Proceedings of the Federated Conference on Computer Science and Information Systems pp. 639–648
DOI: 10.15439/2020F199
ISSN 2300-5963 ACSIS, Vol. 21

Towards Extending UML's Activity Diagram for the Architectural Modeling, Analysis, and Implementation

Mehmet Alp Kose,
Altinbas University, Institute of Graduate Studies, Istanbul, Turkey
Email: alp.kose@ogr.altinbas.edu.tr

Mert Ozkaya,
Yeditepe University, Department of Computer Engineering, Istanbul,Turkey
Email: mozkaya@cse.yeditepe.edu.tr

***Abstract*— SAWUML is a general-purpose software modeling language that extends UML by unifying component and sequence diagrams for the specifications of software architectures. While component diagram is used for modeling the system structures, sequence diagram is extended with the Design-by-Contract approach for the modeling of system behaviors. In this paper, we aim at enhancing the language usability by providing alternative modeling choices for practitioners. To this end, we extended SAWUML's notation set with UML's activity diagram for the behavior modeling. So, practitioners may now use either sequence or activity diagrams, while the system structures are still modeled with component diagrams. We also extended SAWUML's modeling editor for creating software architecture models together with component and activity diagrams and the code generators for automatically obtaining (i) formal models in SPIN's ProMeLa for formal verification and (ii) Java-based implementation. We illustrate our language extension with the gas station case-study.**

I. Introduction

SOFTWARE architecture is the structure of a system that comprises components, their behavioral specifications, and interactions with each other [1]-[3]. The software architecture is concerned with which components a system consists of and whether these components are integrated and working together, as well as what kind of interfaces the components will have, what will be the inter-component communication and dependencies. For modelling software architectures an Architecture Description Language (ADL) plays an important role [4]-[6].

An ADL is a formal specification language for describing the structures and behaviors of components and connectors at an abstraction level for the software architecture of a system. ADLs are designed for different domains (e.g., embedded, automotive, multi-agent, and distributed) and purposes such as modeling software structures, modeling software architectures from different viewpoints (e.g., structure, behavior, concurrency), non-functional property specifications and analysis, formally verifying system behaviors, and code generation.

UML [7] is an ADL that is one of the most widely used modeling languages in industry [8]-[9]. UML is a general-purpose software modeling language that can be used to visually specify the structural and behavioral aspects of any software systems at various levels of abstractions. The structural aspects of software systems can be specified using UML's class diagram, component diagram, or package diagram. The behavioral aspects of software systems can be specified using UML's state diagram, activity diagram, or sequence diagram.

We proposed SAWUML in our previous research [10], which is a UML-based ADL and enables practitioners to use UML's component and sequence diagrams together for the architectural modeling. SAWUML enables to specify the structural aspects of software architectures in terms of component diagrams. SAWUML also enables the behaviors of components to be specified with an extended form of sequence diagrams with Design-by-Contract [11]. SAWUML is supported with a toolset, which consists of a visual modeling editor and a set of code generators. The architectural models in SAWUML can be automatically transformed in SPIN's ProMeLa formal verification language for formally verifying the architectural models against pre-defined (i.e., deadlock and incompleteness) and user-defined linear temporal logic (LTL) [12] properties. Also, SAWUML models can be transformed in Java for facilitating the implementation of software architectures.

In this paper, we aim to improve SAWUML's notation set so as to enhance the language usability. To this end, we extend SAWUML with the activity diagram notation set and intend to offer practitioners two alternative choices for the behavioral modeling. Practitioners may now either use the sequence diagram or the activity diagram depending on what looks more usable and familiar to them. Indeed, while activity diagram is inspired from flowchart and promotes the behavioral modeling in terms of the component activities and their transitions, sequence diagram focuses more on the collaborations of components and promotes the specifications of the order in which the components operate their activities. It should also be noted that we extended with

the activity diagram as we believe that the activity diagram is already familiar to many practitioners with different profiles (including those with very limited technical knowledge) [13]-[15]. We also extend SAWUML's existing toolset. The modeling editor now also supports our new behavior notation set that extends the UML activity diagram. Also, we extended the existing code generators for ProMeLa and Java properly. So, while practitioners who feel more comfortable with the UML sequence diagram (i.e., its notation and syntax) may use SAWUML's sequence diagram extension, those who feel comfortable with UML's activity diagram may use the activity diagram extension introduced in this paper so as to model, analyze, and implement their software architectures.

A. Paper Structure

In the rest of the paper, we firstly provide an overview of SAWUML. Next, we discuss the structure and behavior specifications of software architectures. Then, we introduce the extended SAWUML with activity diagrams and their specifications. After that, SAWUML's generators for translation in SPIN's ProMeLa formal verification language and Java code are introduced. We illustrated the extended SAWUML and its toolset via the gas station system. Lastly, we evaluated the extended toolset of SAWUML for the formal verification and software implementation and then conclude the paper.

II. Overview of SAWUML

SAWUML [10] supports both the structural and behavior modeling of software architectures. While the structural aspects of a system are specified with component diagrams, the behavioral aspects are specified with sequence diagrams.

A. Structural Modeling

There are two types of ports defined in SAWUML, which are required and provided. Fig. 1 shows the types of ports.

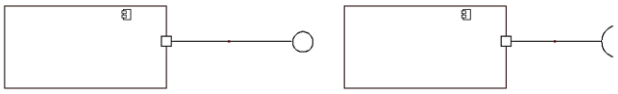


Fig. 1 Components with a provided port and a required port respectively

Every required port is connected to a provided port. Component's required port sends method-call(s) to the provided port of the connected component. The method(s) can take some parameter(s), which may be assigned with arguments upon method-call requests. The required and provided ports are specified in terms of methods that the ports request (if required) or receive the request of (if provided). Note that the required and provided ports of any two components that are connected must be specified with the same set of methods. Indeed, as described in the next section, the required port exhibits the behaviors for sending those method-calls and the provided port exhibits the behaviors for receiving those method-calls.

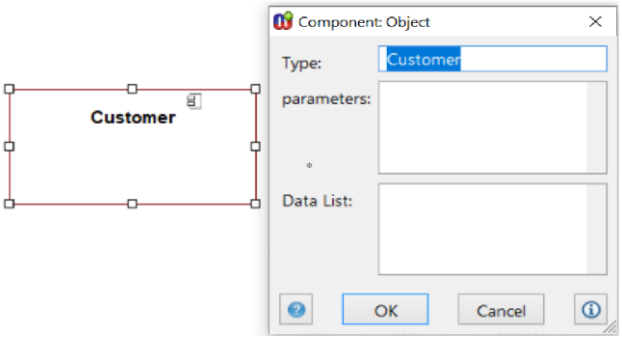


Fig. 2 A component with its specifications

B. Behavioral Modeling

Fig. 2 shows a component with its specifications in SAWUML. When a component box is clicked, a dialog box opens for specifying the component details i.e., component type name, component parameters, and component data list. The type name is unique for every component in a software architecture specifications. Practitioners can pass information to a component through the parameter specification of the component. A component data list represents the state data of the component, which are manipulated by the method-call behaviors operated by the component ports.

For behavioral modeling, a sequence diagram is used. When the relevant port is right clicked, the sequence diagram in a subgraph editor is opened. As seen in Fig. 3 and Fig. 4, there are two life-line objects, the one on the left represents the component with the required port and, the one on the right represents the component with the provided port.

For the sequence diagram of the required port, two arrows are used as depicted in Fig. 3. The solid arrow represents making the method-call to the provided port. The solid arrow herein is supplemented with a contract that consists of pre-condition and promise assignment. The promise herein is used for assigning parameter argument data for the method-call. The dashed arrow represents the method-call response received from the provided port of the connected component. The dashed arrows are supplemented with the pre- and post- condition notations. The pre-condition herein describes the condition on the result data received from the provided port. If pre-condition is satisfied, the post-condition is evaluated, which ensures certain values for the component state data.

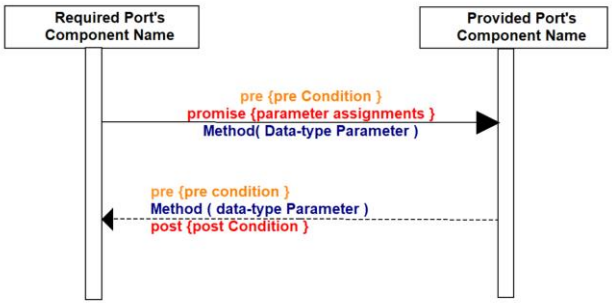


Fig. 3 Required port behavior

There are also two arrows used for the sequence diagram of the provided port as depicted in Fig. 4. The solid arrow represents the receiving method-call from the required port and is supplemented with a contract of pre- and post-conditions. After receiving the method-call from the required port, the pre-condition of the component is checked and if it is satisfied, the data assignments are made in accordance with the post-condition. The dashed arrow indicates a method-call response to the required port back with a return value that is specified via the '\return' notation of the post-condition (*post*).

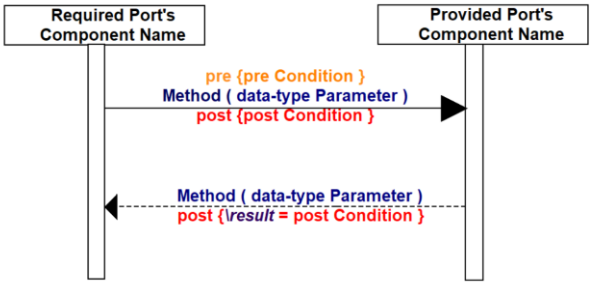


Fig. 4 Provided port behavior

III. Extending SAWUML with Activity Diagrams

In this study, we extend SAWUML's behavior modeling with the activity diagram that has been inspired from UML's activity diagram. In this way, practitioners may have the option of selecting either the sequence or activity diagram notation set for the behavioral modeling. It should be noted that the sequence diagram discussed above and the activity diagram extension to be discussed now are both semantically the same but vary in terms of the modeling notations used.

Whenever practitioners double-click on the required port interface icon, a new sub-editor appears as shown in Fig. 5. Using the sub-editor, practitioners can create the activity diagram model to specify the behaviors of the interacting components.

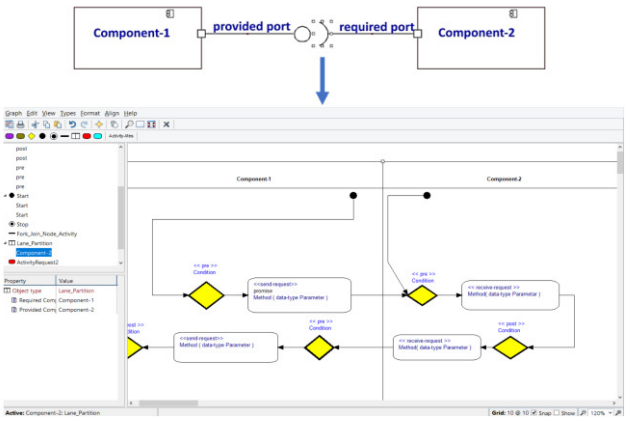


Fig. 5 Accessing to an activity diagram from a component's required port

Fig. 6 shows the notation set for the activity diagram. Although, the activity diagram notation set here is similar to the UML activity diagram, the SAWUML activity diagram has subtle differences, which we discuss in the rest of this section.

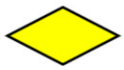
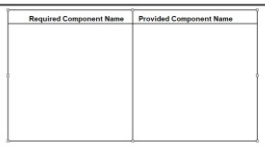
Notation	Visual Symbol
pre/post condition	
an activity of a required component port for sending a request	<< send-request >> promise Method (data-type Parameter)
an activity of a required component port for receiving a response	<< receive-response >> Method (data-type Parameter)
an activity of a provided component port for receiving a request	<< receive-request >> Method(data-type Parameter)
an activity of a provided component port for sending a response	<< send-response >> Method(data-type Parameter)
component lanes	
start stop	

Fig. 6 Design elements of the activity diagram in SAWUML

Practitioners need firstly to use the 'component lanes' notation to separate the activities of the interacting component ports. Note that any activities and pre-post-

conditions of each component port that represent their method-call behaviors need to be placed in the corresponding lane. While the activities of the required component port are placed in the left lane with the name of the component that is written on the top of the left lane, the activities of the provided component port are placed in the right lane with the name of the component that is written on the top of the right lane (Fig. 7). Start and stop nodes are used for starting and stopping the component behaviors respectively.

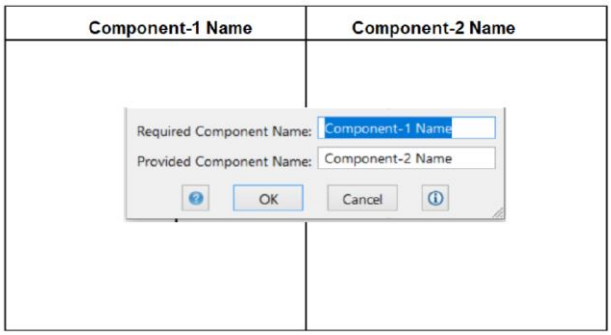


Fig. 7 Component lanes notation with its specifications

Practitioners may use the pre/post condition notation to specify the pre-post conditions of the method-call behaviors that are operated via the component ports. So, the pre/post condition notation needs to precede/follow the activity notations that represent the method-calls and are explained in the next paragraphs. Fig. 8 shows that whenever the pre/post condition notation is clicked, a new dialog box appears for specifying the type of condition (i.e., pre or post) and the condition statement.

Fig. 8 A pre/post notation with specifications

An activity notation of a required component port for sending a request is specified as given in Fig. 6. Whenever the respective notation is clicked, a new dialog box opens as given in Fig. 9. With this dialog box, one can specify the parameter data assignments of the method-call (promise) and the method-call name and parameter list. Note that the pre-condition of the method-call request cannot be specified via the dialog box given in Fig. 9. Practitioners need to use the pre/post-condition notation shown in Fig. 8, which needs to precede the activity notation. So, if the pre-condition is satisfied, the activity specified can be operated.

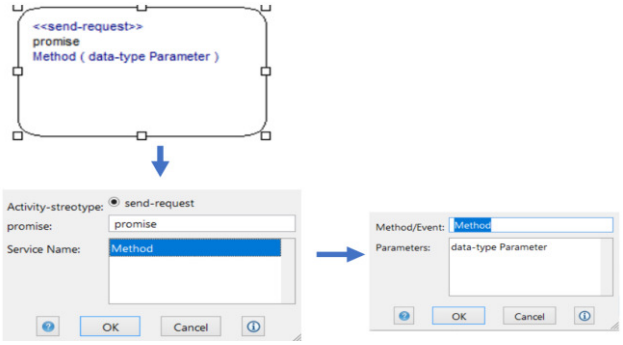


Fig. 9 Specifications of an activity of a required component port for sending a request

Whenever an activity of a required component port for sending a request is operated, this may be followed by the activity of a provided component port for receiving that request. So, the activity for receiving a request is specified as shown in Fig. 10. Note again that the activity for receiving a request here may be preceded by the pre/post-condition symbol to specify the pre-condition on the provided port's method-call receipt.

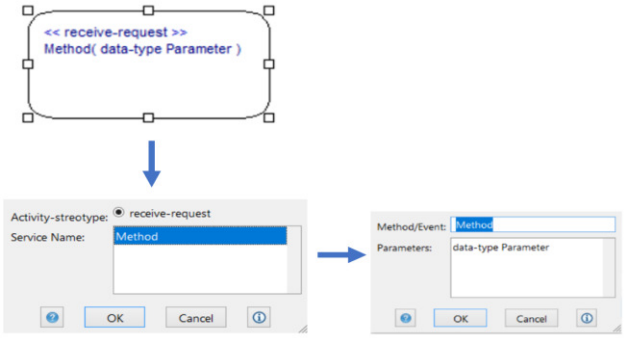


Fig. 10 Specifications of an activity of a provided component port for receiving a request

Whenever an activity of a provided component port for receiving a request is operated, practitioners may use the pre/post-condition notation for the post-condition to ensure that the data will be assigned (if any needed). This is followed by the activity of a provided component port for sending the method-call response. The activity for sending a response is specified as shown in Fig. 11. The pre/post-condition notation may be used after the activity for sending the response so as to the post-condition on the return value of the method-call.

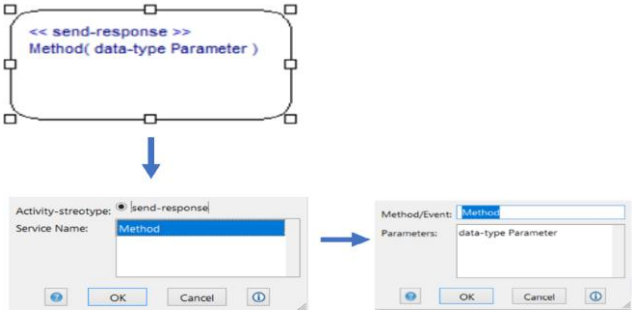


Fig. 11 Specifications of an activity of a provided component port for sending a response

Whenever an activity of a provided component port for sending a response is operated, the pre/post-condition notation may need to be used to check if the pre-condition on receiving a method-call response for the required port is satisfied. If so, the activity of a required component port for receiving that response can be operated. The activity for receiving a response is specified as shown in Fig. 12. After the activity for receiving a method-call response is operated, the pre/post-condition notation may be used to specify the post-condition that ensures the post-state of the component.

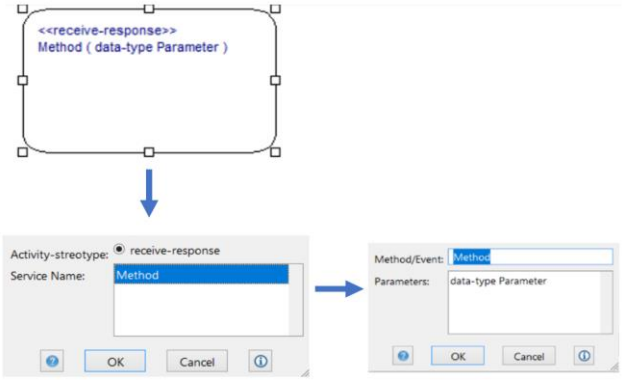


Fig. 12 Specifications of an activity of a required component port for receiving a response

IV. MetaEdit+ Based Tool Support

We used the MetaEdit+ [16] meta-modeling tool to develop the modeling editors and code generators for SAWUML. We defined the language abstract and concrete syntax with MetaEdit+'s GOPPRR meta-modeling framework, which then gave us the supporting modeling editor as depicted in Fig. 13.

In this study, the modeling editor, previously developed with MetaEdit+ has been extended to support the activity diagram notation introduced in the previous section.

We also used MetaEdit+ MERL[1] code generation definition language to extend the ProMeLa and Java code-generators to support the activity diagram notation set.

Selecting the icons (Component, Required port, Provided port) on the tool bar given in the modeling editor (as depicted in Fig. 13) creates the respective of the design element objects in the drawing area of the component diagram editor. Later a component object and a provided/required port object can be connected. When double-clicked, the practitioner is offered two options, to open an activity or a sequence diagram. By pressing the generator icon (ProMeLa/java Translator), a dialog opens, which allows practitioners to select one of the generators (i.e., java generator with activity/sequence diagram or ProMeLa generator with activity/sequence diagram) that are available as in Fig. 14 to run. By pressing the LTL property button, a dialog box opens on the drawing area of the component modeling editor, and the user-defined linear temporal logic (LTL) [12] properties can be entered as shown Fig. 15.

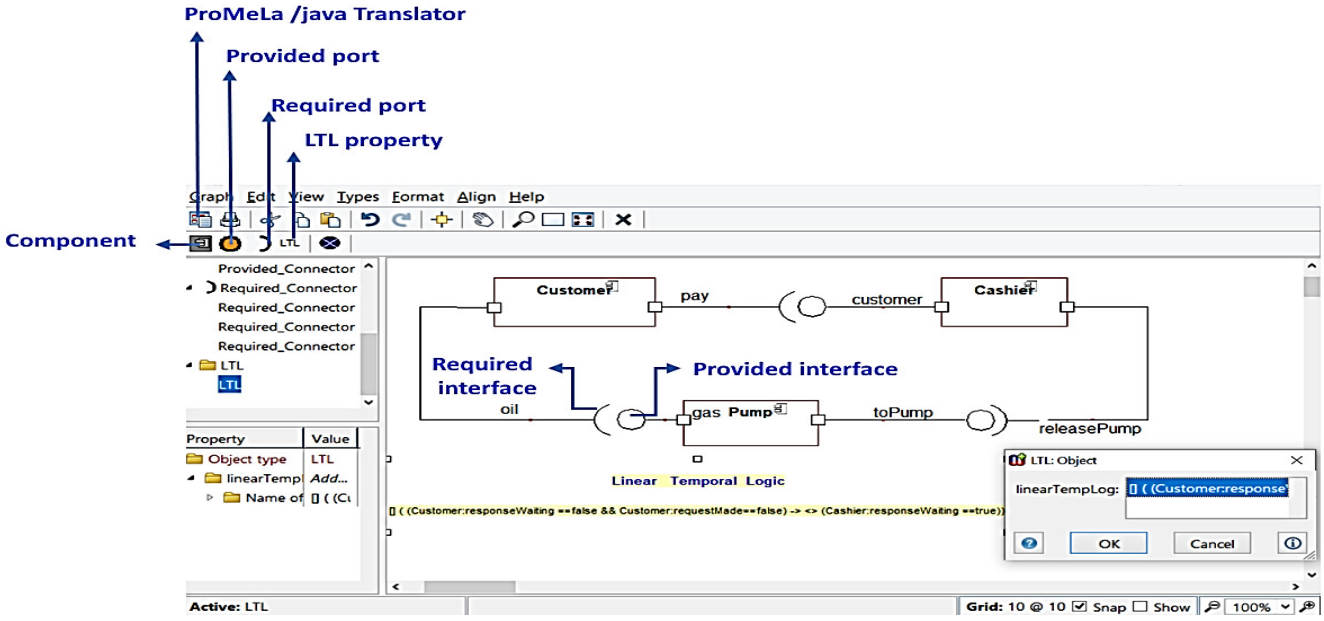


Fig. 13 The modeling editor of a component diagram

[1] MetaEdit+'s MERL language website: https://www.metacase.com/support/55/manuals/mwb/Mw-5_2_1.html

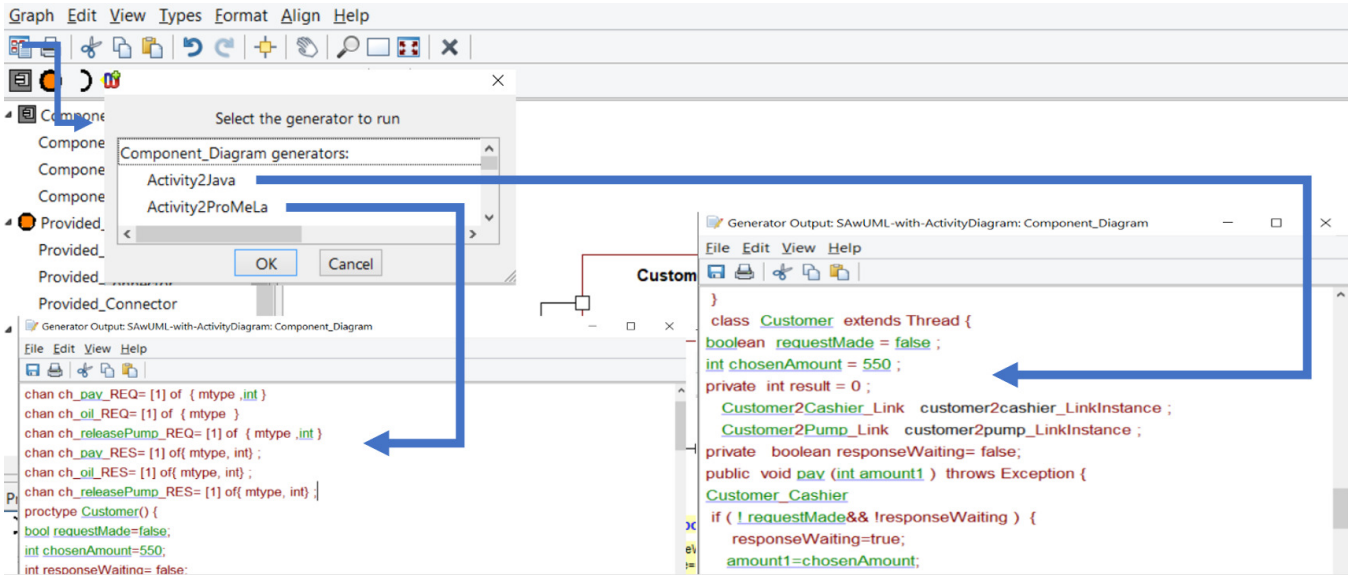


Fig. 14 The results of the ProMeLa generator and the java generator of a model

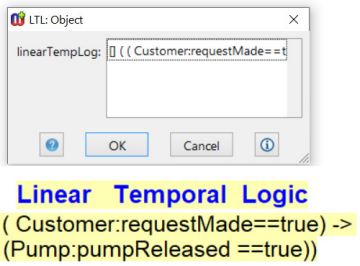


Fig. 15 Example of an LTL property

After running the ProMeLa generator, LTL properties are translated according to the LTL syntax in the ProMeLa language and embedded in the ProMeLA model obtained from the generated SAWUML model. So, using the SPIN model checker [17], the generated ProMeLA model can be formally verified for the LTL-based user-defined properties. Besides the user-defined properties, SPIN also checks a couple of pre-defined properties (i.e., deadlock, incompleteness) that we introduced as part of the ProMeLa translation algorithms. A deadlock error happens when the component processes get stuck executing and none of them will be able to reach their end states. Incompleteness happens if the response behavior specifications for a required port cannot handle all possible cases properly. The code for checking the pre-defined properties have been encoded as part of the ProMeLa code generator. Since there is not enough space in the article, the java generator algorithm[2], ProMeLa generator algorithm[3] and SAWUML toolset[4] can be accessible via the website.

V. Gas Station Case Study

The gas station system [18] is composed of three components that interact with each other. These are the customer, cashier and pump components. The customer component gets gas from the pump component if the customer component pays to the cashier component. Fig. 16 shows the component diagram specification of the gas station system in SAWUML.

Each component's data list specifications are shown in Fig. 17, Fig. 18, Fig. 19. The activity diagrams for the gas station system are shown in Fig. 20, Fig. 21, Fig. 22.

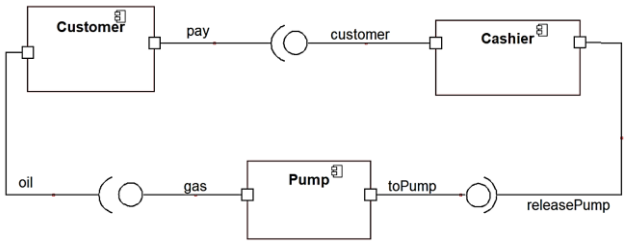


Fig. 16 Gas station in the SAWUML model

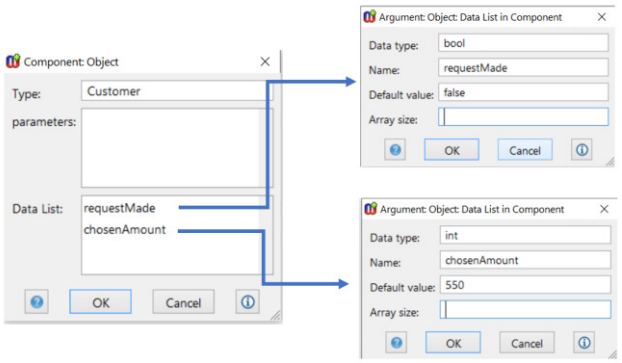


Fig. 17 The data list of the customer component

[2] SAWUML's java generator algorithm website: https://sites.google.com/view/mkose/javaalgorithm
[3] SAWUML's ProMeLa generator algorithm website: https://sites.google.com/view/mkose/promelaalgortihm
[4] SAWUML's toolset website: https://sites.google.com/view/mkose/sawuml-toolset

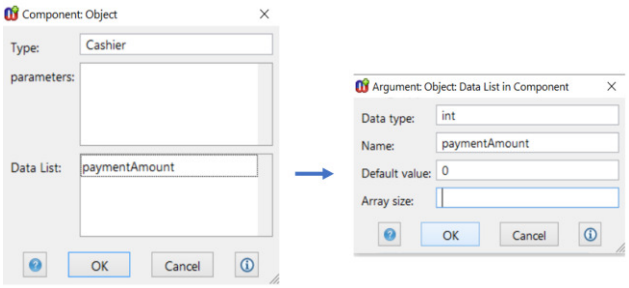


Fig. 18 The data list of the cashier component

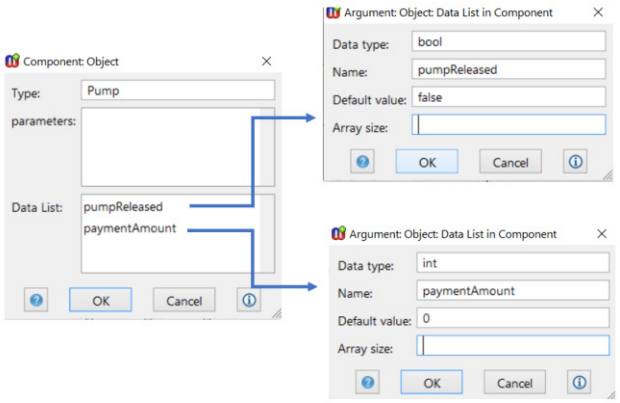


Fig. 19 The data list of the pump component

In the customer-cashier activity diagram in Fig. 20, before making the *pay* method-call via the customer's required port, a pre-condition (*!requestMade*) is checked via the pre/post-condition symbol. If it is satisfied, the activity for making the *pay* method-call is operated. The activity for the *pay* method-call includes a promise data assignment (*amount=chosenAmount*). After the *pay* method-call is received by the cashier's provided port, firstly the pre-condition is checked via the pre/post-condition symbol (*paymentAmount==0*). If it is satisfied, the cashier's activity for receiving the *pay* method-call request is operated. Then, the post-condition is ensured via the pre-/post symbol (*paymentAmount=amount*). Then, the cashier's send-response activity for the *pay* method-call is operated to send the response to the customer. The customer receives back the *pay* method-call response via its receive-response activity under no pre-condition. After the customer operates the activity for the method-call response, the post-condition is ensured via the pre/post-condition symbol (*requestMade=true*).

In the cashier-pump activity diagram in Fig. 21, before making the *releasedPump* method-call via the cashier's required port, a pre-condition (*paymentAmount!=0*) is checked via the pre/post-condition symbol. If it is satisfied, the cashier's activity for making the r*eleasedPump* method-call request activity is operated. The activity for the *releasedPump* method-call includes a promise data assignment (*amount2=paymentAmount*). After the *releasedPump* method-call is received by the pump's provided port, the pre-condition is checked via the pre/post-condition symbol *(!pumpReleased*). If satisfied, the activity for receiving the *releasedPump* method-call is operated and the post-condition is ensured via the pre/post-condition symbol (*pumpReleased=true, paymentAmout=amount2*). Then, the pump's send-response activity for the *releasedPump* method-call is operated to send the response to the cashier. The cashier receives back the *releasedPump* method-call response via its receive-response activity under no pre-condition. A post-condition ensures that the data will be assigned (*paymentAmount=0*) via the pre/post-condition symbol.

Fig. 22 gives the activity diagram specification for the customer and pump relationships. Before the customer makes a *pump* method-call via its required port, a pre-condition (*requestMade==true*) is checked via the pre/post-condition symbol. If it is satisfied, the activity for making the *pump* method-call is operated. Whenever the customer sends the method-call request for the pump, firstly, the pre/post-condition symbol of the pump (*pumpReleased==true*) is checked and if it is satisfied, the activity for receiving the pump request is operated. Then, the pre/post-condition symbol for the post-condition of the pump is ensured (*pumpReleased= false*). Afterward, the pump component operates the activity for sending the *pump* method-call response to the customer. The *pump* method-call is sent back with a return value. The pre/post-condition symbol is employed here to state the post-condition on the return value *(\result == paymentAmount*). When the customer receives the *pump* response, the pre-/post-condition symbol is used to operate the pre-condition that compares the return value (*result*) with the *chosenAmount* variable. If both are equal (*result==chosenAmount*) then the activity for the receiving the response for the *pump* method-call can be operated and then the pre/post-condition symbol for the post-condition is ensured (*requestMade = false*).

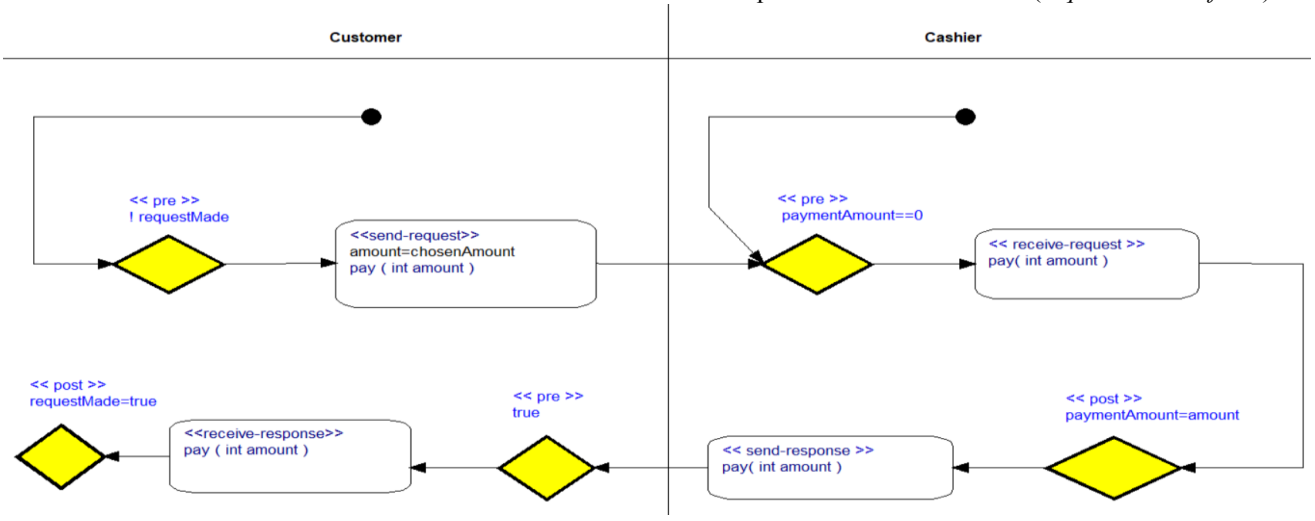


Fig. 20 The activity diagram of customer-cashier components

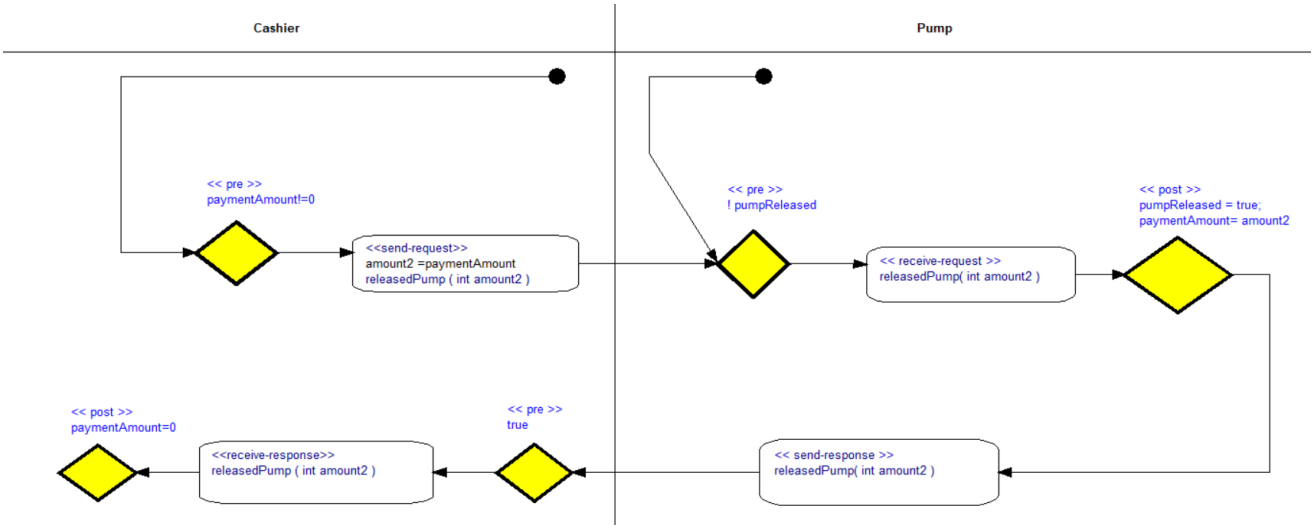


Fig. 21 The activity diagram of cashier-pump components

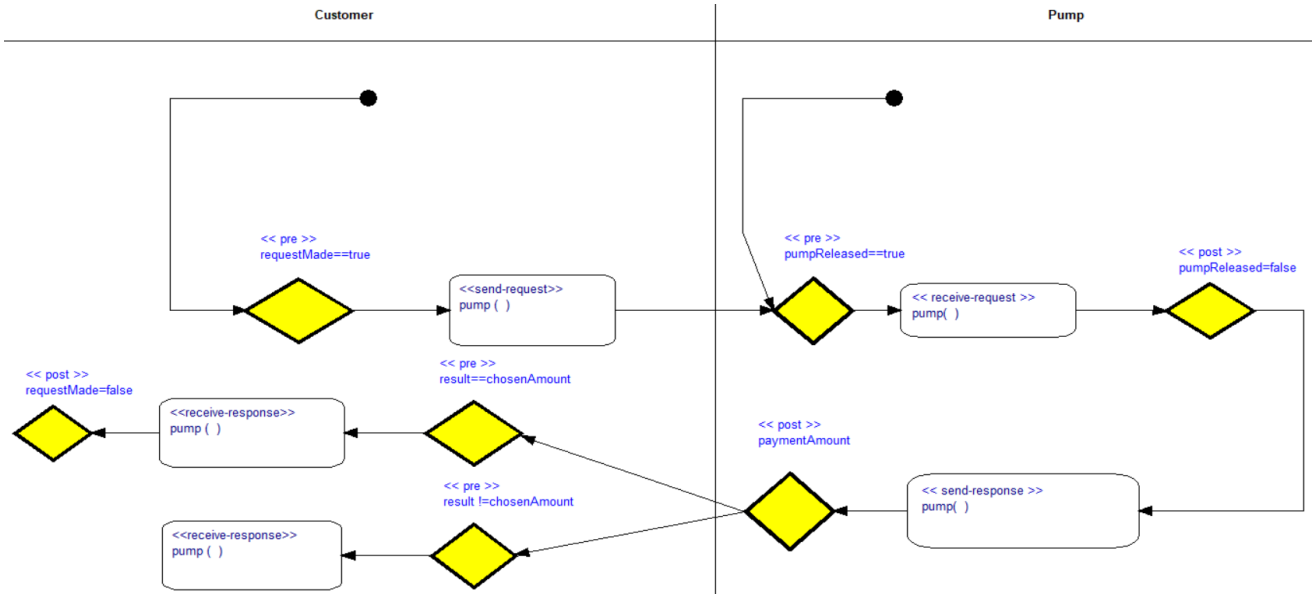


Fig. 22 The activity diagram of customer-pump components

Using the LTL property icon in the editor, the user-defined LTL property has been specified as part of the gas station specification as shown in Fig. 23. The LTL property here states that a particular constraint must always be satisfied. That is, when *requestMade* data of a customer is *true*, then eventually the pump's *pumpReleased* data becomes *true*. This means that whenever the customer sends a gas request to the cashier, the pump will eventually receive a pump-release request from the cashier.

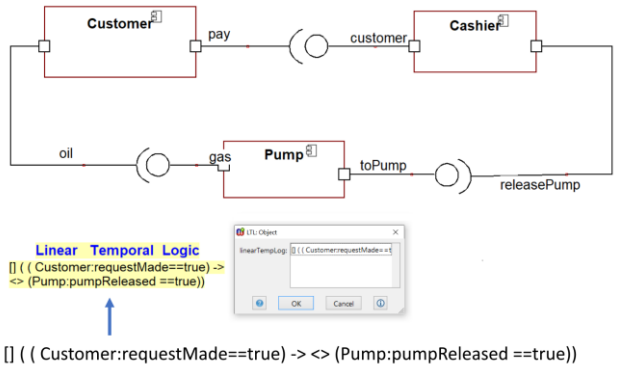


Fig. 23 Specifying an LTL property for the gas station model in SAWUML

TABLE I.
FORMAL VERIFICATION RESULTS IN SPIN

Case Studies	Sector-vector (bytes)	States		Memory (Mb)	Time (s)
		Stored	Matched		
Gas station – 1 customer	144	101	59	64.539	0
Gas station – 3 customers	364	121988	148130	106.922	0.413
Gas station – 5 customers	584	3462574	7813281	2016.661	25.4

VI. TOOL EVALUATION

After we specified the gas station model in SAWUML as discussed in Section 5, we used SAWUML's toolset for automatically transforming the gas station model into a formal ProMeLa model that can be accepted by the SPIN model checker and Java code for obtaining the implementation of the gas station model. We considered three different configurations of the gas station model, which vary depending on the number of customers involved (gas station with 1, 3, and 5 customers).

A. Formal Verification

Table I[5] shows the formal verification results that have been produced by the SPIN model checker for each configuration of the gas station model (namely their ProMeLa translations). The SPIN formal verification results are given with (i) the size of the states in the system's state space, (ii) the number of the stored state in the state space, (iii) the number of the matched states that are revisited during the state space search, (iv) the total actual memory usage of the state space, and (v) the elapsed time for the exhaustive analysis of the state space.

Whenever a deadlock occurs, an *invalid end state error* is generated by the SPIN model checker, which indicates that the running component processes cannot reach at the end of their code. To illustrate a deadlock situation, we used the gas station with one customer. We intentionally changed the *requestMade* data of customer's pay port to *true*. So, customer's both pay and oil port pre-conditions are now the same. The end result is that the customer is waiting for making a payment or a pump request, the cashier is waiting for a payment from the customer, the pump is waiting for receiving a release-gas request from the cashier. So, none of the components reach the end of their states and that causes deadlock.

To illustrate an incompleteness error, we again used the gas station with one customer. The response behavior specification for the customer's pump method consists of two cases. The case when the *result* is equal to *chosenAmount*, and the case when the result is not. So, we did not get any incompleteness error. If however the response behavior specification here failed to consider all possible cases (e.g., suppose "the *result* is equal to the *chosenAmount*" case is absent), then we would get an *assertion violation error* via the SPIN model checker.

Lastly, the gas station model has been verified for the user-defined LTL property specified in Fig. 23. The translated ProMeLa model actually includes the LTL translation in ProMeLa. So, whenever we run the ProMeLa model with the SPIN model checker, we successfully performed the formal verification for the LTL property. Note that if the LTL property was violated, we would get an *assertion violation error* in SPIN.

B. Java Implementation

After formally verifying the gas station model, we automatically produced Java code using SAWUML's toolset. Java implementation was created according to the Adapter Design Pattern to enhance the code modularity and understandability. Basically, the adaptee (e.g., the customer) component sends a request to the adapter via the component interface and the adapter transmits this request to another adaptee component (e.g., cashier in customer to cashier relationship).

It should be noted that the produced code includes the structural and behavioral architectural design decisions of the model. Practitioners can develop their systems with other necessary modules (i.e., network, GUI, database connection, etc.) starting from this code.

Since there is not enough space in the article, the generated java file (Configuration.java)[6], and the ProMeLa file (gasStation.pml)[7] can be accessible via the web site.

[5] Spin Version 6.4.5 is used with 2.4 GHz Intel Core i7-8750H, 16GB of RAM, and Windows 10 Home OS. We run the following the SPIN commands which are *spin -a GasStation.pml*, *gcc -o pan pan.c,* and *pan* (Note that for the gas station with 5 customers *pan -m800000* command is used.)

[6] The java file of the gas station system's web site: https://sites.google.com/view/mkose/javafile

[7] The ProMeLa file of the gas station system's web site: https://sites.google.com/view/mkose/promelafile

VII. Discussion & Conclusions

SAWUML is an ADL that uses UML's component and sequential diagrams for the specification of the structural and behavioral design decisions. SAWUML extends the sequential diagram using the Design-by Contract approach to define behavioral specifications of components' methods to send /receive each other. SAWUML is supported with a modeling editor to design architecture modeling and to specify user-defined properties in the form of LTL. The SAWUML models can be automatically transformed in SPIN's ProMeLa formal verification language for checking pre-defined properties (deadlock, incompleteness) and user-defined LTL properties.

In this study, we extended SAWUML by introducing the notation set for the extended activity diagram for modeling the behavioral design decisions. Practitioners may now have the options of selecting either the activity diagrams or sequence diagrams for the behavioral modeling. This is actually intended for enhancing the language usability and providing practitioners different types of notation sets among which they can choose the one that best fit their expertise. We also extended SAWUML's code-generator toolset to enable the architectural models with activity diagrams to be formally verified via the SPIN model checker and transformed into the Java-based implementation.

We evaluated our approach with the gas station system, where we specified the structural and behavioral design decisions with the component and activity diagrams respectively. We then used SAWUML's code generators to transform the models in SPIN's ProMeLa and used SPIN to formally verify the behavioral design decisions. We further automatically generated Java code from the gas station models, which is based on the Adapter design pattern.

SAWUML may actually be considered by any practitioners who use UML to model their software architectures from the structural and behavioral viewpoints. While UML and many tools that support UML do not allow for formally analyzing UML models, SAWUML does so. Moreover, SAWUML integrates the structural modeling with behavioral modeling – i.e., practitioners actually click on the component ports to specify their behaviors with sequence/activity diagrams. Note that this is not possible with UML and practitioners are forced to specify the structural and behavioral models that are cleanly separated. Moreover, in SAWUML we extend the UML sequence and activity diagrams with Design-by-Contract so as to enable practitioners to specify not only the interactions but also the behaviors in terms of pre- and post-conditions on the component state.

As a future work, aim at developing a tool that can reverse engineer the Java model back to the SAWUML model. By doing so, we aim at enabling the existing (i.e., already implemented) projects to be modeled and analyzed automatically and the developers to determine any architecture erosions [19].

VIII. References

[1] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, 2nd ed. Addison-Wesley Proffesional, 2003, pp. 19-26.

[2] N. Medvidovic and R. N. Taylor, "Software architecture: foundations, theory, and practice," *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Cape Town, 2010, pp. 471-472, doi: 10.1145/1810295.1810435.

[3] David Garlan and Mary Shaw, "An introduction to software architecture". *Advances in Software Engineering and Knowledge Engineering,* 1993, pp. 1-39.
https://doi.org/10.1142/9789812798039_0001

[4] Ozkaya M. "The analysis of architectural languages for the needs of practitioners". *Softw Pract Exper*. 2018; 48: 985– 1018.
https://doi.org/10.1002/spe.2561

[5] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," in *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93, Jan. 2000, doi: 10.1109/32.825767.

[6] P. C. Clements, "A survey of architecture description languages," *Proceedings of the 8th International Workshop on Software Specification and Design*, Schloss Velen, Germany, 1996, pp. 16-25, doi: 10.1109/IWSSD.1996.501143.

[7] Object Management Group. OMG unified modeling language secification –version 2.5. http://www.omg.org/spec/UML/2.5/; 2015. URL http://www.omg.org/spec/UML/2.5/.

[8] Ozkaya M. "Do the informal & formal software modeling notations satisfy practitioners for software architecture modeling?" *Inf Softw Technol* 2017; 95: 15–33. doi: 10.1016/j.infsof.2017.10.008.

[9] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione and A. Tang, "What industry needs from architectural languages: a survey," in *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 869-891, June 2013, doi: 10.1109/TSE.2012.74.

[10] Ozkaya M. and Kose M. A. "SAwUML – UML-based, contractual software architectures and their formal analysis using SPIN". *Journal of Computer Languages, Systems and Structures*, 2018; 54: 71- 94. https://doi.org/10.1016/j.cl.2018.04.005

[11] B. Meyer, "Applying 'design by contract'," in *Computer*, vol. 25, no. 10, pp. 40-51, Oct. 1992, doi: 10.1109/2.161279.

[12] A. Pnueli, "The temporal logic of programs," *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, Providence, RI, USA, 1977, pp. 46-57, doi: 10.1109/SFCS.1977.32

[13] Reggio, G., Leotta, M., Ricca, F., Clerissi, D. "What are the used UML diagrams? a preliminary survey". *Proceedings of 3rd International Workshop on Experiences and Empirical Studies in Software Mod*eling (EESSMod 2013), vol. 1078, pp. 3–12. *CEUR Workshop Proceedings,* 2013.

[14] Wrycza, Stanisław & Marcinkowski, Bartosz. "A light version of UML2:survey and outcomes". 2007, doi:10.13140/RG.2.1.3445.1046.

[15] G. Reggio, M. Leotta and F. Ricca, ""Precise is better than light" a document analysis study about quality of business process models," *Workshop on Empirical Requirements Engineering (EmpiRE 2011)*, Trento, 2011, pp. 61-68, doi: 10.1109/EmpiRE.2011.6046257.

[16] Kelly S, Lyytinen K, Rossi M. "Metaedit+ a fully configurable multi-user and multi-tool CASE and CAME environment". In: Bubenko J, Krogstie J, Pastor O, Pernici B, Rolland C, Sølvberg A, editors. *Seminal contributions to information systems engineering, 25 years of CAiSE*. Springer; 2013. p. 109–29. ISBN 978-3-642-36925-4. doi: 10.1007/978- 3- 642- 36926- 1 _ 9 .

[17] Holzmann GJ. "*The SPIN Model Checker - primer and reference manual*". Addison-Wesley Professional, 2003, ISBN 978-0-321-22862-8.

[18] Naumovich G , Avrunin GS , Clarke LA , Osterweil LJ. "Applying static analysis to software architectures". In: Jazayeri M, Schauer H, editors. *Software engineering–ESEC/FSE'97. Lecture Notes in Computer Science,* 1301. Springer; 1997. pp. 77–93. ISBN 3-540-63531-9.

[19] Dewayne E. Perry and Alexander L. Wolf. 1992. "Foundations for the study of software architecture". *SIGSOFT Softw. Eng. Notes* 17, 4 (Oct. 1992), pp. 40–52. DOI:https://doi.org/10.1145/141874.141884