International Journal of Applied Engineering Research ISSN 0973-4562 Volume 15, Number 2 (2020) pp. 189-196
© Research India Publications. http://www.ripublication.com

Towards Automatically Generating State Transfer Model Integrated in Adaptive Software Development Process

Ngoc-Tho Huynh

The University of Danang, School of Information and Communication Technology, Danang, Vietnam.

ORCID: 0000-0001-6144-642

Abstract

Adaptive software development process is a set of activities to build adaptive softwares. Adaptive software is able to itself change its architecture or behavior at runtime without stopping whole system through an adaptation process. In architecture based adaptive software, an adaptation process includes some actions to change/replace components and their connectors. A challenge when replacing components at runtime is to maintain the state of the system such the value stored in variables, communication messages, etc. To cope with this issue, some approaches use state transfer functions which are manually implemented in the implementation phase of the development process. These are ad-hoc approaches. They do not define a general solution to manage state in the development process. Therefore, we propose an approach to manage state applied in adaptive software development process. This approach is interested in specifying the state in software architecture model at design time and be applied at runtime to automatically generate a state transfer model. This model is used during adaptation process for transferring state.

Keywords: software adaptation, state transfer, modeling, software development, safe adaptations

I. INTRODUCTION

Dynamic adaptation is critical feature of modern software. A software is called adaptive if it is able to change behavior or its architecture to adapt to a changing operation environment. The operating environment includes anything affected the software such as user requirements, end user input, external hardware devices and sensors, or program instrumentation [20]. In architecture based adaptive software, to cope with the changes of the operating environment, the adaptive software must change its architecture, i.e, components or connectors must be replaced at runtime without stopping whole system. In order to replace components or connectors, they must be stopped, then removed. New components or connectors will be added to the system and then started [14]. A challenge when replacing components or connectors in the adaptive software architecture is to ensure the system consistency, i.e., the system functions correctly before and after the replacement of components and/or connectors. One of aspect of the system consistency is to maintain state such as value stored in variables, communication messages, etc, before and after adaptation. That means that the state must be transferred among replacing components and replaced components or event among connectors for communication messages. Existing approaches have taken into account this issue, i.e. [24], [5].

In order to ensure the system consistency when replacing a component, its state must be stored and restored in the new components, i.e, the state must be transferred among components. Some solutions have taken into account this issue. A simple Solution is to take into consideration the homogeneity between two component versions and provide methods, setState() and getState() such as [24], [5], etc. In the case of the inhomogeneity between the component versions, this solution may generate errors: type errors, data errors, etc. A solution is to propose a logic unit such as state transfer net in [11], or state transition loGIC IN [25], that do the state mapping between the components. However, they are ad-hoc approaches, they do not provide a general solution to design the state mapping during development process and apply it for state transfer during adaptation.

In this paper, we extend the paper published in the 2019 International Conference on Computing and Communication Technologies (RIVF2019). The paper proposed a solution using a state transfer model to specify the state mapping between components. However, the model proposed in this paper is manually specified. It is not generated from a development process. Therefore, this extended paper proposes a solution to automatically generate the state transfer model from the software specification at design time and exploited at runtime during adaptation process to identify the state link among components. The remainder of the paper is structured as follows. Section II provides a medium to specify adaptive softwares and shows an overview of adaptive software development process. Section III focuses on the state management and state transfer. The justification through an example is presented in Section IV. Related work is discussed in Section V then the paper concludes.

II. BACKGROUND

A. Variability Specification

Our approach is based on model-driven engineering to develop adaptive softwares. In order to develop an adaptive software architecture, we use CVL meta-models to specify and manage variability. This section introduces an overview of the CVL approach [19] and the required information to be specified for supporting state transfer model generation.

CVL is a domain-independent language, and also an approach for specifying and configuring variability. An overview of the CVL approach is depicted in Figure 1. The base model is used to model the elements (components and connectors) of the architecture. The variability model specifies the variability in the base model, and the resolution model is defined to configure the variability model. In this paper, we use the term "the CVL model" to talk about these three models.

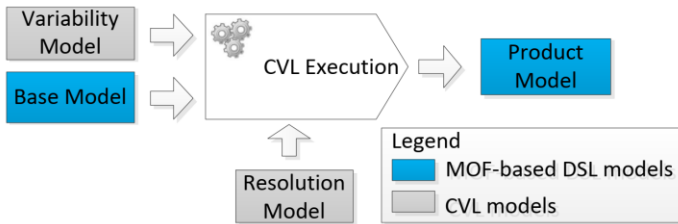

Figure 1. CVL approach (adopted from [21])

1) Base Model: A base model represents the software architecture of a product family. Therefore, it embeds variability using alternative elements. It can be defined in any MOF-defined language [19].

   In our approach, the base model is specified as a component-based architecture and defined by an ADL such as Fractal ADL [7] or ACME [9]. Each component in this model is specified with some information stored in component, for example variables, queues, communication messages, etc.

2) Variability Model: A variability model captures variability of a product line. It represents the differences and the similarities between products in the same family explicit.

   In the CVL approach, the variability model is specified thanks to the CVL meta-model. It contains three parts: the variability specification tree (VSpec tree), variation points, and Object Constraint Language (OCL) constraints.

   - A VSpec tree consists of VSpecs which are similar to features in feature models (FMs) [16]. There are four types of VSpecs: Choice, Variable, VClassifier and CompositeVSpec. A Choice allows to specify binary selections (true/false). A Variable should be used to specify a parameter whose value may change. A VClassifier allows to specify a min and max number of instances of the VSpec. Finally, CompositeVSpec is used for modularity purposes.
   - Variation points link a vspec to the corresponding elements (components or connectors) of the base model.
   - CVL allows to define OCL constraints among VSpecs that cannot be directly captured by hierarchical relations.

3) Resolution Model: In order to build a specific product from the variability model and the base model, the variability model must be configured, i.e., each VSpec should be resolved. To do this, a resolution model needs to be specified. Each element of this model is called VSpecResolution, refers to a VSpec of the variability model. According to the four types of VSpec, there are four kinds of VSpecResolution: ChoiceResolution, VariableValueAssignment, VInstance, and VConfiguration. A ChoiceResolution resolves a Choice VSpec in the variability model. It indicates which of the possible choices is taken for the VSpec. A VariableValueAssignment should be used to give a value to a Variable VSpec. An VInstance will indicate the number of instances that are present in the final product for a VClassifier. Finally, a VConfiguration element resolves a CompositeVSpec

   Once all the VSpec are resolved, the variability model is configured and the variation points are used to identify the elements and parameters or attributes in the base model that are affected by the configuration.

B. Architecture Modeling

An architecture (or base model) in our approach is specified by using ACME meta-model [9]. This model provides definition to specifically describe components and connectors in a system architecture. An extract of this model is shown in Figure 2.

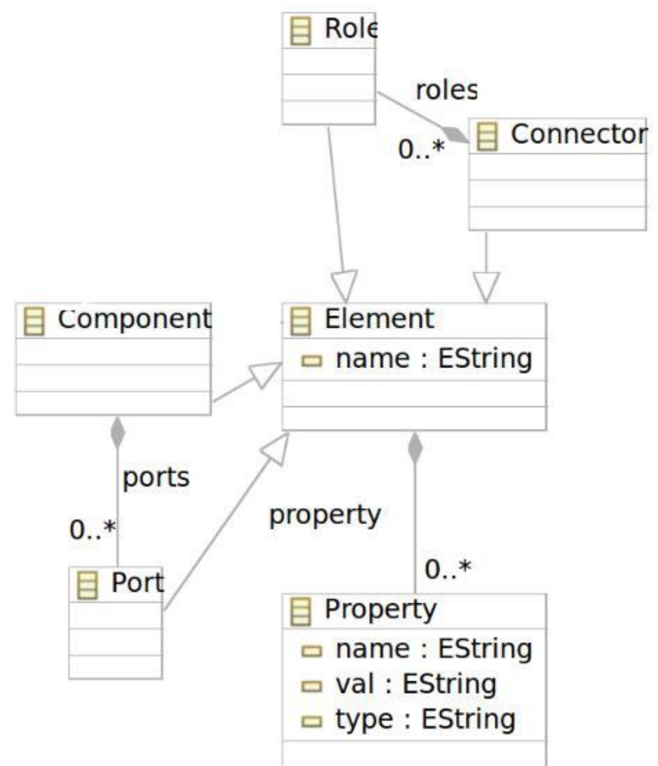

Figure 2. Component model(adopted from [9])

An architecture includes elements such as components and connectors. A component has one or many ports that allow to communicate with other ones through connectors. Each component and connector have properties name, val, type to

identify the information saved in component and connector. These properties can be considered as variables in the system. In this research, we focus on the corresponding of variables between components and connectors to manage state transfers during adaptation process.

C. An Adaptive Software Development Process

This section presents a brief of our adaptive software development process. The interested reader can find more detail in [13]. The process consists of two sub processes: domain engineering and application engineering. The domain engineering defines activities to specify variability and system architecture. The application engineering defines activities to specify a special product and generate adaptive architecture. The information about state transfer in the adaptive architecture should be taken into account at this time.

In this paper, we focus on the application engineering and gives details about state transfer, how to generate the state transfer model, and applying it in the development process. The next section, we mention what is the state and why transferring state.

III. STATE MANAGEMENT

The state of a system includes the local state of all components and all messages in transit [18]. The local state consists of all information such as component properties, data of a component in the system. According to Grondin et al. [11], a system state is defined as the set encompassing values of all variable attributes of all roles in all configurations.

A. State Transfer Analysis

This paper focuses on the component-based architecture. Figure 3 shows two components, client and server, in which the server component provides services and the client component requires this service. Each may offer the control service to manipulate the component such as modifying its implementation, activating/deactivating, reading/writing state into them, etc.

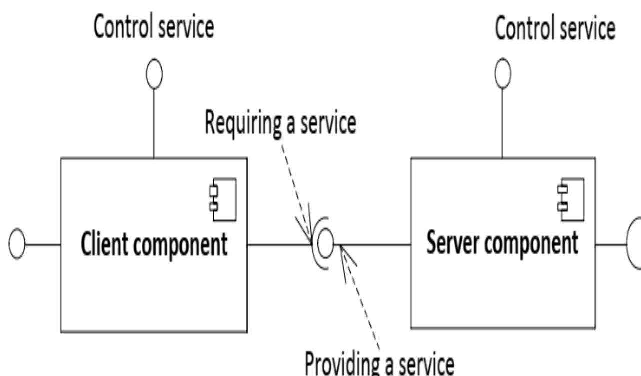

Figure 3. Client-server component model

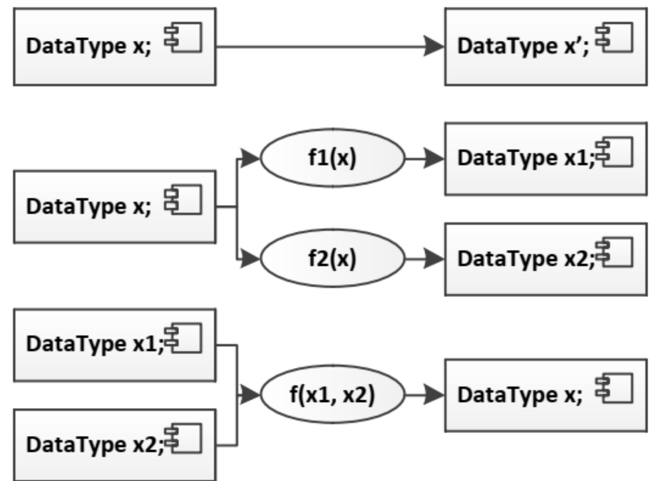

Figure 4. State transfer cases

State transfer is an important issue of dynamic adaptation [10]. It is considered as a process of capturing the runtime state of a component or a group of components and using this state to initialize a new version. In order to consider the state transfer case, we suppose that components, called placement components, will be replaced by another ones, called replacement components. Therefore, the state transfer is the information exchange between the placement components and the replacement components.

As previously mentioned, the state includes all information in the component. In this paper, we are interested in the state stored in the variables of components. Supposing that in a component there is a variable, x, the state transfer case is presented in Figure 4. In the first case, this is a simple case where a variable in the placement component is transferred to a variable in the replacement component. The state in this case is totally maintained in the same structure with the source variable. In the second one, a variable in the placement components can be changed and transferred to many variables in the replacement components. Therefore, functions, *f1*; *f2*, are used to change the state from the source variable, x, to the destination variables, *x1*; *x2*. In the last case, many variables in the placement components, *x1*; *x2* are transferred to a variable, *x*, in a replacement component. The function, *f*, uses the variable, *x1*; *x2* as its input arguments. The result of this function is assigned to a variable in the replacement component. A state mapping between the placement and replacement components is considered as a state exchange point.

In model-driven engineering, a model is used to specify the mapping the state transfer between the placement and replacement components. The next section, we present a state transfer model that is used to identify the mapping for state transfer at runtime adaptation.

B. State Transfer Model

Figure 5 shows a state transfer meta-model. The model consists of many state transfer points. Each point need map to

a destination variable, and zero or many source variables. With no source variable in a point, the destination variable can be assigned by a given value.

Each point may have a function to change the value of source variables into destination ones. For the first case mentioned in Figure 4, there is no function to change the value between source and destination variables. This means that the source variable can be directly assigned to the destination one. For the remaining case, a function is used to change the state in variables from the placement component to the replacement component. It is presented by an expression whose elements include operations, constant, and source variables. An expression can contain other expressions. The operations may be the number operations such as minus, plus, multiple, division, etc, or other operations defined by users.

The variable participated in a state transfer point is specified with a variable name, and its data type. The data type may be primitive or object type. The data type must be exactly specified to ensure the correct of data during state transfer process.

Finally, before assigning a value to a destination variable, the source value can be stored in a temporary variable of a temporary component. For example, a service component can receive requests from clients. In order to replace this component by another one, the service in the component must be blocked. Therefore, without the lost of new requests, it needs have a temporary element to store these requests. This issue is out of the scope of this paper. In this paper, we are interested in the state inside of component.

The state transfer meta-model is used to define the state transfer model. The next section, we present a process to automatically generate the state transfer model that conforms to the state transfer meta-model.

C. Automatically generating the state transfer model

The process to generate the state transfer model is shown as Figure 6. From two configurations specified in two resolution models, the placement and replacement components specified in the base model can be identified. Based on the annotation and properties names specified in the base model, i.e. the corresponding of name and data type, the corresponding of states between the placement and replacement components can be identified to generate the state transfer model. In this research, this model presents simply the first case presented in Figure 4. For other cases, an engineer must modify the model to conform to the adaptation scenario.

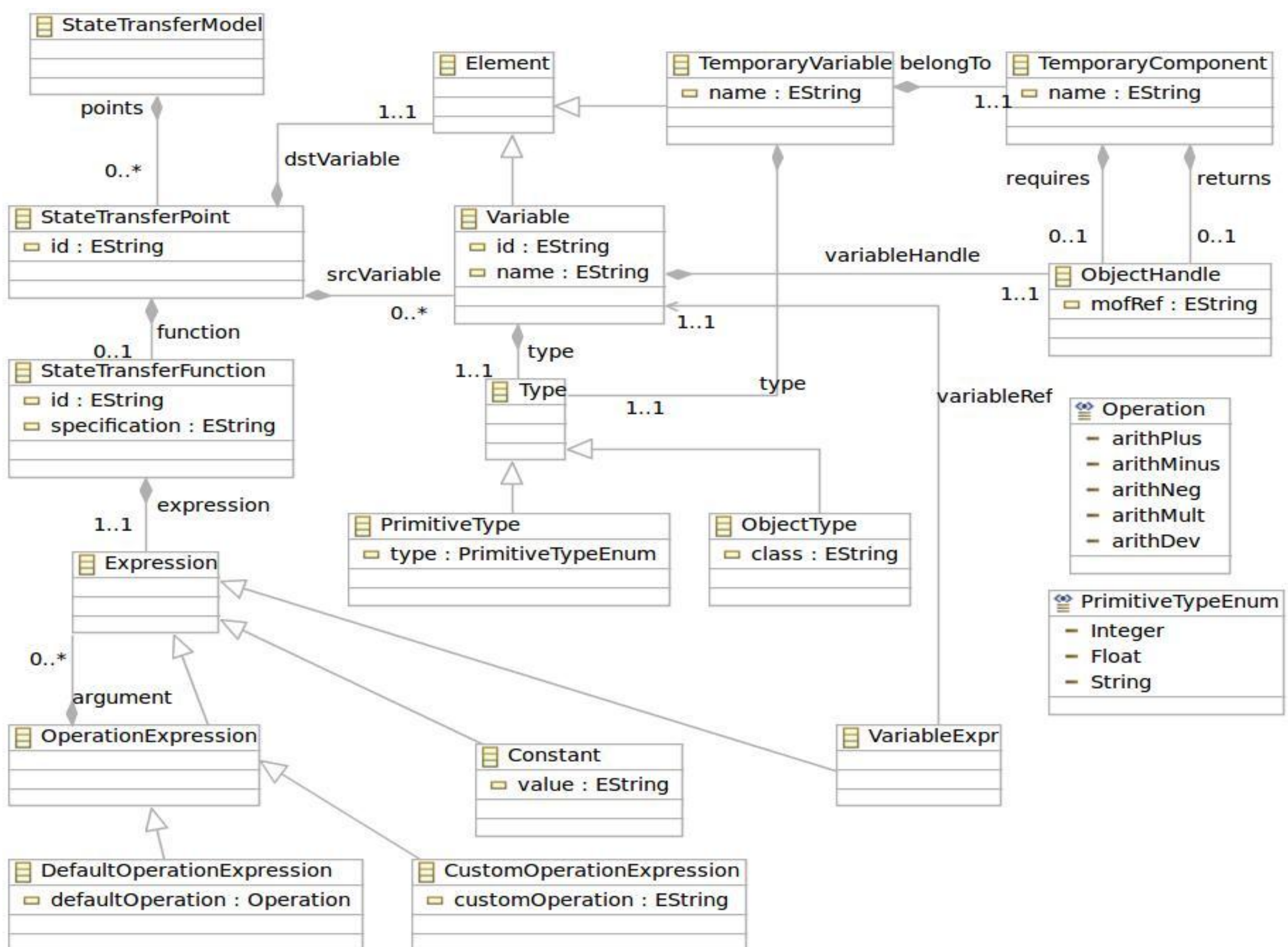

Figure 5. State transfer model

A tool is implemented to exploit these properties for generating the state transfer model. This model is conformed to the state transfer meta-model shown in Figure 5. It is applied at runtime during adaptation process via a state transfer mechanism.

D. State Transfer Mechanism

This mechanism is based on exploiting the state transfer model. As previously mentioned, the state transfer model is used to specify the state mapping in the adaptive software. The mapping information among components in the adaptive software is exploited at runtime by a state transfer mechanism to transfer state during reconfiguration process. A main element in this mechanism is reconfigurator. It controls the adaptation process which includes reconfiguration actions such as stopping/removing components, adding/starting components, changing connections, getting/writing state, etc. In this paper, we focus on the actions, getting/setting state.

The state transfer mechanism is based on the placement components and the replacement ones which are identified by calculating the difference between the current configuration and the new one. From two configurations and the state transfer model, state transfer actions can be identified and a state transfer script can be generated by a code generation module which is implemented using the Xpand generator framework. This script is read and executed by the reconfigurator which controls and realized reconfiguration actions.

In order to execute the script in the running system, the reconfigurator needs to have some functions like getState and setState. To realize these functions, they need to require services, getState, setState, provided by components engaged in a state transfer actions. Moreover, as previously mentioned, before setting a state in a new component, this state can be modified by a function specified in the state transfer model to conform to the destination variable.

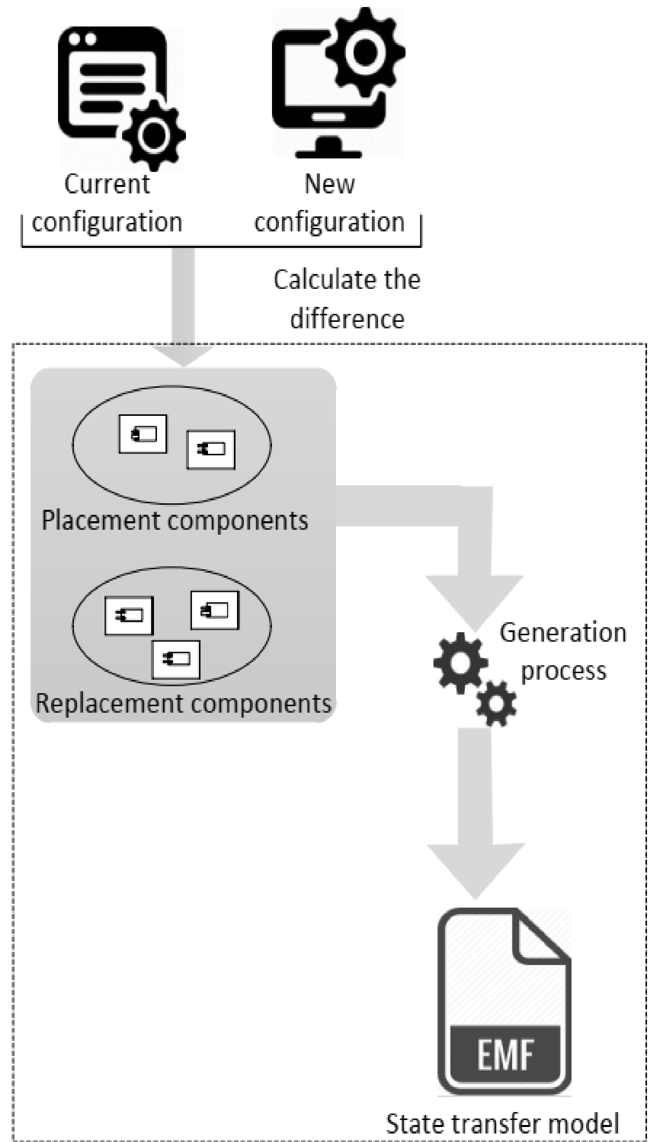

Figure 6. Generating the state transfer model

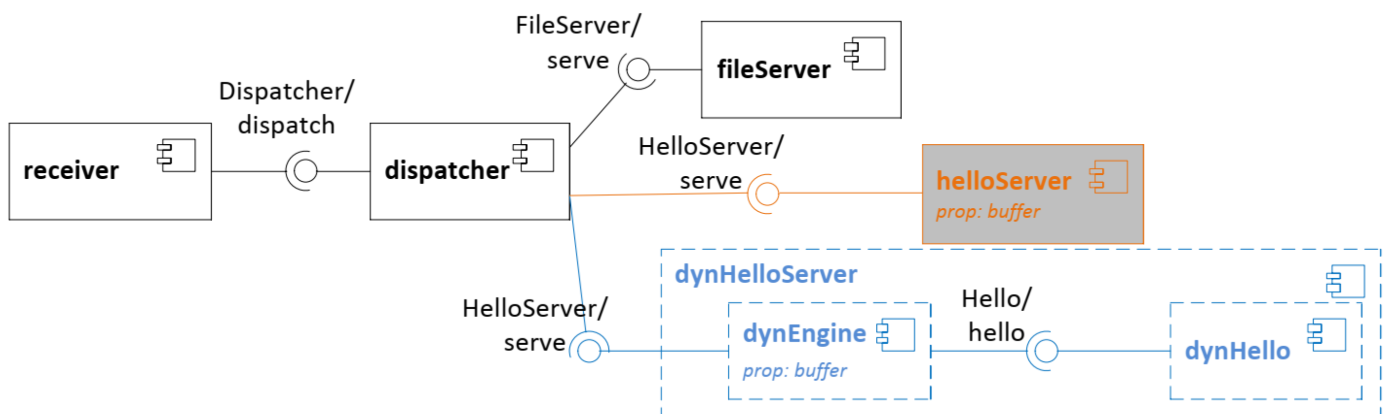

Figure 7. A web service example (adopted from [3])

IV. EXPERIMENTATION

A. Case study

In order to justify our approach, we adopt a simple web service described in [3]. The architecture of this web service is depicted as Figure 7. It also is considered as the base model according to the CVL approach. In the figure, the gray component is removed from the system by the reconfiguration, whereas the dotted line components are added to the system to replace the gray component during adaptation.

The initial service consists of four components, *receiver, dispatcher, fileServer,* and *helloServer*. The *receiver* component receives and decodes HTTP requests. The *dispatcher* component redirects requests to other services, *fileServer* or *helloServer*, according to the request URL. The *fileServer* component identifies a file name in the request and sends a corresponding response. The *helloServer* generates a dynamic web page with a "Hello world" message in its content. Each component provides interface with service for that other component can make requests, such as the *fileServer* component provides an interface, *FileServer*, with a service named *serve*.

The adaptation scenario in this example is to replace the *helloServer* by the *dynHelloServer* that contains two components: *dynEngine* and *dynHello* components. The former generates the dynamic web page, and the latter generates the "Hello world" content. During adaptation process, the request stored in buffer of the *helloServer* will be transferred to the *dynEngine* to be treated and returns to the client.

The *fileServer* component is not affected by the adaptation process. Therefore, it can handle requests during adaptation.

B. Specification and Implementation

The variability model of the example is shown as Figure 8. In the variability model, when selecting the Webservice VSpec, the receiver, dispatcher, fileServer VSpecs must be chosen. The dispatcher VSpec can alternative choice among the dynHelloServer VSpec or helloServer VSpec. The dynHelloEngine includes two VSpecs, dynEngine and dynHello. In order to generate a product model by using the AdapSwAG tool, a resolution model must be specified. Then, the implementation artifacts are able to be generated. Due to space constraints, the variation points and the resolution model are not presented here.

In our approach, the implementation of the example has been realized on top of the iPOJO/OSGi component model. This model allows to build dynamically extensible Java-based applications and facilitates their management by offering features like dynamic dependency, component reconfiguration, component factory, and introspection. In order to connect components, the Apache CXF framework [1] is used.

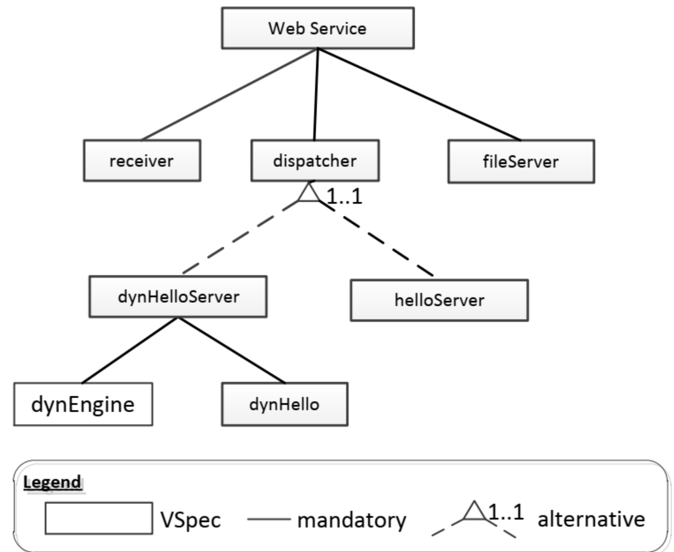

Figure 8. A web service variability model

C. Adaptation process

The adaptation process consists of the main steps: (1) isolate the *helloServer* component and block all new requests to it, wait to respond all previous requests; (2) add and activate new components, *dynEngine* and *dynHello*; (3) change connector from the *dispatcher* component to the *dynEngine* component instead of the *helloServer* component; (4) transfer new requests blocked in the *helloServer* component (stored in buffer) to the *dynEngine* component; (5) reintegrate the *dynEngine* component and treat the requests; (6) remove the *helloServer* component.

In order to identify the fourth step in the adaptation process, the reconfigurator needs have some information related to the mapping between the *helloServer* component and the *dynEngine* component. Such information is specified in a state transfer model that is generated via a process described in Section III-C. This model is shown as follows.

```
<StateTransferModel>

<Points>

<srcVariable name= "buffer">

<type type = "CbjectType" class = "message"/>

<variableHandle MofRef = "helloServer"/>

</srcVariable>

<dstVariable name= "buffer">

<type type = "ObjectType" class = "message"/>

<variableHandle mofRef = "dynengine"/>

</dstVariable>

</Points>

</StateTransferModel>
```

In this specification, a variable named buffer in the *helloServer* component contains new requests received from the *dispatcher* component. A variable named *buffer* in the *dynEngine* component presents the space where the request stored in buffer will be transferred to. This example presents a simple state transfer case, since two components, *helloServer* and *dynEngine* provides the same interface, *HelloServer/serve*. Thanks to the state transfer model, reconfigurator can identify state transfer actions for adaptation.

V. RELATED WORK

In order to transfer state in adaptive software, the corresponding states between two components versions is identified and the getting/setting functions are used for transferring state. In [2], [26], authors defined state transfer function from the current component version to the new one to represent the state mapping between two component versions. Moreover, the running system has to implement the setVar() and getVar() functions, to set and get the variable values. Vandewoude et al. in [25] addressed the state transfer and proposed a methodology to deal with runtime adaptation of components. They proposed a number of steps to perform state transfer. Particularly, at design time, both, old and new component versions, are analyzed and information collected during this analysis is embedded in a structure called the state transition logic. This structure is packaged together with the implementation of the new component version and used by the Dynamic Update Module in which a State Transformation Manager is responsible for transferring the actual state to the new one using the state export and import actions.

Other approaches are just interested in the introspection and intercession aspects in component model. The approach in [22], [24], [23] considered state as the private data encapsulated by components that are designed conforming to the Fractal component model. They propose adaptation scenarios to exploit the introspection and intercession aspects in component model to realize adaptation. Similar to [24], in [4], a component model was proposed in which control interface defines methods such as extractState() and restoreState() for state transfer. They are implemented by programmers and used by a configuration management in adaptation process. The approach in [6] introduces Lux, a pattern designed to extend object oriented languages with unanticipated adaptations. This pattern provides a notion to get and set state between objects defined in adaptive softwares. Existing approaches have been interested in the state transfer without a general solution for developing adaptive system. Our approach proposes a state transfer model which can be applies with any adaptive softwares development process based in models. Based on the state transfer model, it allows us to well manage state at design time. Moreover, thanks to the model, the state transfer mechanism can identify the corresponding state between components.

VI. CONCLUSION

In this paper, we have presented a state transfer model which can be generated from the specifications at design time and applied in an adaptive software development process to present the state mapping between components. The mapping is identified based on the corresponding of variables specified in the base model. It can be explicitly specified with the transition functions to adapt to adaptation scenarios. We defined the model by using EMF framework integrated in Eclipse. A set of API generated from this framework allow easily implementing the state transfer mechanism. The proposed model will be used at runtime during the reconfiguration process by the state transfer mechanism. This mechanism uses the model as its input to identify the state corresponding among components. In this paper, a prototype is defined to justify our approach. A real system is necessary to enforce this approach in the future.

ACKNOWLEDGMENT

This research is funded by Funds for Science and Technology Development of the University of Danang under project number B2018-DN07-03

REFERENCES

[1] C. Apache. An open source service framework. See: http://cxf. apache. org, 111, 2009.

[2] R. Bialek and E. Jul. A framework for evolutionary, dynamically updatable, component-based systems. In Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on, pages 326–331, March 2004.

[3] J. Buisson, F. Dagnat, E. Leroux, and S. Martinez. Safe reconfiguration of coqcots and pycots components. Journal of Systems and Software, 2015.

[4] X. Chen and M. Simons. A component framework for dynamic reconfiguration of distributed systems. In Proceedings of the IFIP/ACM Working Conference on Component Deployment, CD '02, pages 82–96, London, UK, UK, 2002. Springer-Verlag.

[5] S.-W. Cheng, D. Garlan, B. R. Schmerl, J. a. P. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu. Software architecture-based adaptation for pervasive systems. In Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing, ARCS '02, pages 67–82, London, UK, UK, 2002. Springer-Verlag.

[6] S. Costiou, M. Kerboeuf, G. Cavarle, and A. Plantec. Lub: A pattern for fine grained behavior adaptation at runtime. Science of Computer Programming, 161:149 – 171, 2018. Advances in Dynamic Languages.

[7] T. Coupaye and J.-B. Stefani. Fractal component-based software engineering. In Proceedings of the 2006 Conference on Object-oriented Technology: ECOOP 2006 Workshop Reader, ECOOP'06, pages 117– 129,

Berlin, Heidelberg, 2007. Springer-Verlag.

[8] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: An extensible service-oriented component framework. In IEEE International Conference on Services Computing, number Scc, pages 474–481, 2007.

[9] D. Garlan, R. T. Monroe, and D. Wile. Foundations of component-based systems. chapter Acme: Architectural Description of Component-based Systems, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.

[10] M. Ghafari, P. Jamshidi, S. Shahbazi, and H. Haghighi. Safe stopping of running component-based distributed systems: Challenges and research gaps. In Proceedings of the 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE '12, pages 66–71, Washington, DC, USA, 2012.

[11] G. Grondin, N. Bouraqadi, and L. Vercouter. Component reassembling and state transfer in madcar-based self-adaptive software. In 46th International Conference, TOOLS EUROPE 2008, number 11, pages 258–277, Zurich, Switzerland, 2008. Springer Berlin Heidelberg.

[12] M. Harsu. A survey on domain engineering. Technical report, Tampere University of Technology, Institute of Software Systems, 2002.

[13] N. Huynh, M. Segarra, and A. Beugnard. Building adaptive software architectures with useful and available elements for adaptation. In 2018 10th International Conference on Knowledge and Systems Engineering (KSE), pages 258–263, Nov 2018.

[14] N.-T. Huynh, A. Phung-Khac, and M.-T. Segarra. Towards reliable distributed reconfiguration. In Adaptive and Reflective Middleware on Proceedings of the International Workshop, ARM '11, pages 36–41, New York, NY, USA, 2011. ACM.

[15] N. T. Huynh, M. T. Segarra, and A. Beugnard. Ensuring consistent dynamic adaptation: An approach from design to runtime. In 2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA), pages 1–8, Nov 2016.

[16] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[17] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. IEEE Transaction on Software Engineering, 16(11):1293–1306, 1990.

[18] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 245–255, New York, NY, USA, 2011. ACM.

[19] OMG. Common variability language (cvl). OMG Revised Submission, 2012.

[20] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. IEEE Intelligent Systems and their Applications, 14(3):54–62, May 1999.

[21] G. Pascual, M. Pinto, and L. Fuentes. Self-adaptation of mobile systems driven by the common variability language. Future Generation Computer Systems, 2014.

[22] J. Polakovic and J.-B. Stefani. Architecting reconfigurable component-based operating systems. Journal of Systems Architecture, 54(6):562 – 575, 2008. Selection of best papers from the 32nd {EUROMICRO} Con-ference on 'Software Engineering and Advanced Applications' (SEAA 2006).

[23] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. Software: Practice and Experience, 42(5):559–583, 2012.

[24] M. Stoicescu, J.-C. Fabre, and M. Roy. From design for adaptation to component-based resilient computing. In Dependable Computing (PRDC), 2012 IEEE 18th Pacific Rim International Symposium on, pages 1–10, Niigata, 2012. IEEE.

[25] Y. Vandewoude and Y. Berbers. Component state mapping for runtime evolution. In In Proceedings of the 2005 International Conference on Programming Languages and Compilers, pages 230–236, 2005.

[26] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.