code fragments (Line 6 of Class *ServerImp*). Note that the user-defined code may call the methods of other components via its generated code (e.g., Lines 4 and 6 of Class *ServerImp*).

B. DynaMyx Framework

Myx is an existing architecture framework written in the Java programming language. It includes a set of modules as built-in implementations of key architectural elements, which are used to develop an architecture-based application. Myx also encapsulates the logic for bootstrapping the application and regulating interactions (e.g., method calls) between components. This allows application developers to focus on developing application-specific logic. DynaMyx inherits these capabilities (i.e., supporting architecture implementation) from Myx. In particular, DynaMyx maintains Myx's interface (e.g., APIs and lifecycle methods that are underlined in Figure 3b) to the overlying application. All the applications originally built on Myx can still be correctly executed with DynaMyx. This is reflected in Figure 2 as all the DynaMyx modules are underneath Myx and are invisible to the application code above.

DynaMyx extends Myx with the capability of dynamic software updating and hides the related complexity from implementing architectures. This represents a novel and promising approach to supporting architecture-based runtime adaptation. DynaMyx includes five new modules: *Code Monitor*, *Code Reloader*, *Code Wrapper*, *State Manager*, and *Adaptation Finalizer* as shown in Figure 2. It automatically detects source file changes, complies the program from the changed source files, starts it up alongside the old program, transmits its state to the new program, and finally swaps the initialized new program with the old program. The entire process consists of the following five steps.

Step 1 – *Detect code changes*. DynaMyx works with an program development tool (e.g., Eclipse) that automatically complies source code when it is changed. The Code Monitor module of DynaMyx monitors the modified dates of every component's compiled code files. If a change is detected in a component, Code Monitor automatically triggers the steps below to update the component's running code. Other components are not affected during this process.

Step 2 – *Reload changed code into the running syst*em. The Code Reloader module includes a dedicated code loader, which enforces reloading of a modified code file into the system (e.g., Java Virtual Machine). The code reloading will occur if the component is inactive and is not communicating with other components. This is determined based on Myx's capability of managing component communications as mentioned earlier.

Step 3 – *Create new instances from reloaded code*. The new code instances are not bound to the system at this point.

Step 4 – *Transfer application state to new code instances and initialize them afterwards*. State Manager processes user-defined code to transfer state for the updated components. During this process, it bypasses security scope, especially the private and protected scopes, to access and copy state from old stances to new instances. In particular, State Manager is able to transfer state in a class hierarchy and support properties that are inherited and defined in a parent class. After that, the Manager will initialize new instances with the transferred state by calling the component's Myx lifecycle methods (e.g., *init* underlined in Figure 3b).

Step 5 – *Swap (i.e., bind) new instances into the running system and discard old instances*. A challenge involved at this point is updating the references of other components to old code instances, which will be swapped out of the memory. It is difficult to detect all the related references as this is essentially a problem of dynamic code analysis. DynaMyx addresses the challenge by wrapping the implementation of each component with Code Wrapper (implemented based on Java Proxy). The Wrapper, instead of the component's code, is referenced by the code of other connected components. The Wrapper serves as a delegate that intercepts and redirects the function calls from the connected components. At the end of the adaptation, the Adaptation Finalizer module updates the Wrapper to refer to new instances of the component's code and bind new instances into the system.

Table 1. Mapping Runtime Architectural Changes to both Source Code and Running Code.

Runtime architectural changes	Mapping to source code	Mapping to running code
Component addition/removal	Regenerate code to include/exclude the component's architectural code.	Load/unload code and create/destroy instance.
Component replacement	Regenerate code to switch to a different user-defined core module.	Reload new code; create new instance; transfer state; swap code.
Provided interface addition/removal	Regenerate code to include/exclude the interface's architectural code.	Load/unload code and create/destroy instance.
Provided interface replacement	Regenerate code to switch to a different user-defined variability module.	Reload new code; create new instance; transfer state; swap code.
Required interface addition/removal	Regenerate code; process annotations and code fragments in user-defined modules.	Reload new code; create new instance; transfer state; swap code.
Connection addition/removal	Regenerate the bootstrapper program.	Reload the bootstrapper program.

C. Mapping Architectural Changes to Code

Table 1 summarizes our approach's capabilities of mapping typical kinds of runtime architectural changes to both source code and running code based on the implementation model and DynaMyx framework presented in this section. It distinguishes a *provided* (input) component interface from a *required* (output) interface. A provided interface contains the methods implemented within the component, while a required interface contains the methods that are implemented by another component and used by the current component. The connection changes are handled by Myx as mentioned in Section II.B.

III. Preliminary Experience

We developed a prototype of the approach in the ArchStudio open-source system as mentioned in Section I. The prototype includes a code generator, an annotation processor, and the DynaMyx framework. The code generator is built using the Eclipse JET code generation engine [6] that follows a template-based code generation paradigm. The code generation templates capture routine implementations of software architecture. The annotation processor is built using the ANTLR parser generator [20]. It automatically identifies and updates code annotations and code fragments corresponding to an architectural change. We integrated these tools with the ArchFeature architectural modeling tool mentioned in Section II. This provides us with a platform where we can assess our approach.

We will consider the architecture-implementation mapping approach presented in this paper successful if (1) it maintains conformance between a software's architecture, its source code, and its running code when the planned architecture changes are deployed; (2) the dynamically-updated running software behaves appropriately (with the updated behavior) and continuously (with the transferred state); (3) the overhead in terms of executing time and memory requirement during adaptation is acceptable. To validate our approach along all these three dimensions, we applied the approach and tools to a chat application.

The chat application was implemented by two Masters students based on the approach presented in this paper. It has a list of features (e.g., *File Sharing*, *Game*, and *Template*) and an explicit architectural model developed using the ArchFeature tool. It has around 15K SLOC, including generated code, user-defined code, and code annotations. The architecture and code are consistent with each other. This was validated using a consistency checking tool of ArchStudio. We assessed the approach by executing the chat application and exercising some of its functions (e.g., chat) to generate runtime state (e.g., chat messages). We then used the ArchFeature tool to change its architecture while the application was running. We exercised different kinds of architectural changes as discussed in the paper, such as component removal and interface addition, which were eventually reflected into the running code by our approach. In the end, we checked architecture-implementation conformance using the tool mentioned above. We also inspected the behavior and application data of the updated chat application to validate whether it still functions correctly.

Figure 4 shows screenshots of the chat application's architectural model (opened in our ArchFeature modeling tool) and user interface (i.e., a chat client window). In one of the experiments, we removed the *Template* feature (selected in the feature list) and its related architectural elements while the application was running. Our approach was then triggered to automatically update the application's source code (via code regeneration and annotation processing) and running code (with DynaMyx) without terminating its execution. When the adaptation was completed, we noticed that a related user interface element (e.g., the button circled in the figure) disappeared since the corresponding code was dynamically removed by our approach. Meanwhile, the application state (e.g., chat messages) was successfully preserved. We created a video demo [7] to illustrate the process described above.

Overall, our approach was able to automatically update both the source code and running code of the chat application when its architecture was changed at runtime. We validated

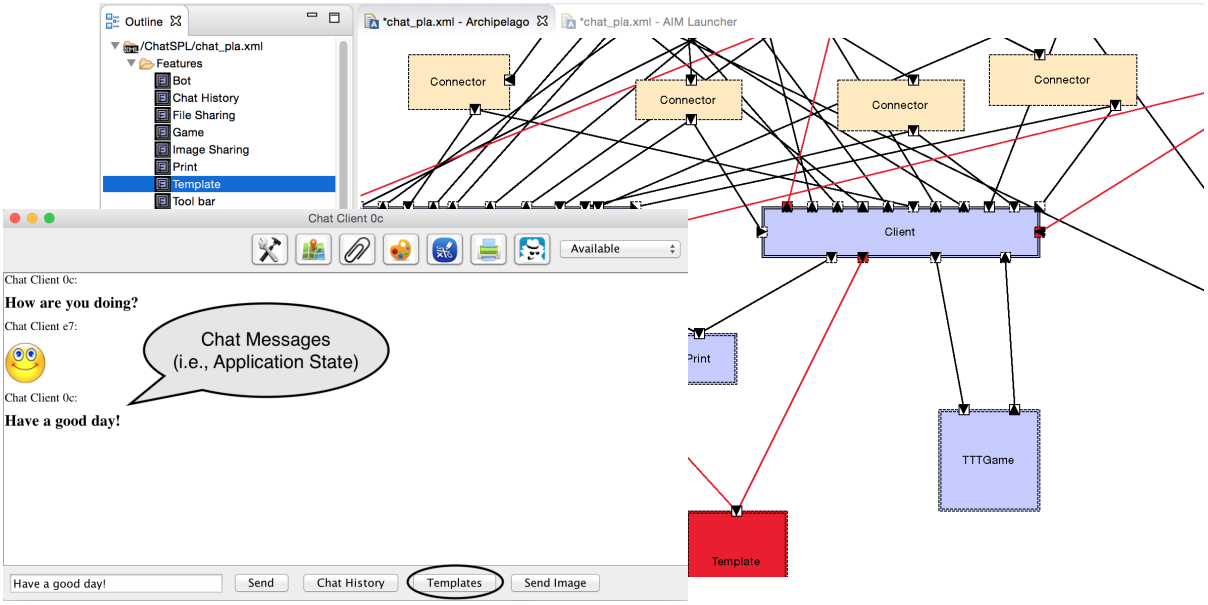

Fig. 4. Architecture-based runtime evolution of a chat application.

conformance between the updated architecture and implementation after each evolutionary operation. The system's new behavior also matched the corresponding architectural changes, and we did not notice any performance degradation during the adaptations. A limitation of DynaMyx that we found is that the new state must be determined and transferred from the existing state. Our approach does not address inferring new state information, which usually requires manual intervention (due to lack of information). For example, it cannot automatically transfer state to a new field added in the new code since this information does not exist in the old code.

IV. Related Work

Several architecture-implementation mapping approaches exist, including programming language design [11], code generation [6], and architecture frameworks [9, 16]. These approaches successfully address the challenge of bridging the abstraction gap between architecture constructs and program elements during the initial development of a software system. They can maintain conformance between the architecture and source code along certain criteria, such as *style conformance* [16], *communication integrity* [11], or *quality concerns* [14]. However, none of them addresses *runtime conformance* between the architecture and running code in architecture-based self-adaptation. Existing architecture frameworks, such as C2 [16] and Myx [9], provide fairly well understood source code that assists developers in implementing systems conforming to an architecture style. They do not support the mapping of architecture changes to code and require an additional mapping approach (e.g., the presented work) to maintain architecture-implementation conformance.

Existing architecture-based runtime adaptation approaches address some important issues in this area, such as adaptation infrastructure [5, 12, 17] and architecture styles [18]. These approaches reveal the benefits of a self-managed software architecture. In terms of mapping architectural changes to running code, they mainly rely on existing architecture-implementation mapping approaches, such as architecture frameworks and code generation, which are not sufficient as described above. For example, existing approaches in this area cannot support architectural changes (e.g., replace an interface) involving the challenges of automatically updating user-defined code and dynamic software updating.

V. Conclusion

This paper presents an approach that maintains runtime conformance between the architecture and running system. This is essential to architecture-based runtime adaptation, but fails to be addressed by the existing approaches of dynamic software updating and architecture-implementation mapping. The approach has two main contributions: (1) a variability-specific architecture implementation approach that enables automatic modifications of source code (e.g., user-defined code) during architectural adaptation, and (2) an architecture framework that encapsulates dynamic software updating mechanisms and enables automatic modifications of running code. The initial assessment reveals that our approach is capable of supporting architecture-based runtime software adaptation. We intend to further evaluate the approach through a long-term study with a large software system in the future.

References

[1] A. Orso, A. Rao and M. J. Harrold, "A technique for dynamic updating of Java software," International Conference on Software Maintenance, 2002. Proceedings., Montreal, Quebec, Canada, 2002, pp. 649-658.

[2] Archstudio. An Architecture-based Development Environment. http://www.isr.uci.edu/projects/archstudio/, Institute for Software Research, University of California, Irvine.

[3] C. Cu, X. Ye, and Y. Zheng. "XLineMapper: a product line feature-architecture-implementation mapping toolset". In Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE 2019). IEEE Press, 87–90. 2019.

[4] D. Garlan, R. Allen and J. Ockerbloom, "Architectural mismatch: why reuse is so hard," in IEEE Software, vol. 12, no. 6, pp. 17-26, Nov. 1995.

[5] D. Garlan, S. Cheng, A. Huang, B. Schmerl and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," in Computer, vol. 37, no. 10, pp. 46-54, Oct. 2004.

[6] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Professional, 2008.

[7] DynaMyx. https://youtu.be/2zCHz6jovX4

[8] E.M. Dashofy, A. van der Hoek, and R.N. Taylor, "A Comprehensive Approach for the Development of Modular Software Architecture Description Languages," ACM Transactions on Software Engineering and Methodology (TOSEM). 14(2), p. 199-245, April, 2005.

[9] E.M. Dashofy, Myx and myx.fw. http://www.isr.uci.edu/projects/archstudio/myx.html.

[10] H. Seifzadeh, H. Abolhassani, and M.S. Moshkenani, "A survey of dynamic software updating," Journal of Software: Evolution and Process, 25(5), 535-568, 2013.

[11] J. Aldrich, C. Chambers and D. Notkin, "ArchJava: connecting software architecture to implementation," Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, Orlando, FL, USA, 2002, pp. 187-197.

[12] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," Future of Software Engineering (FOSE '07), Minneapolis, MN, 2007, pp. 259-268.

[13] M. Hicks, and S. Nettles, "Dynamic Software Updating," ACM Transactions on Programming Languages and Systems (TOPLAS) 27(6), p. 1049-1096, 2005.

[14] M. Mirakhorli and J. Cleland-Huang, "Detecting, Tracing, and Monitoring Architectural Tactics in Code," in IEEE Transactions on Software Engineering, vol. 42, no. 3, pp. 205-220, 1 March 2016.

[15] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," ACM Trans. Auton. Adapt. Syst. 4, 2, Article 14, 42 pages, May 2009.

[16] N. Medvidovic, N.R. Mehta, and M. Mikic-Rakic, "A Family of Software Architecture Implementation Frameworks," In Proceedings of the 3rd IFIP Working International Conference on Software Architectures. Montreal, Canada, August, 2002.

[17] P. Oreizy et al., "An architecture-based approach to self-adaptive software," in IEEE Intelligent Systems and their Applications, vol. 14, no. 3, pp. 54-62, May-June 1999.

[18] R.N. Taylor, N. Medvidovic and P. Oreizy, "Architectural styles for runtime software adaptation," 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, Cambridge, 2009, pp. 171-180.

[19] R.N. Taylor, N. Medvidovic, and E.M. Dashofy, Software Architecture: Foundations, Theory, and Practice. 736 pgs., John Wiley & Sons, 2010.

[20] T. Parr, The ANTLR Parser Generator. http://www.antlr.org/.

[21] Y. Zheng, C. Cu and H. U. Asuncion, "Mapping Features to Source Code through Product Line Architecture: Traceability and Conformance," 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, 2017, pp. 225-234.

[22] Y. Zheng, C. Cu, and R. N. Taylor, "Maintaining Architecture-Implementation Conformance to Support Architecture Centrality: From Single System to Product Line Development," ACM Transactions on Software Engineering and Methodology. 27, 2, Article 8, 52 pages, June 2018.