Abstract

UML 2 communication diagrams are a well known graphical language and are widely used to specify the dynamic behaviors of transaction-oriented systems. However, communication diagrams are expressed in a semi-formal modeling language and need a well-defined formal semantic base for their notations. This formalization enables analysis and verification tasks. Many efforts have been made to transform sequence diagrams into formal representations including Petri nets, but very few works have been proposed for transforming communication diagrams into formal methods, and none, to our knowledge, has used Büchi automata. Büchi automata are a mathematical tool allowing formal specification of the system dynamics and they are commonly used in Model Checking based on automata theory. For these reasons, in this paper, we present a transformation approach that consists of a source metamodel for UML 2 communication diagrams, a target metamodel for Büchi automata and transformation rules. This approach has been implemented using Atlas Transformation Language (ATL). A Cellular Phone System is considered, as a case study.

Keywords UML 2, Communication diagrams, Büchi automata, Model checking, Model transformation, Metamodelling, Transformation rules, ATL
UML is widely accepted by the software engineering community as a standard in industry and research.

The UML 2 provides several categories of diagrams to specify different aspects of the system, like structural or behavioral aspect. For behavioral-intensive systems, the dynamic behavior is the most critical aspect to take into account.

Communication Diagrams (CDs) - which are considered in this paper - belong to the behavioral diagrams like sequence diagrams. They are collectively known as interaction diagrams. The communication diagrams are used to understand and document the interactions between the objects and also in order to show how the classes are working together to achieve a goal [16]. We selected CDs from UML 2 interactions diagrams since, in contrast to sequence diagrams, they do not only supply the message flow information of an interaction, but also the underlying structural information building the context of the interaction, i.e., the links via which messages are sent.

UML models of the interaction category are generally transformed for verification and validation purposes. This is because dynamic models, such as CDs, lack sufficient formal semantics [14]. Moreover, UML was created as a semi-formal modeling language, it does not include a formal semantics [4]. This limitation makes rigorous analysis difficult, which leads to an ambiguous model and problems with modeling the process concurrency, synchronization, and mutual exclusion [22]. On the other hand, one of the most important problems of designing phase in software engineering is to verify all designed things before going to the implementation phase because starting the implementation phase before verifying design phase is a big risk in big projects [16].

Thus, production of the new technologies for verification and validation of UML models seem very crucial and converting UML to some mathematical models, in order to formalize and validate them can be a very important task. Many researchers have been performed in order to only transform the UML models into a formal model [16]. In our approach, the formal model is Büchi Automata (BA) [2]. Büchi automata can model, among others like Petri nets, the behavior of systems having concurrency. Since BAs are a formal model and they have a mathematical representation with a well-defined syntax and semantics, they do not carry any ambiguity and thus, are able to be validated, verified and simulated. One of the main advantages of using BA for model checking is that both, the modeled system and the specification are represented in the same way i.e. with automata [5]. Then, the SPIN [10] model checker can be used as a formal method.

The suggested approach is mainly based on the technique of metamodel transformations [6]. Such approach consists in: defining the source metamodel of communication diagrams, defining the target metamodel of Büchi automata, and defining the transformation rules. Our transformation contributes to the on-going attempt to develop a formal semantics of UML [17] based on model transformations [6]. All these reasons motivate the work to map or to transform UML 2 communication diagrams to Büchi automata. To achieve this goal, this paper proposes a set of rules for this transformation.

The rest of this paper is organized as follows. In section 2 we discuss related work. In section 3 we briefly review the features of UML 2 interactions and communication diagrams and we give an informal introduction of Büchi automata. Section 3 also describes both source and target metamodels suited to the transformation. We then show in section 4 how we translate a communication diagram into behaviorally equivalent Büchi automaton. In section 5 is presented the application of the proposed transformation rules with a Cellular Phone System. Section 6 presents the implementation of the system design transformation.
process. We finally conclude our work in section 7 with some remarks and future work.

2: Related Works

Many research works have been done on model transformations and especially to transform sequence diagrams into Petri Nets in order to perform formal verification. UML sequence diagrams have been very considered. In our knowledge, only two works addressed directly communication diagrams for verification purposes.

Elmansouri [7] proposed an approach to transform UML 1 statechart and collaboration diagrams to Coloured Petri Net models. They used graph transformation to specify transformation rules.


These two approaches propose the usage of Petri Nets. Our work essentially differs in two ways. First, we use Büchi automata rather than Petri Nets as a target model for verification. Second, we do not use graph transformation to specify transformation rules.

This paper deals with transforming UML 2 communication diagrams into Büchi automata models for analysis and verification purposes by using some transformation rules expressed in the ATL language. Our work is a step forward in a project that is exploring means to define a semantics for UML 2 communication diagrams and to exploit the model checker SPIN [10].

3: The Basic Metamodels

3.1: UML 2 Diagrams for Interaction

UML 2 divides diagrams into two categories: structural modeling diagrams and behavioral modeling diagrams:

- Structural diagrams illustrate the static features of a model. Static features include classes, objects, interfaces and physical components. In addition they are used to model the relationships and dependencies between elements. Structural diagrams include Class diagram, Object diagram, and some others.

- Behavioral diagrams describe how the modeled resources in the structural diagrams interact and how they execute each other capabilities. The behavioral diagram puts the resources in motion, in contrast to the structural view, which provides a static definition of the resources [19]. Behavioral diagrams include the Interaction diagrams, Use Case diagram, Activity diagram, State Machine diagram and others.

Interaction diagrams [20] are defined by UML 2 to emphasize the communication between objects, not the data manipulation associated with that communication. Interaction diagrams focus on specific messages between objects and how these messages come together to realize functionality. An interaction can be displayed in several different kinds of diagrams: Communication Diagrams, Sequence Diagrams, Interaction Overview Diagrams, and Timing Diagrams.
Communication diagrams are one of the kinds of interaction diagrams that focuses on the elements involved in a particular behavior and what messages they pass back and forth. Communication diagrams emphasize the objects involved more than the order and type of the messages exchanged.

Sequence diagrams are one of the kinds of interaction diagrams that emphasize the type and order of messages passed between elements during execution.

Interaction overview diagrams are simplified versions of activity diagrams. Instead of emphasizing the activity at each step, interaction overview diagrams emphasize which element or elements are involved in performing that activity.

Timing diagrams are designed to specify the time constraints on messages sent and received in the course of an interaction. They are often used to model real-time systems such as satellite communication or hardware handshaking [20].

Both communication and sequence diagrams concentrate on the presentation of dynamic aspects of a software system, each from a different perspective. Sequence diagrams stress time ordering while communication diagrams focus on organisation. Despite their different emphases they share a common set of features. Booch, Rumbaugh, and Jacobson [3] claim that they are semantically equivalent since they are both derived from the same sub-model of the UML metamodel, which gives a systematic description of the syntax and semantics of the UML. In this work we concentrate on communication diagrams.

### 3.1.1: Communication Diagrams

Communication Diagrams (CDs) and Sequence Diagrams (SDs) are two views of the same scenario where CD gives structural view of a scenario and SD gives the temporal one. CDs record the same information as SDs and, hence, scenarios. They just provide a different view - one that focuses on the structural view of the object interactions, rather than the temporal view. The communication is implicit in a SD, rather than explicitly represented as in a CD. Some tools even generate CDs from SDs (or vice versa).

CDs appear at first glance, to offer little to the Analyst that is not available from the SD. They are more often used by designers when planning for object interaction at run-time.

Many experienced Analysts [21], however, find that the more modeling they do, the more often certain arrangements of objects or classes seem to occur, and these are often associated with particular tasks. These arrangements become recognisable as patterns.

Patterns have become very important part of the modeler’s tool-set, and an awareness of which kinds of patterns can be used to tackle which kinds of problems can often make the modeler’s job a little more manageable. It is easier to recognise these patterns in communication diagrams than in sequence diagrams. CDs also help to define the shape of the Class Diagram.

This is one of the reasons that this paper deals with communication diagrams rather than sequence diagrams.

### 3.1.2: Communication Diagrams Metamodel

Communication diagram expresses interactions between objects by exchanging messages. We provide UML 2 communication diagram a metamodel, which graphically displays the abstract syntax in terms of class diagram. The metamodel complies with the interaction metamodel provided by OMG [17], whereas showing only the essential syntax constructs.
of a communication diagram, to facilitate the mapping to the Büchi automata. In the metamodel, the syntax elements are represented as classes, shown as boxes, and relations elements are represented as associations, shown as lines among classes in terms of class diagram. A hollow diamond on an association represents aggregation relationship (has-a), while a filled diamond represents a composition relationship (part-of). A triangle on an association represents a generalization between a superclass and its subclass. The numbers attached to an association are called multiplicities, which describe how many objects may exist in the association. A star denotes zero or more. If no multiplicity is present, a one-to-one relationship is implied.

Figure 1. A simplified metamodel for UML 2 communication diagrams

Figure 1 shows our simplified metamodel for UML 2 communication diagrams. A communication diagram is represented by a set of Objects. An Object has a top-down ordered sequence of Occurrences. Occurrences may receive and send messages. A Message consists of a sent event and a received event, which are placed on two different Occurrences. A Message has five attributes:

- **msg**: of type String that represents the content of the message to send (the prototype of the function to trigger)
- **msgInitial, msgFinal**: of type Boolean, to differentiate between initial and final messages from other messages.
- **beginLoop, endLoop**: of type Int describe the existence of loops in the communication diagram (the iteration in the “sending of messages”).

The message scheduling is described by the Occurrence class, through its order attribute of type Int. An Occurrence depends on a single Object through the comeFrom association which connects it with the Object class. The Object in turn, covers a set of Occurrences of sending and receiving for incoming and outgoing messages of this Object. This property is described by the covered association with multiplicity 0 ..* between the two classes Object and Occurrence.

Figure 2 shows an adapted metamodel for technical constraints of ATL language. The Root class is only added for these constraints. Effectively, without the class Root, the metamodel would have been split in multiple files depending on the number of classes available in the metamodel.
3.2: The target Metamodel

Infinite words conveniently represent the infinite behaviors exhibited by a non-terminating system and model checking traditionally deals with non-terminating systems. The simplest computation model for infinite behaviors is the $\omega$-automaton [8], which accepts words of infinite length. Büchi automata are the simplest kind of $\omega$-automata that accepts infinite words and were introduced by Büchi in [2]. A Büchi automaton has the same structure as a finite-state automaton but it has a different notion of acceptance. It is also given by a 5-tuple $(\Sigma, Q, \Delta, q_0, F)$:

1. $\Sigma$ is a finite set of symbols (the alphabet),
2. $Q$ is a finite set of states,
3. $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accepting states.

An infinite run of a BA is accepting if it visits at least one acceptance state infinitely often. Figure 3 shows such an automaton.

$S_1$ is a final state

$w_1 = S_0S_1S_2S_2S_2S_2\ldots$ accepted

$w_2 = S_0S_1S_2S_1S_1\ldots$ accepted

$w_3 = S_0S_1S_2S_1S_1\ldots$ rejected
In the automata metamodel shown in Figure 4, an automaton consists of *States* and *Transitions*. Each *State* can have *Transitions* to another *State*. Each *Transition* knows its *source* and its *target State*. Similarly, each *State* knows its implicit incoming and outgoing *Transitions*. A *State* is identified by a name (the *nameOrder* attribute of type *String*). A *State* may be initial, final or intermediate, according to the values of the two attributes *initialState* and *finalState*.

**Figure 4. Finite-state Automata Metamodel**

### 4: Transformation Approach

#### 4.1: The Transformation Process

To make easier the rules’ specification of the transformation, our efforts address the transformation at the metamodel level of UML 2. This also allows the mapping between the concepts of both metamodels source and target. The metamodeling based transformation approach for transforming UML 2 communication diagrams into Büchi automata is shown in Figure 5. Communication diagrams are assumed to be syntactically and static-semantically correct. The transformation process is achieved by the application of *rules*. A transformation rule consists in transforming a concept outlined in the source metamodel to a corresponding concept in the target metamodel.

#### 4.2: Transformation Rules

In the following, we define the rules for transforming communication diagrams into automata. The transformation rules describe the interactions that exist between classes of the "communication diagrams" metamodel and "automata" metamodel. These rules consist essentially of:

**Rule1:** All objects in the communication diagram become states of the produced automaton. Any object of *Object* class of the source metamodel becomes an object of *State* class of the target metamodel.

**Rule2:** All messages in the communication diagram become constituent states of the produced automaton. Any object of *Message* class of the source metamodel becomes an object of *State* class of the target metamodel.

**Rule3:** The sending of a message in the communication diagram becomes a transition that connects the state corresponding to the emitting object and the state corresponding...
Rule 4: The reception of a message in the communication diagram becomes a transition that connects the state corresponding to the sent message and the state corresponding to the object that will receive it. The target relationship in the metamodel source becomes an object of the Transition class.

Rule 5: The final state of a received message of order "n" is the same as the initial state of the sent message of order "n+1". Two successive states unrelated and with the same order become one state bringing the same order, and replacing them by linking all the transitions that were related to the two states to be transformed.

Rule 6: The iteration of a message in the communication diagram becomes a loop for a transition to a state that corresponds to it in the Büchi automaton. We add an object of class Transition labelled epsilon that connects two states where endLoop of the first state is equal to beginLoop of the second one.

The transformation is carried out in three steps. The first step produces a non connex graph whose states are not all connected because a message sent in the source model of communication diagram is transformed into three interconnected states with two transitions.

The second step has the objective to link all states that are not connected, that is to say that for every two states not connected and with the same order, we create for them a single state which has the same order and that will be the reference of all transitions that have been related to the transformed state. In other words, all connex components are connected to obtain a connex graph. The result of this second step is then a finite-state automaton.

The third and final step of the transformation is to take the finite-state automaton and
add to this automaton empty transitions in order to transform it into a Büchi automaton. This will be done as follows: An empty transition is added for each iterative message, this transition will allow, whenever the iterative function of the message is triggered, to return to the first state of the iteration to move through the same states corresponding to the scope of the occurred iteration, so the transition gives the opportunity to continue the states of the cycle of the iteration whenever the iterative message is resent. All this allows us to obtain a Büchi automaton.

---

**Rule1, Rule 2 and Rule 3**

```plaintext
rule Message2State {
  - the name of the principal rule in the first step of transformation that produces
  3 States and 2 Transitions for each Message in the source model that conforms to the Communication
  Diagram's Metamodel
from M : CmnMessage
to
S1 : Stat!State(  - it produces the first State
  nameOrder <- 'S' + thisModule.Num.toString(),
  msgInitial <- true,
  msgFinal <- false,
  beginLoop <- M.beginLoop,
  endLoop <- 0
),
T1 : Stat!Transition(  - it produces the Transition that corresponds to the fact of sending the Message
  symbol <- 'Send' + '(' + M.sent.comeFrom.name + ',' + M.msg + ',' + M.received.comeFrom.name + ')',
  source <- S1,
  target <- S2
),
S2 : Stat!State(  - it produces the second State
  nameOrder <- 'S' + M.incrmt1Num().toString(),
  msgInitial <- false,
  msgFinal <- false,
  beginLoop <- 0,
  endLoop <- 0
),
T2 : Stat!Transition(  - it produces the Transition that corresponds to the fact of Receiving the Message
  symbol <- 'Receive' + '(' + M.sent.comeFrom.name + ',' + M.msg + ',' + M.received.comeFrom.name + ')',
  source <- S2,
  target <- S3
),
S3 : Stat!State(  - it produces the third State
  nameOrder <- 'S' + M.incrmt2Num().toString(),
  msgInitial <- false,
  msgFinal <- true,
  beginLoop <- 0,
  endLoop <- M.endLoop
)
do { thisModule.Num <- thisModule.Num+2; }  - a global variable that count the number of States
to give each one of them it’s order number
}
```
Rule 4 and Rule 5

rule transitionChange {  
  - the name of the first principal rule in the second step of transformation  
  that returns the Transitions of the State that will be replaced and relates it to the replaced State  
from  
  S : Stat!Transition (S.changement())  
to  
  T : realStat!Transition (  
symbol <- S.symbol,  
source <- S.getStateCorrespond(),  
-target <- S.target  
)  
do { thisModule.first <- thisModule.first+2; }  
}

– – – – – – – – – – – – – – – – – –

rule stateDouble {  
  - the name of the second principal rule in the second step of transformation that  
  replace two similar separated States by only one who will have all their transitions  
from  
  S : Stat!State ((S.msgFinal = true and S.msgInitial = false and not (S.estILeCas() and not  
S.existeSuivant(S.nameOrder))))  
to  
  R : realStat!State (  
nameOrder <- S.nameOrder,  
msgInitial <- false,  
msgFinal <- false,  
beginLoop <- S.getBeginLoop(S.nameOrder),  
endLoop <- S.endLoop  
)  
}

Rule 6

rule stateLoop {  
  - the name of the principal rule in the third step of transformation that adds an  
ε-Transition, to allow loops in the States that correspond to the Iteration Messages  
from  
  S : RealStat!State (S.endLoop > 0)  
  - A condition that checks ’returns True’ if the current State  
  represents the last State of an Iteration  
to  
  R1 : Automate!State (  
nameOrder <- S.nameOrder,  
initialState <- S.msgInitial,  
finalState <- S.msgFinal  
),  
R2 : Automate!Transition (  
  - Creating the ε-Transition  
symbol <- 'epsilon',  
source <- R1,  
  - the attribute source references the State where the Loop ends ’the state we have just  
  kept’  
target <- S.getLoopTransitions(S.endLoop)  
  - the attribute target references the State where the Loop  
begins  
)  
}

5: A Case Study: A Cellular Phone

To validate the proposed transformation rules we plan to apply them to several case  
studies, so that we can also evaluate their practical usefulness. Currently we consider the  
software that controls a very simple Cellular Phone System [15] as a case study. Such a
phone has Buttons for dialing digits, and a Send button for initiating a call. It has Dialer hardware and software that gathers the digits to be dialed and emits the appropriate tones. It has a CellularRadio that deals with the connection to the cellular network. It has a microphone, a Speaker, and a Display. How does the Cellular Phone work? To keep things simple, let us just look at how a customer might make a phone call. The use case for this interaction or communication looks like this:

1. User presses the digit Buttons to enter the phone number.
2. For each digit, the Display is updated to add the digit to the phone number.
3. For each digit, the Dialer generates the corresponding tone and emits it from the Speaker.
4. User presses Send.
5. The in use indicator is illuminated on the Display.
6. The CellularRadio establishes a connection to the network.
7. The accumulated digits are sent to the network.
8. The connection is made to the called party.

There are a variety of ways that this can be accomplished [15]; but they can all be simplified to having a Button object that sends a Digit message. The Dialer object receives the Digit message. The Dialer must then tell the Display to show the new digit, and must tell the Speaker to emit the appropriate tone. The Dialer must also remember the digits in the list that accumulates the phone number. Each new Button press follows the same procedure until the Send button is pressed. When the Send button is pressed, the appropriate Button object sends the Send message to the Dialer. The Dialer then sends a Connect message to the CellularRadio and passes along the accumulated phone number. The CellularRadio then tells the Display to illuminate the In Use indicator, the CellularRadio still resending the InUse message until the caller hangs up.

This simple procedure is depicted in the communication diagram in the well-known concrete syntax in Figure 6, while Figure 7 shows a possible abstract syntax of the same diagram according to the metamodel we have defined above.

The Cellular Phone model after applying the three steps of the transformation is seen in Figure 8. We have now reached a Büchi automaton corresponding to the Cellular Phone. This model contains two loops. The first inside the state named “S1”, marked in orange and the second in the state named ”S13”, marked in blue. Since we have at least one loop in the final state ”S13”, we can conclude that this model represents well a Büchi automaton. This is illustrated in Figure 9.

6: Implementation

We have chosen Atlas Transformation Language (ATL) [13][12] under the Eclipse development platform [11] to express the transformation rules. ATL is a model transformation language that contains a mixture of declarative and imperative constructs. ATL is accompanied by a set of tools built on top of the Eclipse platform. According to the adopted transformation process, the implementation of this process requires the following steps:

1. The representation of the source metamodel described in UML 2 communication diagram in Ecore Diagram Tool which generates an Ecore file named CD_Metamodel.ecore
Figure 6. Communication diagram for the phone system in concrete syntax

Figure 7. Communication diagram for the phone system in abstract syntax
Figure 8. Büchi Automaton for the phone system

described in XMI language [18].

2. The representation of the target metamodel described in Büchi automaton in Ecore Diagram Tool which generates an Ecore file named BA_Metamodel.ecore described in XMI language.

3. The representation of a model instance, i.e. a communication diagram, of the source metamodel in Ecore file.

4. Applying the rules of model transformation specified in ATL language to the source model. This process generates an XMI file containing a Büchi automaton describing formally the behavior of the source communication diagram.

7: Conclusion and Perspectives

In this paper, we proposed a transformation from UML 2 communication diagrams into Büchi automata. A set of rules was defined to govern the transformation process. On the basis of this transformation it is possible to accomplish verification of the dynamic model of the real system expressed by a communication diagram. Our approach was implemented using the ATL language. A Cellular Phone case study was used to illustrate the transformation technique.

This work still in progress so we plan to complete it further. First, one direction for future work can be to extend our methodology so that it supports asynchronous communication and conditional messages. Second, we need to better tune the rules, to realize if they can be
automated [1]. Third, is to generate Java code automatically from UML 2 communication diagrams [23].

Finally, we plan to integrate timing constraints with an aim of taking into account the asynchronous communications with propagation delay. Then it will be suitable to use the timed Büchi automata model.

References


Authors

Dr Elkamel Merah is an Associate Professor. He is with the Department of Computer Science, Faculty of Sciences and Technology, University of Khenchela, Algeria. He received an Engineer degree in Computer Science in 1986 from the University of Constantine. He received a "Magistere" degree in Computer Science in 1990 from the "Haut Commissariat à la Recherche", Algiers and his PhD degree in 1999 from the University of Constantine. His research interests are model driven architecture, model transformations, UML modeling, UML based code generation and mobile code.

Nabil Messaoudi is an Associate Lecturer. He is with the Department of Computer Science, Faculty of Sciences and Technology, University of Khenchela, Algeria. He received an Engineer degree in Computer Science in 1992 from the University of Annaba. He received his "Magistere" degree in Computer Science in 2009 from the University of Khenchela. His research interests are model driven architecture, model transformations, UML modeling, UML based code generation.

Halima Saidi completed his Bachelors and Masters degree in 2012 in Computer Science from Khenchela University, Algeria. This publication is based on his work in Master Thesis. His research interests are model driven architecture and model transformations.

Dr Allaoua Chaoui is a Professor. He is with the Department of Computer Science, Faculty of Engineering, University Mentouri Constantine, Algeria. He received his "Magistere" degree in Computer Science in 1992 and his PhD degree in 1998 from the University of Constantine. His field of research is formal specification and verification of distributed systems, model driven architecture, model transformations, UML modeling, UML based code generation.