Towards Formalizing Adaptive Software Services

Sonali Sharma, Rajeev Raje
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN, USA, 46202
sharmaso@umail.iu.edu/rraje@cs.iupui.edu

Ruchika Malhotra
Department of Software Engineering
Delhi Technological University
Delhi, India
ruchikamalhotra2004@yahoo.com

***Abstract*—More and more complex, distributed and software-intensive systems are built using independently developed services. Due to various reasons, such as changes in the execution environment, these systems may need to adapt their behavior. Although, adaptation at the system level has been extensively studied, developing adaptive services to start-with has not received any significant attention. This paper describes a framework for formalizing the concept of adaptation at the service level, leading to the "service adaptation by construction" approach. Hence, the proposed work will help software developers in identifying the important adaptation categories at the service level.**

Keywords—services; formalism; adaptation; taxonomy.

I. Introduction

Adaptation can be considered as a key factor for the survival of species in nature. Similar to the natural species, current and future distributed, complex, and software-intensive systems (henceforth referred as "*DCS systems*") will also need to adapt their behaviors – due to various reasons such as changes in the requirements and execution environments, demands for reuse and potential economic benefits. Such DCS systems are often composed as an ensemble of independently developed and deployed software services, thus, these individual services also need to be adaptive. Software Adaptation is widely recognized as an important problem in software engineering [1, 2]. It offers benefits such as high degree of flexibility, low maintenance cost, reliability and dependability and improved Quality of Service (QoS) [3].

Despite of its importance, adaptation in most cases is considered as an afterthought. Such an ad-hoc approach generally involves the use of wrappers or adapters on the top of existing services or DCS systems to enable the desired adaptation. Such approaches can significantly increase the risk of unintended behavior of a DCS system or a service as they might not operate as desired. Also, such a service or a DCS system would not be properly tested as it might not be possible to test all adaptation scenarios when they are considered as an afterthought. Although, adaptation at the level of DCS systems has been extensively studied, developing adaptive services to start-with has not received any significant attention. The current methods and techniques are not powerful (see the next section) to develop such services using the "adaptation by construction approach".

Hence, there is a need for a formalized approach towards software service adaptation – this requires a systematic study of the nature of adaptations that are applicable for software services and their appropriate categorization. Such a formalization and associated taxonomy will not only be crucial but also helpful for the developer of an adaptive service (and hence, an adaptive DCS system). In this paper, we focus on the adaptation by construction for individual services and not on their composed DCS systems. This paper describes a framework for formalizing the concept of adaptation at the service level, leading to the "service adaptation by construction" approach.

The rest of the paper is organized as follows: next section briefly discusses prominent related efforts; Section 3 presents the details of the proposed approach and illustrates the proposed approach by indicating a few examples; the paper concludes with lessons learnt and an outline of future efforts.

II. Related Works

Many frameworks have been proposed to support different types of software adaptations. In FUSION framework [4], a learning-based approach is presented for run-time software adaptation in which the adaptation decisions are learned to make them more accurate. The adaptive behavior of the system is analyzed and tuned for unanticipated changes. In [5], the concept of control data is introduced and the relationship between the behaviors of a component with the control data is identified. Change of the control data triggers change in behavior which is viewed as adaptation. [6] describes a framework which is based on a model-driven middleware based approach for adapting applications and services dynamically. In [7], an architectural model based approach to self-adaptation is presented in which system level properties and constraints are exposed with the help of the model. A majority of these approaches are focused on system-level adaptation or adaptation at run-time – they do not consider a formal notion of adaptation while developing individual services, as advocated by our approach.

In literature, a few taxonomies have also been proposed in the past. For example, in [8], a classification scheme has been described for self-adaptive systems. Five major characteristics have been defined, namely, origin, activation, system-layer, operation and controller distribution. In [9], the authors propose another taxonomy for self-protecting systems which can detect and tackle security threats at runtime. A signature level taxonomy is proposed in [10], which describes the

This is the author's manuscript of the article published in final edited form as:
Sharma, S., Raje, R., & Malhotra, R. (2016). Towards formalizing adaptive software services. In 2016 1st India International Conference on Information Processing (IICIP) (pp. 1–6). https://doi.org/10.1109/IICIP.2016.7975331

properties of signature changes in software systems based on their kind, frequency and evolution patterns. McKinley et al. [11] describes a taxonomy for adaptation with composition as the mechanism for adaptation. Their focus is on techniques of computational reflection and Aspect-oriented Programming. Again, these approaches are at the system level and not at the individual service level, as emphasized in our approach.

III. Proposed Approach

A. Taxonomy of Adaptation and Associated Formalism

As indicated earlier, we advocate the "adaptation by construction approach" while designing individual services that need to adapt. As services are independently developed and deployed, each service needs to be associated with its formal specification. Here, we assume that service specifications are multi-level as proposed by [12]. They have suggested four levels of specifications – syntax, semantics, synchronization and QoS. Hence, in this paper, adaptations are considered at all these levels, resulting in adaptive specifications similar to proposal discussed in [13].

The first step towards achieving this goal is to develop taxonomy of adaptation and formalize it appropriately. An adaptation, in the context of a service can be classified and formally defined using the following grammar. The non-terminals in the grammar start with an uppercase letter, while the terminals start with a lowercase letter or are enclosed in quotes. '&&' indicates the logical AND, while '||' indicates the logical OR operators. We use the prevalent notation (such as the use of '<', '::=', '|', and '>') for describing this grammar.

<Adaptation> ::= <Basic> | <Compound>

*<Basic> ::= <Type-Adaptation> | <Size-Adaptation> |
<Order-Adaptation> | <Number-Adaptation>|
<Time-Adaptation> | <Range-Adaptation> |
<Syntax-Adaptation> | <Input-Adaptation> |
<Output-Adaptation> | <Behavior-Adaptation> |
<Synchronization-Adaptation> |
<QoS-Adaptation>*

*<Compound> ::= <Basic> '&&' <Basic> |
<Basic> '||' <Basic>*

<Type-Adaptation> ::= <Type> <Type-Function> <Type>

<Type> ::= <Basic-Type> | <Constructed-Type>

<Basic-Type> ::= int | float | real | bool | string ...

<Constructed-Type> ::= array | struct | union | class | ...

*<Type-Function> ::= subset | equivalence |
reverse-implication | implication | ...*

<Size-Adaptation> ::= <Range> <Size-Function> <Range>

<Range> ::= int '..' int

<Size-Function> ::= increase | decrease

<Number-Adaptation>::= int <Num-Function> int

<Num-Function> ::= equal | increase | decrease

<Order-Adaptation>::= set <Order-Function> set

*<Order-Function> ::= all-permutation |
restricted-permutation | pre-defined permutation*

<Time-Adaptation> ::= <State> <Time-Function> <State>

<State> ::= s_1 | s_2 | s_3 *...*

*<Time-Function> ::= before | after | at | every | where | once |
so-far | awaits | until | eventually ...*

<Range-Adaptation> :: = range <Range-Function> range

<Input-Adaptation> ::= set <Input-Function> set

<Input-Function> :: = <Type-Function> | <Order-Function>

<Output-Adaptation> ::= set <Output-Function> set

*<Output-Function> ::= <Type-Function> |
<Order-Function> | <Num-Function> | <Size-Function> |
<Time-Function>*

*<Syntax-Adaptation> ::= <Input-Adaptation> |
<Output-Adaptation>*

<Behavior-Adaptation> ::= <State> <Pattern> <State>

*<Pattern> ::= insert pattern | delete pattern | move pattern |
replace pattern | swap pattern | ...*

<Synchronization-Adaptation> ::= <Policy> '=>' <Policy>

<Policy> ::= mutex | monitor | first-come-first-serve |...

*<QoS-Adaptation> ::= <Value> '=>' <Value> |
<Range> '=>' <Range>*

<Value> ::= int | float | double | ...

B. Types of Adaptation and Examples

In this section, we illustrate the abovementioned taxonomy and formalism by describing a few of these adaptations and associated examples. In the discussion below, each adaptation is denoted as A_{name}, where *name* is either a *Basic* or *Compound* category of adaptation as described in the above grammar. Also, we associate with each adaptation a generic function, *R,* which maps an appropriate entity (e.g., a particular type in the case of the type adaptation), to another appropriate entity (e.g., another type in the case of the type adaptation). All non-terminals that contain the word *Function (e.g., Type-Function, Time-Function, Size-Function, etc.),* in the above grammar, are described using R (and its specific variants) either formally or semi-formally.

1) Type Adaptation – Types, in the theory of programming languages [14], are either basic types (e.g., int) or constructed types (e.g., arrays). As indicated in the above grammar, a type adaptation applies to both these categories. Formally, we define the Type-Adaptation, A_T, as:

$$A_T ::= \mathrm{T} \xrightarrow{\mathrm{R}} \mathrm{T'}$$

… where, T is type of any element before adaptation, T' is type of the element after adaptation, and R is the relation between T and T'. R can be the subset, equivalence, implication, reverse implication, or a specific transformation relation (e.g. type converting function). Examples that use relations such as subset or equivalence are fairly common – as they relate to the subtyping or inheritance relationships. In contrast, an example of such type adaptation that may use a specific transformation relation can be described in the context of the URL Shortner Service (from Google Marketplace [15]). The *insert()* method of this service is responsible for shortening the URL. This method takes an *API key (indicating the URL)* as an argument – this key is of the type String – and shortens it into as few characters as possible in order to make the URL more readable, shareable and easy to email. This method can support type conversion by accepting a QR-code as an argument instead of a URL.

2) Size Adaptation – The Size adaptation is denoted by A_S. It is defined as:

$$A_S ::= \mathrm{S} \xrightarrow{\mathrm{R}} \mathrm{S'}$$

...where S is the size of an element, before adaptation, and S' is the size of that element, after adaptation. The computation of the size of an element depends on the nature of the element. For example, if the element is a set, then its size could be the cardinality of the set. R is the relation that maps S to S'. R can be either increase or decrese function and is defined as:

- R_{dec}: *S → S'… iff S < S'.*
- R_{inc}: *S → S'… iff S > S'.*

The size adaptation is generally associated with size of a service's input or output parameters. General triggers for this adaptation could be the QoS constraints, the QoS adaptation, and resource availability for service or network conditions. For example, the *URL Shortner* service mentioned above does perform the size adaptation when it shortens a URL into a smaller form. This also emphasizes the fact that a service can exhibit more than one kind of adaptation.

3) Number Adaptation – The Number adaptation is denoted by A_N. It is defined as:

$$A_N ::= \mathrm{N} \xrightarrow{\mathrm{R}} \mathrm{N'}$$

… where let a set of elements of an entity, before adaptation, is denoted as X and after adaptation is denoted as X', such that X = {x_1, … , x_p} and X' = {x'_1, … , x'_q} where p>0, q>0 and p != q. Also, let N = |X| and N' = |X'| where |X| and |X'| indicate the cardinality of sets X and X' respectively. R can be equal, increase or decrease function and is defined as:

- R_{eq}: *N → N'… iff N = N'.*
- R_{dec}: *N → N'… iff N < N'.*
- R_{inc}: *S → S'… iff N > N'.*

The number adaptation can be triggered when the service has a need to adapt to the change in the number of its inputs or outputs. Generally, such a scenario will occur if the client of the service provides fewer arguments than expected or expects a different set of outputs. If there was no adaptation in this situation, the service will throw an exception. However, such a situation could be handled by the number adaptation of the inputs or outputs. There could be various strategies to achieve this but the choice will be dependent on the developer of the service. As an example of number adaptation scenario, consider the case of missing parameters. In this case, the number of arguments provided by the client is less than expected number of arguments described by the service interface. One of the most common strategies suggested, in this case, is to replace the missing argument values with default values. Another approach could be to obtain some essential details of the input and form a complete new set of inputs with a reduced number of parameters. Again, the strategy may differ from developer to developer, but the general principles remain the same. Some of the common triggers of this type of adaptation are QoS requirement, size adaptation or when the required number of inputs and/or outputs are not provided or generated. A concrete example for this adaption is the Google's *Books* Service [16] – it has a resource called *Bookshelf* which contains a method called *list.* This method takes a required parameter *UserId* and an optional parameter, *source.* If an invocation is made with only one parameter, the number adaptation takes place by ignoring the optional argument.

4) Order Adaptation – The Order Adaptation is denoted by A_O. It is defined as:

$$A_O ::= \mathrm{O} \xrightarrow{\mathrm{R}} \mathrm{O}$$

… where O represents the set of *n* entities. For any such set O, a permutation can be defined as a bijective function R: O→O . Following permutations are defined for R:

- R_{all}: *an n-permutation of O.*
- $R_{restrictive}$: *an r-permutation of O where r < n.*
- $R_{pre\text{-}defined}$: *a pre-defined sequence of elements of O.*

This adaptation indicates a change in order of an entity associated with a software service. This could mean changing the order of inputs to changing the order of a sequence of function calls. For example, the first function, R_{all}, described above, denotes adapting to any of the possible permutations. One such approach was proposed by [13]. In this approach, the function calls are performed with named parameters or arguments identified by keywords. The clients then can call the function by providing name of each argument along with the parameters. Another approach could be a restricted or group order adaptation. In this approach, the service may allow only certain deviation from the expected order or may allow the change in order for only certain input arguments. Thus, the adaptation function would change order based on only some permitted permutations of the expected input order. The criteria of filtering out valid permutations will be application specific. Another example of this adaptation is the use of named arguments in Python.

5) Time Adaptation – In order to define the time (or temporal) adaptation, it is required to define the meaning of a temporal entity. In [17], Hobbs et al. have proposed time based ontology to formally represent temporal entities and their properties as a way to efficiently describe time-based information on the web, more specifically, web pages and web services. Also, one set of temporal relations have been proposed by Allen [18,19] which is also known as Allen's Interval Algebra. The relations defined in these works are used while defining the temporal adaptation as indicated below. The Time adaptation is denoted by A_{Time}. It is defined as:

$$A_{\mathrm{Time}} ::= S \xrightarrow{R} S'$$

...where, S is the initial state of the entity at time t and S' is its state after adaptation at time t'. Let p and q be temporal formulas. These formulae can be represented as basic time-based propositions. For example, p is true if the time value is a time, t_1. The sequence of states at any time t can be denoted by E_t and is represented as $\{S_t, S_{t+1}, S_{t+2} \dots S_n\}$ – for t = 0, 1, 2…n. Following functions, based on [17,18,19], are defined for R:

- *R_1: S → S'… iff After p.*
- *R_2: S → S'… iff Before p.*
- *R_3: S → S'… iff At p.*
- *R_4: S → S'… iff Every p.*
- *R_5: S → S'… iff [-]p where [-] represents So-far.*
- *R_6: S → S'… iff <->p where <-> represents Once.*
- *R_7: S → S'… iff p since q.*
- *R_8: S → S'… iff p Backto q (either p since q or[-]p).*
- *R_9: S → S'… iff ◯p where ◯ represents Next.*
- *R_{10}: S → S'… iff p Awaits q.*
- *R_{11}: S → S'… iff pUq where U represents Until.*
- *R_{12}: S → S'… iff <> p where <> represents Eventually.*
- *R_{13}: S → S'… iff []p where [] represents Henceforth.*
- *R_{14}: S → S'… iff □ p where □ represents Globally.*
- *R_{15}: S → S'…iff (<O> && C) where <O> is any of the above operators and C is the conditional expression.*

While considering the time adaptation for an entity, it is important to understand the relationship of this entity with time. For example, let us consider an adaptation scenario for an input to a service. This input can be either a time itself or could be an input in any other form but dependent on time. For the former case, application of the abovementioned time-based functions can be straight forward, but for the latter case, there might be more work required. This scenario is common in services which have real-time inputs. In such scenarios, the goal of adaptation is, generally, to modify the input representation in an effective way so that the changing input values may be utilized to adapt the service under various conditions. Thus, it becomes important to establish the relationship of time with the input. A common method is to time-stamp the input. This method is found among services with real-time data such as financial services and traffic routing services. A timestamp can describe the apprpriate time value by listing the year, month, day, hour, minute, second and millisecond. Different standards are available for time stamping values.

An example of time adpation is the GoogleNow wallpaper [20] application – it changes the color scheme of the background based on the time of the day. Rhapsody [21], an application for playing music, suggests songs to users based on the time and day and generates a user's listening pattern. As another example, consider a map service that assists in real-time directions based on traffic conditions. Such a service would have an incoming temporal data as input, which will, in most cases, be the data collected from GPS of vehicles on road. A map service could be designed to adapt its route suggestions not only based on current traffic but also the current time of day. Traffic patterns are known to vary based on the time of day or the season of the year, or based on some other real-time feed of a live event in the city.

6) Range Adaptation – The Range adaptation is denoted by A_R. It is defined as:

$$A_R ::= RN \xrightarrow{R} RN'$$

… where RN is the range of an entity before adaptation and RN' is the range of the entity after adaptation. RN is the set [x,..,y] that contains monotonically increasing integer or real values from x to y inclusive of x an y. Similarly, RN' is the set [x',..,y'] that contains values from x' to y' inclusive of x' and y'. Following functions are defined for R based on the relations between RN and RN':

- *R_{before}: RN → RN'… iff y < x'.*
- *R_{meets}: RN → RN'… iff y = x'.*
- *$R_{overlap}$: RN → RN'… iff [x,..,y] ∩ [x',.., y'] != Φ i.e., RN and RN' are not disjoint.*
- *R_{start}: RN → RN'… iff ((x = x') && (y < y')).*
- *R_{con}: RN → RN'… iff [x,..,y] ∩ [x',.., y'] = [x,..y,].*
- *R_{fin}: RN → RN'… iff ((x < x') && (y = y')).*
- *R_{eq}: RN → RN'… iff ((x = x') && (y=y')).*

An example of such an adaptation is the Set-*SPServerScaleOutDatabaseDataSubRange* associated with the SharePoint Server [22].

7) Input Adaptation – The input adaptation is denoted by A_I. It is defined as:

$$A_I ::= I \xrightarrow{R} I'$$

… where, I is the set of input elements before adaptation, and I' is the set of input elements after adaptation. The input adaptation, in turn, uses many of the earlier defined adaptations. Specifically, the input adaptation uses the type adaptation (A_T), the order adaptation (A_{order}), the number adaptation (A_{number}), the size adaptation (A_{size}), the range adaptation (A_{Range}) and the time adaptation (A_{Time}). Hence, the function R for the input adaptation takes the appropriate forms that are defined for these related adaptations – that is why, here, we neither specifically enumerate these functions not indicate any concrete examples of the input adpatation.

8) Output Adaptation – The output adaptation is denoted by A_O. It is defined as:

$$A_O ::= O \xrightarrow{R} O'$$

… where, O is the set of output elements before adaptation, and O' is the set of output elements after adaptation. The output adaptation has the same semantics (and hence, the use of related adaptations) as the input adaptation, except it deals with the outputs of the function than the inputs of the function. Hence, again here, we do not elaborate details of the output adpatation.

9) Syntax Adaptation – A service is typically specified by the function that it supports. Thus, at the service syntax level, a function's signature maybe required to change – resulting in the syntactical adaption for that service. The signature of a function in an object-oriented language such as Java, consists of a function name, list of parameters along with their types, return type of function, access control modifier or some specialized keywords specific to the language (e.g., Java language has abstract and final as keywords). Based on this signature structure, the syntax adaptation will either be the input adaptation or the output adaptation or both. As these adaptations are described earlier, we do not repeat them here while discussing the syntax adaptation. In the syntax adaptation, typically, the functionality of the service may not change At this level, the functionality of the service does not change but the function signature will need to adapt based on the specified parameter. An example of syntax adaptation is the Classifier Service described in [13].

10) Behavior Adaptation – A behavior (or semantic) adaptation is the change in the core functionality of a software service. Thus, the behavior adaptation can be defined as the adaptation of the functionality of the service. For an adaptive service, there would be a number of semantic variants available. In literature, several approaches have been proposed for adaptation at the behavior level in software services. One such list of different adaptation patterns is proposed in [23]. These patterns are: *Insert Process Fragment, Delete Process Fragment, Move Process Fragment, Replace Process Fragment, Swap Process, Extract Sub Process, Inline Sub Process, Embed Process Fragment in Loop, Parallelize Process Fragment, Embed Process Fragment in Conditional Branch, Add Control Dependency, Remove Control Dependency, and Update Condition.* Any of these patterns could be the choice(s) for the behavior adaptation. The Classifier Service [13] uses the Update Condition pattern.

11) Synchronization Adaptation – The next level in the multi-level contract, advocated by [12], is the synchronization level. Services need to support concurrent accesses, in most of the application scenarios, and hence, this level is needed. In [24], the specification of the synchronization level is divided into two parts – the policy part and the implementation part and a catalog of policies is presented. A few examples of the policy, indicated in [24], are *mutex, barrier, first-come-first-serve (FCFS)* and *monitor*. [25] discusses relations between these policies, and thus, provides a possible mechanism for adapting one policy to another. The *relaxed matching function,* defined in [25], is the relation associated with the synchronization adaptation and is defined as:

- R_{relax}: *SP → SP'… iff SP implies SP'.*

Again, the Classifier Service, from [13], is an example of synchronization adaptation which can adapt from *mutex* to *first-come-first-serve*.

12) QoS Adaptation – The last level in the multi-level contract, advocated by [12], is the QoS level. Services, especially the ones which deal with real-time constraints, need to guaranteed a specific value or a range values for specific QoS parameters (e.g., response time). Dynamic QoS parameters, such as the response time, are highly dependent upon the execution environment and hence, need to adapt for different operating scenarios. The QoS adaptation can either indicate change from a specific value to another specific value or from a range to another range. The latter scenario is most likely – thus, this adaptation will use the range adaptation described earlier. Again, the Classifier Service [13] is an example of such a QoS adaptation, where it can provide the classification in 3 to 120 ms.

13) Compond Adaptation -- Compound adaptations are created by a combination of two or more basic adaptations. This categorization is important to identify the relationships amongst various adaptations. The compound adaptation allows a designer to identify the impact of a basic adaptation on other types of basic adaptations or on the system as a whole. One such example of a compound adaptation is the *Synchronization-QoS Adaptation*. This adaptation is obtained by a serial combination of the *synchronization* and *QoS* adaptations. The underlying principle here is that the change in the synchronization policy will lead to a different QoS outcome. The Classifier Service [13], for example, when changes its synchronization behavior from *mutex* to *FCFS* may result in a different response time – due to changing its behavior from a non-deterministic to a more deterministic manner. In some cases, such a change in the QoS outcome may be significant but in others it may not matter. Hence, the developer is required to consider such compound adaptations while developing services, in addition to the basic adaptations.

IV. Conclusion and Future Work

As more and more services will need to adapt, they should be developed using the "adaptation by construction" approach – i.e., developers of such services should consider the notion of adaptation right from the beginning and not as an afterthought. The proposed formalism and taxonomy will help developers in identifying the important adaptation categories and ensure that they are appropriately met. Also, selected adaptation categories will have to be described in the specification of services – perhaps, using an alternative such as [13]. The approach discussed in this paper will allow a semi-automatic code generation for such services from their adaptive specifications.

Our future work includes the creation of a toolset which enables the developer to choose the different adaptations that are consistent with each other and generate an adaptive multi-level specification for the service. Such a tool can also develop skeleton code, which the developer can complete by filling in the application specific details. Another direction of our future work is to extend the current taxonomy based on specific domains. Finally, impact of adaptation of one type on another type and on the service as a whole will be investigated in future.

References

[1] P. Alencar and H. Weigand, "Challenges in Predictive Self-Adaptation of Service Bundles", in IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technologies, vol.3, pp.457-461, 2009.

[2] M. Papazoglou, K. Pohl, M. Parkin, and A. Metzger (Eds.), Service Research Challenges and Solutions for the Future Internet. S-Cube – Towards Engineering, Managing and Adapting Service-Based Systems, 2010.

[3] S. Kell, "A Survey of Practical Software Adaptation Techniques", J. UCS 14.13 (2008): pp. 2110-2157.

[4] N. Esfahani, A. Elkhodary, and S. Malek, "FUSION: A Learning-Based Approach for Engineering Self-Adaptive Software Systems", IEEE transactions on Software Engineering, no. 11, vol. 39, pp. 1467-1493, 2013.

[5] R. Bruni, et al. "A Conceptual Framework for Adaptation, "Fundamental Approaches to Software Engineering", in 15th International Conference, FASE 2012, pp. 240-254, 2012.

[6] S. Hallsteinsen, et al. "A Development Framework and Methodology for Self-adapting applications in Ubiquitous Computing Environments", Journal of Systems and Software 85.12 (2012): pp. 2840-2859.

[7] D. Garlan, et al. "Rainbow: Architecture-based Self-adaptation with Reusable Infrastructure", Computer 37.10 (2004): pp. 46-54.

[8] M. Rohr, et al. "A Classification Scheme for Self-adaptation Research", (2006): pp. 1-5.

[9] E. Yuan, and S. Malek, "A Taxonomy and Survey of Self-protecting Software Systems", in the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 109-118, 2012.

[10] S. Kim, E. Whitehead, and J. Bevan, "Analysis of Signature Change Patterns", ACM SIGSOFT Software Engineering Notes. Vol. 30. No. 4. ACM, 2005.

[11] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng, "A Taxonomy of Compositional Adaptation" Technical Report No. MSU-CSE-04-17, Michigan State University, 2004.

[12] A. Beugnard, J. Jezequel, N. Plouzeau, and D. Watkins, "Making Components Contract Aware", IEEE Computer, vol. 32, issue:7, pp.38-45, 1999.

[13] S. Phatak, "Multilevel Specification for Adaptive Services", M. S. Thesis, Department of Computer and Information Science, Indiana University-Purdue University Indianapolis, 2009.

[14] L. Cardelli, and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computing Surveys (CSUR), 17(4), 471-523, 1985.

[15] Google URL Shortner: https://goo.gl/

[16] Google Books: https://books.google.com/

[17] J. Hobbs, and P. Feng Pan, "An Ontology of Time for the Semantic Web", ACM Transactions on Asian Language Information Processing (TALIP) 3.1 (2004): 66-85, 2004.

[18] J. Allen, James, "Towards a General Theory of Action and Time", Artificial intelligence 23.2 (1984): 123-154, 1984.

[19] J. Allen, "An Interval-Based Representation of Temporal Knowledge", J. Allen, 7th International Joint Conference on Artificial Intelligence (IJCAI), 1981.

[20] GoogleNow Wallpaper Application: https://play.google.com/store/apps/details?id=com.bongoman.gnowwallpaper

[21] Rhapsode Music Service: http://www.rhapsody.com/.

[22] Set-SPServerScaleOutDatabaseDataSubRange, Share Point: https://technet.microsoft.com/en-us/library/jj871009.aspx.

[23] B. Weber, S. Rinderle, and M. Reichert, "Identifying and Evaluating Change Patterns and Change Support Features in Process-aware Information System", Technical Report No. TR-CTIT-07-22, University of Twente, The Netherlands, 2007.

[24] A. Kumari, "Synchronization and Quality of Service Specifications and Matching of Software Components", M. S. Thesis , Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2004.

[25] R. Raje, P. Katuri, A. Kumari, O. Tilak, "Multi-level Matching of Distributed Software Components", International Conference on Computer, Communication, and Instrumentation, Mumbai, India, 2009.