JOURNAL OF AERONAUTICS AND SPACE TECHNOLOGIES
(ISSN : 1304-0448)
July 2020 Volume 13 Number 2
www.jast.hho.edu.tr

Research Article

The General-Purpose Modeling, Analysis, and Implementation of Software Architectures

Mert ÖZKAYA

Yeditepe University, Department of Computer Engineering, Istanbul, Turkey, mozkaya@cse.yeditepe.edu.tr, https://orcid.org/0000-0002-3464-6489

Article Info

Received: *February 20, 2020*
Accepted: *May 15, 2020*
Online: *July 27, 2020*

Keywords: UML, software modeling languages, formal-verification, code generation, JML

Abstract

In this paper, a new software modeling language called SAMP is proposed, which is inspired from UML and enables the general-purpose modeling of software architectures but at the same time promotes the multiple-viewpoints modeling, formal verification of the models for the desired requirements, and combining model and code together. SAMP supports the high-level modeling of software architectures from the requirements, logical, behavioral, and deployment perspectives and supports checking the consistencies between the software models in different perspectives. SAMP is also supported with a modeling toolset that allows for the visual modeling of software architectures in those perspectives. The toolset further generates formal ProMeLa models that can be accepted by the SPIN model checker for the exhaustive verification of the software behaviors against the user-defined properties and some pre-defined properties (e.g., deadlock, race-condition, wrong and incomplete preconditions). Moreover, the toolset can also generate the Java Modeling Language (JML) code that combines the contractual models with the Java program for ensuring the consistency between model and code throughout the software development.

To Cite This Article: M. Özkaya, "The General-Purpose Modeling, Analysis, and Implementation of Software Architectures", Journal of Aeronautics and Space Technologies, Vol. 13, No. 2, pp. 165-177, July 2020.

Yazılım Mimarilerinin Genel-amaçlı olarak Modellenmesi, Analizi, ve Koda Dönüştürülmesi

Makale Bilgisi

Geliş: *20 Şubat 2020*
Kabul: *15 Mayıs 2020*
Yayın: *27 Temmuz 2020*

Anahtar Kelimeler: UML, yazılım modelleme dilleri, biçimsel sınama, koda dönüşüm, JML

Özet

Bu makalede, endüstride sıklıkla kullanılan UML modelleme dilinden esinlenen SAMP adında yeni bir yazılım mimarisi modelleme dili önerilmektedir. SAMP ile, modellenen yazılım mimarileri, biçimsel yöntemler kullanılarak analiz edilebilir, ve sonrasında otomatik olarak koda dönüştürülebilir. SAMP, yazılım mimarilerinin gereksinim, mantıksal, davranışsal, ve fiziksel bakış açılarına göre ayrı ayrı modellenebilmesini desteklemektedir. Ayrıca, bir araç kümesi aracılığıyla, SAMP modelleri otomatik olarak ProMeLa biçimsel sınama dilinde modellere dönüştürülerek ProMeLa'yı destekleyen SPIN model sınayıcısıyla kapsamlı olarak analiz edilebilir ve kullanıcı-tanımlı ve önceden-tanımlı birçok özelliğin sağlanıp sağlanmadığı anlaşılabilir. Yine aynı araç desteği ile, analiz edilen modeller Java Modeling Language (JML) dilinde koda dönüştürülebilir. JML sayesinde, model ile Java kodunu bir arada yürütmek mümkün olup kodun modeli sağlayıp sağlamadığı JML'i destekleyen araçlar ile sürekli test edilebilir.

1. INTRODUCTION

Many different software modeling languages are available today for the architectural modeling of software systems, including UML [1] and its derivatives [2], architecture description languages (ADLs) [3], formal specification and verification languages (e.g., Eiffel [4], Java Modeling Language [5], ProMeLa [6], pi-calculus [7] and CSP [8]), and any domain-specific languages [9]. Software modeling languages provide modeling notations of varying abstraction levels and may differ in many aspects such as general-purpose/domain-specific scope, visuality, formal semantics, non-functional properties, extensibility, multiple viewpoints, and tool

support. UML is considered as one of the top-used software modeling languages in industries, which offers a general-purpose notation set and provides several types of diagrams for the high-level modeling of software systems from different perspectives. UML gains its popularity due to many supporting CASE tools[1] that facilitate the UML modeling and provide various capabilities such as collaborative modeling, code generation, reverse engineering, scripting, etc. UML is also extensible with its profiling mechanism, which has been used by many UML-based modeling languages that extend UML for particular domains [2]. However, UML suffers from some issues, which essentially restrict the practitioners' modeling capabilities and make them use UML basically for documenting and communicating their software models. UML lacks in formally defined and executable semantics, which sometimes cause the UML diagrams to suffer from ambiguities. Due to UML's informal semantics, most of the UML CASE tools do not provide any mechanism for processing the UML models and detecting any design errors - except performing some basic syntax and well-formedness rule checking. Indeed, the exhaustive analysis support (i.e., model checking or theorem proving) or non-functional property checking is highly difficult (if not impossible). UML also does not consider preserving the consistency between the UML models and software implementation. It is not possible with any UML CASE tools to combine modeling and coding together and ensure that code always satisfies the modeled design decisions. Another issue with UML is concerned with the UML CASE tools' lack of support for the multiple-viewpoints modeling. While the UML tools allow for creating various types of UML diagrams that can be used for modeling in different viewpoints (e.g., logical, behavioral, and deployment), most tools do not support organizing the UML diagrams created into the separate viewpoints and checking the consistency relationships between different viewpoint models. Last but not least, UML is actually supported with a huge notation set that consists of many different diagrams with various constructs. This unfortunately increases UML's complexity and sometimes make the UML models harder to learn and use by practitioners [10, 11].

In this paper, the goal is to propose a new software modeling language called SAMP, which is inspired from UML so as to maximize the adoption of practitioners. Just like UML, SAMP offers a general-purpose notation set for the high-level modeling of software architectures. Moreover, SAMP addresses the issues that are discussed in the previous paragraph which UML suffers from. To support the multiple-viewpoints modeling, SAMP considers the four perspectives that are believed to be so important for software development: requirements, logical, behavioral, and deployment. SAMP is supported with a modeling toolset, which offers a modeling editor for modeling software systems from the requirements, logical, behavioral, and physical perspectives and checking the consistency relationships between the software models. The toolset translates the contractual behavior models in the formal ProMeLa verification language [6] and enables the formal verification of the behavior models via the SPIN model checker for some pre-defined and user-defined properties. The formally-verified software models can be translated into a software implementation in Java. To this end, SAMP's modeling toolset uses the Java Modeling Language (JML) [12] and produces a formal model in JML, which is a contract-based formal specification language for Java and allows for combining the Java programs with the contract specifications. The resulting JML models can be improved with the low-level design and implementation decisions and the final, executable Java program can be formally verified for the contract specifications via the JML verifier tools.

2. RELATED WORK

Given our previous analysis on the existing software architecture modeling languages [2, 3], only 11 languages support the modeling of software systems from the logical, behavior, and deployment perspectives. However, none of those 11 languages consider all the requirements that are supported by SAMP - i.e., a high-level, general purpose notation set, requirements modeling and its traceability with the other software models, formal verification, implementation code generation, and combining modeling and implementation for ensuring their consistencies.

SimPL [13] offers a UML profile for modeling the integrated control systems. SimPL does not support any formally defined semantics, nor does it provide any toolset for modeling, analysis, and code generation. UWE [14] offers a UML profile that extends various UML diagrams for modeling the web applications. UWE supports various viewpoints such as requirements, content modeling, navigation modeling, and process modeling. UWE's code generator produces Java code for the JavaServer Faces (JSF) web application framework. UWE is defined using the FACPL formal language and supported with a tool for translating the security

[1] The list of UML CASE tools: https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

properties specified in UML into the FACPL models for their evaluation. UWE does not support the formal verification of system behavior models. AADL [15] and East-ADL [16] are essentially the continuation of Meta-H [17], which focus on the real-time embedded systems domain. EAADL [18] extends AADL with variability modeling. AADL, East-ADL, and Meta-H consider the modeling of embedded systems and their analysis for many quality properties including security, resource-consumption, performance, and reliability. AADL's tools enable the translation of the AADL models into software code, while East-ADL, and Meta-H do not. Modeling software requirements and their consistency analysis with the architectural models are not considered either. Ambient-Prisma [19] supports the aspect-oriented modeling of software architectures for distributed and mobile systems. Note that Ambient-Prisma is essentially the extension of the well-known aspectual language Prisma [20] that is based on Ambient Calculus [21]. Ambient-Prisma is supported with a toolset that generates .NET code from the aspectual specifications. However, the facilities such as the formal analysis, requirements specifications and their traceability, and consistency checking are not considered by Ambient-Prisma. DremsML [22] focuses on modeling the distributed systems. DremsML's toolset analyses the architectural models for well-formedness rules and some quality properties (i.e., network resource, security, scheduling, and timing) and also generates some skeleton C++ code. The requirement specifications, the model consistency analysis, and formal analysis are not considered by DremsML. Palladio [23] focuses on the component-based software systems and offers a simulator to simulate the performance of the system components. Palladio also offers a Java code generator. The system requirements are modeled in Palladio as a usage model that describes the concurrent users and the their behaviors in terms of the users' service calls. UNAS/SALE [24] focuses on the air traffic control systems, which can be modeled in terms of Kruchten's 4+1 viewpoint model (i.e., logical, process, physical, and development viewpoints). While UNAS/SALE's toolset generates Ada code, UNAS/SALE does not support the formal verification. EADL [25] focuses on modeling the hardware and software components of embedded systems. The behaviors of the components can be formally specified in the form of a temporal logic, which can be formally verified by a model checker. EADL's toolset generates C code from software models. However, the requirements specifications and their traceability are not considered by EADL.

3. SOFTWARE REQUIREMENTS AND ARCHITECTURE MODELING IN SAMP

The meta-model definition of the SAMP language is depicted in Figure 1. SAMP essentially provides four different types of notation sets, one for each modeling perspective supported (i.e., requirements, logical, behavioral, and deployment). Each notation set consists of a set of modeling elements, which may also be hierarchically composed of other elements. Figure 2 gives the visual symbols for the elements composing SAMP's notation set and their use for creating models in different perspectives. In the rest of the section, we describe for each perspective the syntax of the notation set and how they can be used to specify the design decisions on the relevant concerns.

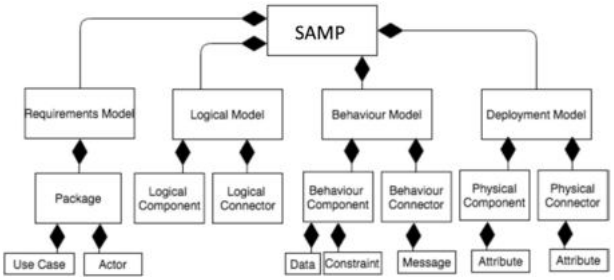


Figure 1. The meta model for SAMP's notation set

3.1 Requirements Modeling

The requirements model notation set consists of the use-case, actor, and package elements. The package element is the unit of decomposition and allows for decomposing systems into cohesive units so that each consists of a relevant set of functionalities. As shown in Figure 3, each package is specified with actors and some use-cases. An actor in a package is linked with one or more use-cases that are cohesive and this creates the use-case diagram for the package.

The actor herein describes the component of the package that performs the functionalities represented with the associated use-cases.

3.2 Logical Modeling

The logical software model describes the system structure with high-level components and connectors. While the logical components are the units of computations, the logical connectors are the units of interactions. SAMP's connectors support various types of interactions, which are indicated via the stereotype centered on the connector notation. The interaction types herein have been determined from Mehta et al.'s connector classification framework [26] - i.e., procedure call, event, stream, data access, adaptor, distributor, and arbitrator.

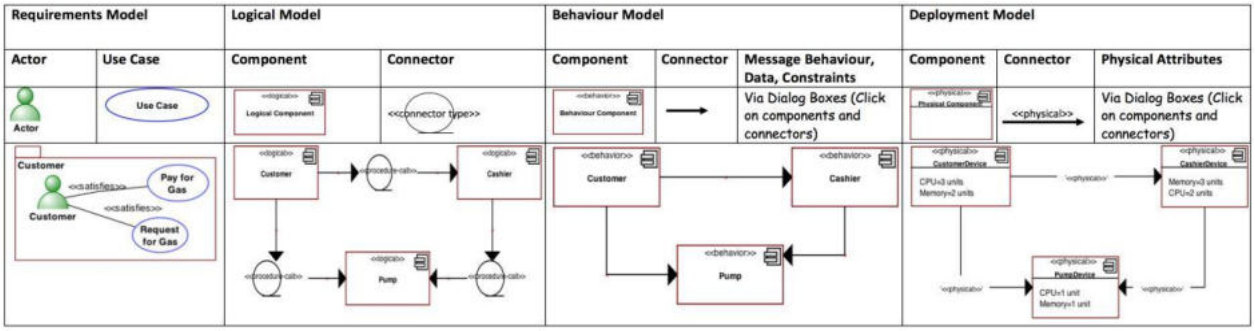


Figure 2. The symbols for SAMP's notation set

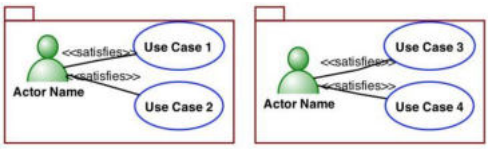


Figure 3. Modeling the software requirements in SAMP



Figure 4. The logical component specifications in SAMP

Whenever the logical component notation is clicked, a new window pops up as shown in Figure 4. Thus, the component's name that appear on the component box and the use-cases, that the component is expected to handle, can be specified. SAMP's modeling toolset checks the consistency between the requirements and logical models for a software system and ensures that a use case specified in the requirements model is performed by any one of the logical components. Clicking on a connector notation also pops up a new window as shown in Figure 5. Using this window, one can specify the values for the connector attributes that have been determined from Mehta et al.'s connector classification framework.

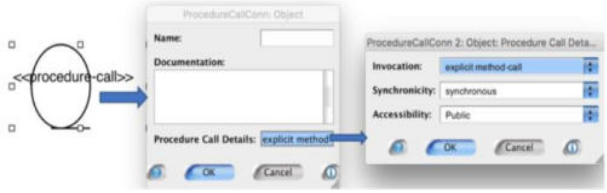


Figure 5. The logical connector specifications in SAMP

3.3. Behavior Modeling

To model the behaviors of the components, firstly the behavior model's structure is specified. That is the components which are simply linked with each other without detailing their interactions as is the case with the logical model. SAMP's modeling toolset checks for ensuring that any logical component specified in the logical model exists as a behavior component with the same name exactly once, and vice versa too. For any connector link that connects any two behavior components, the logical components associated with the two behavior components must also be connected with some logical connector.

Whenever a component is clicked on the behavior model, a new window pops up as shown in Figure 6. This enables to specify for a component (i) the state data and (ii) the constraints that are expected to be satisfied. Each component state data is specified as a variable whose data can be changed dynamically during the message exchanges between the components. The component constraints are used for specifying the local interaction protocols for the components. As shown in Figure 7, a component constraint is specified as a sequence of messages that the component is expected to follow in their message exchange with their environment.



Figure 6. Specifying component state data in SAMP



Figure 7. Specifying a constraint for a component in SAMP

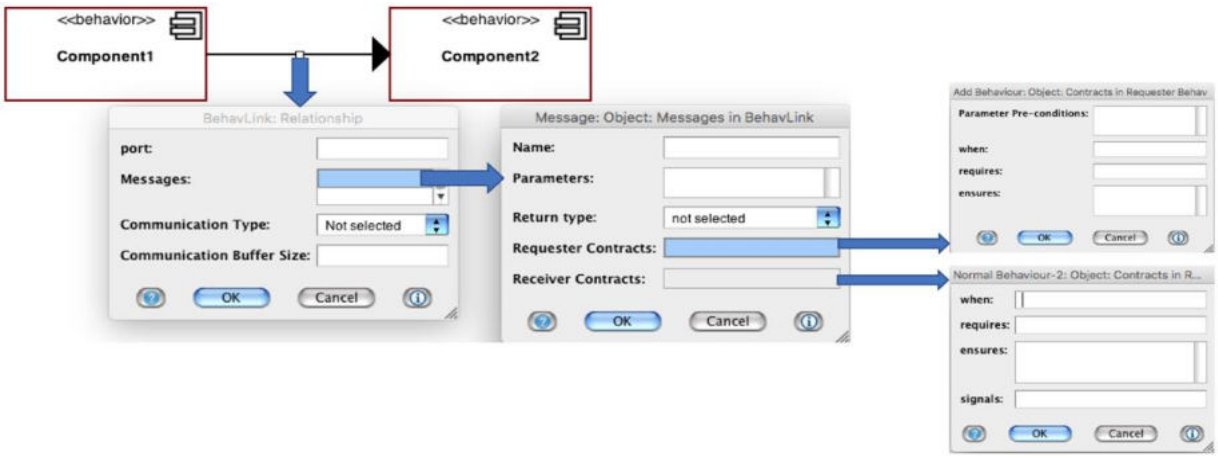


Figure 8. Specifying the contractual message behaviors in SAMP

3.3.1. Component Messages and their Contractual Behaviors

The messages exchanged between any two components can be specified by clicking on the link connectors. Besides the message specifications, the communication type can be specified, which can be either a two-way request & response communication or one-way request communication. The communication buffer size can be specified too, which determines the number of messages that can be queued for that communication link. If the buffer size is 0, this leads to a rendezvous (i.e., synchronous) communication. Whenever a message specification is clicked, another window is opened as shown in Figure 8 for specifying (i) the message name, (ii) the message parameters, (iii) the message return type, (iv) the contracts for specifying the message request behavior that is performed by the requester component and (v) the contracts for specifying the message receipt behavior that is performed by the receiving component. Note that each message may be attached with multiple contracts. A message request is specified contractually in terms of (i) a blocking precondition (when), (ii) the precondition on the message parameters and the component data (requires), and (iii) the postcondition on the component data (ensures). A message receipt is contractually specified in terms of (i) a blocking precondition (when), (ii) a precondition on the component state (requires), (iii) postcondition on the component data and the message result to be returned (ensures), and (iv) the type of any exception that wishes to be thrown if the precondition is not satisfied. The semantics for the contractual behaviors of the messages are discussed in terms of the translations in the ProMeLa formal verification language in Section 4.

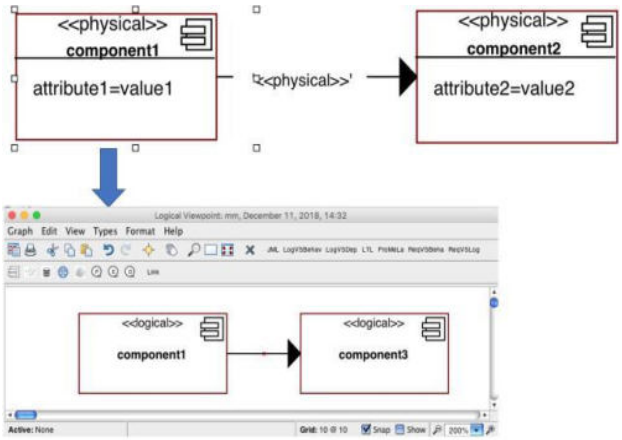


Figure 9. Specifying a deployment model in SAMP

3.4. Deployment Modeling

The deployment software model is concerned with the physical architecture of software systems and the mapping between the physical and logical components. The visual notation set for the deployment model basically consists of the physical component notation and the physical connection notation for connecting the physical components together. The physical components can be supplemented with the generic attribute specifications for describing any quality properties for the components (e.g., CPU usage, memory consumption, encryption, and response time). The physical connection links can also be supplemented with the generic attributes for specifying the expected quality properties of the component communications, e.g., HTTPS and SSL protocols. The attribute values (e.g., the CPU and memory units employed) can be adjusted depending on the logical component(s) that the physical components

include. Also, to specify the logical components that the physical components include, one needs to right click on the physical components that will open up a new sub-editor for drawing the logical components as depicted in Figure 9. SAMP's modeling toolset checks for ensuring that each logical component is mapped with exactly one physical component and each physical component is mapped with one or more logical components that exist in the logical model.

4. SAMP'S FORMAL VERIFICATION SUPPORT

SAMP defines the algorithms for translating the behavior models in the ProMeLa formal verification language, which can be exhaustively checked using the SPIN model checker [6]. To enhance understanding, the entire translation algorithm is presented in the rest of the section in terms of sub-algorithms that are each discussed separately. Also, the dot notation (.) is employed in the translation algorithms for reaching the sub-elements of the elements.

```
1 Model2ProMeLa()
2  FORALL connectorLink ∈ Model
3   chan ch_connectorLink.name-REQ
4                  = [connectorLink.bufferSize] of {....};
5   IF connectorLink.CommunicationType = 'Request and Respons
6     chan ch_connectorLink.name-RES
7                  = [connectorLink.bufferSize] of  {....}
8  FORALL component ∈ Model
9    Component2ProMeLa(component)
```

Figure 10. Translating a SAMP specification ProMeLa

```
 1 Component2ProMeLa(Component component)
 2
 3  proctype component.instanceName {
 4     FORALL data ∈ component.DataList
"5       data.Type data.Name = data.DefaultValue;
 6
 7   do
 8     FORALL link ∈ component.RequesterConnectorLinkLis
 9       requesterBehaviour(component, link)
10     FORALL link ∈ component.ReceiverConnectorLinkList
11       receiverBehaviour(component, connectorLink)
12   od
13 }
```

Figure 11. Translating a component specification in ProMeLa

4.1. Translating an Entire SAMP Model in ProMeLa

Figure 10 gives the algorithm for translating a software behavior model in SAMP into a formal ProMeLa model. In lines 2-7, the ProMeLa channels through which the component processes send/receive the messages are defined. One channel is created for exchanging the message request and another channel is created for exchanging the message response. Then, in lines 8-9, for each component specified in the behavior model, the algorithm given in Figure 11 is consulted for defining a concurrent process. Indeed, the system configuration is translated as the concurrent execution of the component processes that interact with each other via the request and response channels created.

4.2. Translating a Component in ProMeLa

In Figure 11, a ProMeLa process is created for each component in the behavior model (lines 1-13). Each process definition starts with the variable declarations for the component state data (lines 4-5). Then, an infinite loop is specified as given in lines 7-12 using ProMeLa's do::od construct. In this infinite loop, the goal is to execute the message request and response operations performed by the component infinitely. For each message that the component requests, the algorithm referred to in line 9 is consulted. For each message that the component receives a request for, the algorithm referred to in line 11 is consulted.

4.3. Translating the Messages Requested by Components in ProMeLa

Figure 12 gives the translation algorithm for the messages that are requested by the components. For each message that the component requests, two guarded sequence of actions are created (lines 4-11 and 13-21). While the first guarded action sequence is used for sending the message request (lines 4-11), the second guarded action sequence is used for receiving the message response (lines 13-21). For each contract specification of the message, a different guarded action sequence is created for the message request.

In the first guarded action sequence for the message request (created one for each contract), its guard in line 4 is satisfied when the contract's blocking precondition (*when*) is satisfied. Note here that the guarded action sequence to be operated is chosen non-deterministically whose *when* precondition is satisfied. Then, the message's contract *requires* precondition is checked in line 6. If the *requires* precondition is not satisfied, an error occurs indicating the wrongly specified pre-condition and the process execution is halted (lines 9-10). Otherwise (if satisfied), the message is written into the communication channel, including its name and the parameter arguments. So, the message request can be received by the connected component. In the second guarded action sequence for receiving the message response, its guard in line 13 is satisfied when the response channel has a message response for the current message. If so, the message response is accepted along with any result data (if any). Then, the message contract's post-conditions are realized in lines 16-20. Therein, for each post-condition, the condition on the received result is checked, and if satisfied, the component state data is updated in accordance with the postcondition (lines 17-18). If none of the specified conditions is satisfied, an

assertion error is generated in line 20 and the process execution is halted. This indicates that the postcondition cannot be satisfied.

```
1 requesterBehaviour(component, connectorLink)
2   FORALL message ∈ connectorLink.messageList
3     FORALL m_req ∈ message.requesterContractList
4       :: m_req.when →
5         if
6         :: m_req.requires →
7            ch_connectorLink.name-REQ!
8                 message.name, message.parameterVariabl
9         ::!(m_req.requires) →
10           assert(false); printf("wrong pre-condition")
11        fi
12
13    :: ch_connectorLink.name-RES ? message.name,
14                                   result_message.name →
15        if
16        FORALL ensure ∈ m_req.ensuresList
17          :: ensure.EnsuresCondition →
18             ensure.Data = ensure.Value;
19        ::!(∨_m_req.ensuresList EnsuresCondition)
20          → assert(false); printf("incomplete post-cond")
21        fi
```

Figure 12. Translating a component message to be requested in ProMeLa

4.4. Translating the Messages Received by Components in ProMeLa

In Figure 13, the translation algorithm for receiving message requests is given. For each message received, a different guarded sequence of actions is created for each contract specified for receiving the message (lines 4-24). One of those guarded action sequences is chosen to be processed non-deterministically. The guarded sequence chosen may only be processed if there is a message in the request channel that is ready to be received (line 4). Whenever the request message is received, if the blocking precondition for the message contract is not satisfied the received message is written back to the request channel as given in lines 21-23. If the blocking precondition is satisfied (line 7), the *requires* precondition of the message contract is checked in line 9. If that is satisfied too, then the component state data are updated according to the *ensures* postcondition of the message contract (lines 10-11). Then, the result (if any) is determined from the postcondition and written into the response channel (lines 12-13). So, the component that requests the message may receive the result. Note that if the message communication is specified to be one-way (i.e., only request), then the message response will not be considered to be sent back. If an exception type is specified via the *signals* clause of the message contract (lines 14-16), then the case when the *requires* precondition is not satisfied leads to an assertion error with the specified exception information in the signals clause and the process execution is halted. Otherwise, if the requires precondition is not satisfied without any exception specification (lines 18-19), there is again an assertion error occurring that indicates the wrong precondition this time.

```
1 receiverBehaviour(component, connectorLink)
2   FORALL message ∈ connectorLink.messageList
3     FORALL m_rec ∈ message.receiverContractList
4       ::ch_connectorLink.name-REQ?
5             message.name, message.ParameterVariables →
6         if
7         :: m_rec.when →
8            if
9            ::m_rec.requires →
10              FORALL ensure ∈ m_req.ensuresList
11                ensure.Data = ensure.Value;
12              ch_connectorLink.name-RES !
13                          message.name, result_message.name;
14              IF m_rec.signals ≠ NULL
15                :: !(m_rec.requires) →
16                   assert(false); printf(m_rec.signals)
17              ELSE
18                 :: !(m_rec.requires) →
19                    assert(false); printf(wrong pre-condition);
20           fi;
21        ::!(m_rec.when)→
22           ch_connectorLink.name-REQ!
23             message.name, message.ParameterVariables
24        fi;
```

Figure 13. Translating a component message to be received in ProMeLa

5. COMBINING MODEL WITH CODE IN SAMP

To preserve the consistency between the software models and implementation, SAMP defines the translation of the behavior models in the Java Modeling Language (JML) [12], which is a formal specification language that combines the Java program development with the Design-by-Contract [27] based software specifications. Basically, JML allows practitioners for attaching the Java program modules (i.e., interfaces, classes, and methods) the contractual annotations that describe the behavior of the modules. Then, practitioners can use the Java program herein to develop their software applications in Java and use the powerful verification tools (e.g., JACK[2], OpenJML[3], KeY[4], etc.) that support JML and formally verify the Java program modules for their contractual specifications.

The translation algorithms are explained in the rest of the sections in terms of the connector and component translations respectively. Also, the dot notation (.) is employed in the translation algorithms for reaching the sub-elements of the elements.

5.1. Translating Connectors in JML

Figure 14 gives the algorithm for translating each connector link that connects any two components. Each connector link is translated into a Java class (lines 12-29)

[2] JACK web-site: http://www-sop.inria.fr/everest/soft/Jack/jack.html

[3] OpenJML web-site: http://www.openjml.org/

[4] KeY web-site: https://www.key-project.org/

that implements a Java interface (lines 3-10). The Java interface for the connector link consists of a Java method definition (lines 6-9) for each message that the components exchange via the connector link. The connector class has a constructor (lines 17-21) that initializes the instance variable in line 15 with the class instance which corresponds to the receiver component in that connection. The methods of the interface that the connector class implements are filled in the connector class (lines 23-30). Each method includes a single statement that basically calls the same method of the class instance which is referenced via the instance variable (line 28). As discussed in the next part, the requesting components essentially use the connector class instances to make a call for their methods and thus send their message requests to the receiver component.

5.2. Translating Components in JML

In Figure 15, the algorithm for translating a component into a Java class is given. In lines 4-5, the component state data are translated as the instance variables in the Java class. In lines 7-9, for each connector link that the component uses for requesting messages from another component, a reference variable is created.

The reference variable herein is of the interface type created for the connector link in Figure 14. Using this variable, the component class instance sends method-calls to the connector instance that directs the call to the receiver component's instance. The reference variable herein is initialized with the connector class instance that implements the interface in the main class. Note that the main class herein is created for the system configuration and instantiates the component and connector class instances and passes the components to the connector class constructors. So, the adapter design pattern has been applied herein, and the requesting and receiver components do not know each other and may only communicate via the connector that acts as the adapter.

For each message that the component requests via their connector link, its Java method declaration is as given in lines 11-19. The method herein uses the interface variable defined in lines 7-9 and directs the method call to the class instance of the corresponding connector link that is supposed to call the component class instance which provide the message. For each message that the component receives a request for, its Java method declaration is as given in lines 21-28. The method herein does not perform any delegation of method-calls. Indeed, that is the actual method to be called via the connector class instance. Note that the method body parts are expected to be filled by the developers, who can implement the methods by considering other necessary modules such as the network, distribution, or graphical user interface modules. Each method declaration is annotated with the JML contracts, which are attached at the top of the method declarations as given in lines 13 and 23 - right before the method headers.

```
1 Connector2Java(connectorLink)
2
3   public interface connectorLink.Name '_Inteface' {
4
5     FORALL message ∈ connectorLink.MessageList
6       message.Type message.Name (
7         FORALL param ∈ message.ParameterVariables
8           param.Type param.Name
9         );
10  }
11
12  public class connectorLink.Name
13        implements connectorLink.Name '_Interface'   {
14
15    connectorLink.ReceiverComponent.Type receiver;
16
17    public connectorLink.Name (
18          connectorLink.ReceiverComponent.Type
19                                          recParameter){
20          receiver = recParameter;
21    }
22
23    FORALL message ∈ connectorLink.MessageList
24      public message.Type message.Name (
25        FORALL param ∈ message.ParameterVariables
26          param.Type param.Name
27        ){
28          receiver.message.Name();
29        }
30  }
```

Figure 14. Translating a connector link into a Java class

```
1 Component2Java(component)
2 public class component.Name {
3
4   FORALL variable ∈ component.DataList
5     variable.Type variable.Data = variable.Value;
6
7   FORALL connectorLink ∈ component.RequesterConnectorLinkList
8     connectorLink.Name '_Interface'
9                        connectorLink.Name '_Instance';
10
11  FORALL connectorLink ∈ component.RequesterConnectorLinkList
12    FORALL message ∈ connectorLink.MessageList
13      Requester_Contract(message)
14      message.Type message.Name (
15        FORALL param ∈ message.ParameterVariables
16          param.Type param.Name
17        ){
18          connectorLink.Name '_Instance'.message.Name();
19        }
20
21  FORALL connectorLink ∈ component.ReceiverConnectorLinkList
22    FORALL message ∈ connectorLink.MessageList
23      Receiver_Contract(message)
24      message.Type message.Name (
25        FORALL param ∈ message.ParameterVariables
26          param.Type param.Name
27       ){
28        }
29  }
```

Figure 15. Translating a component into a Java class

5.3. Translating the Message Request Behaviour

In Figure 16, the JML contract definition is given for describing the behaviors of the message requests made by the components. The JML contracts are specified within the comment blocks /*@ ... @*/ and each contract for a message request is translated as JML's normal behavior case notation (*@public normal_behavior* clause) in lines 4-11. The contracts of the message request are combined with JML's *also* keyword given in line 13. So, for each contract specification of a message request, JML's *normal behavior* case consists of the when, requires, and ensures clauses that are translated from the message's contract specification. For the method-call to be executed, initially, the when condition is evaluated, which is a blocking condition that keeps the method-call waiting until it is called in the right state [28]. When satisfied, the

method-call is executed and the component (i.e., class instance) state is changed under the control of its pre- and post-conditions. The requires precondition here is the logical *AND* combination of the contract's requires precondition and parameter precondition (lines 6-7). The postconditions are also combined with the logical *AND*, and each postcondition is specified using the logical implication which states that if the condition specified over the message result received is satisfied, the post-condition on the component state may be evaluated then (lines 8-10).

5.4. Translating the Message Receipt Behaviour

In Figure 17, the JML contract definition is given for describing the behaviors of how the message is received by the components. The JML contracts are again specified within the comment blocks /*@ ... @*/. This time, each contract for a message receipt is translated as the combination of a JML's *normal behavior* case and *exceptional behavior* case notations (*@public normal_behavior* and *@public exceptional_behavior* clauses respectively) given in lines 4-8 and 11-14 respectively. The contracts of the message request are combined with JML's *also* keyword given in lines 10 and 12. The *@public normal_behavior* clause consists of the when, requires, and ensures clauses of JML that are translated from the message contracts and their corresponding elements. So, whenever the when condition is satisfied, the method-call can be executed and its pre- and post-condition clauses are evaluated. Moreover, if the message contract includes an exception specification for an abnormal situation, that is translated as JML's exceptional behavior (*@public exceptional_behavior*) as given in lines 10-14. Note that the exceptional behaviour's requires precondition is the logical negation of the normal behaviour's requires precondition. So, whenever the when condition satisfied, if the normal behavior for the message is not satisfied and an exception is specified, this causes the exceptional behavior to be evaluated.

```
1 Requester_Contract(message)
2 /*@
3 FORALL contract ∈ message.ContractList
4  @public normal_behavior
5  @when: contract.when
6  @requires: contract.requires ∧
7    ∧param∈contract.ParameterList param.data=param.value
8  @ensures:
9     ∧ensures∈contract.ensuresList
10       ensures.assignmentCondition==>
11                          ensures.data=ensures.value;
12 IF contract ≠ ContractList.last
13   @also
14
15 @*/
```

Figure 16. Translating a message request into a JML contract

```
1 Receiver_Contract(message)
2 /*@
3  FORALL contract ∈ message.ContractList
4    @public normal_behavior
5    @when: contract.when
6    @requires: contract.requires
7    @ensures:
8     ∧ensures∈contract.ensuresList ensures.data=ensures.value;
9    IF contract.signals ≠ NULL
10     @also
11     @public exceptional_behavior
12     @when: contract.when
13     @requires: !(contract.requires)
14     @signals: contract.signals
15   IF contract ≠ ContractList.last
16     @also
17 @*/
```

Figure 17. Translating a message receipt into a JML contract

6. EVALUATING SAMP'S MODELING TOOLSET

SAMP's modeling toolset has been developed using Metaedit+ [29], which offers a language design and development platform along with a language for developing the necessary generators. SAMP's toolset offers a modeling editor for the users to specify their software architectures from different perspectives (i.e., the requirements, logical, behavioral, and deployment). SAMP's toolset further provides two code generators that process the behavior models of the software systems specified in SAMP. While one generator produces a ProMeLa model, the other generator produces a JML model. The generated ProMeLa model is intended to be used for the formal verification of the high-level behavioral design decisions. The generated JML model is intended to be used for the implementation of the behavioral design decisions in Java and validating the implementation for the behavior models throughout the software development. SAMP's toolset can be downloaded via the project web-site[5].

The code generators have been illustrated via two well-known case studies, i.e., the gas station system [30] and aegis combat system [31]. The gas station system consists of the pump, cashier, and a number of customer components that interact with each other so that each customer gets the gas from the pump upon making the payment to the cashier. The aegis combat system has been developed by the US Navy Forces for the purpose of enabling the navy ships to control their weapons against enemies. The aegis system is essentially a client-server system, which consists of many sensor components to determine the current state of the ships and other components that analyze the data provided by sensor components in order to react to potential threats. Due to the space limit, the specifications of the gas station and aegis systems could not be presented in the paper, which are however available via the project web-site[5].

[5] SAMP's project web-site:
https://sites.google.com/site/ozkayamert1/srad

Table 1. Formal verification results for the gas station and aegis combat systems

Case-studies	State-vector (bytes)	States		Memory (Mb)	Time (sec)
		Stored	Matched		
Gas Station System (1 customer)	348	668	251	128.925	0
Gas Station System (2 customers)	476	110216	47118	175.116	0.15
Aegis Combat System	1656	4645276	18414384	7023.981	25.1
Aegis Combat System - BITSTATE	1656	56859562	1.8260127e+08	33.319	226

After modeling the gas station and aegis combat systems from different perspectives in SAMP, the toolset has been used for generating the formal ProMeLa models in accordance with the translation algorithms given in Section 4. The resulting ProMeLa models have been formally verified with SPIN's model checker for the pre-defined properties and component constraint specifications. The pre-defined properties are deadlock, race-condition, and wrong and incomplete behaviors. The deadlock property is checked by the SPIN model checker automatically, which determines the components that are blocked executing and wait for each other infinitely. The race-condition checking has been encoded as part of the ProMeLa translation algorithms and is concerned with the message requests performed in two actions (i.e., sending the request and receiving the response). So, whenever a component requests for a message and waits for the response, the same component may attempt at operating some other message(s) in between. If a data to be assigned by the message request contract has been changed due to the messages processed in between, this is considered as a race-condition. Note that due to the page limit, we omitted the race-condition encoding while discussing the translation algorithms. The wrong precondition has been encoded as part of the ProMeLa translation algorithms and is used to determine for any component message whether at least one of its contract preconditions is satisfied in all possible system executions or not. If, for a message contract, none of its preconditions are satisfied, this is considered as the wrong precondition error. The incomplete post-condition has also been encoded as part of the translation algorithms and is concerned with the component messages to be requested. So, for any message request, whenever its response is received, one of the contracts should include a postcondition that needs to be satisfied on the message result. If none of the post-conditions are satisfied for the message request contract, this is considered as the incomplete post-condition error. Concerning the user-defined properties, the component constraints specified by the users can be translated into LTL formulas using the SAMP toolset. So, the resulting ProMeLa models can further be formally verified for the LTL properties using the SPIN model checker. The gas station system has been verified for different configurations, one with a single customer involved and another with two customers. The formal verification results for the gas station and aegis systems that have been obtained from SPIN's verification report are as given in Table 1. No any verification errors have been detected for the pre-defined properties and user-defined component constraints. Note however that for the aegis combat system, the verification could not be completed with the available memory space (i.e., 7024 MB), and therefore, SPIN's bit-state hashing mode has been applied [32].

After formally verifying the gas station and aegis combat system models, the gas station and aegis combat system models have been translated into the JML models via SAMP's toolset, which runs the translation algorithms discussed in Section 5. The JML models are essentially the Java programs annotated with the contractual specifications. JML is supported with some powerful verifier tools such as KeY [33] and OpenJML [34], which use the formal theorem proving techniques for the exhaustive verification of the Java programs with regard to the contract specifications. The author used the JML verifiers to perform the type checking on the two systems' JML models for any syntax errors that violate the syntactic rules for Java and JML. While KeY does not accept the when clause of JML for concurrency modeling, OpenJML do so. However, OpenJML does not allow any contract with public visibility to refer to any instance variable with private or protected visibility. So, JML's *spec_public* annotation[6] needs to be used for the variable declarations, which can then be referred to in the contract specifications. Also note that OpenJML is also available as online, which liberates the users from having to download and install any tools[7]. However, when the Java programs for both systems have been attempted to be validated via the verifier tools, the verifiers gave warning messages that indicates the violations of the contracts. This is essentially what the author expects. Indeed, in the generated JML models, the implementation of the class methods corresponding to the component messages are left empty, while the messages' behavior specifications encoded as the JML annotations attached to the methods. The class methods need to be filled in by the users with the actual implementation code, which can then be

[6] JML spec_public description: http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_6.html#SEC44

[7] OpenJML's online tool: http://www.rise4fun.com/OpenJMLESC/

verified against the JML annotations. To justify this, the author provided for each software method some implementation code that reflect the contractual specifications. When the verifiers have been re-run, the warning messages disappeared and the implementation has been proven to satisfy the JML annotations corresponding to the contractual model.

7. CONCLUSION

In this paper, a UML-based, general-purpose software modeling language called SAMP has been introduced, which focuses on the modeling of software systems from multiple perspectives (i.e., requirements, logical, behavioral, and deployment), and their analysis and implementation. SAMP is supported with a toolset that can perform a number of operations automatically. These include (i) checking the consistency relationships between the requirements and architecture models and (ii) the translation of the software behavior models in the formal ProMeLa language and Java Modeling Language (JML). The generated ProMeLa models can be formally analyzed via the SPIN model checker for some predefined properties (i.e., deadlock, race-condition, wrong precondition, incomplete postcondition). SAMP's toolset also translates any user-defined constraints specified as an activity diagram into LTL formulas that can be accepted by SPIN and used for the formal verification of the ProMeLa models. Concerning the JML code generation, the generated code is essentially the combination of a Java program with the contractual behavior specifications. So, developers can use the resulting JML models to develop their software systems in Java and then use the JML verifier tools (e.g., KeY and OpenJML) for formally verifying the Java program for the contractual specifications. SAMP and its toolset have been evaluated using the gas station system and aegis combat system.

We are currently improving the SAMP tool for teaching purposes so as to teach the undergraduate CS students the model-driven software development, writing quality code with JML, and applying design-by-contract for reliable software development.

8. REFERENCES

[1] J. E. Rumbaugh, I. Jacobson, and G. Booch, *The unified modeling language reference manual,* Addison-Wesley-Longman, 1999.

[2] M. Ozkaya, "Analysing uml-based software modelling languages," *Journal of Aeronautics and Space Technologies*, vol 11, pp. 119-134, July 2018.

[3] M. Ozkaya, "The analysis of architectural languages for the needs of practitioners," *Softw., Pract. Exper., vol.* 48, pp. 985-1018, January 2018.

[4] B. Meyer, J.-M. Nerson, and M. Matsuo, *Object-oriented design for software engineering.* Lecture Notes in Computer Science, vol. 289, pp. 221-229. Springer, 1987.

[5] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: A behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1-38, May 2006.

[6] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*, Addison-Wesley, 2004.

[7] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes", *i. Inf. Comput.*, vol. 100, pp. 1-40, September 1992.

[8] C. A. R. Hoare, "Communicating sequential processes," *Commun*. ACM, vol. 21, pp. 666-677, August 1978.

[9] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Not.*, vol. 35, pp. 26-36, June 2000.

[10] L. T. W. Agner, I. W. Soares, P. C. Stadzisz, and J. M. SimaO, "A brazilian survey on uml and model-driven practices for embedded software development," *J. Syst. Softw.*, vol. 86, pp. 997-1005, April 2013.

[11] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, *Empirical assessment of mde in industry*. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, New York, NY, USA May 2011. pp. 471-480.

[12] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, *Beyond assertions: advanced specification and verification with jml and esc/java2*. In Proceedings of the 4th international conference on Formal Methods for Components and Objects, FMCO'05, Berlin, Heidelberg, Springer-Verlag., November 2006. pp. 342-363.

[13] R. Behjati, T. Yue, L. Briand, and B. Selic, "Simpl: A product-line modeling methodology for families of integrated control systems," *Inf. Softw. Technol.*, vol. 55, pp. 607-629, March 2013.

[14] A. K. Andreas Kraus and N. Koch, *Model-driven generation of web applications in uwe*. In Proceedings of the 3rd International Workshop on Model-Driven Web Engineering, MDWE2007, pp. 23-38. CEUR-WS, 2007.

[15] P. H. Feiler, B. A. Lewis, and S. Vestal, *The SAE architecture analysis & design language (AADL): A standard for engineering performance critical systems*. In IEEE Intl Symp. on Intell. Control, pp. 1206-1211, October 2006

[16] V. Debruyne, F. Simonot-Lion, and Y. Trinquet, "EAST-ADL | An Architecture Description Language", pp. 181-195. Springer US, Boston, MA, 2005.

[17] J. H. McDuffie, *Using the architecture description language metah for designing and prototyping an embedded reconfigurable sliding mode flight controller*. In Proceedings of the 21st Digital Avionics Systems Conference, vol. 2, pp. 8B1-1-8B1-17, 2002.

[18] Y. Oh, D. H. Lee, S. Kang, and J. H. Lee, *Extended architecture analysis description language for software product line approach in embedded systems*. In Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE '07, Washington, DC, USA, 2007. pp. 87-88.

[19] N. Ali, I. Ramos, and C. Solis, "Ambient-prisma: Ambients in mobile aspect-oriented software architecture," *Journal of Systems and Software*, vol. 83, pp. 937-958, October 2010.

[20] J. Perez, I. Ramos, J. J. Martinez, P. Letelier, and E. Navarro, *Prisma: Towards quality, aspect oriented and dynamic software architectures*. In QSIC, IEEE Computer Society, 2003. pp. 59-66

[21] L. Cardelli and A. D. Gordon, "Mobile ambients", Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 140-155. 1998.

[22] D. Balasubramanian, A. Dubey, W. Otte, T. Levendovszky, A. Gokhale, P. Kumar, W. Emfinger, and G. Karsai, "DREMS ML: A wide spectrum architecture design language for distributed computing platforms," *Science of Computer Programming*, vol. 106, pp. 3-29, August 2015.

[23] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3-22, January 2009.

[24] P. Kruchten and C. J. Thompson, *An object-oriented, distributed architecture for large-scale ada systems*. In Proceedings of the Conference on TRI-Ada '94, TRI-Ada '94, New York, NY, USA, ACM. 1994. pp. 262-271

[25] J. Li, N. T. Pilkington, F. Xie, and Q. Liu, "Embedded architecture description language," *Journal of Systems and Software*, vol. 83, pp. 235-252, February 2010.

[26] N. R. Mehta, N. Medvidovic, and S. Phadke, *Towards a taxonomy of software connectors*. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, ICSE, ACM, 2000. pp. 178-187

[27] B. Meyer, "Applying Design by Contract," *IEEE Computer*, vol. 25, pp. 40-51, October 1992.

[28] E. Rodriguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby, *Extending jml for modular specification and verify cation of multi-threaded programs.* In Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05, Berlin, Heidelberg, Springer-Verlag. 2005. pp. 551-576,

[29] S. Kelly, K. Lyytinen, and M. Rossi, Metaedit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In J. A. B. Jr., J. Krogstie, O. Pastor, B. Pernici, C. Rolland, and A. S lvberg, editors, Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE, Springer 2013. pp. 109-129.

[30] G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, *Applying static analysis to software architectures*. In M. Jazayeri and H. Schauer, editors, ESEC / SIGSOFT FSE, volume 1301 of Lecture Notes in Computer Science, Springer, 1997. pp. 77-93.

[31] R. Allen and D. Garlan, *A case study in architectural modelling: The aegis system*. In Proceedings of the Eighth International Workshop on Software Specification and Design (IWSSD-8), Paderborn, Germany, March 1996. pp. 6-15.

[32] G. J. Holzmann, "An analysis of bitstate hashing," *Formal Methods in System Design*, vol. 13, pp. 289-307, November 1998.

[33] B. Beckert, R. Hahnle, and P. H. Schmitt, *Verification of Object-oriented Software: The KeY Approach*, Springer-Verlag, Berlin, Heidelberg, 2007.

[34] D. R. Cok, Openjml: *Jml for java 7 by extending openjdk*. In M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, NASA Formal Methods, Springer Berlin Heidelberg. 2011. pp. 472-479.

9. VITAE

Mert ÖZKAYA is an Associate Professor of Computer Engineering at Yeditepe University and leads the software engineering research group at Yeditepe. Dr. Özkaya conducts active researches on model-driven software development, software architectures, formal verification, and empirical software engineering. Dr. Özkaya also conducts researches on improving the academia-industry collaborations on software engineering, software engineering education, and improving productivity in software engineering. Dr. Özkaya contributes and contributed to several national and international research projects on software

engineering and published many article and conference papers on model-driven software engineering and empirical software engineering. Dr. Özkaya acts as a technical/academic consultant of many software development companies in Turkey.