Runtime Software Architectural Models for Adaptation, Recovery and Evolution

Hassan Gomaa
Dept. of Computer Science
George Mason University
Fairfax, Virginia, USA
hgomaa@gmu.edu

Emad Albassam
Dept. of Computer Science
George Mason University
Fairfax, Virginia, USA
ealbassa@gmu.edu

Daniel A. Menascé
Dept. of Computer Science
George Mason University
Fairfax, Virginia, USA
menasce@gmu.edu

***Abstract*—This paper describes approaches for dynamic software adaptation using runtime models of the software architecture. Software adaptation patterns consist of interaction models and state machine models that are used during dynamic software adaptation. Software adaptation and recovery concerns are off-loaded from components by incorporating them into connectors, which are responsible for dynamically adapting and recovering components. Both centralized and decentralized approaches to adaptation and recovery are considered. Two approaches to dynamic software product lines are described, a dynamic software adaptation approach for service-oriented product lines and the design of variable adaptation and recovery connectors.**

Keywords*—*dynamic software adaptation; autonomic computing; dynamic software product lines; runtime models; recovery and adaptation connectors.

I. INTRODUCTION

A software architectural model describes the design of the software system in terms of components and connectors. Architectural models are frequently developed at design time before the implementation of the software system. However, architectural models can also be used at runtime to enable architecture recovery and architecture adaptation. Runtime architectural models are software models that coexist with the executing software system, such that runtime decisions about dynamic changes to the executing system are made by analyzing the architectural model and then applied to the executing system.

This paper describes approaches for dynamic software adaptation using runtime models of the software architecture for both planned and unplanned adaptation. Planned adaptation is proactive in which manual or automated decisions are made to dynamically change the software system at runtime. Unplanned adaptation is triggered by unexpected events, such as node failure, which necessitate reactive decisions to dynamically recover the software system from failure.

This paper describes and discusses the research conducted in dynamic software adaptation at George Mason University, in particular the research directions taken and why they were taken, the software engineering concepts and technology that the research is based on, the main results of the research, and directions for future research. Section II considers dynamic software adaptation from different perspectives before addressing adaption of software architectures in Section III, comparing in Section IV adaption without knowledge of the software application with adaptation in which the application's software architectural patterns are known. From these patterns, software adaptation patterns are developed consisting of interaction models and state machine models that are used during dynamic software adaptation, as described in Section V.

Next, in Section VI, the paper describes how software adaptation concerns can be separated from component development concerns by incorporating adaptation concerns into adaptation connectors. The paper then describes in Sections VII and VIII how adaptation connectors can be extended to address recovery in a decentralized autonomic MAPE-K approach for self-healing and self-adaptation.

The paper goes on to describe in Section IX, two approaches to dynamic software product lines (SPL). The first approach is a dynamic software adaptation approach for service-oriented product lines. The second approach is the design of variable adaptation and recovery connectors used in the software adaptation of dynamic software product lines (DSPL). In Section X, the paper discusses related work and future directions for dynamic software adaptation.

II. DYNAMIC SOFTWARE ADAPTATION

Software adaptation addresses software systems that need to change their behavior during execution. In self-managed and self-healing systems, systems need to monitor the environment and adapt their behavior in response to changes in the environment [3]. Garlan and Schmerl [1] have proposed an adaptation framework for self-healing systems, which consists of monitoring, analysis/resolution, and adaptation. Kramer and Magee

[2] have described how in an adaptive system, a component needs to transition from an active (operational state) to a quiescent (idle) state before it can be removed from a configuration. Kramer and Magee have also advocated an architectural approach to self-managed systems [5]. Autonomic systems address software adaptation by following the MAPE-K model that consists of four activities (monitoring, analysis, planning, and execution) that operate on a knowledge-base of the system [3].

Software adaptation can take many forms. It is possible to have a self-managed system that adapts the algorithm it executes based on changes it detects in the external environment. If these algorithms are pre-defined, then the system is adaptive but the software structure and architecture is fixed. The situation is more complex if the adaptation necessitates changes to the software structure or architecture. In order to differentiate between these different types of adaptation, adaptations can be classified as follows within the context of distributed component-based software architectures:

a) Algorithmic adaptation. The system dynamically changes its behavior within its existing structure. There is no change to the system structure or architecture.

b) Configuration adaptation. Dynamic adaptation involves dynamically relocating components of the architecture to different nodes of a distributed configuration, for example if a node fails. The dynamic replacement of a failed component with a replacement component has to be performed while the system is executing.

c) Architectural adaptation. The software architecture has to be modified as a result of the dynamic adaptation. Old component(s), which may not provide the same interface, must be dynamically replaced by new component(s) while the system is executing. In this situation, in addition to architectural changes, there are also likely to be configuration changes as old components are removed and new components are instantiated on new or existing nodes.

III. Model-driven Software Architecture for Dynamic Software Adaptation

In model-driven software architecture, models of the software architecture are developed prior to implementation. An architecture model can be a platform-independent model (PIM), which is a precise model of the software architecture before commitment to a given platform, or a platform-specific model (PSM), where the architecture is deployed to a distributed hardware configuration [4]. Both PIM and PSM models can be used in runtime adaptation of software systems. In dynamic architecture-based software adaptation, it is necessary to have runtime knowledge before and after adaptation of (a) the software architecture in terms of components and connectors, and (b) the mapping of the software components and connectors to the hardware configuration, in particular the computer nodes to which they are deployed.

If the dynamic adaptation involves a change to the software architecture, then (a) the runtime PIM is used to determine what components need to be added, removed, or replaced in the architecture – this corresponds to architectural adaptation in section II (b) the runtime PSM is used to determine the nodes that are affected. If the adaptation does not involve a change in the software architecture but involves a change to the software configuration (e.g., after a node failure, one or more components need to be recovered and relocated to different nodes), then the runtime PSM is used to determine how to reconfigure the system. This corresponds to configuration adaptation in section II.

IV. Dynamic Change Management of Software Architectures

Kramer and Magee [2][5] investigated how dynamic adaptation (also called dynamic reconfiguration) could be carried out at the software architecture level. The software architecture consists of distributed components deployed to a distributed configuration. They described how a component must transition to a quiescent state before it can be removed or replaced in a dynamic software configuration. However, each component of the software architecture is treated as a black box with no knowledge of the executing software application.

A change management model [5] defines the precise steps involved in dynamic reconfiguration to transition from the current software run-time configuration to the new run-time configuration. Thus, a component that needs to be replaced has to stop being active and become quiescent, the components that it communicates with need to stop communicating with it; the component then needs to be unlinked, removed, and potentially replaced by one or more new components, after which the configuration needs to be relinked and the affected components restarted.

In order to drive a target component to quiescence, it is frequently necessary to also drive neighboring components to quiescence. However, if the application behavior of the neighboring components is known, it is possible to drive these components to partial quiescence so that they cease to communicate with the target component but continue communicating with other components, thereby providing less disruption to the runtime software system as described next.

V. Software Adaptation Patterns and Connectors

The Kramer/Magee approach to software adaptation assumes no knowledge of the executing application. If the software application is known, then knowledge of the patterns of behavior in the software architecture can be taken advantage of during software adaptation.

A software architecture is composed of distributed software architectural patterns, such as client/server, master/slave, and distributed control patterns, which describe the software components that constitute the pattern and their interconnections [4, 6]. For each of these architectural patterns, there is a corresponding *software adaptation pattern*, which models how the software components and interconnections can be changed under predefined circumstances, such as replacing one client with another in a client/server pattern, inserting a control component between two other control components in a distributed control pattern, etc.

A software adaptation pattern defines how a set of components that make up an architectural pattern dynamically cooperate to change the software configuration to a new configuration [9]. A software adaptation pattern requires state- and scenario-based reconfiguration behavior models to provide for a systematic design approach. The adaptation patterns are described by adaptation interaction models (using communication or sequence diagrams) and adaptation state machine models [9]. Several adaptation patterns have been developed including the Master-Slave Adaptation Pattern, Centralized Control Adaptation Pattern, Decentralized Control Adaptation Pattern [8], and service-oriented architectural patterns [9].

A. Software Adaptation State Machines

An adaptation state machine defines the sequence of states a component goes through from a normal operational state to a quiescent state [9]. A component is in the Active state when it is engaged in its normal application computations. A component is in the Passive state when it is not currently engaged in a transaction it initiated, and will not initiate new transactions. A component transitions to the Quiescent state when it is no longer operational and its neighboring components no longer communicate with it. Once quiescent, the component is idle and can be removed from the configuration, so that it can be replaced with a different version of the component. Fig. 1 shows the basic adaptation state machine model for a component as it transitions from Active state to Quiescent state. The adaptation framework sends a Passivate command to the component. If the component is idle, it transitions directly to the Quiescent state. However, if the component is busy participating in a transaction, it transitions to the Passive state. When the transaction is completed, it then transitions to the Quiescent state.

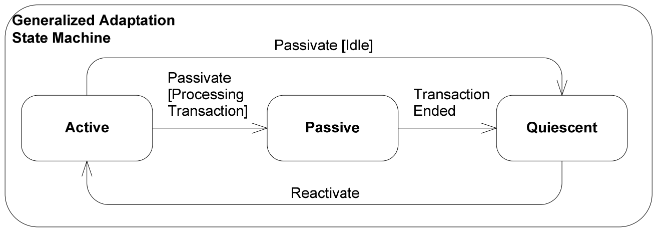
Generalized Adaptation State Machine
Passivate [Idle]
Passivate [Processing Transaction]
Transaction Ended
Active
Passive
Quiescent
Reactivate

Fig. 1. Basic adaptation state machine

In early research on software adaptation patterns [8], the state machine for each component was modeled using two orthogonal state machines, an operational state machine (modeling normal component operation) and an adaptation (also referred to as reconfiguration) state machine (Fig. 1). However, for more complicated adaptation patterns, there is often some interaction between the two state machines, which complicates the adaptation process.

In later research on service-oriented systems [9, 10], the operational state machine was separated from the adaptation state machine. This is done by moving the adaptation state machine from the component and encapsulating it in the connector. This separation of adaptation concerns from application concerns ensures that the adaptation patterns, as well as the corresponding code that realizes each pattern, are more reusable.

VI. Software Adaptation Connectors

Connectors in component-based software architectures (CBSA) are objects that interconnect components and encapsulate a communication protocol [4]. Connectors encapsulate frequently used communication patterns such as asynchronous communication and synchronous communication with reply. For software adaptation, a connector is enhanced to also address adaptation concerns of the component it is connected to. As long as the connector receives all incoming messages to the component and all responses from that component, it can track the behavior of the component in terms of messages received and responses sent, steer the component to a quiescent state [6], and thereby carry out the adaptation of the component.

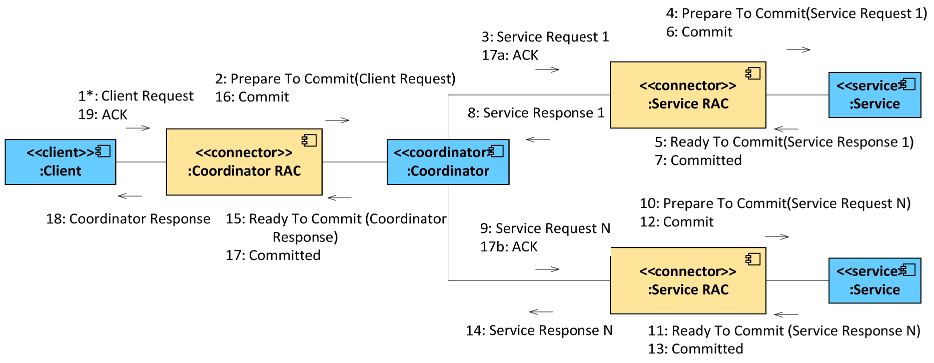

Fig. 2. SOA architectural pattern

The SASSY framework [7, 9, 10, 17, 20] provides two different types of adaptation connector, coordinator connector and service connector, as shown in Fig. 2. A service adaptation connector behaves as a proxy for a service, such that its clients can interact with the connector as if it was the service. The goal of an adaptation connector is to separate the concerns of an individual service from dynamic adaptation, i.e., the adaptation connector implements the adaptation mechanism for its corresponding service, including the interaction with the change management service [9] and tracking the operational states of the service. The adaptation state machine for a given adaptation pattern is encapsulated in the corresponding adaptation connector.

Each adaptation connector encapsulates a state machine that has the states depicted in Fig. 1. However, the events and actions of the state machine depend on the characteristics of the component that the connector communicates with. For example, in Fig. 2, a service connector that interacts with a service behaves differently from a connector that interacts with a coordinator.

SASSY uses an adaptive Change Management Model [9] to establish a region of quiescence [2] that allows dynamic adaptation of the component(s) to take place. For each adaptation pattern, the change management model controls and sequences the steps in which the configuration of components in the pattern is changed from the old configuration to the new configuration. In Fig. 2, a service could be dynamically replaced by a newer version of the service. However, to dynamically add a new service to the SOA would also necessitate a dynamic change of the coordinator to a newer version that is capable of interacting with the new service.

VII. Recovery and Adaptation Connectors

As a sequel to the SASSY project, we started work on the Resilient Autonomic Software Systems (RASS) project. After the SASSY research demonstrated that software adaptation concerns could be separated from component application concerns, the runtime modeling problems investigated by the RASS project [15, 16] are to (a) dynamically discover a software architecture at run-time [14] (b) investigate connectors that address software recovery in addition to software adaptation and (c) to investigate a decentralized autonomic MAPE-K approach to self-healing and self-configuration.

RASS [15] introduced the following concepts:

A *recovery pattern* defines how components in an architectural pattern can be dynamically relocated and recovered to a consistent state after a component has failed.

A *Recovery and Adaptation Connector (RAC)* that is used to separate adaptation and recovery concerns from component concerns so that a component can be dynamically adapted and recovered from failures.

A RAC (fig. 2) behaves as a proxy for a component or service by receiving requests from clients and then forwarding these requests to the service. The RAC also receives responses from the service, which are then forwarded to requesting clients. To ensure safe adaptation at run-time and recoverability of service failures, the RAC must keep track of the transactions that the service is currently engaged in and must maintain messages (i.e., requests and responses) that pass through it so that these messages can be held during adaptation and can be recovered when the service fails.

The reason why the RAC is able to address both planned adaptation and unplanned adaptation (i.e., recovery) is that in both cases, the messages that constitute each transaction need to be stored in the connector until the transaction has completed and manipulated during adaptation if required. However, the state machines and algorithms for planned and unplanned adaptation are different, as described next.

VIII. Distributed Adaptation and Failure Recovery

Based on the concept of RACs, we designed and implemented DARE (Distributed Adaptation and Recovery), a decentralized, integrated adaptation and recovery framework [16] for providing both self-healing and self-adaptation properties to complex and highly dynamic CBSAs. DARE is based on a decentralized version of the MAPE-K model. Every node in the system hosts an identical instance of the DARE middleware whose architecture is shown in Fig 3.

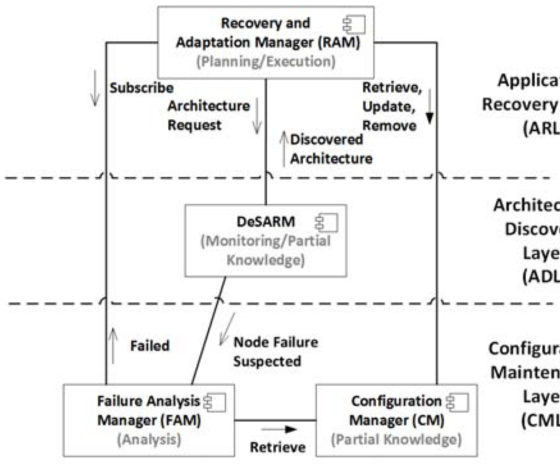
Recovery and Adaptation Manager (RAM)
(Planning/Execution)
Subscribe
Architecture Request
Retrieve, Update, Remove
Discovered Architecture
Application Recovery Layer (ARL)
DeSARM
(Monitoring/Partial Knowledge)
Architecture Discovery Layer (ADL)
Failed
Node Failure Suspected
Configuration Maintenance Layer (CML)
Failure Analysis Manager (FAM)
(Analysis)
Retrieve
Configuration Manager (CM)
(Partial Knowledge)

Fig. 3. The DARE architecture at each node

The *Configuration Maintenance Layer (CML)* tracks mapping of components to nodes and provides services to higher layers for retrieving and modifying this map. The Configuration Manager (CM) is responsible for maintaining the current configuration map of the software system, which is stored in a distributed hash table that supports replication of its entries [23] in order to tolerate failures and enable distribution of the map. The table contains entries that map the IP address of a node to the set of identifiers of components and RACs hosted by the node with this IP address.

The recovery of a failed component must be handled by exactly one Recovery and Adaptation Manager at one node. To ensure consistent recovery, our approach involves electing the node with the lowest IP address to become the *recovery coordination node* for coordinating recovery of other failed nodes. The Failure Analysis Manager (FAM) in the recovery node is the *only* FAM that proceeds with the recovery by analyzing the failure.

The *Architecture Discovery Layer (ADL)* consists of DeSARM [14], a decentralized architecture discovery mechanism, which is responsible for automatically discovering at runtime the current architecture of the software system, in particular components and connectors as well as synchronous and asynchronous communication patterns between components. DeSARM's decentralized architecture discovery mechanism is based on *selective* gossiping techniques [21] and message tracing. DeSARM notifies the Configuration Maintenance Layer when it suspects a node failure due to absence of gossip messages from that node.

The *Application Recovery Layer (ARL)* consists of the Recovery and Adaptation Manager (RAM) that is responsible for overseeing dynamic adaptation and recovery of the CBSA. When a node fails, the ARL ensures that the software system recovers to a consistent configuration in which every failed component is relocated and instantiated on a healthy node, and that the connections between a recovered component and its neighbor components are re-established.

The RAM interacts with the appropriate RAC for each component to be adapted or recovered. The RAC ensures that (1) any transactions that were interrupted due to a run-time failure are recovered and restarted at the recovered component and (2) a component is only adapted after it has become quiescent [2].

IX. DYNAMIC SPL RUN-TIME ARCHITECTURE

Software product line (SPL) engineering [18] provides a means for systematic planning and design for dynamic software adaptation. With a conventional SPL approach, such as the PLUS method [19, 22], a family of software architectures, consisting of common and variant components, is built at design time in advance of deployment and is managed using a feature model. By means of application feature selection, a given member of the SPL is derived and then deployed. However, with dynamic software adaptation, a member of the SPL can, after deployment, be dynamically adapted at run-time to a different member of the SPL.

Dynamic software product lines (DSPL) [24] are needed when SPL members need to adapt after deployment while the system is operational. Dynamic software adaptation for SPLs is the process of changing the SPL member at run-time to a different SPL member. In other words, dynamic software adaptation is concerned with changing the application configuration at runtime after it has been deployed and is needed for SPLs that have to dynamically adapt after original deployment. To address dynamic adaptation, the SPL life cycle for the PLUS method [19] is extended as shown in Fig. 4. PLUS consists of two phases: 1) Product Line (Domain) Engineering and 2) Application Engineering. For dynamic adaptation, a third phase is required: 3) Target System Reconfiguration. During this third phase, the executable target system is dynamically changed from the target system run-time configuration for one product line member to a new target system run-time configuration for a different SPL member [25].

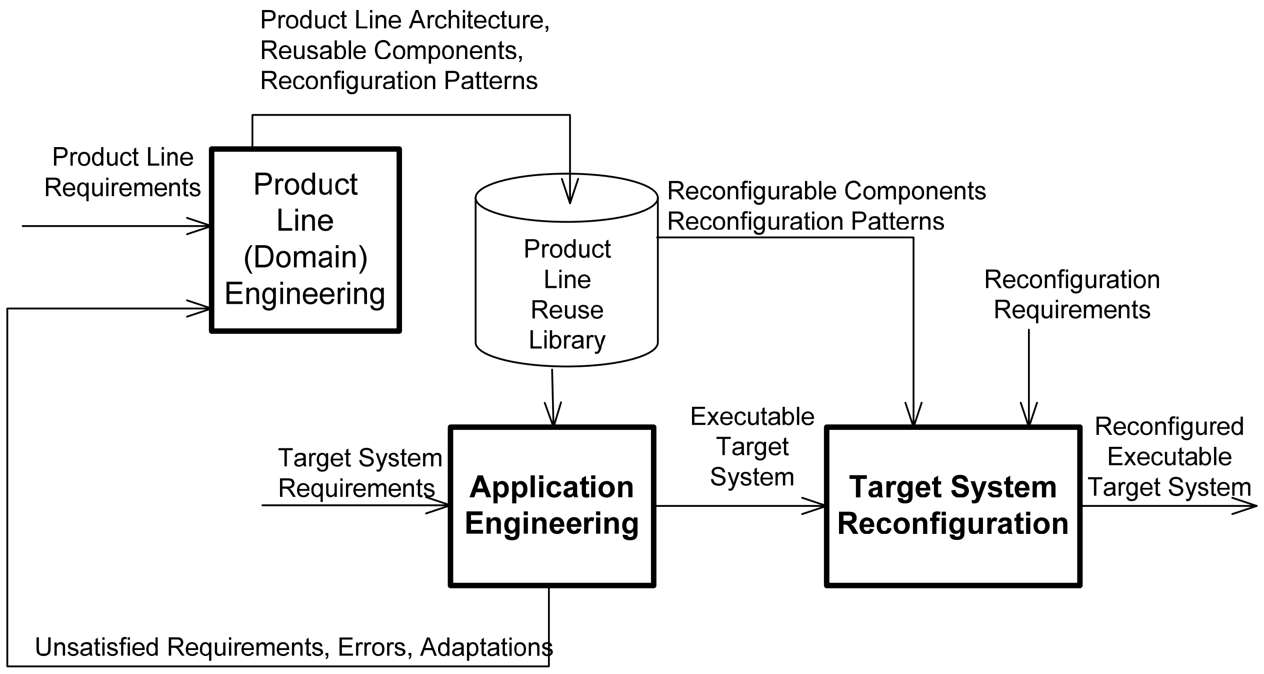
Product Line Architecture,
Reusable Components,
Reconfiguration Patterns
Product Line Requirements
Product Line (Domain) Engineering
Product Line Reuse Library
Reconfigurable Components
Reconfiguration Patterns
Reconfiguration Requirements
Target System Requirements
Application Engineering
Executable Target System
Target System Reconfiguration
Reconfigured Executable Target System
Unsatisfied Requirements, Errors, Adaptations

Fig. 4. Software life cycle for dynamic SPL

Using feature modeling, the SPL feature model has to capture all features of the SPL corresponding to the different application run-time architectures. The run-time feature model captures the features of the currently executing target system and is initially deployed at application derivation time. Furthermore, this feature model can change at run-time by deactivating those features no longer required and dynamically replacing them with new features (from the original SPL feature model) that are required. In order to dynamically adapt the software architecture in response to these feature changes, it is necessary to have a feature/component table that relates each feature to the components that realize the feature.

Dynamic SPL concepts have been used in SASSY for dynamic software adaptation of service-oriented architectures [25]. RASS [26] investigated the design of variable adaptation and recovery connectors used in the software adaptation of dynamic software product lines. The approach integrates software product line and feature modelling concepts with autonomic properties of self-healing and self-adaptation.

X. DISCUSSION

Although there is a large body of literature in the area of autonomic and self-adaptive systems, most of them use a centralized approach [11]. The main challenge with decentralized approaches is carrying out dynamic adaptation and recovery using partial knowledge of the system [12]. Schneider et al. [13] concluded in their survey on self-healing frameworks that the systematic integration of the self-* properties is one of the main challenges in this area. This paper has described how the above challenges have been systematically addressed. The SASSY and DARE approaches have progressively addressed the issues of decentralizing approaches to dynamic adaptation and recovery; DARE has addressed the integration of self-healing and self-adaptation properties.

Two other areas investigated by one of the authors are evolutionary software architectures [27] and designing reusable secure connectors [28]. The evolutionary software development approach uses SPL and feature modeling concepts for evolving multi-version systems. The multi-version systems constitute a family of systems with some common functionality and some variable functionality. The goal is to model all versions of the system, including previously deployed systems as a software product line. The addition of optional and alternative features necessitates the adaptation of the original software architecture by designing optional and variant components to realize these features.

Secure connectors [28] are designed separately from application components by reusing the appropriate communication pattern between components as well as the security services required by these components. Each secure connector is designed as a composite component that encapsulates both security service components and communication pattern components. Integration of security services and communication patterns within a secure connector is provided by a security coordinator. The main advantage is that secure connectors can be reused in different secure applications.

Dynamic SPL and dynamic software adaptation approaches could be enhanced by incorporating both the above approaches. Thus a dynamic adaptive SPL [25] could also evolve after initial deployment by allowing new features and components to be added to address evolving requirements. Dynamic and variable recovery and adaption connectors [26] could be further enhanced by incorporating security patterns. Thus variable connectors could be evolved and adapted at run-time to incorporate different recovery, adaptation, and/or security patterns.

XI. Conclusions

This paper has described and discussed the research conducted in dynamic software adaptation at George Mason University, including the research directions taken and why they were taken, the software engineering concepts and technology that the research is based on, the main results of the research and directions for future research. In particular, the research has been progressively addressing more challenging problems, building on dynamic software architecture models and software product line models, progressing from software architectural patterns to software adaptation and software recovery patterns, separating adaptation concerns from application concerns, decentralizing software adaptation planning and decision making, and addressing both planned and unplanned adaptation using a decentralized autonomic MAPE-K model. If a system failure occurs during adaptation planning, the planning needs to be aborted and restarted. However, if a system failure occurs during adaptation execution, both the adaptation planning and execution need to be aborted and restarted.

This paper has described approaches for dynamic software adaptation using runtime models of the software architecture, comparing adaptation with no knowledge of the software application with adaptation in which the application's software architectural patterns are known. From architectural patterns, software adaptation patterns were developed consisting of interaction models and state machine models that are used during dynamic software adaptation.

The paper then described how software adaptation and recovery concerns could be off-loaded from components by separating these concerns and incorporating them into adaptation and recovery connectors, which are responsible for dynamically adapting and recovering components from failure. The paper has considered both centralized and decentralized approaches to adaptation and recovery. The paper also described two approaches to dynamic software product lines, a dynamic software adaptation approach for service-oriented product lines and design of variable adaptation and recovery connectors used in the software adaptation of dynamic SPL.

For future work, the DSPL approach could be further enhanced by addressing software evolution and incorporating security patterns into recovery and adaptation connectors.

Acknowledgements

This work is partially supported by the AFOSR award FA9550-16-1-0030. The authors thank Koji Hashimoto and Mohamed Hussein for their contributions to this research.

References

[1] D. Garlan and B. Schmerl, "Model-based Adaptation for Self-Healing Systems", Proc. Workshop on Self-Healing Systems, ACM Press, Charleston, SC, 2002.

[2] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Tr. Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[4] H. Gomaa, *Software modeling and design: UML, use cases, patterns, and software architectures.* Cambridge University Press, 2011

[5] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering, 2007. FOSE'07.* IEEE, 2007, pp. 259–268.

[6] H. Gomaa, "A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines", Keynote paper, Proc. 9th International Conference on Model-Driven Engineering, Languages, and Systems, Genova, Italy, October 2006.

[7] D. Menascé, H. Gomaa, S. Malek, J. Sousa, "SASSY: A Framework for Self-Architecting Service-Oriented Systems", IEEE Software, Vol. 28, No. 6, pp. 78-85, November/December 2011.

[8] H. Gomaa and M. Hussein, "Software reconfiguration patterns for Dynamic Evolution of Software Architectures", Proc. Fourth Working IEEE/IFIP Conference on Software Architecture, Oslo, Norway, June, 2004.

[9] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. A. Menascé, "Software adaptation patterns for service-oriented architectures," in *Proc. 2010 ACM Symp. Applied Computing*. ACM, 2010, pp. 462– 469.

[10] H. Gomaa and K. Hashimoto, "Dynamic self-adaptation for distributed service-oriented transactions," in *Proc. 7th Intl. Symp. Software Engi- neering for Adaptive and Self-Managing Systems*. IEEE Press, 2012, pp. 11–20.

[11] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II.* Springer, 2013, pp. 76–107.

[12] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, “A survey on engineering approaches for self-adaptive systems,” *Pervasive and Mobile Computing*, vol. 17, pp. 184–206, 2015.

[13] C. Schneider, A. Barker, and S. Dobson, “A survey of self-healing systems frameworks,” *Software: Practice and Experience*, vol. 45, no. 10, pp. 1375–1398, 2015.

[14] J. Porter, H. Gomaa, and D. Menascé, “DESARM: A decentralized mechanism for discovering software architecture models at runtime in distributed systems,” in *11th Intl. Workshop on Models@run.time*, 2016

[15] E. Albassam, H. Gomaa, and D. Menascé, “Model-based Recovery Connectors for Self-adaptation and Self-healing: Design and Experimentation,” in Software Technologies, Revised Selected Papers from the 11th International Joint Conference, ICSOFT 2016, Lisbon, Portugal, July, 2016; Published by Springer, CCIS 743 pp. 108-131, 2017.

[16] E. Albassam, J. Porter, H. Gomaa, & D. Menascé, "DARE: A Distributed Adaptation and Recovery Failure Recovery Framework for Software Systems", Proc International Conf. on Autonomic Computing (ICAC), 2017.

[17] H. Gomaa and K. Hashimoto, “Model-based Run-Time Software Adaptation for Distributed Hierarchical Service Coordination”, Proc. Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications, Venice, May 2014.

[18] P. Clements & L. Northrop, *Software Product Lines: Practices and Patterns*, Boston: Addison-Wesley, 2001.

[19] Gomaa, H., “Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures”, Addison-Wesley, 2005**.**

[20] D. A. Menascé, J. P. Sousa, S. Malek, and H. Gomaa, “QoS Architectural Patterns for Self-Architecting Software Systems”, 7th IEEE Intl. Conf. on Autonomic Computing and Communication, Washington, DC, June, 2010.

[21] A.-M. Kermarrec and M. Van Steen, “Gossiping in distributed systems,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 2–7, 2007.

[22] M. Abu-Matar and H. Gomaa, "Variability Modeling for Service Oriented Product Line Architectures", Proc. International Software Product Line Conference, Munich, Germany, August 2011.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[24] M. Hinchey, S. Park, and K. Schmid, “Building Dynamic Software Product Lines,” *Computer*, vol. 45, no. 10, pp. 22–26, Oct. 2012.

[25] H. Gomaa & K. Hashimoto, “Dynamic Software Adaptation for Service-oriented Product Lines,” *Proc. Intl. Software Product Line Conf., Volume 2*, New York, USA, 2011.

[26] E. Albassam, H. Gomaa, and D. Menascé, “Variable Recovery and Adaptation Connectors for Dynamic Software Product Lines”, Proc. Tenth International Workshop on Dynamic Software Product Lines, SPLC Conference, Sevilla, Spain, September 2017.

[27] H. Gomaa, “Towards Feature-Based Evolutionary Software Modeling”, Proc. International Workshop on Models and Evolution, MODELS Conference, Wellington, New Zealand, October 2011**.**

[28] M. Shin, H. Gomaa, and D. Pathirage, “Reusable Secure Connectors for Secure Software Architectures”, Proc. 15th International Conference on Software Reuse, Limassol, Cyprus, June 2016.