An Approach to Automated Realization and Validation of Software Architecture Model – A Case Study on E-Commerce

Zhijiang Dong Yujian Fu Xudong He

School of Computer Science
Florida International University
E-mail: {zdong01, yfu002, hex}@cs.fiu.edu

Abstract

The benefits of architecture description languages (ADLs) cannot not be fully captured without an automated realization of software architecture designs because manually shifting from a model to its implementation is error-prone. In addition, validation of the realization process is necessary to ensure system properties kept after the realization. We proposed an integrated approach for automatically translating software architecture design models to the implementation code and validating the translation as well as the implementation code by exploring runtime verification technique and aspect-oriented programming. Specifically, system properties are not only verified against design models, but also verified during execution of the generated implementation of software architecture designs. A prototype tool, SAM Parser, is developed to demonstrate the approach on an ADL – SAM (Software Architecture Model). In SAM Parser, all the realization and verification code can be automatically generated without human interference.

In this paper, we report on a case study conducted at an e-commerce scenario, an online shopping system to assess the benefits of automated realization of software architecture design and validation in a web service domain.

1. Introduction

Software architecture plays a critical role in software development processes since it helps us further understand the system through the construction of high-level system structure and it becomes the corner stone for subsequent software development activities. Therefore, lots of work have been done in software engineering community to validate and verify architecture against system requirements or specifications.

However, a complete and correct software architecture at design level does not ensure the correctness of its implementation because manually shifting from model to its implementation is error-prone, and system properties are not guaranteed to hold in the implementations [2]. In order to attack this problem, on the one hand, an automatic translation with tool support is necessary to prevent man-made errors. Although automatic programming from a formal specification is in general impossible [4], generating the implementation from design models automatically is viable since architectural design provides enough details. Currently, some architecture description languages (ADLs) support the implementation of architectural design in a number of ways [31, 21], but none of them can enforce communication integrity [22, 19] in the implementation that is necessary to enable architectural reasoning about an implementation [2]. On the other hand, properties held in the architecture should be guaranteed to hold in the implementations. This means, the transformation not only ensures the functionality correctness of implementations, but also guarantees the correctness of implementations with regard to architectural properties. Unfortunately, none of current ADLs tools can achieve this goal.

In this paper, we propose an approach to achieve above goals, i.e. not only implements software architectures automatically, but also verifies if an architectural property is satisfied at the implementation that validates the implementation indirectly. We applied the approach to an e-commerce scenario to show its feasibility and practicability. As Fig. 1 indicates, the structure and the behavior of software architectures are realized in Java and ArchJava [2] respectively by SAM Parser. These codes, called functionality code, simulate the execution of architectures. The property specifications that describe important behavioral properties are implemented as aspects of components or connectors. The aspects containing runtime verification code are weaved into functionality code through hooks (join-points) provided by aspect-oriented programming. To our best knowledge, we are the first to integrate automated realization, runtime verification and aspect-oriented program-
ming seamlessly for software architectures, which brings some benefits that cannot be achieved by using individual technique: verifying architecture design models at implementa-
tion level, presenting counter examples for property violation, validating automated realization process, providing potentiality to detect exceptions and steer model execution at runtime, and most importantly these work can be done automatically.

![Software Architecture Diagram](image)

**Figure 1. Framework of Our Approach**

2. Preliminaries

In our work, SAM (Software Architecture Model) [15] is chosen as the architectural description language because not like other ADLS, SAM not only provides means to define structure and behavior of software architecture, but also provide means to specify behavioral properties for components and connectors that should hold in the architecture.

SAM is an architectural description model based on Petri nets [23], which are well-suited for modeling distributed systems. SAM [15] has dual formalisms underlying – Petri nets and Temporal logic. Petri nets are used to describe behavioral models of components and connectors while temporal logic is used to specify system properties of components and connectors.

SAM architecture model is hierarchically defined as follows. A set of compositions $C = \{C_1, C_2, ..., C_k\}$ represents different design levels or subsystems. A set of component $C_m$ and connectors $C_n$ are specified within each composition $C_i$ as well as a set of composition constraints $C_s$, e.g. $C_i = (C_m, C_n, C_s)$. In addition, each component or connector is composed of two elements, a behavioral model and a property specification, e.g. $C_{ij} = (S_{ij}, B_{ij})$. Each behavioral model is described by a Petri net, while a property specification by a temporal logical formula. The atomic proposition used in the first order temporal logic formula is the ports of each component or connector. Thus each behavioral model can be connected with its property specification. A component $C_m$ or a connector $C_n$ can be refined to a lower level composition $C_l$ by a mapping relation $h$, e.g. $h(C_m) = C_l$.

SAM gives the flexibility to choose any variant of Petri nets and temporal logics to specify behavior and constraints according to system characteristics. In our case, Predicate Transition (PrT) net [11] and linear temporal logic (LTL) are chosen.

Predicate Transition (PrT) net [11] is a high level Petri net. A PrT has a net structure: $(P, T, F)$, where $P$ is a set of places represented by circles, $T$ is a set of transitions represented by rectangles and $T$ is disjoint from $P$, and $F$ is a relation between $P$ and $T$ represented by arcs. Each place is assigned a sort indicating what kind of tokens it can contain. The tokens in a place can be viewed as a multi-set over the sort. A marking of a PrT net is a function that assigns tokens to each place. A label is assigned to each arc to describe types and numbers of tokens that flow along this arc. Each transition has a boolean expression called guard, which specifies the relationship among arcs related with the transition. A transition is enabled if there is an assignment to all variables occurred in arcs related with the transition such that each incoming place contains the set of tokens specified by the label of the arc, and the guard of the transition is satisfied. An enabled transition is fired under an assignment by removing tokens from incoming places and adding tokens to outgoing places. Fig. 2 shows a simple PrT net of dining philosopher problem. In the figure, transition pickup is fired with variable assignment $(x=1, r=1, l=2)$, which describes the action of the philosopher 1: picking up his fork to eat.

3. Overview of the Approach

Fig. 3 shows the framework of our approach. The approach is implemented as the SAM parser, which is responsible for automatically generating functionality code and runtime verification code. Runtime verification code is weaved into functionality code through joinpoints provided by aspect-oriented programming. The system structure of SAM Parser is shown in Fig. 3.

The input of the SAM parser is a XML file, which specifies SAM structures (such as components, connectors, ports and their relationships) and property specifications. In the XML file, SAM behavior is defined by referring to a Petri
Net Markup Language (PNML) [6] file, which is an XML-based interchange format for Petri nets. The logic engine is responsible to construct a piece of pseudo code called monitoring code for each temporal logic formula. A piece of monitoring code is invoked to check if the corresponding formula is satisfied whenever an interesting event occurs. The logic server is a middleware that translates monitoring code to target language, here Java. The SAM parser merges all translated monitoring code for properties of a component or a connector into an aspect. All generated aspects are called runtime verification code.

Figure 2. Petri Net of Dining Philosopher

3.1. Automated Generation of Functionality Code

Automated realization of functionality code for SAM models consists of two parts: generating code for the structure and the behavior respectively. In order to generate code for behavior (PrTs), we predefine a set of classes called templates, which specify structure and dynamic semantics of high level Petri nets. For example, the basic elements of Petri nets such as places, arcs, transitions, guards, inscriptions are defined by individual classes. We also provide dynamic semantics of Petri nets in Java classes Net and Transition. In our work, a class is constructed as a child of templates for each net, arc inscription, and guard. The user can provide a more efficient way to check the enable-ness of a transition and the way to fire it by overloading methods of corresponding classes without any side effects on other transitions. The execution of generated code is non-deterministic, i.e. an enabled transition and a valid assignment is randomly chosen to fire.

It is hard to generate code automatically for any given a Petri net due to the complexity of sorts, guard conditions of transition and arc labels [18]. In our work following some restrictions of Petri nets as we specified we can achieve:

- The sorts of Petri nets are Java predefined types with its methods.
- If a variable is a product type, whose elements are Java predefined types.
- The type of variables occurred in the label of an incoming arc of a transition is the same as the token type of the incoming place.
- Only labels of incoming arcs of transitions can introduce variables.

SAM structure is translated as ArchJava [2] code by the SAM parser. ArchJava is an extension to Java that seamlessly unifies software architecture with implementation, which uses a type system to ensure that the implementation conforms to architectural constraints. In other words, ArchJava is proposed to avoid inconsistency, confusion, and violation of architecture properties when decoupling implementation code from software architecture. To our best knowledge, ArchJava is the best candidate to the target language for the implementation of SAM structure – not only because it provides architecture concepts such as
components, ports as first-level entities, but also because it enforces communication integrity. A system has communication integrity of implementation if components only communicate directly with the components they are connected to in the architecture.

In the SAM Parser, it is straightforward to realize components/ connectors, compositions, and ports as ArchJava entities such as components, component compositions, and ports respectively. More specifically, an incoming/outgoing port in SAM is realized by a ArchJava port that declares a provides/requires method. A SAM component/connector is realized as a ArchJava component class that consists of the declarations of ports, the mapping between ports and places of its behavioral Petri net, reference to its generated behavioral code, and other necessary methods. A SAM composition is realized as a ArchJava component composition that specifies and establishes dynamic connections among sub-components and contains port declarations if necessary. More detailed information about automated generation of functionality code can be found at [10].

3.2. Automated Generation of Runtime Verification Code

Runtime verification [27, 28, 29, 30] has been proposed as a lightweighted formal method applied during the execution of programs. It can be viewed as a complement to traditional methods of proving design model or programs correct before execution. Aspect-oriented software engineering [25, 13, 24] and aspect-oriented programming [9] were proposed to separate concerns during design and implementation. Aspect-Oriented Programming complements OO programming by allowing the developer to dynamically modify the static OO model to create a system that can grow to meet new requirements. In other words, it allows us to dynamically modify models or implementations to include code required for secondary requirements (in our case, it is runtime verification) without modifying the original code.

The SAM parser generates runtime verification code automatically and weaves it into functionality code seamlessly without side effects on the functionality code. In order to generate monitoring codes for properties (linear temporal formulae), a logic server, Maude [20] in our case, is necessary. Maude, acting as the main algorithm generator in the framework, constructs an efficient dynamic programming algorithm (i.e., monitoring code) from any LTL formula [26]. The generated algorithm can check if the corresponding LTL formula is satisfied over an event trace.

The SAM parser weaves monitoring code into functionality code by integrating them as aspects. In aspect-oriented programming, AspectJ [3] in our case, aspects wrap up pointcuts, advice, and inter-type declarations in a modular unit of crosscutting implementation where pointcuts pick out certain join points in the program flow, advice brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points), and Inter-type declarations are declarations that cut across classes and their hierarchies. In our case, for each component or connector, pointcuts specify time spots: whenever a port sends or receives a message; pieces of code brought together by advice with pointcuts are the generated monitoring code; and Inter-type declaration specifies helper variables and methods. Fig. 4, which is a part of generated aspect for composition Customer in Fig. 6, clearly shows the way to weave runtime verification code into functionality code through aspects. Currently the SAM parser can handle future time linear temporal formulae as well as past time linear temporal formulae.

By combining runtime verification and automated implementation of software architecture, we can obtain the following benefits:

- The transformation from design models to implementations is generally informal, therefore error-prone. Automated implementation provides a means to prevent man-made errors, and runtime verification can validate transformation indirectly.

- Runtime verification at implementation level is a natural complement to analysis techniques of design level. Not all properties can be verified against a design model either due to the state space explosion problem or due to characteristic of open-systems. In either case, runtime verification can be explored to verify the correctness of design models.

- Runtime verification provides a mechanism to handle exceptions of implementations that are not detected during development or testing.

4. Case Study – An E-Commerce Scenario

In this paper we present an application of our approach to the basic electronic commerce process of online shopping with credit card transaction as a case in point to demonstrate SAM Parser system and breads of scope. However, our approach is domain independent, i.e. SAM Parser does not depend on a particular domain of discourse but rather on fundamental definitions of an ontology that captures what a component and a connector are. Due to this generic nature of SAM Parser, it is found to be most suitable for developing systems in a large variety of domains.

Fig. 5, adapted from [11] and [33], describes online shopping processing in a free natural language, while an accompanying pictorial scheme provides the details of electronic shopping that precedes the payment processing.
The description in Fig. 5 concerns details of the shop system, which is the focus of the e-business depicted in [33]. The two modules - the front platform for the user for the graphically and textually representation of shopping operations and the back platform for the retailer for organization and billing, etc. are separated into 5 blocks of services – Costumer, ShoppingCart, Warehouse, OrderProcessing, and CreditCardProcessing. The retailer bank is a simple entity in the online shopping system and we don’t concern the black box financial organization of business company. In what follows, we examine the SAM specification constructed, translated code and verification process to study several unique features of SAM Parser.

4.1. SAM Model of Online Shopping System

The top topology of an online shopping system specified in SAM model is demonstrated in Fig. 6. Each description block is captured in a component (or a composition). The compositions, (different concept from the composition in SAM, we subscript with a letter v) between e-services are defined by connectors. Vertically, components/compositions in a lower level can form a composition in the upper level. For instance, block Customer and ShoppingCart is modeled in the component Customer and component Cart, and both components form a composition User in the topology of SAM. In the composition User, a customer can browse the web, choose category, select and checkout items. In addition, a user can also adjust the order and cart list. All these internal behaviors are represented by the refined inner components or Petri nets. Description blocks Warehouse and OrderProcessing are specified in the component Warehouse and Order, which form another composition E-Company. An E-company can evaluate user requests, confirm or reject user requests, submit user information forms and generate orders, etc.. Finally, the component PaymentProcess captures the behavior of the description block CreditCardProcessing. Description block RetailerBank is simply represented in the component RetailerBank.

In SAM, components communicate with each other through ports represented as semicircles. An incoming port, represented by a semicircle inside of the component, only receives messages from other components at the same hierarchy, like port response_15 of composition User in Fig. 6. Similarly, an outgoing port, represented by a sem-
The service provided here is to collect or refund money from or to the customer.

RetailerBank
The authorisation department checks the credit card information, such as the card has not been reported CreditCardProcessing the related information, i.e., order number, is sent to the customer. Otherwise, the transaction is failed.

Information out and waiting for response. If the credit card information is valid, the order is processed and information such as credit card number, credit line, expiration date. It then sends the form of credit card information to the retailer's online platform.

OrderProcessing
The web calculates the total price of all items that a customer requested, and prevalidates the customer information. If the quantity of a required item exceeds the storage in the warehouse, the whole transaction is discarded.

Warehouse
The web keeps all available information of the products of retailers, such as product name, category, price, etc. If a user requests to browse a product, a proper statement for the product is shown up; if a user puts a "checkout" request with products (s)he wants, eventually (s)he gets either an order and shipment information or is rejected because of payment issues, in the later case, an error message is popped up to the user. In SAM model, we have to make sure that compositions User and E-Company behave as expected from description in Fig. 5. In other words, composition Customer obtains a message \((m_2 = \langle \text{uid, pname, category, price} \rangle)\) in the port response_15(m2), it indicates that sometime before the customer sent out a message \(m_1 = \langle \text{uid, pReq, category} \rangle\) through port request_12(m1). This can be expressed by the following formulae on composition Customer:

\[
\Diamond(\text{response}_15(\langle \text{uid, pname, category, price} \rangle)) \quad (1)
\]

\[
\Box(\text{request}_12(\langle \text{uid, pReq, category} \rangle) \rightarrow \\
\Diamond(\text{response}_15(\langle \text{uid, pname, category, price} \rangle))) \quad (2)
\]

The atomic predicate in above formula is in the form \(\text{Port}(m)\), which is evaluated true if specified port contains the message \(m\). For example, predicate \(\text{request}_12(\langle \text{uid, pReq, category} \rangle)\) is true if the port request_12 of composition User has a message \(\langle \text{uid, pReq, category} \rangle\). Our work supports future time linear temporal logic and past time linear temporal logic. Formula 1 is a future time LTL formula, while formula 2 is a past time LTL. In above formulae, \(\Box\), \(\Diamond\), and \(\lor\) are future time operator eventually, and past time operator SometimesInThePast (sometime in the past), and AlwaysInThePast (always in the past) respectively.

There are many key system properties need to be enforced. We list three of them for the composition Customer and \(E-Company\) as follows.

1. Selective response property. When a customer sends a checkout request with products (s)he wants, eventually (s)he gets either an order and shipment information or
the error response on this payment process. The formula is

\[ \square((\text{checkout}_{12}(<\text{uid}, \text{"checkout"}>) \land \text{prdLCt}_{12}(<\text{uid}, \text{name}, category, price, quantity >)) \rightarrow (\diamond(\text{ordNum}_{15}(<\text{uid}, \text{ordNum}>) \land \text{shipEmail}_{15}(<\text{uid}, \text{"shipEmail"}>) \land \text{chkoutErr}_{15}(<\text{uid}, \text{"ccErr"})))) \] (3)

2. Conjoined reverse reasoning property. This property involves more than one reasons about a result. When a customer receives an order information, we can reversely deduce that first the customer got a response on a request for the product(s), second the customer provided his/her credit card information sometime before, finally the customer got the shipment information. The LTL formula for this property is

\[ \Box((\text{ordNum}_{15}(<\text{uid}, \text{ordNum}>) \rightarrow \Phi((\text{request}_{12}(<\text{uid}, \text{"req"}, category) \rightarrow \Phi(\text{response}_{15}(<\text{uid}, \text{name}, category, price}) \land \Phi(\text{shipEmail}_{15}(<\text{uid}, \text{name}, category, price, price >)) \land \Phi(\text{order}_{15}(<\text{uid}, \text{ordNum}>) \land \Phi(\text{shipEmail}_{15}(<\text{uid}, \text{"shipEmail"}>) \land (\text{shipEmail}_{15}(<\text{uid}, \text{"shipEmail"}>) \land (\text{ordNum}_{15}(<\text{uid}, \text{ordNum}>) \land \text{chkoutErr}_{15}(<\text{uid}, \text{"ccErr"})))) \) (4)

3. Cause-and-effect property. This involves more than two components/compositions, and there is a timing order between each two. For instance, the E-Company request to authorize a customer’s payment information (credit card form). After component PaymentProcess responses, the E-Company sends out either an order and shipment information or the credit card error information to the component Customer. The LTL formula for this property is
4.3. Results

The translation of SAM model and generated code of verification takes about 1.5 seconds for the SAM parser on a P4 2.4Ghz machine with 512MB RAM. The generated implementation has 280 files, and it is executable without any modification. Most of the files are the implementation of components or connectors behavior (Petri nets). The statistical data is shown in Table 1. The reason of generating so many files is due to the most important principle for the SAM parser: We have to make the generated code easy to understand and minimize the cost of modification. It takes about 9.6 seconds for the generated implementation to execute and verify 8 formulae involved four compositions or components. The execution of the generated implementation fires transition 31 times, i.e. almost one transition is fired every second. Most of the time is spend on the search of enabled transition and valid assignments to variables. The code can be manually optimized for critical transitions by overriding methods that judge if a transition is enabled.

From the log file, we also can see that the formulae 1, 3, 5 holds. However, the evaluations of formulae 2 is neither true nor false. This seems strange at first since the purpose of runtime verification was to check if formulae are satisfied or not. However this result is correct because these two formulae are past time LTL, which are supposed to be always satisfied. In other words, runtime verification code only reports exception for past time LTL formulae if it is violated, just like code in Fig. 4. Therefore, the unsure result for formula 2 indicate that there are no violation detected. Same reason for the formula 4. This means these formulae hold during the program execution, which assures the behavioral correctness of Customer, E-Company, and PaymentProcess.

5. Related Work and Discussion

We discuss related work in two parts – one part related with ADLs and their tool supports, another one related with runtime verification technique.

Currently, most ADLs such as MetaH [32], Unicon [31] and Weaves [12], support semi-automatic code generation from an architecture model. However, none of them can enforce communication integrity [22, 19] in the implementation that is necessary to enable architectural reasoning about an implementation [2]. Moreover, there little work on the validation of the code generation for these tools.

Several tools are developed for the runtime verification of programs. Java PatheXplorer [14] (JPaX) is designed for monitoring Java programs on the execution and the concreteness analysis. JPaX uses the Maude as the logic engine to transform system properties into pseudo code for monitors. MaC [17] is a general framework for runtime checking of programs. The features of MaC are user can define their logics and monitors are separated from programs. JavaMaC [16] is the MaC application on Java programs. Several other approaches to runtime verification (especially for Java) exist. Temporal Rover/DBRover [7, 8] checks time-dependent specifications and handles assertions embedded in comments by source-to-source transformation. Jass [5] is a pre-compiler that translates annotated assertions into Java programs using Design by Contract, etc..

The difference of SAM Parser exists in the following aspects. First, SAM Parser is designed for the software architecture model or ADLs, which is more abstract, general and accurate than common Java programs. Second, event trace in SAM Parser is online while in JPaX/Java-MaC is offline because we use AspectJ in our work. Last, JPaX/MaC separate the monitoring code from the monitored code in which the benefits are the distributed implementation and parallel execution of the monitors and the program, and lower performance penalty. In comparison, in SAM Parser, the monitors are intermixed with the program. However, we still gain the good performance because of the usage of AspectJ. Moreover, intermixed monitors are more efficient than that separated ones by saving the data conveying time.

6. Conclusions

We have proposed an integrated framework for validating conformance of an architecture model to system properties automatically. The architecture model is automated realized in ArchJava/Java through a SAM parser. SAM parser not only generates code for structure and behavior of an architecture design model, but also generate runtime verification codes for system properties. The generated functionality code and runtime verification code is executable without any manual modification if SAM specifications follow the restrictions mentioned in section 3.1. By integrating automated realization technique and runtime verification technique on architecture design models, we not only verify the correctness of design models against system properties, but also validate the automated realization process indirectly.

To our best knowledge, this is the first work to combine runtime verification, aspect-oriented programming with automated realization of architectural design models.

We applied the approach on a case study of an e-commerce application – an online shopping system. In this application we model each service or group of services as a
component/composition in SAM. Two services can be composed through a connector. A connector is geared through ports. This illustrates that the connector in SAM can be used for modeling and analysis of service composition, and e-service composability. Moreover, the SAM parser can be a tool for automated composition, of e-services. Finally, this work opens perspectives for SAM specification with runtime validation on the dynamic inter-operation of highly distributed, heterogeneous network services. Future work can be several aspects. We expect to extend our approach to first order LTL formulae. We also plan to conduct more case studies on the web applications to explore more features of service-oriented architecture using our approach of automated realization and validation of ADLs.

Acknowledgements This work is supported in part by NSF under grant HRD-0317692 and by NASA under grant NAG 2-1440.

References


Table 1. Statistic Data of Generated Files

<table>
<thead>
<tr>
<th>ArchJava Files</th>
<th>AspectJ Files</th>
<th>Java Files</th>
<th>Properties</th>
<th>Types</th>
<th>Folders</th>
<th>Total Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>6</td>
<td>273</td>
<td>8</td>
<td>17</td>
<td>30</td>
<td>291</td>
</tr>
</tbody>
</table>


