A Spin / Promela Application for Model checking UML Sequence Diagrams

Cristian L. Vidal-Silva[1*], Rodolfo Villarroel[2], José Rubio[3], Franklin Johnson[4], Erika Madariaga[5], Camilo Campos[6], and Luis Carter[6]

[1]Ingeniería Civil Informática, Escuela de Ingeniería, Universidad Viña del Mar, Viña del Mar, Chile
[2]Escuela de Ingeniería Informática, Facultad de Ingeniería, Pontificia Universidad Católica de Valparaíso, Valparaíso, Chile
[3]Área Académica de Informática y Telecomunicaciones, Universidad Tecnológica de Chile INACAP, Santiago, Chile
[4]Depto. Disciplinario de Computación e Informática, Facultad de Ingeniería, Universidad de Playa Ancha, Valparaśo, Chile
[5]Ingeniería Informática, Facultad de Ingeniería, Ciencia y Tecnología, Universidad Bernardo O'Higgins, Santiago, Chile
[6]Ingeniería Civil Industrial, Facultad de Ingeniería, Universidad Autónoma de Chile, Talca, Chile

***Abstract*—UML sequence diagrams usually represent the behavior of systems execution. Automated verification of UML sequence diagrams' correctness is necessary because they can model critical algorithmic behaviors of information systems. UML sequence diagrams applications are often on the requirement and design phases of the software development process, and their correctness guarantees the accurate and transparent implementation of software products. The primary goal of this article is to review and improve the translation of basic and complex UML sequence diagrams into Spin / Promela code taking into account behavioral properties and elements of combined fragments of UML sequence diagrams for synchronous and asynchronous messages. This article also redefines a previous proposal for a transition system for UML sequence diagrams by specifying Linear Temporal Logic (LTL) formulas to verify the model correctness. We present an application example of our modeling proposal on a modified version of a traditional case study by using UML sequence diagrams to translate it into Promela code to verify their properties and correctness.**

***Keywords*—Spin / Promela; UML Sequence Diagrams; Fault Tolerance; LTL formulas; Combined Fragment**

I. Introduction

Developing computerized systems requires the use of modeling languages like UML to identify and characterize the systems structural and behavioral elements along with their properties [1] [2]. Namely for behavior modeling, UML offers use cases, sequences and state diagrams [3].

UML use case diagrams are usual at the beginning of the software development process [1] [2] [3]. UML sequence diagrams are useful to identify participant objects in use case scenarios and how those objects interact each other during the use case execution process. Usually participant objects in UML sequence diagrams modeled scenarios are part of the system's classes and respect their properties and communication methods [1] [2] [3].

UML state diagrams and UML sequence diagrams are usual in the design stage of the software development process. Even though UML state diagrams can represent states, state transitions, and events for those state changes either of individual objects or an instance of the complete system, those models do not give details about objects involved in the state changes (UML state diagrams do not represent interactions among objects in the system). On the other hand, UML sequence diagrams model the execution of systems by representing each communication (message) among their participant objects. UML sequence diagrams traditionally model only one execution scenario of the model system (a representative and critical execution situation) because multiple scenarios exist. Taking into consideration the model of algorithmic interactions by combined fragments of UML 2.0 sequence diagrams and the use of gates to represent modular behavior, without a doubt UML sequence diagrams are usable for modeling all of the interactions within a system.

Each model of an entirely consistent computerized system should be consistent (valid) with the user requirements as well as with each other valid product [4]. Using that base idea and assuming consistent previous models, this article describes steps to reach consistent UML sequence diagrams for a software system.

A system model is formally consistent with the user requirements if those requirements can be written and verified as correct LTL (Linear Temporal Logic) formulas [5] [6] [7] [8]. Thus, it is relevant to consider that Promela language [7] [9] allows verifying the correctness of LTL formulas.

Each model of an entirely consistent computerized system should be consistent (valid) with the user requirements as well as with each other valid product [4]. For that base idea and valid models for the system's requirements, this article describes steps to reach consistent and valid UML sequence diagrams for a software system.

Formally talking, a system model is consistent with the user requirements if those requirements are writeable and verifiable as correct LTL formulas [6] [7]. Thus, it is relevant to consider that Promela language [7] [9] allows verifying the correctness of LTL formulas.

Taking into account current references about the correctness verification of UML sequence diagrams [6] [10] [11] with the common application of the model checker Spin / Promela and other formal tools [12] [13], the main goals of this article are to improve previous concerning the representation in Promela code of UML sequence diagrams [12] [10] and to give the necessary steps for establishing LTL formulas to verify properties of execution scenarios for the modeled system.

* Corresponding author

This article promotes the use of UML sequence diagrams for modeling system behavior. This article uses a modified version of an existing case study [10] [11] for translating UML sequence diagrams into Promela code just to show improvements in the translation algorithm and highlight new considerations for the system transition states in the specification of LTL formulas to verify the consistency and correctness of systems' models. This article takes into account synchronous and asynchronous messages in the translation process, as well as additional details for the modeling and consideration of combined fragments of UML sequence diagrams. Thus, the correctness of a UML 2.0 sequence diagram for our case study is completely valid.

This article organizes as follows: Section 2 describes the main characteristics of UML sequence diagrams and describes a case study modeling its execution using a UML sequence diagram. Section 3 presents algorithms to translate UML sequence diagrams into Promela code with their application to the case study. Section 4 describes how to define a state transition system for UML sequence diagrams to specify LTL formulas of the modeled system for their correctness verification. In this section LTL formulas of the case study are obtained as well. Finally, section 5 indicates the pros and cons of this proposal along with future works and conclusions for this study.

II. UML Sequence Diagrams

In object-oriented software development, after defining actors and use cases for the application, a common task is to model and analyze the behavior or execution of use cases. Likewise, after establishing the main structural elements of the system, that is, classes and their components (attributes and methods), a traditional modeling task is to know the behavior and interaction of participant objects of those classes on critical scenarios to analyze behavioral characteristics of the participants. UML sequence diagrams permit describing system scenarios and understanding how their participant objects interact and react to special conditions over time [1] [3]. Participant elements, named or unnamed blocks, interact and communicate using synchronous or asynchronous messages (open and solid arrows, respectively).

UML sequence diagrams allow establishing combined fragments to support concurrent and parallel behavior, alternative and optional behavior, cycles, and exceptions [1] [2] [3]. Even though, there may be a high number of system scenarios, modeling critical scenarios of a system along with their participant instances is a relevant task for verifying a correct system execution.

Concerning the works of [2] and [3], UML sequence diagrams show collaborative behavior among participant objects, but sequence diagrams are not adequate to define the complete behavior and details for a particular object. For behavioral modeling of a single object, applying UML state diagram is more convenient. Even though UML state diagrams allow modeling the whole system behavior (the system as an object), to deduce all participant objects for each state change is neither simple nor direct. UML sequence diagrams allow modeling particular scenarios of the system execution, modularizing algorithmic behavior (combined fragments), and mixing behavioral scenarios (combined fragments and gates). A complete system behavior model is reachable.

As a practical application, this article models a modified version of a traditional case study [10] [11], the ATM system, to show new considerations to translate UML sequence diagrams into Promela code: to define guards of the associated state transition system to consistently specify LTL formulas for their correctness verification, and to be able to reach a correctness verification of the complete model.

Figure 1 shows the UML sequence diagrams for a scenario of the case study for interactions among anonymous instances of the classes User, ATM, and Bank. A user first inserts his / her card in the ATM (InsertCard() message); then, the ATM in parallel (combined fragment par), first, communicates with the Bank to validate the card status; and, second, asks for and receives the user introduced PIN. Two possible results exist regarding the card status (combined fragment alt): 1. if the card status is OK (Cardok = true), ATM asks for and receive the PIN validation; 2. if the card is not valid (Cardok = false), then the card is ejected. Two possible results exist concerning the PIN validation (combined fragment alt): 1. if the PIN were not valid, the card is ejected; 2. if the PIN were valid, the user can proceed to operate with its bank account using the ATM. A combined fragment alt exists to proceed with a bank transaction: 1. if the card or PIN were not valid, then the card is ejected; 2. if the card and the PIN were valid, then the User provides his account and the desired bank operation for the ATM to proceed. Assuming only 2 operations exist in the ATM (CashAdvance and ejectCard), the last combined fragment alt contains a combined fragment loop concerning the operation selection that iterates as long as the chosen operation is not ejectCard. For CashAdvance, the User indicates the required quantity of cash, then the ATM checks the ATM card balance and delivers its status. After, because BalanceOk = true or BalanceOk = false, there is an alt combined fragment to indicate either t pick the cash or insufficient money. Following this combined fragment alt, inside the loop the User can choose a new operation in the ATM. Finally, after finishing the loop, the card is ejected.

III. Algorithm to Translate a UML Sequence Diagram into Promela Code

In general, each participant of a UML sequence diagram is a process in Promela (Process or Protocol Meta Language). Basically, for two instances A and B in a UML sequence diagram, when A sends a message to B, either that corresponds to a signal or to a request of a method of B [14]. Modeling this situation in Promela, because Promela supports sending and receiving messages between processes by channels, A is the sender and B is the receiver of the message in a channel with the same name as the original message in the UML sequence diagram. Furthermore, when A needs return values, B sends them using an additional channel, $R_{OriginalName}$, including its output parameters.

It is entirely relevant to consider the synchronization nature of each message (synchronous or asynchronous), because in Promela the size of a channel allows determining that channels behavior [9]. A channel with size =1 represents an asynchronous channel while a channel with size = 0 represents a rendezvous communication or synchronous channel.

It is entirely relevant to consider the synchronization nature of each message (synchronous or asynchronous), because

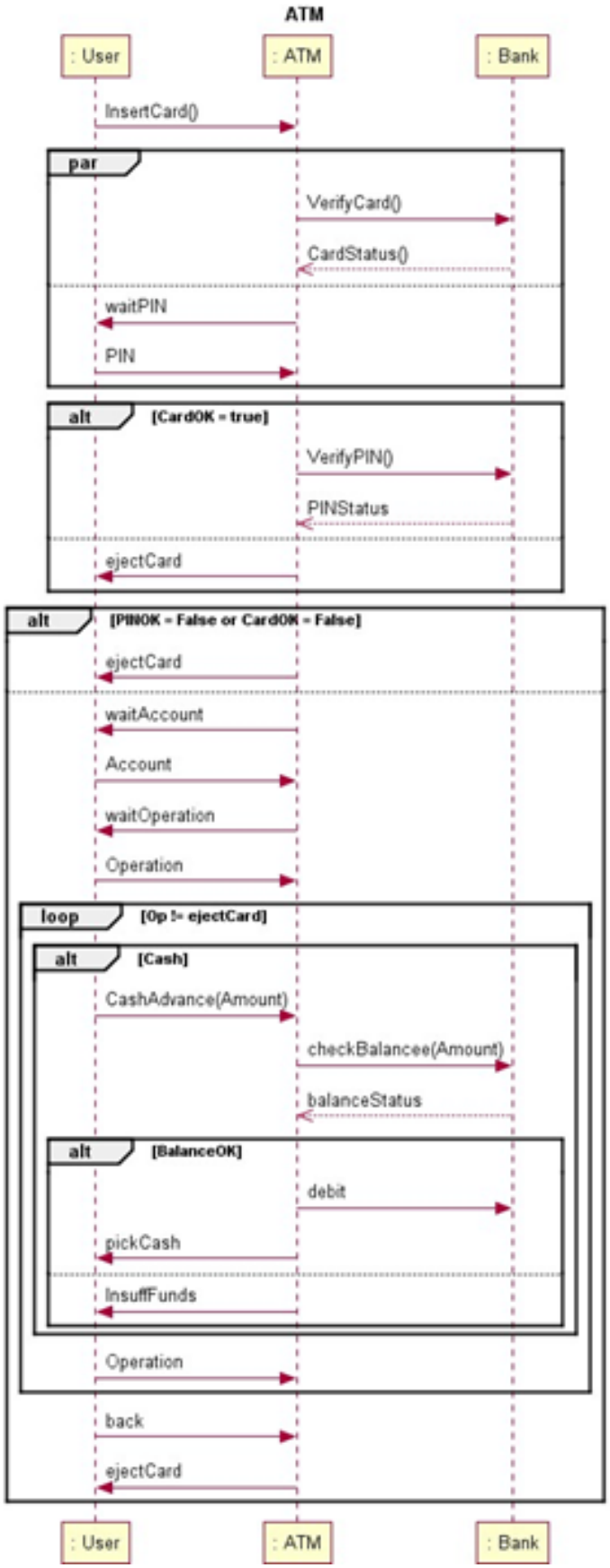

Fig. 1. UML Sequence Diagram for a Scenario of the ATM System.

in Promela the size of a channel allows determining that channels behavior [7]. A channel with size =1 represents an asynchronous channel while a channel with size = 0 represents a rendezvous communication or synchronous channel.

Furthermore, each synchronous message implies that the sender of the message has to wait until getting informed about a complete reception of the message. Thus, the sender needs to receive a reception confirmation to continue its work. Usually, each synchronous message in UML sequence diagrams requires a return value. This situation was solved in the classic UML by the use of return messages, but this situation is implicit in UML 2.0, that is, a return message is not necessary. For example, for the first message in the case study which is synchronous, there is an associated return value which should be known by the User, status of the card (CardOK), to continue in this scenario. One solution is to represent these required values as global variables in Promela, but global variables are visible for all the system processes, so messages are not directly required for communicating. A different solution for those synchronous messages which require a return value is to include an asynchronous channel to return those required values. This option claims for establishing communication because to see the result of any synchronous operation, a return message (R Message) is necessary. Nevertheless, as we later review, global variables are used for sharing elements among the main process and sub-processes. Thus, in the case study, to know the card status, there should be a return message in the first section of the combined fragment par, after ATM receives that information from the Bank, but this action is part of a section of the combined fragment par, and the process User communicates in a different section as well. So, this particular scenario includes a return message to know the card status as well as a global shared variable for the process User and its sub-processes User1 and User2. This modeling idea represents an essential element not considered by previous articles about this topic.

Moreover, a situation exists for which the use of asynchronous messages is typical: when a participant sends a message to itself. The simplest solution to represent the previous case in Promela is using an asynchronous channel of buffer size 1 to send and receive the message.

Through defining a channel in Promela, one can determine a message and its parameter. Besides, we define a symbolic name Parameters by using mtype. The main goal of this symbolic type is to support messages without parameters like signals in UML sequence diagrams. Thus, in Promela code the name of a message is represented by a channel name which also includes the type of the original parameters of the message with an additional parameter mtype for the symbolic name Parameters.

Promela uses special symbols for communication by sending and receiving values through channels, symbols ? and ! respectively. Therefore, so far there is an algorithm for translating a simple UML sequence diagram without combined fragments into Promela:

- Defining a symbolic name for the parameters.
- Identifying the synchronization nature of messages by creating buffer channels of size 0 for synchronous messages, and buffer channels of size 1 for asynchronous messages. These channels represent messages of the UML sequence diagram using the same name as those messages. If a synchronous message requires of returns values, a new synchronous channel R MessageName is defined along with the type of its return values. When a channel is defined, it has the same name as the original message and has a parameter mtype along with the original kind of message parameters.
- Defining a process for each participant in the UML sequence diagram, and assuring that each process sends and receives messages in the same order as in the sequence diagram.
- Defining an init process that runs the defined process associated with the UML sequence diagram participants.

After establishing the main step of a basic algorithm for translating simple UML sequence diagrams into Promela code, it is time to define a complete algorithm for that translation and thus be able to verify properties of the modeled scenario. After getting an algorithm with the established purpose, we will apply it to the model of the case study in Figure 1 for its translation into Promela code. To proceed, it is necessary to review the translation of the UML sequence diagram combined fragments into Promela code. Consequently, following the structure of the case study, we review first the translation of combined fragment par and apply it to the first part of the case study. Second, we review the translation of combined fragment alt and also apply it to the second combined fragment of the case study. In the translation of the alt combined fragment into Promela code, because the combined fragment opt is a reduced version of alt and both share the same logic, the translation into Promela code of the combined fragment opt is also reviewed. Finally, we detail the translation of the combined fragment loop and apply it for translating the remaining part of the case study by using the already reviewed combined fragments translation into Promela solutions. The code of Figures 4, 5, 6, 7 and 8 in the Appendix present a translation into Promela code for the complete case study of Figure 1.

A. Combined Fragment par into Promela

UML sequence diagrams allow modeling parallel interactions among processes which are so useful for modeling distributed and parallel scenarios. Assuming that different objects can communicate in parallel, this combined fragment is relevant for modeling parallel scenarios in which there are multiple sources and multiple targets, or only one source and multiple targets.

First, for a combined fragment par with multiple sources and destinations without communication interference. That is a simple situation for modeling in Promela because we can model each sender and receiver as different processes not using the same communication channels, that is, each couple or pair of processes in communication do not interfere with the communication of other couples. However, this situation is not directly simple for only one source and multiple destinations which is a common occurrence as in the case study of this article.

Second, for translating a combined fragment par into Promela code in which there is only one sender process and multiple target processes, the sender executes in parallel some sub-processes depending on the required number of parallel processes. This set of sub-processes sends messages and receives messages in parallel, and has coordinated access to the primary sender process properties. In that sense, each property of the primary sender process is a shared resource for the set of sub-processes, and each property has a token for guaranteeing exclusive access when that is required. Therefore, these shared properties and their tokens are global variables in Promela code for modeled scenarios. Thus, if sub-processes access different properties of the sender, those accesses are in parallel; and token variables permit exclusively obtaining a property value. Furthermore, it is relevant to distinguish between active and inactive processes in a combined fragment par. An active process in a combined fragment par sends, receives, or executes both actions in more than one division of the combined fragment.

Considering the case study combined fragment par, in the section Appendix Codes 3 and 4 show the Promela code translation for that combined fragment.

For a secure mutual exclusion, this article assumes that the properties of the sender are global variables in Promela code, and each property has a token to guarantee mutual exclusion. Also, asynchronous channels of buffer size1 models each token and each of them initially contains one stored value. Thus, when a process wants to access a shared resource, the process asks for the token, asks for a message in the associated channel, and if it were free (not used), then it has access to the resource. Note that during the combined fragment par, if a shared resource is free, there is a message on the channel. Moreover, if another sub-process wants to access to the same resource, it must wait for the token, the existence of a message in the channel, and for a sub-process that delivers the token sending a message in the associated token channel to wake up a process or sub-process that waits for the sent message. This translation that guarantees exclusive access on shared resources among multiple processes, participant objects of a UML sequence diagram translated into Promela code, is an additional improvement to previous translations of UML sequence diagrams into Promela code.

B. Combined Fragment alt into Promela

This combined fragment presents different interaction alternatives. Therefore, combined fragment alt represents a set of execution options for sending and receiving messages.

Promela allows using the conditional structure for sending messages. In that situation, each sender must include the condition associated with its actions. In the Appendix section, Code 2, Code 3, and Code 5 present the combined fragment alt use.

It is relevant to mention that a combined fragment opt behaves analog to a combined fragment alt with only one conditional division. Thus, this translation of combined fragment alt is also applicable for combined fragment opt.

Note that presenting examples with the use of messages with parameters is an additional issue not shown in previous mode ls for translating the original version of this case study [10] [11].

C. Combined Fragment loop into Promela

A combined fragment loop permits a cycle of actions when a defined guard condition is true. When that variable is false, the combined fragment loop finishes proceeding with the next action outside the loop. In each iteration of an instance of this combined fragment, there can be active and inactive participants, that is, active and inactive processes in the associated Promela code. Each active process in the Promela code presents a do cycle with a logical condition to iterate, and inactive processes only present a conditional statement to know the end of their cycle. In practice, in a Promela code of a UML sequence diagram combined fragment loop, condition variables are global because those variables are visible for active and inactive processes.

In a combined fragment loop, it is relevant to differentiate among active processes that communicate with synchronous and asynchronous messages. A sender of asynchronous messages does not wait for a confirmation of reception and continue with the iteration meanwhile the receiver can be waiting for receiving a possible previously lost message. The presence of asynchronous messages can originate faults (missed messages), and model checking reveals these situations. Since asynchronous messages are part of the nature of distributed systems [15], it is relevant to understand their nature and include forms of synchronization for processes that communicate by asynchronous messages.

Even though in a combined fragment loop, it is relevant to define a number of iterations, there exist loops for which that number is imprecise. In those scenarios, the end of the loop is not directly determined, and this situation can generate infinite cycles (a potential failure).

To illustrate the use of a combined fragment loop and its representation in Promela code, see Code 3 in the section Appendix. In iterations of a combined fragment loop there should be actions to finishing the cycle. This situation should be in any cycle, including those cycles without an exact number of iterations. The use of non-deterministic options is relevant to select values for simulating execution of system scenarios, specifically for those which include at least two potential options.

IV. LTL Formulas for Verifying the Correctness of UML Sequence Diagrams

Defining LTL formulas for UML sequence diagrams require to determine the associated state transition system (STS system). Logically, STS should represent the set of events in a UML sequence diagram (each event potentially determines a state change). Following and extending the original ideas of [8], a set of tuples including guard variables for the sending or reception of messages characterizes the states change in an STS system:

- $receive_{NameOfMessage}$: Identifies the reception of the message.
- $send_{NameOfMessage}$: Identifies the sending of the message.

- proc1 $Participant_{NameOfMessage}$: Identifies the participant source of the message.
- proc2 $Participant_{NameOfMessage}$: Identifies a participant tar

Even though state variables allow writing LTL formulas to verify properties of the modeled system execution, there are no guards that represent return values of reply messages. According to [6], return values are relevant because their change allows verifying the existence of transient and Byzantine faults. To represent reply values a new guard, variable-NameOfMessage, is added to the tuple. Thus, an LTL formula for the case study, after the participant User inserts the card in the ATM (after the user sends the message $send_{InsertCard}$, the User eventually receives a message to see the card state (receiveR InsertCard), and that value is assigned to the variable CardOKR InsertCard. Therefore, with the fact that a message cannot be received if it was not sent, the associated LTL formula for the sending and reception of a reply message is CardOkR InsertCard; receiveR InsertCard. Note that this LTL formula does not verify the existence of sources and destination. Considering that execution, combined fragment divisions usually depend on guard variables, those variables are also part of the messages inside those divisions.

Figures 2 and 3 show the set of transition tuples for the ATM system. The guards of these formulas permit verifying properties of the ATM system. Taking into account the syntax and semantic of symbols for defining LTL formulas (G for always, globally; F for eventually, in the future; X for next; and U for until), we define LTL formulas using variables of the STS system to verify properties of the ATM system.

Taking into account syntax and semantic of symbols to define LTL formulas (G for always, globally; F for eventually, in the future; X for next; and U for until), it is possible to define LTL formulas using variables of the STS system are defined to verify properties of the ATM system. The following sets of formulas are verified (they do not give a counterexample):

- G($receive_{InsertCard} \wedge (\neg CardOK_{CardStatus} \vee \neg PINOK_{PINStatu}) \rightarrow \neg(proc1\_User_{waitAccount} \wedge receive_{waitAccount})$. This LTL formula establishes that always a card is inserted and either that card of the associated got it PIN are not valid; then the User will not be asked for his account. There is not a counter-example for this formula..
- $\neg(proc1\_User_{pickCash} \wedge receive_{pickCash} \cup (proc1\_Bank_{debit} \wedge rceive_{debit}))$. This LTL formula indicates that a User does not receive a message pickCash meanwhile the Bank does not receive a debit message. There is not a counter-example for this formula.
- G($(proc1\_User_{insufFund} \wedge receive_{insufFund}) \rightarrow (\neg proc1\_User_{ejectCard} \cup (proc1\_ATM_{waitOperation} \wedge send_{waitOperation}))$. This LTL formula indicates that a User that has received a message InsufFund for insufficient funds is not the main actor of the message for the eject card action until the bank does not send a message waitOperation.

V. Discussion

We implemented these proposals using Eclipse [16] along with their PlantUML [17] and Spin / Promela [18] plugins. This technique permits validating behavioral modeling of software systems using UML Sequence diagrams which is of great value to guarantee software quality products. This proposal solution performs an automated validation which is one of its great properties for applying it to model the behavior of software products in the software development process. Nevertheless, current fast-development of software approaches such as RUP and XP in the practice demand use less time for modeling and formal modeling.

This article shows the importance of UML sequence diagrams and the benefits of modeling the behavior of software systems. The use of combined fragments in UML sequence diagrams gives the capacity for modeling algorithmic behavior, and by their translation into Promela code along with the definition and correctness verification of LTL formulas, detecting algorithmic, and faults in the requirement for their correction seem possible. Simulating faults in modeled systems behavior to see their effect on the software system would permit defining and applying solutions before their occurrence.

VI. Related Works

Primarily, Mellor et al. [19] detail about executable UML models which are like code for their examination. Those models do not work on model checking.

Baresi et al. [20] present an efficient solution for modeling checking graph transformation systems. This proposal would do not entirely support the model checking of model checking of UML sequence diagrams for their UML class diagram relations.

The works [21] and [22] applies model checking on UML sequence diagrams using labels to identify combined fragments for getting an ordered code in Eclipse Java [16] using PlantUML [17] and Spin / Promela plugging [18], that is, to accept a PlantUML sequence diagram as input and generate its translation to Promela code. The use of labels is recommendable to identify combined fragments for getting an ordered code, even though those labels do not affect the execution. These works do not directly describe the translation into Promela code of UML sequence diagrams combined fragments.

The work of [23] mainly describe and exemplify the development of a relational database schema from a conceptual UML schema in the form of a UML class diagram and OCL constraints, but they do not link UML class diagrams and UML sequence diagrams.

VII. Conclusions

This article shows new considerations for translating UML sequence diagrams into Promela code expanding this process for more case studies, specifically:

- distinguishing the synchronization nature of messages;
- defining how to work with shared resources through processes and associated sub-processes;

$\{send_{InsertCard}; proc1\_User_{InsertCard}; proc2\_ATM_{InsertCard}\}$
$\{receive_{InsertCard}; proc1\_ATMInsertCard; proc2_2\_User_{InsertCard}\}$

$\{send_{VerifyCard}; proc1\_ATM_{VerifyCard}; proc2\_Bank_{VerifyCard}\}$
$\{receive_{VerifyCard}; proc1\_Bank_{VerifyCard}; proc2\_ATM_{VerifyCard}\}$
$\{send_{CardStatus}; proc1\_Bank_{CardStatus}; proc2\_ATM_{CardStatus}\}$
$\{receive_{CardStatus}; proc1\_ATM_{CardStatus}; proc2\_Bank_{CardStatus}; CardOK_{CardStatus}\}$
$\{send_{R\_InsertCard}; proc1\_ATM_{R\_InsertCard}; proc2\_User_{R\_InsertCard}\}$
$\{send_{waitPIN}; proc1\_ATM_{waitPIN}; proc2\_User_{waitPIN}\}$
$\{receive_{waitPIN}; proc1\_User_{waitPIN}; proc2\_ATM_{waitPIN}; CardOK_{CardStatus}\}$
$\{send_{PIN}; proc1\_User_{PIN}; proc2\_ATM_{PIN}\}$
$\{receive_{PIN}; proc1\_ATM_{PIN}; proc2\_User_{PIN}\}$

$\{CardOK_{CardStatus}; send_{verifyPIN}; proc1\_ATM_{verifyPIN}; proc2\_Bank_{verifyPIN}\}$
$\{CardOK_{CardStatus}; receive_{verifyPIN}; proc1\_Bank_{verifyPIN}; proc2\_ATM_{verifyPIN}\}$
$\{CardOK_{CardStatus}; send_{PINStatus}; proc1\_Bank_{PINStatus}; proc2\_ATM_{PINStatus}\}$
$\{CardOK_{CardStatus}; receive_{PINStatus}; proc1\_Bank_{PINStatus};$
$proc2\_ATM_{PINStatus}; PINOK_{PINStatus}\}$
$\{!CardOK_{CardStatus}; send_{ejectCard}; proc1\_ATM_{ejectCard}; proc2\_User_{ejectCard}\}$
$\{!CardOK_{CardStatus}; receive_{ejectCard}; proc1\_User_{ejectCard}; proc2\_ATM_{ejectCard}\}$

$\{!CardOK_{CardStatus}; !PINOK_{PINStatus}; send_{ejectCard}; proc1\_ATM_{ejectCard};$
$proc2\_User_{ejectCard}\}$
$\{!CardOK_{CardStatus}; !PINOK_{PINStatus}; receive_{ejectCard}; proc1\_User_{ejectCard};$
$proc2\_ATM_{ejectCard}\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; send_{waitAccount}; proc1\_ATM_{waitAccount};$
$proc2\_User_{waitAccount}\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; receive_{waitAccount}; proc1\_User_{waitAccount};$
$proc2\_ATM_{waitAccount}\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; send_{Account}; proc1\_User_{Account};$
$proc2\_ATM_{Account}\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; receive_{Account}; proc1\_ATM_{Account};$
$proc2\_User_{Account}\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; send_{waitOperation}; proc1\_ATM_{waitOperation};$
$proc2\_User_{waitOperation}\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; receive_{waitOperation}; proc1\_User_{waitOperation};$
$proc2\_ATM_{waitOperation}\}$

Fig. 2. First Part of State Transition System for ATM System.

$\{CardOK_{CardStatus}; PINOK_{PINStatus}; send_{Operation}; proc1\_User_{Operation};$
$proc2\_ATM_{Operation}\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; receive_{Operation}; proc1\_ATM_{Operation};$
$proc2\_User_{Operation}; (Op = 1); Cash\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$send_{checkBalance}; proc1\_ATM_{checkBalance}; oc2\_Bank_{checkBalance}\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$receive_{checkBalance}; proc1\_Bank_{checkBalance}; proc2\_ATM_{checkBalance}\}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$send_{balanceStatus}; proc1\_Bank_{balanceStatus}; proc2\_ATM_{balanceStatus}; \}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$receive_{balanceStatus}; proc1\_ATM_{balanceStatus};$
$proc2\_Bank_{balanceStatus}; BalanceOK\}$

$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$BalanceOK; send_{debit}; proc1\_ATM_{debit}; proc2\_Bank_{debit}; \}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$BalanceOK; receive_{debit}; proc1\_Bank_{debit}; proc2\_ATM_{debit}; \}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$BalanceOK; send_{pickCash}; proc1\_ATM_{debit}; proc2\_User_{pickCash}; \}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$BalanceOK; receive_{pickCash}; proc1\_User_{pickCash}; proc2\_ATM_{pickCash}; \}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$!BalanceOK; send_{InsuffFunds}; proc1\_ATM_{InsuffFunds}; proc2\_User_{InsuffFunds}; \}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$!BalanceOK; receive_{InsuffFunds}; proc1\_User_{InsuffFunds};$
$proc2\_ATM_{InsuffFunds}; \}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$send_{Operation}; proc1\_User_{Operation}; proc2\_ATM_{Operation}; \}$
$\{CardOK_{CardStatus}; PINOK_{PINStatus}; (Op = 1); Cash;$
$receive_{Operation}; proc1\_ATM_{Operation}; proc2\_User_{Operation}; \}$

$\{send_{ejectCard}; proc1\_ATM_{ejectCard}; proc2\_User_{ejectCard}\}$
$\{receive_{ejectCard}; proc1\_User_{ejectCard}; proc2\_ATM_{ejectCard}\}$

Fig. 3. Second Part of State Transition System for ATM System.

- giving steps to translate the combined fragment loop regardless of their end condition; and
- translating into Promela reception and sending of messages with parameters.

The steps to define a state transition system to establish and verify LTL formulas in a Promela code for a UML sequence diagram gives a way to check the correctness of UML sequence diagram with the elements analyzed by this article. By having a general algorithm to translate UML sequence diagrams into Promela code along with knowing how to define a state transition system for UML sequence diagrams, establishing and verifying LTL formulas in the model system is a possible task. Without a doubt, this articles proposal permits verifying the correctness of UML sequence diagrams.

Promela code along with the LTL formula verification facilitates detection of faults in a diagram written in Promela, and UML sequence diagram translations into Promela code would permit their refinement. However, for complete verification of correctness about UML sequence diagrams messages syntax, the associated UML class diagrams are necessary to know their methods language.

This research proposal establishes steps for translating UML sequence diagrams into Promela code. Those steps are implementable as a software application using free tools for obtaining the Promela code and testing the UML sequence diagrams correctness.

Because this article gives the necessary steps for translating UML sequence diagrams into Promela code for the LTL formulas verification, using the mentioned steps and producing a tool with them would enable to find mistakes for their correction and producing accurate software according to the correctness of the software requirements. Even though Spin and Promela are more linked to distributed environments, this article demonstrates that is possible to use those tools in a non-distributed environment like traditional UML models.

As a future work, extending currently produced tools to support the translation into Promela code of other combined fragments of UML sequence diagrams (break, strict, ignore, consider, assert, and neg). Even though, the already reviewed combined fragments in this article are algorithmically the most relevant, our goal of producing a complete tool for correctness verification takes us to work on the way to translate these additional combined fragments. Now, there are steps to translate a UML sequence diagram in Promela code for implementing a software tool to generate Promela code on similar case studies. However, that does not guarantee the support for other UML diagram models such as class diagrams. For a complete consistency among UML class diagrams and UML sequence diagrams, UML class diagrams have to be an additional input for a refinement process. Therefore, a future goal is to produce a tool for refinement and consistency verification among UML class and sequence diagrams.

REFERENCES

[1] S. G. Akl, "Superlinear performance in real-time parallel computation," *J. Supercomput.*, vol. 29, no. 1, pp. 89–111, Jul. 2004. [Online]. Available: https://doi.org/10.1023/B:SUPE.0000022574.59906.20

[2] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[3] R. Miles and K. Hamilton, *Learning UML 2.0*. O'Reilly Media, Inc., 2006.

[4] T. Pender, *UML Bible*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[5] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Proceedings of the VIII Banff Higher Order Workshop Conference on Logics for Concurrency : Structure Versus Automata: Structure Versus Automata*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996, pp. 238–266. [Online]. Available: http://dl.acm.org/citation.cfm?id=239519.239527

[6] M. Usman, A. Nadeem, T.-h. Kim, and E.-s. Cho, "A survey of consistency checking techniques for uml models," in *Proceedings of the 2008 Advanced Software Engineering and Its Applications*, ser. ASEA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 57–62. [Online]. Available: https://doi.org/10.1109/ASEA.2008.40

[7] Y. KAWAKAMI, T. YOKOGAWA, H. MIYAZAKI, S. AMASAKI, Y. SATO, and M. HAYASE, "Symbolic model checking of interactions in sequence diagrams with combined fragments by smv," vol. 4, pp. 1692–1695, 11 2010.

[8] F. U. Muram, H. Tran, and U. Zdun, "A model checking based approach for containment checking of uml sequence diagrams," in *23rd Asia-Pacific Software Engineering Conference (APSEC 2016)*, December 2016. [Online]. Available: http://eprints.cs.univie.ac.at/4830/

[9] J. Chen and S. S. Kulkarni, "Application of automated revision for UML models: A case study," in *Distributed Computing and Networking - 13th International Conference, ICDCN 2012, Hong Kong, China, January 3-6, 2012. Proceedings*, 2012, pp. 31–45. [Online]. Available: https://doi.org/10.1007/978-3-642-25959-3_3

[10] M. Ben-Ari, *Principles of the Spin Model Checker*, 1st ed.

[11] V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and M. Pourzandi, "Formal verification and validation of uml 2.0 sequence diagrams using source and destination of messages," *Electron. Notes Theor. Comput. Sci.*, vol. 254, pp. 143–160, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2009.09.064

[12] M. Debbabi, F. Hassane, Y. Jarraya, A. Soeanu, and L. Alawneh, *Verification and Validation in Systems Engineering: Assessing UML/SysML Design Models*, 1st ed. Berlin, Heidelberg: Springer-Verlag, 2010.

[13] A. Nimiya, T. Yokogawa, H. Miyazaki, S. Amasaki, Y. Sato, and M. Hayase, "Model checking consistency of uml diagrams using alloy," vol. 71, pp. 547–550, 11 2010.

[14] C. Vidal, R. Villarroel, X. L'opez, and J. Rubio, "Una propuesta de algoritmo spin / promela para el anlisis y diagnstico de errores en diagramas de secuencia uml," *Información Tecnológica*, vol. 30, no. 1, 2019.

[15] M. A. Oubelli, N. Younsi, A. Amirat, and A. Menasria, "From uml 2.0 sequence diagrams to promela code by graph transformation using atom3," in *Proceedings of the Third International Conference on Computer Science and its Applications, CIIA*, Saida, Algeria, 2011.

[16] "Eclipse foundation: The platform for open innovation and collaboration," http://www.eclipse.org/, accessed: 2018-13-08.

[17] "Plantuml in a nutshell," http://en.plantuml.com/, accessed: 2018-13-08.

[18] "Eclipse plug-in for spin," http://matrix.uni-mb.si/en/science/tools/eclipse-plug-in-for-spin/, accessed: 2018-13-08.

[19] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[20] L. Baresi, V. Rafe, A. T. Rahmani, and P. Spoletini, "An efficient solution for model checking graph transformation systems," *Electron. Notes Theor. Comput. Sci.*, vol. 213, no. 1, pp. 3–21, May 2008. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2008.04.071

[21] "Review of: Distributed systems: An algorithmic approach (2nd edition) by sukumar ghosh," *SIGACT News*, vol. 47, no. 4, pp. 13–14, Dec. 2016, reviewer-de Vera,Jr., Ramon. [Online]. Available: http://doi.acm.org/10.1145/3023855.3023860

[22] S. Ghosh, *Distributed Systems: An Algorithmic Approach, Second Edition*, 2nd ed. Chapman & Hall/CRC, 2014.

[23] M. Gogolla and K.-H. Doan, "Quality improvement of conceptual uml and ocl schemata through model validation and verification." in *Conceptual Modeling Perspectives*, J. Cabot, C. Gmez, O. Pastor, M.-R. Sancho, and E. Teniente, Eds. Springer, 2017, pp. 155–168. [Online]. Available: http://dblp.uni-trier.de/db/conf/birthday/olive2017.htmlGogollaD17

VIII. Appendix

```
mtype = {Parameters}

chan InsertCard   = [0] of {mtype};
chan R_InsertCard = [1] of {mtype, bit};


chan VerifyCard   = [0] of {mtype};
chan CardStatus   = [1] of {mtype, bit}
chan waitPIN      = [0] of {mtype}
chan PIN          = [1] of {mtype, byte}
chan R_PIN        = [1] of {mtype, bit}


chan VerifyPIN  = [0] of {mtype, byte}
chan PINStatus  = [1] of {mtype, bit}
chan ejectCard  = [0] of {mtype}


chan waitAccount    = [0] of {mtype}
chan Account        = [0] of {mtype, byte}
chan waitOperation  = [0] of {mtype}
chan Operation      = [0] of {mtype, byte}


chan CashAdvance    = [0] of {mtype, int}
chan checkBalance   = [0] of {mtype, int}
chan balanceStatus  = [1] of {mtype, bit}

chan debit          = [0] of {mtype, int}
chan pickCash       = [0] of {mtype}
chan InsuffFunds    = [0] of {mtype}

chan back  = [0] of {mtype}

////Shared Resource and Tokens
chan Token1 = [1] of {mtype}; char Token2 = [1] of {mtype};

//Variables shared by processes User, User1 & User2...
bit CardOK; byte PINNumber;

//Variables shared by ATM, ATM1 & ATM2...
bit CardOK_ATM; byte PINNumber_ATM;
```

Fig. 4. Case Study - Promela Code (Part I)

```
proctype User()
{ bit BalanceOK; bit PINOK; byte Op; bit Cash; byte Acc; int Amount;
    InsertCard!Parameters; //First Action...
    CombinedFragment_Par:
        Token1!Parameters; run User1();
        Token1!Parameters; run User2();
        //Wait until obtain a final result...
        Token1?Parameters; Token1?Parameters;
    CombinedFragment_Alt1:
    if
      :: (!CardOK) -> ejectCard?Parameters
      :: else          -> R_PIN(PINOK)
    fi;
    CombinedFragment_Alt2:
    if
      :: (!CardOK || !PINOK) -> ejectCard?Parameters
      :: else  -> waitAccount?Parameters;
            Acc = 1; //#Account = 1...
            Account!Parameters(Acc); waitOperation?Parameters;
            Cash = 1; Op = Cash; //Operation = CashAdvance
            Operation!Parameters(Op);
            CombinedFragment_Loop:
            do
            :: (Op == 1) ->
                   CombinedFragment_Alt3:
                       if
                       ::(Cash) -> Amount = 1000; //Money
                          CashAdvance!Parameters(Amount);
                          R_CashAdvance(BalanceOK)
                          CombinedFragment_Alt4:
                             if
                             :: (BalanceOK)->pickCash!Parameters
                             ::  else -> skip;
                             fi
                       :: else        -> skip;
                       fi;
                       if
                       :: (1) -> Op = 1; //Cash Advance
                       :: (1) -> Op = 0; //Eject Card
                       fi;
                       Operation!Parameters(Op);
            :: (Op != 1) -> break;
            od;
            back!Parameters; ejectCard?_Parameters; //Scenario End
    fi;
}
```

Fig. 5. Case Study - Promela Code (Part II)

```
proctype ATM()
{  bit PIN_Status; byte AccATM; byte ATMOp; int Amount;
   bit  BalanceStatusATM;
   InsetCard?Parameters; //First action of ATM...
   CombinedFragment_Par:
       Token!Parameters; run ATM1();
       Token!Parameters; run ATM2();
       //Wait until obtain a final result...
       Token?Parameters; Token?Parameters;
    CombinedFragment_Alt1:
    if
    :: (CardOK) -> VerifyPIN!Parameters(PINNumberATM);
            PINStatus?Parameters(PIN_Status);
            R_PIN!Parameters(PIN_Status);
    :: else -> ejectCard!Parameters;
    fi;
    CombinedFragment_Alt2:
    if
      :: (!CardOK || !PINOK) -> ejectCard!Parameters;
      :: else -> waitAccount!Parameters;
           Account?Parameters(AccATM); waitOperation!Parameters;
           Operation?Parameters(ATMOp); //a Cash Advance
           CombinedFragment_Loop:
           do
           :: (ATMOp == 1) ->
                Cash = ATMOp;
                CombinedFragment_Alt3:
                if
                :: (Cash) -> CashAdvance?Parameters(Amount);
                  checkBalance!Parameters(Amount);
                  balanceStatus?Parameters(BalanceStatusATM);
                  CombinedFragment_Alt4:
                  if
                  :: (BalanceStatusATM) -> debit!Parameters(Amount);
                  pickCash!Parameters;
                  :: else -> InsuffFunds!Parameters;
                  fi;
                :: else        -> skip;
                fi;
                Operation?Parameters(ATMOp);
           od
       back?Parameters;
       ejectCard!Parameters;
    fi
}
```

Fig. 6. Case Study - Promela Code (Part III)

```
 ////Sub-Processes User1 & User2
proctype User1()
{
  CombinedFragment_Par:
    //Asking for the Token
    Token?Parameters;
    R_InsertCard?Parameters(CardOK); //Receive Status the Card status...
    //Releasing the Token
    Token!Parameters;
 }
 proctype User2()
 {
  CombinedFragment_Par:
    //Asking for the Token
    Token?Parameters;

    waitPIN?Parameters;
    PINNumber = 1; PIN!Parameters(PINNumber);

    //Releasing the Token
    Token!Parameters;
 }
 ////Sub-Processes ATM1 & ATM2
proctype ATM1()
{
CombinedFragment_Par:
  //Asking for the Token
  Token?Parameters;
  VerifyCard!Parameters();
  CardStatus?Parameters(CardOKATM);
  R_InsertCard(CardOKATM);
  //Releasing the Token
  Token!Parameters;
}
proctype ATM2()
{
CombinedFragment_Par:
  //Asking for the Token
  Token?Parameters;
  waitPIN!Parameters;
  PIN?Parameters(PINNumberATM);
  //Releasing the Token
  Token!Parameters;
}
```

Fig. 7. Case Study - Promela Code (Part IV)

```
proctype Bank()
 {
   int V; byte _PIN_; bit BalanceOKBank;

   VerifyCard?Parameters;
   if
     :: (1) -> CardStaus!Parameters(1); //Valid Card
     :: (1) -> CardStatus!Parameters(0) //Valid Card
   fi;

   CombinedFragment_Alt1:
   if
     :: (CardOK) -> VerifyPIN?Parameters(_PIN_);
              if
              :: (1) -> PINStatus!Parameters(1) //Valid PIN
              :: (1) -> PINStatus!Parameters(0) //Valid PIN
              fi;
    ::  else      -> skip;
  fi;
  CombinedFragment_Alt2:
  if
    :: (CardOK) -> checkBalance?Parameters(V);
              if
              :: (1) -> balanceStatus!Parameters(1); //Valid Balance.
              :: (1) -> balanceStatus!Parameters(0); //Not valid Balance.
              fi;
    ::  else     -> skip;
  fi;
  CombinedFragment_Alt3:
     if
       :: (CardOK) -> checkBalance?Parameters(V);
              if
              :: (1) -> BalanceOKBank = 1; balanceStatus!Parameters(BalanceOKBank);
              //Valid Balance.
              :: (1) -> BalanceOKBank = 1; balanceStatus!Parameters(BalanceOKBank);
              //Not valid Balance.
              fi;
       ::  else     -> skip;
     fi;
 }
init {run ATM(); run User(); run Bank();}
```

Fig. 8. Case Study - Promela Code (Part V)