Towards Containment Checking of Behaviour in Architectural Patterns

FAIZ UL MURAM, HUY TRAN, UWE ZDUN
University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria

---

The behaviour of architectural patterns must be consistent in terms of the artefacts produced in the various activities of the software development process, such as requirements, software architecture, detailed design and implementation. In this context, high-level models are mainly used to convey the core concepts or principles of the reality they represent in an abstract and/or concise way (e.g., requirements or architecture design). If a specific architectural pattern like MODEL-VIEW-CONTROLLER is used in such high-level models, the corresponding detailed designs and implementations are also based on the particular pattern. Low-level or detailed design models are used to provide a (more) precise specification of the source code. However, because of the involvement of different stakeholders and independent evolution of software systems, inconsistencies might occur in architectural patterns' behaviour at those different abstraction levels. Previous studies have not considered the checking of architectural patterns' behaviour. In this paper, we present a solution to the containment checking problem that verifies whether the behaviour described by a low-level model still is consistent with the pattern specifications provided in its high-level counterparts. Here, the interactions between architectural pattern elements are captured using UML2 sequence diagrams. This paper also aims at providing more informative and comprehensive feedbacks to the stakeholders for identification of violation causes and their resolutions. The applicability of the proposed solution is demonstrated by applying it on three architectural patterns, namely MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER. The proposed solutions can also be applied to other types of behaviour models, such as state machines, activity diagrams and BPMN models, as well as other architectural patterns.

CCS Concepts: •**Software and its engineering → Software architectures;**

Additional Key Words and Phrases: Architectural Pattern, Consistency, Containment Checking, Modelling, UML.

ACM Reference Format:
Faiz UL Muram, Huy Tran, Uwe Zdun. 2017. Towards Containment Checking of Behaviour in Architectural Patterns. *EuroPLoP* (July 2017), 19 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

This work is supported by the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001, University of Vienna, and Austrian Science Fund (FWF), Grant No. I 2885-N33.
Author's addresses: F. UL Muram (faiz.ulmuram@univie.ac.at), H. Tran (huy.tran@univie.ac.at) and U. Zdun (uwe.zdun@univie.ac.at), University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroPLoP '17, July 12-16, 2017, Irsee, Germany
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-4848-5/17/07...$15.00
https://doi.org/10.1145/http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

A typical development scenario for modelling the system behaviour is that a business analyst or software architect uses a high-level model for outlining the system and discussing it with the customers and developers. The development team will expand the high-level model to include one or more low-level models. The low-level model, for example a sequence diagram showing the detailed interactions, is closer to or even very closely related to the source code of the system. As the software development process involves various activities, such as requirements elicitation, software architecture design, detailed design and implementation that are created and evolved independently by different stakeholders and teams, inconsistencies often occur among them. For instance, high-level models might be changed according to new requirements, and low-level models are changed as the implementation is modified. If each change is not systematically propagated to all other models of the same system (or reality), the evolved models may become inconsistent. Hence, detecting inconsistencies in early phases of the software development life-cycle is crucial to eliminate as many anomalies as possible before the systems are actually deployed. Such inconsistencies concern all kinds of constraints that a high-level model imposes on the low-level model. This is important for architectural patterns, as they impose various kinds of design constraints on the detailed designs and implementations that should not be violated. To date, however, none of the published studies have considered the consistency checking of architectural patterns' behaviour [Muram et al. 2017].

The main idea of architectural patterns is to resolve the recurring design problems that arise in a specific context at the level of software architectures, including those related to helping in documentation of architectural design decisions, facilitating the communication between stakeholders through a common vocabulary, and describing the quality attributes of a software system [Avgeriou and Zdun 2005]. There have been many attempts at modelling the structure of architectural patterns [Gamma et al. 1995; Shaw and Garlan 1996; Medvidovic and Taylor 2000; Zdun and Avgeriou 2008]; however, only a very few studies have focused on behaviour modelling of patterns [Garlan et al. 1994; Kamal and Avgeriou 2008; Perronne et al. 2006]. In practice, the most popular languages for modelling of architectural patterns and pattern variants in software design are various kinds of informal and semi-formal box-and-line diagrams [Rozanski and Woods 2005], the Unified Modelling Language (UML) [Group 2011b], Architecture Description Language (ADL) [Shaw and Garlan 1996; Medvidovic and Taylor 2000], and Domain Specific Language (DSL) [Mernik et al. 2005].

This work focuses on a special type of consistency checking, containment checking, which can be categorized as *vertical consistency*, i.e., consistency of the same model at different levels of abstraction [van der Straeten 2005; Muram et al. 2017]. The idea of containment checking is to verify whether the behaviour (or functions) described by a low-level design and implementation encompasses the behaviour specified in the high-level counterpart. The containment relationship mainly aims at unidirectional consistency because the low-level behaviour models are often constructed by refining and extending the high-level model. The containment checking of architectural patterns' behaviour not only deals with missing elements or interactions but also misplacement of elements at different levels of abstraction. Please note that there are severe negative effects of containment inconsistencies that may cause serious delays in and therefore increased costs of the system development process, jeopardize properties related to the quality of the system, and make it more difficult to maintain the system [Spanoudakis and Zisman 2001]. Unfortunately, modelling or mapping of architectural patterns' behaviour to sequence diagram is also a challenging task due to different variants of architectural patterns and different semantics of pattern elements and UML.

In order to support containment checking, we conducted a systematic literature review on behaviour consistency checking research [Muram et al. 2017]. In addition, we have investigated the containment relationship for various behaviour models in our previous work, including activity diagrams [Muram et al. 2014; Tran

et al. 2015], sequence diagrams [Muram et al. 2016] and Business Process Model and Notation (BPMN) [Group 2011a] process, choreography and collaboration models [Muram et al. 2015; Muram et al. 2017]. We have also investigated possible solutions that are based on model-checking techniques [Muram et al. 2014; 2015; 2016; Muram et al. 2017] and graph algorithms [Tran et al. 2015]. The major contributions of this paper can be summarised as follows:

—For modelling and analysing the behaviours of architectural patterns, we illustrate the containment checking solution using UML2 sequence diagrams [Group 2011b]. Our solution provides informative and comprehensive feedback to the software architects and/or developers to identify the violation causes and their resolutions.

—In order to support modelling or mapping of architectural patterns' behaviour to sequence diagram as well as to guide the user to follow a specific architectural pattern and its variants, we extend the UML metaclasses using UML profile mechanism. In particular, we use stereotypes to extend the properties of existing UML metaclasses.

—The applicability of the proposed solution is demonstrated for the MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER patterns. The proposed solution can also be applied to other types of behaviour models such as BPMN, UML activity diagrams and state machines.

The remainder of this paper is structured as follows: Section 2 summarises the related work on modelling and formalization of architectural patterns, and behavioural consistency checking. Section 3 describes the proposed containment checking solution in detail. Section 4 demonstrates the applicability of the proposed solution to the MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER patterns. Section 5 discusses the various aspects and challenges of supporting containment checking. Section 6 concludes the paper and discusses future work.

2. RELATED WORK

This section gives an overview of existing work on modelling and formalization of architectural patterns and consistency checking of behavioural models.

2.1 Modelling and Formalization of Architectural Patterns

The literature describes a number of attempts to model the structure of patterns [Gamma et al. 1995; Medvidovic and Taylor 2000; Zdun and Avgeriou 2008]. For instance, Giesecke et al. [Giesecke et al. 2007] extend the UML metamodel by creating profiles based on patterns. Their work maps the MidArch ADL to the UML metamodel for describing patterns in software design. Garlan and Kompanek [Garlan and Kompanek 2000] introduce four strategies (classes and objects, classes and classes, UML components, and subsystems) for encoding the architectural elements in UML typically found in modern ADLs. Clements et al. [Clements et al. 2003] demonstrate how UML can be used to represent the fundamental architectural concepts in a number of architectural views. Selic [Selic 1998] describes a UML profile for real-time systems, which demonstrates several architectural concepts such as components, connectors, and ports. These approaches have not considered the behaviour modelling of architectural patterns.

Mehta and Medvidovic [Mehta and Medvidovic 2003] propose an approach, called Alfa, for composing elements of architectural patterns using a small set of architectural primitives. In particular, they identified eight forms and nine functions as architectural primitives. Similarly, Bass et al. [Bass et al. 2003] have also proposed a predefined set of unit operations, such as abstraction, compression, separation and resource

sharing as the building blocks for all architectural and design patterns. Zdun et al. [Zdun and Avgeriou 2005; Kamal et al. 2008] present a generic and extensible approach for modelling architectural patterns by means of architectural primitives. They use a vocabulary of pattern elements in parallel to architectural primitives to capture the missing semantics of architectural patterns. The primitives for systematic modelling of architectural patterns' behaviour are not considered in the existing approaches.

There are many approaches for modelling or representing software patterns, which focuses on the design patterns from [Gamma et al. 1995]. A number of such approaches attempted to formally specify the patterns (see for instance [Mikkonen 1998; Eden et al. 1999; Soundarajan and Hallstrom 2004; Mak et al. 2004]). These approaches have not been used for architectural patterns or whole pattern languages, but just for some isolated patterns from [Gamma et al. 1995].

There have also been attempts to support the modelling of patterns' behaviour in software design. For instance, Garlan et al. [Garlan et al. 1994] propose an object model for representing architectural designs. The authors characterise architectural patterns as specialisation of the object models. Perronne et al. [Perronne et al. 2006] describe a modelling framework consists of two design patterns to support behaviour specification of patterns. The first, polymorphic behaviour pattern provides the integration and the execution of new behaviours for a system. The second, structured behaviour pattern provides the means to use finite state machines for behaviour switching. Kamal and Avgeriou [Kamal and Avgeriou 2008] describe the use of primitives for systematically modelling the behaviour of architectural patterns. However, the published studies have not considered the consistency checking of architectural patterns' behaviour so far.

2.2 Behavioural Consistency Checking

Many approaches tackled different types of models and/or model checking techniques [Spanoudakis and Zisman 2001; Lucas et al. 2009; Muram et al. 2017]. Some of them focus on checking the consistency of behavioural models against structural models [Rasch and Wehrheim 2003; Tsiolakis and Ehrig 2000; Kim and Carrington 2002] or checking different types of behaviour models (models and other representations of the same reality such as the requirements or implementations) [Lam and Padget 2005; Yao and Shatz 2006; Gherbi and Khendek 2007; Martens 2005]. To the best of our knowledge, very few of them consider the consistency checking problem for behaviour models at different levels of abstraction. The major difference of these approaches and our approach is that we consider the consistency of the same model at different levels of abstraction, i.e., "vertical consistency" [van der Straeten 2005]. In particular, we focus on checking the consistency of the containment of the high-level model in the low-level model, rather than checking the consistency of elements of two different representations.

In some studies, the notion of behaviour inheritance has been studied in the realm of consistency checking of behaviour diagrams, in particular, the inheritance of object life cycles in statecharts. Stumptner and Schrefl introduce specialisations of object life cycles by examining extension and refinement in the context of UML statecharts [Stumptner and Schrefl 2000]. Van der Aalst presents a theoretical framework for defining the semantics of behaviour inheritance [van der Aalst 2002]. In this work, four different inheritance rules, based on hiding and blocking principles, are defined for UML activity diagram, statechart and sequence diagram. However, the application of these inheritance concepts in the context of actual scenarios is not clarified. It might also be noted that the outcomes of these techniques do not assist the stakeholders in understanding cause(s) of consistency problems and their resolutions.

In our earlier work, we have investigated the containment checking problem for various behavioural models. Particularly, our previous research not only supports automated transformation of activity diagrams [Muram et al. 2014], sequence diagrams [Muram et al. 2016], and BPMN process, choreography and collaboration

diagrams [Muram et al. 2015; Muram et al. 2017] into equivalent formal specifications and consistency constraints, but also interprets the counterexamples for locating the cause(s) of inconsistencies and their resolutions [Muram et al. 2015; 2016; Muram et al. 2017]. Besides model checking techniques, graph-based solutions for addressing the problem of containment checking are also investigated [Tran et al. 2015]. This research deals with the checking of architecture patterns' behaviour at different levels of abstraction, which has not been addressed so far [Muram et al. 2017].

3. APPROACH

This research aims at identification and resolution of containment checking problems for architecture patterns' behaviour. In particular, we assume that a high-level model of a software system contains an architecture pattern, and focus on the question whether the behaviour described by a low-level model of that system encompasses the specifications made in the high-level model. Typically, a high-level model is created by a business analyst or software architect for outlining the system and discussing with the customers and developers. The low-level models are created by the development team or otherwise reverse-engineered from the source code; they are closer to or even very closely related to the source code of the system. In the course of software system modelling and implementation, as models are created and evolved independently by different stakeholders and teams, inconsistencies among models often occur. Therefore, containment checking improves the quality and reduces the complexity of big and complex system by determining and resolving the deviations between the low-level behaviour models and a high-level counterpart in the design phase. To guide the user to follow a specific architectural pattern and its variants, we extend UML metaclasses using the UML profile mechanism. In particular, we use stereotypes to extend the properties of existing UML metaclasses; for example, the Object/Lifeline metaclass is extended to model the participants of patterns. An overview of our containment checking approach is shown in Figure 1. The main focus of the approach is depicted by the solid lines whilst the dashed lines illustrate relevant modelling and developing activities of the involved stakeholders.

As emphasized, this paper deals with the problem of checking the containment between generic behaviour of architectural patterns at different level of abstraction. We modelled the generic behaviour of architectural patterns using UML2 sequence diagrams as these diagrams can be used to capture the interaction between architectural pattern elements. Sequence diagrams show the sequence of operations/methods entailed by the architectural patterns, occurrence of events to invoke specific operations, and use messages to show the interaction among pattern elements. In particular, a sequence diagram consists of lifelines/objects representing the individual participants in the interaction that communicate via messages. A message is sent from its source object to its target object (represents an operation/method on the objects) and has two endpoints. Each endpoint is an intersection with an object and is called an *OccurrenceSpecification* (*OS*). In particular, each message associates normally two *OS s* (aka events): one is the sending *OS* and the other is the receiving *OS*. An *OccurrenceSpecification* is a specialisation of a *MessageEnd*. *ExecutionOccurrence* is represented by two event occurrences, the start event occurrence and the finish event occurrence. Messages can be asynchronous or synchronous. The source object continues to send and receive other messages after an asynchronous message is sent. In contrast, when a synchronous message is sent, the source object blocks and waits to receive a response (i.e., reply message) from the target object. As the definitions and semantics of UML2 sequence diagrams are rather informal [Group 2011b], we derive a representative description of sequence diagrams constructs, to provide the basis for formally analysing the generic behaviour of architectural patterns.

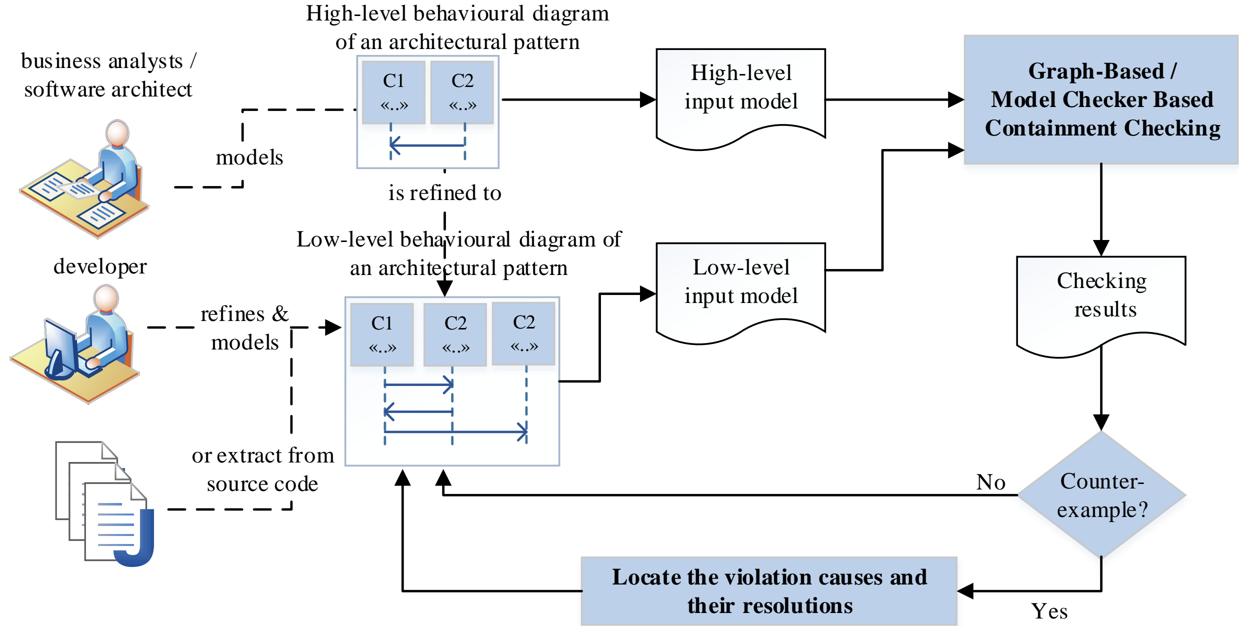

Fig. 1. Approach Overview

Definition 3.1 (*Sequence Model*). A sequence model Seq is a tuple $(Objects, Interactions)$ to express the generic behaviour of architectural pattern where

$$
\begin{aligned}
Object &::= \text{is a finite set of objects/lifelines} \\
type : Interaction &::= (Object_Source, Message, Snd) \mid (Object_Target, Message, Rec) \\
Snd &::= \text{is a sending } OccurrenceSpecification \text{ of a message on a lifeline } Object_Source \\
Rec &::= \text{is a receiving } OccurrenceSpecification \text{ of a message on a lifeline } Object_Target
\end{aligned}
$$

The main goal of our approach is to verify whether the generic behaviour described by the low-level sequence model of an architectural pattern encompasses those specified in the high-level counterpart. Thus, our approach takes as inputs a high-level sequence model Seq_H and a low-level sequence model Seq_L to verify whether the containment relationship between these models are satisfied (denoted below as: $Seq_H \prec Seq_L$). The following rules for sending and receiving messages must be considered as the generic behaviour specification of patterns:

—The sending and receiving occurrence specifications of messages on the same object must occur in the same order in which they are described.

—A receiving occurrence specification Rec of a message is enabled for execution if and only if the sending occurrence Snd of the same message has already occurred.

—In the case that a synchronous message is sent, the source object cannot send or receive the other messages until it has received the reply message from the target object.

Two possible ways of containment checking are model checking techniques and graph-based search. Model checking is an automatic property verification approach that systematically and exhaustively explore the

states of software systems. It is most frequently used for the formal verification of safety-critical systems, for example, air traffic control systems, medical equipment systems, train signalling systems, and automotive control systems [Rozier 2011]. The advantage of model checking is that it can be performed in early phases of software modelling and development where no executable products are produced yet. The model checking techniques require the transformation of high-level behaviour model into design constraints, whereas the low-level behaviour model can be mapped into formal descriptions. This can be done using either manual mapping of input models into formal descriptions and consistency constraints (e.g., specifying the transformation rules) or automated techniques. In [Muram et al. 2014; 2016] we have introduced the transformation rules grounded on formal expressions that can support the automated transformation of the high-level behaviour models into design constraints and low-level behaviour models into formal descriptions. In particular, the behaviour models are created in Eclipse Papyrus[1] and the Eclipse Xtend framework[2] is used to realise the transformation of behaviour models to formal descriptions and design constraints. The model checker takes the formal descriptions and design constraints as inputs, and exhaustively explore all executions of the formal descriptions by traversing the complete state space to determine whether the design constraints hold. In case formal descriptions do not satisfy the design constraints, it implies that the low-level model deviates improperly from the high-level counterpart, and the model checker will generate a counterexample. Note that a counterexample provides only limited information for understanding the causes of inconsistencies but not how to fix the inconsistencies. Therefore, an efficient analysis of the generated counterexample is supported in our proposed solution that provides the concrete information about the causes of inconsistencies (i.e., missing elements and misplacement of elements) and their resolutions [Muram et al. 2015; 2016; Muram et al. 2017]. The containment relationship to be validated by model checking is defined in the following Equation 1:

$$\begin{aligned} Seq_H \prec Seq_L & \\ = \quad & noMissingElements(Seq_H, Seq_L) \\ \wedge \quad & noMisplacedElements(Seq_H, Seq_L) \end{aligned} \tag{1}$$

The containment checking problem can be broken down into smaller graph-based tasks (or functions), which has a number of advantages. First, the tasks are independent of each other, and therefore, can be performed in any order. Moreover, the tasks can also be executed in parallel to gain better performance. Finally, each task produces concrete and precise information about the violations of the containment relationship. For instance, no missing nodes (i.e., no missing expected functions), no missing transitive links (i.e., no missing edges between nodes or functions), and no missing cycles (i.e., no missing loop executions) are implemented in [Tran et al. 2015]. The graph based techniques might require intermediate representation of behaviour models, i.e., mapping of elements to nodes and edges [Fryz and Kotulski 2007; Truong et al. 2009; Tran et al. 2015]. Subsequently, the graph search algorithms are used to verify the containment relationship between input models. In case the containment relationship is not satisfied (i.e., the input behaviour models are inconsistent), the checking results have to provide concrete information about the causes of inconsistencies such as missing elements, missing execution paths, or missing cycles as well as the involved model elements. The containment relationship to be validated by the graph-based algorithm is defined in the following Equation 2:

[1]See https://www.eclipse.org/papyrus
[2]See https://eclipse.org/xtend

$$
\begin{aligned}
Seq_H \prec Seq_L \\
= \quad & noMissingNodes(Seq_H, Seq_L) \\
\wedge \quad & noMissingTransitiveLinks(Seq_H, Seq_L) \\
\wedge \quad & noMissingCycles(Seq_H, Seq_L)
\end{aligned} \tag{2}
$$

$$
noMissingNodes(Seq_H, Seq_L) = \forall e_H : Seq_H \bullet \exists e_L : Seq_L \bullet match(e1, e2) \tag{3}
$$

The function *match*(), which is used by *noMissingNodes*(), takes two model elements as inputs and returns *true* if two elements are matched and *false* otherwise, as shown in Equation 3. For the function *noMissingTransitiveLinks*() the conventional definitions of the *adjacency matrix* and *transitive closure* of a directed graph can be used. Let $G = (V, E)$ be a directed graph where V is the set of nodes and E is the ordered set of arcs. The adjacency matrix A_G of G is an $n \times n$ boolean matrix whose elements $A_G[i, j]$ is *true* if $e(i, j) \in E$ and *false* otherwise. Based on the adjacency matrix A_G, a *reachability matrix* $R_G = A_G^*$ can be derived to represent the transitive closure of G. It is denoted as $R_G[i, j] = true$ if there is a directed *path* from node i to node j and $false$ otherwise. In order to define *noMissingCycles*(), Tarjan's algorithm [Tarjan 1972] can be used to obtain a set of *strongly connected components* (SCC)[3]. For more details see [Tran et al. 2015].

4. APPLICATION OF OUR APPROACH TO ARCHITECTURAL PATTERNS

4.1 Containment Checking in Model-View-Controller

The section discusses the identification and resolution of containment inconsistencies in the model-view-controller (MVC) pattern's behaviour at different levels of abstraction. In the MVC pattern the system is divided into three different parts: The *Model* concerns the object or objects that encapsulate some application data and the logic that manipulates that data independently of the user interfaces. One or multiple *Views* display a specific portion of the data to the user. The *Controller* associated with each view receives user input and translates it into a request to the model. In particular, the views and controllers constitute the user interface. The users interact strictly through the views and their controllers, independently of the model, which in turn notifies all different user interfaces about updates. There are many variations of the MVC pattern, for instance, passive model and classic MVC[4][Sokolova et al. 2013]. The former is used when one controller manipulates the model exclusively. The controller modifies the model and then notifies the view about the changed model, which should be updated. The later is employed when the model changes state, and it notifies the view without the controller involvement.

UML sequence diagrams need to be extended to express the specific semantics of MVC, especially to denote which objects are which participants of MVC and which kinds of messages are sent. Accordingly, a set of stereotypes is required to help architects and developers in correctly mapping the elements of the MVC architectural pattern to UML sequence diagrams. This will also reduce the occurrence of containment violations between the high-level and low-level behaviour models. We selected five stereotypes from the existing

[3]A graph is strongly connected if every vertex is reachable from every other vertex. The strongly connected components of a directed graph form a partition into subgraphs that are themselves strongly connected.
[4]See https://msdn.microsoft.com/en-us/library/ff649643.aspx

vocabulary of design elements [Kamal and Avgeriou 2008]. The «Model», «View», «Controller» stereotypes are used to extend the semantics of lifelines/objects in UML sequence diagrams for modelling the model, view and controller participants of the pattern, respectively. UML sequence diagrams support the asynchronous messaging; however, the semantics defined in the UML standard do not clearly define the difference between the return values from the receiver lifeline (i.e., target object). It is difficult to determine whether the return value is merely a notification/acknowledgement event about the receipt of message or the actually processed data; therefore the «AsynchMessage» stereotype is used for modelling the asynchronous communication among the objects. Similarly, the «SynchMessage» stereotype is used when the source object blocks and waits to receive a response from the target object to update the status of the operation that invoked the synchronous communication. In summary:

—The «Model» stereotype extends the Lifeline/Object metaclass of UML and owns occurrence specifications for interaction with Controller and/or View objects.

—The «View» stereotype extends the Lifeline metaclass of UML and owns occurrence specifications for interaction with Model and Controller objects.

—The «Controller» stereotype extends the Lifeline metaclass of UML and owns occurrence specifications for interaction with Model and View objects.

—The «SynchMessage» stereotype extends the Message metaclass and uses the existing UML synch-message operations to ensure that an end-to-end connection is established with the receiver lifeline (target object), which covers the receiving occurrence specification (event end). A return operation is mandatory for the synchronous communication to update the status of the operation that invoked the synchronous communication.

—The «AsynchMessage» stereotype extends the Message metaclass to ensure that the invocation flag is active when an operation is invoked. The asynchronous communication is further constrained to ensure that the method invoking the operation is not bound to receive the reply message and only a notification/acknowledgement can inform about the receipt of message.

Furthermore, there is need for additional stereotypes to cover the missing aspects concerning the containment relationships. For instance, the User/Client object is covered by any stereotypes explained so far. We also consider the case in which the object(s) in the low-level model are broken down into multiple entities, so that each of them not necessarily receives an update message, in particular, if the View is broken down into *mobileView* and *desktopView*, the message will be sent to only one specific view. For this purpose we introduce the «ViewPart» stereotype. Similarly, model and controller can be broken down into sub-components.

—The «Actor» stereotype extends the Object/Lifeline metaclass of UML and contains occurrence specifications for the interaction with View.

—The «ViewPart» stereotype extends the Lifeline metaclass of UML and divides the View into parts for interaction with a Controller and/or Model.

—The «ModelPart» stereotype extends the Lifeline metaclass of UML and divides the Model into parts for interaction with a Controller and/or View.

—The «ControllerPart» stereotype extends the Lifeline metaclass of UML and divides the Controller into parts for interaction with a Model and Views.

The high-level sequence diagram of an itinerary management system is shown in Figure 2. It follows the MVC pattern and involves five objects, namely Client, Website, User Controller, Travel Agency, and Airline.

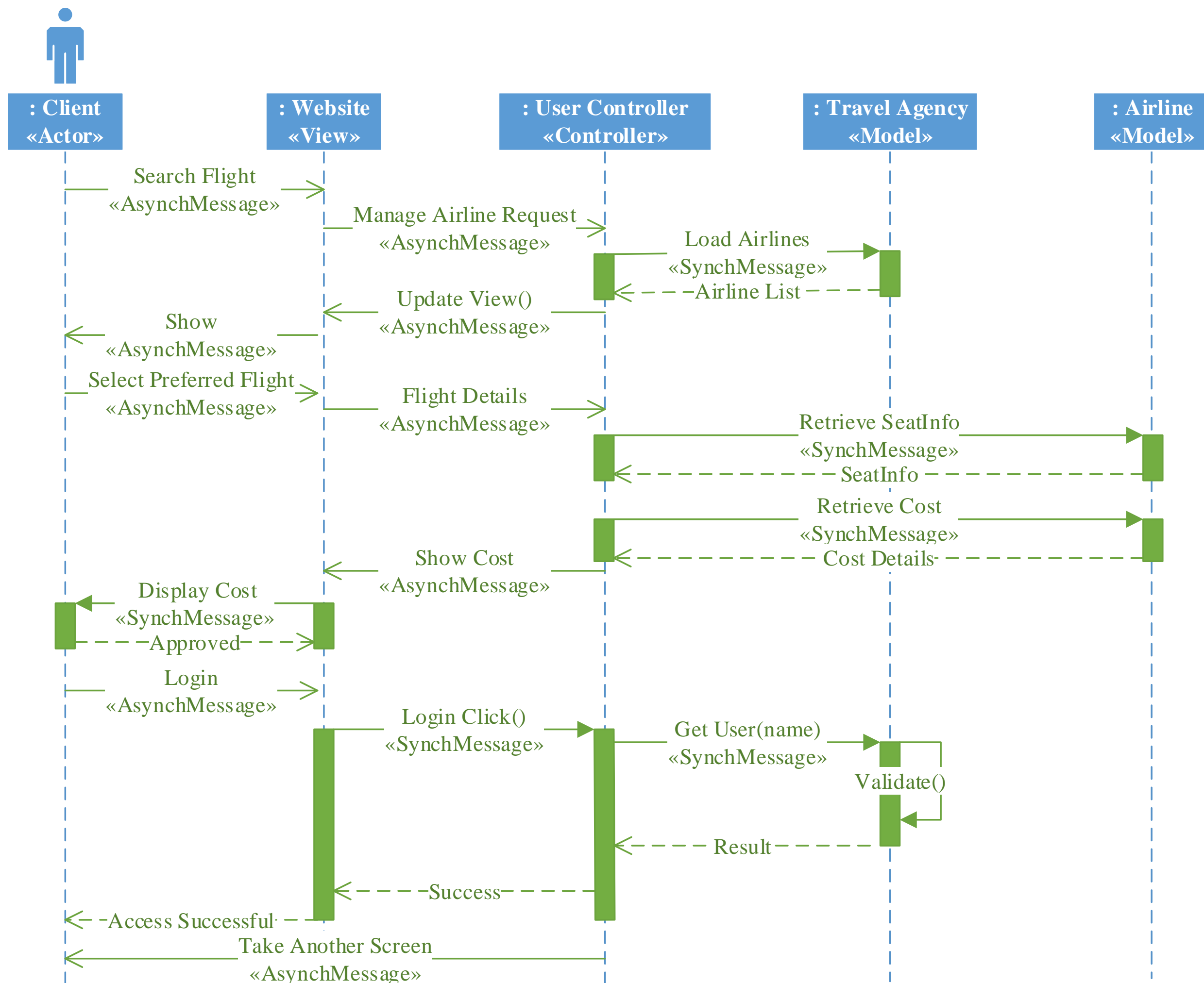

Fig. 2. High-Level Model of Itinerary Management System

Specifically, Airline and Travel Agency objects represent the models; Website concerns view which forward Client requests to User Controller; and User Controller stores and retrieves data from Airline and Travel Agency models and updates the Website view accordingly. The core functionality of the itinerary management system can be described as follows: the process starts when Client sends search flight request by invoking an event through the Website, the User controller, in turn, contacts the Travel Agency model for loading airlines. The Travel Agency replies with a flight list to the User Controller which in turn is asked to update the view. When the Client selects the preferred flight, the User Controller asks the Airline model about seat info and cost details, respectively.

The low-level itinerary management system is a refined and extended version of the high-level diagram that provides more detailed information about the system (an example of a low-level model is shown in Figure 3 in the context of detected violations and their causes). For instance, it contains additionally a Payment model to calculate the price of an itinerary, as well as an Email model and Service for managing the user registration

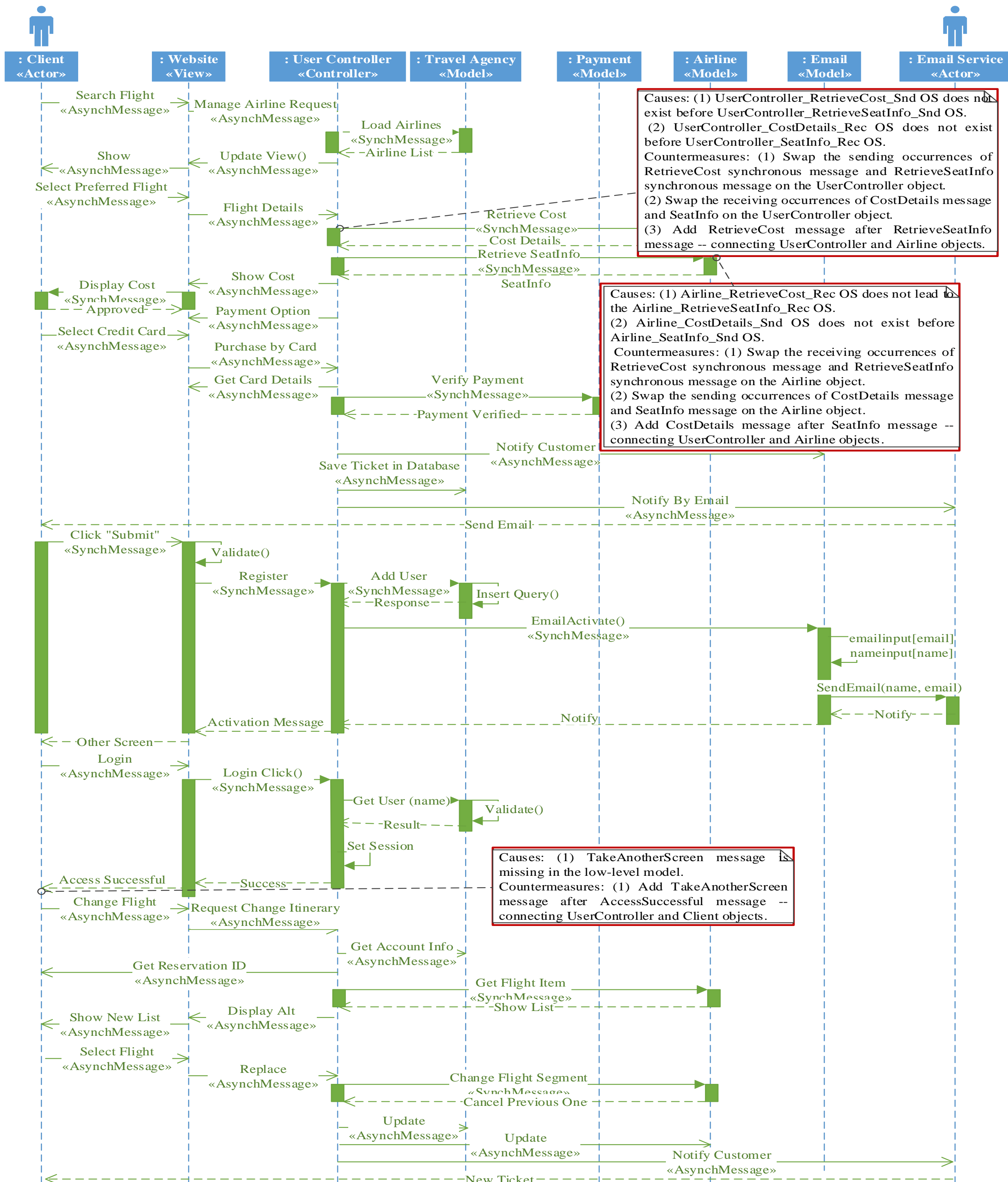

Fig. 3. Feedback of Containment Results in the Low-Level Model

and login strategy. The low-level model may contain new messages that can be inserted in-between existing ones, or new objects and messages in parallel with existing ones, and so on, but the elements should not be inserted arbitrarily. Our containment checking for generic behaviour of the MVC pattern aims to verify whether the elements of high-level model correspond to those of a detailed design of a system.

The containment checking solution presented in the aforementioned Section 3 first verifies whether all the objects (i.e. model, view, and controller) that exist in the high-level sequence diagram of the itinerary management system (Seq_H) are also present in the low-level sequence diagram (Seq_L). For each object the respective interactions (e.g., $Client_SearchFlight_Snd$, $Client_Show_Rec$) are also matched. If an object present in the Seq_H no longer exists in the Seq_L it means that interactions corresponding to this object are also deleted. For this, the "missing element cause" (either one, multiple, or all elements could be missing) is detected and a corresponding countermeasure (i.e., insert the missing element at a particular position in the low-level model) is suggested.

In this case, the `TakeAnotherScreen` message is sent from the `UserController` to `Client` present in the high-level model, but does not exist in the low-level model, can be seen as a reason for the containment violation. As we can see in Figure 3, the third box displays the actual cause in particularly `TakeAnotherScreen` message is missing in the low-level model for which the proposed countermeasure is add `TakeAnotherScreen` message after `AccessSuccessful` message – connecting `UserController` and `Client` objects.

If all objects present in Seq_H are also present in Seq_L, the next check is, whether the corresponding interactions have a different structure. The preceding and succeeding interactions of a corresponding object of Seq_L are matched with the interaction of Seq_H to locate the causes of inconsistencies. In our example, the sending OS of the `RetrieveCost` message ($UserController_RetrieveCost_Snd$) and the receiving occurrence of the `CostDetails` reply message ($UserController_CostDetails_Rec$) covered on the `UserController` object are violated. The former is violated because the `RetrieveCost` is sent prior to the sending OS of the `RetrieveSeatInfo` message ($UserController_RetrieveSeatInfo_Snd$); whereas the latter is violated because the receiving occurrence of the `SeatInfo` message ($UserController_SeatInfo_Rec$) does not exist before the `CostDetails` reply message in the low-level model. Similarly, a misplacement of message occurrences exists for the `Airline` object. These violations can be resolved by putting the `RetrieveCost` and `CostDetails` messages after the `RetrieveSeatInfo` and `SeatInfo` messages – connecting the `UserController` and `Airline` objects in the Seq_L. In Figure 3, the first and second boxes show the actual causes and potential countermeasures of misplacement of elements. Once the causes are located, causes are eliminated by updating the responsible elements of the low-level sequence diagram.

4.2 Containment Checking in LAYERS

So far, we have presented a scenario from a realistic use case representing the MVC pattern that illustrates how our proposed solution works to identify and resolve the containment inconsistencies of the MVC pattern at different levels of abstraction. As our proposed solution aims to support the software architects and/or developers to verify the containment relationship during their development tasks, it is crucial to assess whether our solution is also applicable for other architecture patterns, like the LAYERS architecture pattern, as well.

In the LAYERS pattern a system is structured into *Layers* in which each *Layer* provides a set of services to the layer above and uses the services of the layer below. Within each layer all constituent components work at the same level of abstraction and can interact through connectors. Between two adjacent layers a clearly defined interface is provided. In the pure form of the pattern, layers should not be by-passed: higher-level layers access lower-level layers only through the layer beneath. However, a relaxed layered scheme loosens

the constraints and allows the by-passing such that a component can interact with components from any lower-level layer. The components in the layer should be organized in such a way that they share a set of common behaviours and one layer member cannot be part of multiple layers.

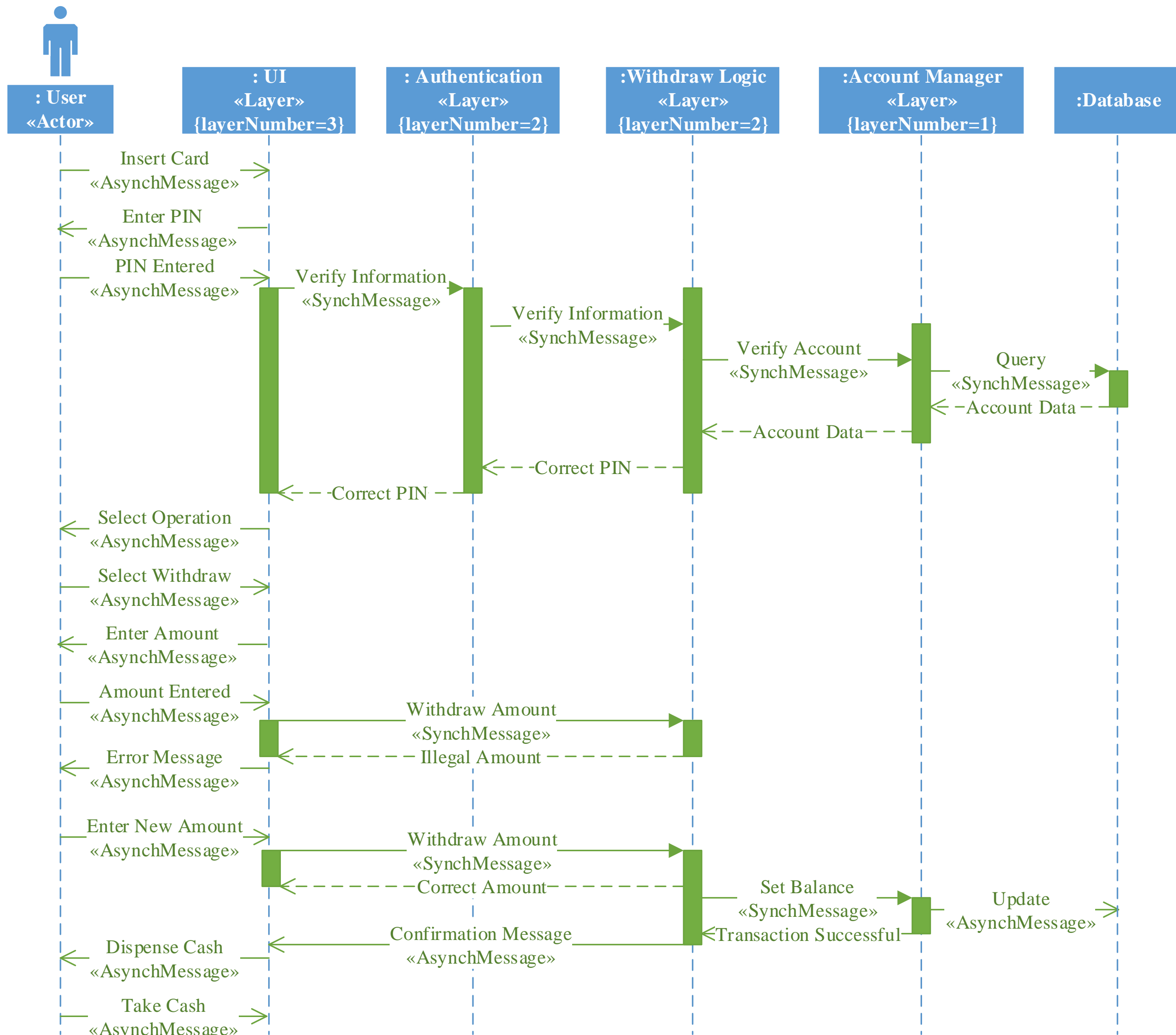

Fig. 4. Modelling Layers Architecture Pattern Using Stereotypes

The generic behaviour of the LAYERS pattern at different levels of abstraction satisfies the containment relationship if elements or behaviours of the high-level model are contained in the detailed design of a LAYERS-based architecture. It is also important that the detailed design follows the same definition of LAYERS patterns as the high-level model. The semantics of UML sequence diagram elements (i.e., lifelines/objects and messages) again need to be extended for modelling the concerns of LAYERS pattern. Therefore, we used the two stereotypes «SynchMessage» and «AynchMessage» from the previous section to map the LAYERS

pattern into sequence diagrams. Specifically, they support synchronous and asynchronous communication between two adjacent layers and members within a layer. The «Actor» stereotype might also be used to model the client/user. In addition, we need to ensure that interactions between lifelines residing in different layers do not allow by-passing; also one layer member cannot be part of multiple layers. Therefore, an additional stereotype is needed to cover the missing aspects concerning these containment relationships – a similar stereotype for components is proposed in [Zdun and Avgeriou 2008].

—«Layer»: A stereotype that extends the Lifeline metaclass of UML and owns occurrence specifications for interaction with lower layer. The «Layer» stereotype allows lifelines who are members of the upper layer (e.g., layer N) to interact with their fellow lifelines in layer N, as well as lifelines in layer N-1 but does not allow them to communicate with other layers (e.g., N-2 and below). We also support the tag definition *layerNumber* (`+layerNumber:Integer`) for layers – representing the number of the layer in the ordered structure of layers.

Figure 4 shows a realistic scenario, namely, an ATM (Automated Teller Machine) system modelled with the stereotypes for expressing the LAYERS pattern. The User (actor) of an ATM machine, can access his/her bank accounts in order check account balance, deposit funds, make cash withdrawals and/or transfer funds. In particular, the User interacts with the ATM through a UI (User Interface, Layer 3) by inserting a bank card and entering a PIN (Personal Identification Number). This information will be sent to the Account Manager (Layer 1) for validation via the Withdraw Logic (Layer 2). After the correct verification, the User chooses the withdraw cash operation from the menu and enters the amount. The UI layer connects to the Withdraw Logic (Layer 2) for verifying the customers' balance. If the requested withdrawal amount is less than or equal to the user's available balance, the transaction will be performed, and the customer will take the money from dispenser. Otherwise, an error message is sent and the transaction prompts the User to enter a new amount. After every withdrawal, the updated balance, the withdrawal amount, and other details of the transaction will be stored in the Bank Database. Using our extensions to UML sequence diagrams it is possible to support containment checking for the explained containment relationships akin to the support for MVC explained before.

4.3 Containment Checking in PIPE AND FILTER

In this section, we describe the identification and resolution of containment inconsistencies in the PIPE AND FILTER pattern's behaviour at different levels of abstraction. In a PIPE AND FILTER architecture a complex task is divided into several sequential subtasks. Each of these subtasks is implemented by a separate, independent component (or filter), which handles only this task. Filters are connected through pipes, each of which transmits outputs of one filter to the inputs of another filter [Buschmann et al. 1996]. A Data Source produces an output stream without any input and supplies data streams to the first pipe. A Data Sink consumes an input stream but does not produce any output. The elements in the PIPE AND FILTER pattern can vary in the functions they perform. For instance, filters can be characterised into active and passive filters based on their input/output behaviour. An active filter starts processing on its own as a separate program or thread. It pulls in data and pushes out the transformed data. A passive filter is activated by being called either as a function (pull output data) or as a procedure (push input data). Pipes can buffer data between filters, form feedback loops or synchronize the filters.

Similar to the MVC and LAYERS patterns, UML sequences diagrams need to be extended in order to enable containment checking. Based on this, the containment checking can be performed to ensure that the generic behaviour described by the low-level model of a software system that is based on the PIPE AND FILTER pattern encompasses those specified in the high-level counterparts, akin to the support for the MVC pattern described

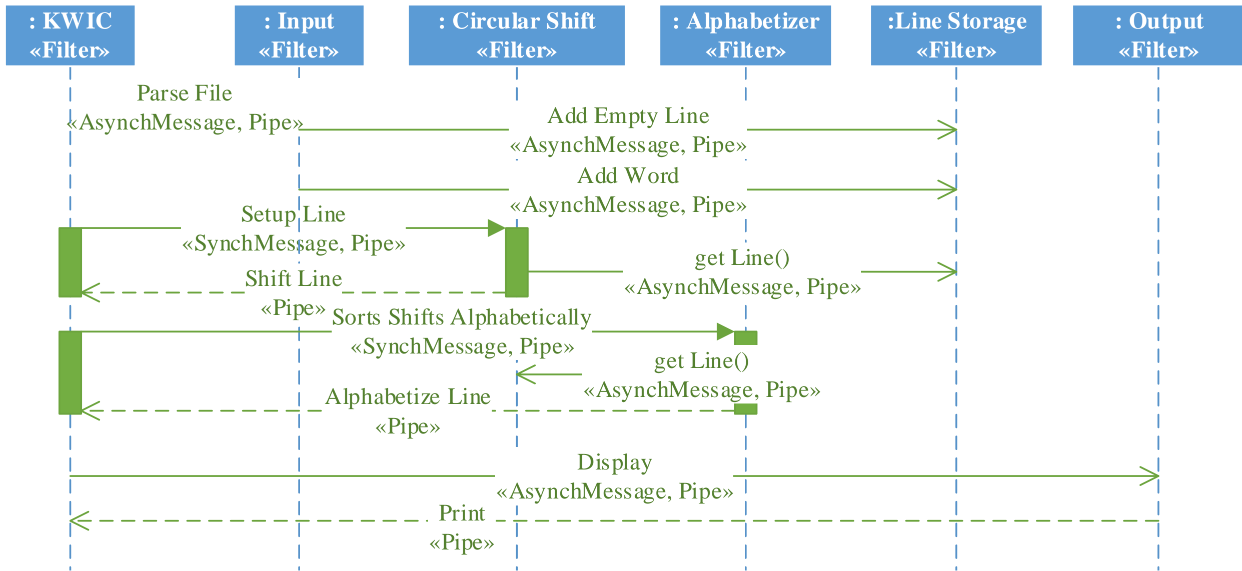

Fig. 5. Modelling Pipe And Filter Pattern Using Stereotypes

above. Here, we require at least «Filter» and «Pipe» stereotypes to denote the participants of the pattern (as also introduced in [Kamal and Avgeriou 2008]), and again the «AynchMessage» and «SynchMessage» stereotypes described above. The «Filter» stereotype is used to depict the lifelines/objects that transmit streams of data, and «Pipe» is used for message interaction from source object (filter) to target object (adjacent filter). The «AynchMessage» and «SynchMessage» stereotypes are used to specify asynchronous and synchronous communication from one filter to the next filter in the chain, respectively.

—The «Filter» stereotype extends the Lifeline metaclass of UML and covers the occurrence specification of the associated pipes.

—The «Pipe» stereotype extends the Message metaclass of UML and connects the occurrence specification of a sender lifeline to the occurrence specification of a receiver lifeline.

Figure 5 shows the stereotypes used for expressing the PIPE AND FILTER pattern in a KeyWord In Context (KWIC) system. The modelled system is composed of KWIC, Input, Circular Shift, Alphabetizer, Line Storage and Output filters; whereas the pipes act as intermediate buffers to facilitate communication between filters. The KWIC process is initiated upon receiving an input file that contains a set of lines. The Input filter reads the input file and writes the parsed lines to the Line Storage filter. Subsequently, the Circular Shift and Alphabetizer filters are invoked to perform the circular shifts and sorting respectively. Finally, the Output filter is invoked to write the results on an output file. It is therefore possible to perform the identification and resolution of containment inconsistencies for the PIPE AND FILTER patterns' behaviour.

5. DISCUSSION

The proposed containment checking approach not only provides a set of stereotypes for correctly modelling and mapping the elements of architectural patterns to UML2 sequence diagrams but also gives informative and comprehensive feedbacks to software architect and/or developer for identifying the causes of containment violations and their resolutions. At the current level of development, our approach is applied to

the MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER architectural patterns. Nevertheless, the proposed approach is applicable to other patterns such as CLIENT-SERVER, BROKER and PRESENTATION-ABSTRACTION-CONTROL patterns and to other behaviour models (used for modelling architectural patterns) such as UML state machines, communication diagrams and activity diagrams.

A limitation of our approach is the extra effort needed for defining stereotypes and formalization of rules, but this extra effort is necessary only once per pattern (variant) and behaviour model type. Then the same benefits as presented in this paper might be gained for other behaviour model types and architectural patterns. However, for each model, modelled using our approach, the comparatively small extra effort for annotating the model with the proposed stereotypes remains.

Note that the containment relationship between the generic behaviour of architectural patterns at different levels of abstraction is based on the assumption that element names of a high-level model and its corresponding low-level counterparts are aligned to a common ontology respected by all stakeholders. The assumption is rather realistic because a low-level model is mainly achieved through a refinement of a high-level model where existing high-level elements are often enriched with more details and elements [Tran et al. 2011]. However, in cases of mismatches of their names and types, one possibility to alleviate this problem, like in the approaches on checking behaviour similarity [Becker and Laue 2012], is to employ supporting text matching techniques [Navarro 2001].

6. CONCLUDING REMARKS

This study focuses on the identification and resolution of containment violations in architecture patterns' behaviour at different abstraction levels. In this context, we have performed a systematic review of behaviour consistency checking research [Muram et al. 2017], investigated various behaviour models including activity diagrams, sequence diagrams and BPMN models, as well as possible solutions based on model-checking techniques and graph algorithms. This research helped us in the identification of violation causes of an architectural patterns' behaviour in various activities of the development process. Our approach helps to automatically ensure the consistency of the pattern realization in the software architecture, the detailed design and the implementation. The applicability of the proposed solution is demonstrated for the MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER patterns. We have suggested one specification extension of UML2 for modelling the behaviour of each of those patterns; please note that our containment checking approach is not dependent on those specific formalization, but can also be applied on different ones and also other variants of the covered and other patterns.

For future work we plan to develop a process related to refactoring, a catalogue of applicable refactorings, and evaluations of that catalogue of refactorings. The intention is that the architecture should not be violated during code or model refactorings.

7. ACKNOWLEDGEMENTS

The authors would like to thank our EuroPLoP 2017 shepherd Michael John for his valuable comments. This work is supported by the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001, University of Vienna, and Austrian Science Fund (FWF), Grant No. I 2885-N33.

REFERENCES

Paris Avgeriou and Uwe Zdun. 2005. Architectural Patterns Revisited - A Pattern Language. In *EuroPLoP*, Andy Longshaw and Uwe Zdun (Eds.). UVK - Universitaetsverlag Konstanz, Konstanz, Germany, 431–470.

Len Bass, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice* (2 ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Michael Becker and Ralf Laue. 2012. A Comparative Survey of Business Process Similarity Measures. *Computers in Industry* 63, 2 (2012), 148–167. DOI:http://dx.doi.org/10.1016/j.compind.2011.11.003

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing.

Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford. 2003. Documenting Software Architectures: Views and Beyond. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 740–741. DOI:http://dx.doi.org/10.1109/ICSE.2003.1201264

Amnon H. Eden, Yoram Hirshfeld, and Kristina Lundqvist. 1999. LePUS - Symbolic Logic Modeling Of Object Oriented Architectures: A Case Study. In *NOSA '99 Second Nordic Workshop on Software Architecture, University of Karlskrona/Ronneby*. Citeseer.

Lukasz Fryz and Leszek Kotulski. 2007. Assurance of System Consistency During Independent Creation of UML Diagrams. In *Proceedings of the 2Nd International Conference on Dependability of Computer Systems (DEPCOS-RELCOMEX '07)*. IEEE Computer Society, Washington, DC, USA, 51–58. DOI:http://dx.doi.org/10.1109/DEPCOS-RELCOMEX.2007.11

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

David Garlan, Robert Allen, and John Ockerbloom. 1994. Exploiting Style in Architectural Design Environments. In *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '94)*. ACM, New York, NY, USA, 175–188. DOI:http://dx.doi.org/10.1145/193173.195404

David Garlan and Andrew J. Kompanek. 2000. Reconciling the Needs of Architectural Description with Object-modeling Notations. In *Proceedings of the 3rd International Conference on The Unified Modeling Language: Advancing the Standard (UML'00)*. Springer-Verlag, Berlin, Heidelberg, 498–512. DOI:http://dx.doi.org/10.1007/3-540-40011-7_37

Abdelouahed Gherbi and Ferhat Khendek. 2007. Consistency of UML/SPT Models. In *Proceedings of the 13th International SDL Forum Conference on Design for Dependable Systems (SDL'07)*. Springer-Verlag, Berlin, Heidelberg, 203–224. DOI:http://dx.doi.org/10.1007/978-3-540-74984-4_13

Simon Giesecke, Matthias Rohr, Florian Marwede, and Wilhelm Hasselbring. 2007. A Style-based Architecture Modelling Approach for UML 2 Component Diagrams. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications (SEA '07)*. ACTA Press, Anaheim, CA, USA, 530–538. http://dl.acm.org/citation.cfm?id=1647636.1647728

Object Management Group. 2011a. Business Process Model And Notation. http://www.omg.org/spec/BPMN/2.0. (2011). Last accessed: 2017-04-10.

Object Management Group. 2011b. UML 2.4.1 Superstructure Specification. http://www.omg.org/spec/UML/2.4.1. (2011). Last accessed: 2017-05-20.

Ahmad Waqas Kamal and Paris Avgeriou. 2008. Modeling Architectural Patterns' Behavior Using Architectural Primitives. In *Proceedings of the 2Nd European Conference on Software Architecture (ECSA '08)*. Springer-Verlag, Berlin, Heidelberg, 164–179. DOI:http://dx.doi.org/10.1007/978-3-540-88030-1_13

Ahmad Waqas Kamal, Paris Avgeriou, and Uwe Zdun. 2008. Modeling variants of architectural patterns. In *Proceedings of 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008)*, Vol. 610. CEUR-WS.org, Irsee, Germany, 1–23.

Soon-Kyeong Kim and David A. Carrington. 2002. A Formal Model of the UML Metamodel: The UML State Machine and Its Integrity Constraints. In *Proceedings of the 2Nd International Conference of B and Z Users on Formal Specification and Development in Z and B (ZB '02)*. Springer-Verlag, London, UK, UK, 497–516. DOI:http://dx.doi.org/10.1007/3-540-45648-1

Vitus S. W. Lam and Julian A. Padget. 2005. Consistency Checking of Sequence Diagrams and Statechart Diagrams Using the pi-Calculus. In *Integrated Formal Methods, 5th International Conference, IFM 2005, Eindhoven, The Netherlands, November 29 - December 2, 2005, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 347–365. DOI:http://dx.doi.org/10.1007/11589976_20

Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. 2009. A systematic review of UML model consistency management. *Inf. Softw. Technol.* 51, 12 (Dec. 2009), 1631–1645. DOI:http://dx.doi.org/10.1016/j.infsof.2009.04.009

Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. 2004. Precise Modeling of Design Patterns in UML. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 252–261. DOI:http://dx.doi.org/10.1109/ICSE.2004.1317447

Axel Martens. 2005. Consistency Between Executable and Abstract Processes. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service (EEE '05)*. IEEE Computer Society, Washington, DC, USA, 60–67. DOI:http://dx.doi.org/10.1109/EEE.2005.53

Nenad Medvidovic and Richard N. Taylor. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.* 26, 1 (Jan. 2000), 70–93. DOI:http://dx.doi.org/10.1109/32.825767

Nikunj R. Mehta and Nenad Medvidovic. 2003. Composing Architectural Styles from Architectural Primitives. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11)*. ACM, New York, NY, USA, 347–350. DOI:http://dx.doi.org/10.1145/940071.940118

Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-specific Languages. *ACM Comput. Surv.* 37, 4 (Dec. 2005), 316–344. DOI:http://dx.doi.org/10.1145/1118890.1118892

Tommi Mikkonen. 1998. Formalizing Design Patterns. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*. IEEE Computer Society, Washington, DC, USA, 115–124. DOI:http://dx.doi.org/10.1109/ICSE.1998.671108

Faiz UL Muram, Muhammad Atif Javed, Huy Tran, and Uwe Zdun. 2017. Towards a Framework for Detecting Containment Violations in Service Choreography. In *Proceedings of the IEEE International Conference on Services Computing (SCC '17)*. IEEE Computer Society, Washington, DC, USA, 172–179. DOI:http://dx.doi.org/10.1109/SCC.2017.29

Faiz UL Muram, Huy Tran, and Uwe Zdun. 2014. Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking. In *11th Int'l Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA), Grenoble, France*. 93–107. DOI:http://dx.doi.org/10.4204/EPTCS.147.7

Faiz UL Muram, Huy Tran, and Uwe Zdun. 2015. Counterexample Analysis for Supporting Containment Checking of Business Process Models. In *Business Process Management Workshops - BPM 2015, 13th International Workshops, August 31 - September 3, 2015, Revised Papers (Lecture Notes in Business Information Processing)*, Vol. 256. Springer, 515–528. DOI:http://dx.doi.org/10.1007/978-3-319-42887-1_41

Faiz UL Muram, Huy Tran, and Uwe Zdun. 2016. A Model Checking Based Approach for Containment Checking of UML Sequence Diagrams. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC '16)*. IEEE Computer Society, Washington, DC, USA, 73–80. DOI:http://dx.doi.org/10.1109/APSEC.2016.021

Faiz ul Muram, Huy Tran, and Uwe Zdun. 2017. Systematic Review of Software Behavioral Model Consistency Checking. *ACM Comput. Surv.* 50, 2, Article 17 (April 2017), 39 pages. DOI:http://dx.doi.org/10.1145/3037755

Gonzalo Navarro. 2001. A Guided Tour to Approximate String Matching. *Comput. Surveys* 33, 1 (2001), 31–88. DOI:http://dx.doi.org/10.1145/375360.375365

Jean-Marc Perronne, Laurent Thiry, and Bernard Thirion. 2006. Architectural Concepts and Design Patterns for Behavior Modeling and Integration. *Math. Comput. Simul.* 70, 5-6 (Feb. 2006), 314–329. DOI:http://dx.doi.org/10.1016/j.matcom.2005.11.004

Holger Rasch and Heike Wehrheim. 2003. *Checking Consistency in UML Diagrams: Classes and State Machines.* Springer Berlin Heidelberg, Berlin, Heidelberg, 229–243. DOI:http://dx.doi.org/10.1007/978-3-540-39958-2_16

Nick Rozanski and Eóin Woods. 2005. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Addison-Wesley Professional, Upper Saddle River, NJ.

Kristin Y. Rozier. 2011. Survey: Linear Temporal Logic Symbolic Model Checking. *Computer Science Review* 5, 2 (2011), 163–203. DOI:http://dx.doi.org/10.1016/j.cosrev.2010.06.002

Bran Selic. 1998. Using UML for Modeling Complex Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98)*. Springer-Verlag, London, UK, UK, 250–260. DOI:http://dx.doi.org/10.1007/BFb0057795

Mary Shaw and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Karina Sokolova, Marc Lemercier, and Ludovic Garcia. 2013. Android passive MVC: a novel architecture model for the android application development. In *International Conference on Pervasive Patterns and Applications*.

Neelam Soundarajan and Jason O. Hallstrom. 2004. Responsibilities and Rewards: Specifying Design Patterns. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 666–675. DOI:http://dx.doi.org/10.1109/ICSE.2004.1317488

George Spanoudakis and Andrea Zisman. 2001. *Handbook of Software Engineering and Knowledge Engineering*. World Scientific, Singapore, Chapter Inconsistency management in software engineering: Survey and open research issues, 329–380.

Markus Stumptner and Michael Schrefl. 2000. Behavior Consistent Inheritance in UML. In *Proceedings of the 19th International Conference on Conceptual Modeling (ER'00)*. Springer-Verlag, Berlin, Heidelberg, 527–542. DOI:http://dx.doi.org/10.1007/3-540-45393-8_38

Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1 (1972), 146–160.

Huy Tran, Faiz UL Muram, and Uwe Zdun. 2015. A Graph-Based Approach for Containment Checking of Behavior Models of Software Systems. In *Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Conference (EDOC '15)*. IEEE Computer Society, Washington, DC, USA, 84–93. DOI:http://dx.doi.org/10.1109/EDOC.2015.22

Huy Tran, Uwe Zdun, and Schahram Dustdar. 2011. Name-based view integration for enhancing the reusability in process-driven SOAs. *IJBPIM* 5, 3 (2011), 229–239. DOI:http://dx.doi.org/10.1504/IJBPIM.2011.042527

Ninh-Thuan Truong, Thi-Mai-Thuong Tran, Van-Khanh To, and Viet-Ha Nguyen. 2009. Checking the Consistency Between UCM and PSM Using a Graph-Based Method. In *Proceedings of the 2009 First Asian Conference on Intelligent Information and Database Systems (ACIIDS '09)*. IEEE Computer Society, Washington, DC, USA, 190–195. DOI:http://dx.doi.org/10.1109/ACIIDS.2009.66

Aliki Tsiolakis and Hartmut Ehrig. 2000. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In *Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*. Open Publishing Association, 77–86.

Wil M. P. van der Aalst. 2002. Inheritance of dynamic behaviour in UML. In *2nd International Workshop on Modelling of Objects, Components and Agents (MOCA)*. Aarhus, Denmark, 105–120.

Ragnhild van der Straeten. 2005. *Inconsistency Management in Model-Driven Engineering An Approach using Description Logics*. Doctoral Dissertation. Vrije Universiteit Brussel.

Shuzhen Yao and Sol M. Shatz. 2006. Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. In *Proceedings of the 15th International Conference on Computing (CIC '06)*. IEEE Computer Society, Washington, DC, USA, 289–297. DOI:http://dx.doi.org/10.1109/CIC.2006.32

Uwe Zdun and Paris Avgeriou. 2005. Modeling Architectural Patterns Using Architectural Primitives. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 133–146. DOI:http://dx.doi.org/10.1145/1094811.1094822

Uwe Zdun and Paris Avgeriou. 2008. A Catalog of Architectural Primitives for Modeling Architectural Patterns. *Inf. Softw. Technol.* 50, 9-10 (Aug. 2008), 1003–1034. DOI:http://dx.doi.org/10.1016/j.infsof.2007.09.003