Visual Decomposition of UML 2.0 Interactions

Abdelkrim Amirat and Ahcen Menasria
Department of Computer Science, University of Souk-Ahras, Algeria

Abstract: *Interaction Fragment model (IF) is a specific notion added in Unified Modeling Language (UML) 2.0 superstructures. Using the graphical notation, it can be used to represent the behavioral aspect of a system in a given scenario. Transforming such models, at early stages, requires the identification of elementary elements and their chronology. In this paper, we propose a visual and intuitive solution to identify and isolate each of which of graphical components while preserving the initial control flow. To that end, we suggest a reusable graph grammar to establish and update the control flow leading to a decomposed interaction. Our proposal can be used as first step to each transformation process whose having an UML 2.0 interaction as a source model.*

Keywords: *UML 2.0 interaction, control flow, graph transformation, AToM^3.*

Received September 4, 2013; accepted March 27, 2014; published online June 11, 2015

1. Introduction

The Unified Modeling Language (UML) is a graphical modeling language for object-oriented software and systems. It has been specifically designed for visualizing, specifying, constructing and documenting several aspects of or views on systems. Different diagrams are used for the description of the different views.

To model the behavior of a system, a modeler must use the interaction diagrams. Except state machine diagram which models the internal behavior of an object at run-time, sequence, communication, timing and global interaction overview are diagrams that capture the interaction between objects or participants involved in a given situation. Sequence Diagram (SD), also called language of scenarios, focuses on the chronological order of messages exchanged.

The Interaction Fragment (IF) is a specific notion for UML 2.0 with new graphical notation and semantics [15]. Defining its concrete syntax, this new notation is essentially borrowed from Message Sequence Chart (MSC) [17] to change the graphical notations of UML 1.x SD to support more complex logical structures such as concurrency, branching, iteration and their hierarchical composition in a compact and concise manner via Combined Fragments (CF) which are also responsible for the change of control flow.

Using CF and their hierarchical constructs lead to complex models. To verify such models at early stages, one has to translate them in a formal model. To do this, we must first identify and isolate their elementary components with preserving the initial control flow. This situation is a hard task when we must deal with the graphical model level regardless the internal representation.

On the other hand, graph transformation [4, 10] approach is a thoroughly studied area with many potential applications domain. The main idea of graph transformation consists in the local manipulation of graphs via the application of a rule. There exist various tools that implement graph transformation and may be used to specify visual languages or to generate diagram editors. Since, UML SD can be represented as graphs in a straightforward way, graph transformation rules can be employed to visually specify transformations between models.

In this paper, we propose an approach of model transformation based on graph rewriting which translate a SD in the rough with an arbitrary level of imbrications of structures as source model to the same but decomposed model as target model. We consider our approach as a simple and visual alternative to the "first step" of the unwinding algorithm based on locations and horizontal cuts proposed by Brill *et al.* [6, 19] for Live Sequence Chart (LSC) [7] whose MSC is the ancestry and adopted by Knapp and Wuttke [20] for UML 2.0 interactions.

We propose a meta-model, a visual environment for modeling an infinity of valid UML 2.0 SDs and a reusable graph grammar for their decomposition. This work is the central part of our global project for formal verification of UML's models by model-checking techniques [5], where we translate SDs into interaction automata and PROMELA code [2, 3]. We use the same grammar, as startup, for both cases.

In this research paper, we focus on SD's syntactical aspect and visual appearance, for different semantics [22] where many meanings of UML 2.0 interaction are summarized.

The remainder of this paper is structured as follows: In section 2, we review the basic concepts of UML 2.0 SD. A brief presentation of model and graph transformation is introduced. In sections 3 and 4 we describe our approach for SDs decomposition. Section 5 reports on the results on an anonymous example. Section 6 concludes with an outlook on our future work.

2. Basic Concepts

SDs has been popular ever since Jacobson *et al.* [18] introduced them as a means of documenting behavior within use cases. Because of their practical ability to show what is happening in a use case, SDs are popular with both business analysts and system designers. Rumbaugh *et al.* [27] describe a SD as a two-dimensional chart. The vertical dimension is the time axis, which runs from the top to the bottom of the diagram. The horizontal dimension shows the classifier roles that represent the individual objects collaborating with each other by sending messages. The SD thus, shows the interactions among the collaborating objects between two given points in time.

Object-Modeling Language (OMG) [24] the OMG adopts the MSC's visual notation in UML 2.0 superstructures to introduce structured control constructs into SD holding more complex structures such as concurrency, branching, iterations and imbrications in a simple and compact manner. Hence, the concept of IF emerges. Each IF alone is a partial view of the system behavior but when combined all together by means of the new Interaction Operators (IO), interactions provide relatively a whole system description.

An IF consists of a Basic Interaction (BI) and optionally, one or more CF. We refers to a SD without CF as a BI. A CF defines the structural articulations and comprises an IO defining the meaning of the particular fragment, and one or more IO. The IO themselves are IFs; they can be guarded by an optional condition or Interaction Constraints (IC), limiting the possibilities for when this operand may be executed. The unary operators are option, loop, break and neg. The others have more than one operand, such as alt, par, strict and seq. An IF can refer another one by an Interaction Use (IU) using keyword Ref.

A Life Line (LL) is a vertical line representing participating (objects, components, actor, etc.,) involved. A horizontal line between lifelines is a Message, which has a name. Each message is sent from its source lifeline to its target lifeline and has two endpoints. Each endpoint is an intersection with a lifeline and is called an Occurrence Specification (OS), denoting a sending or receiving Observable Event (OE). OSs can also be the beginning or end of an Execution Specification (ES), indicating the execution of a unit of behavior within a LL.

To describe the syntax or visual appearance, we refer to Figure 1 which shows an example with annotated syntactic constructs. It contains an interaction with three participants (A, B and C), exchanging five messages (m1, m2, m3, m4, m5, and m6) with an enclosing CF (Strict) and three IO. The second operand contains a nested CF (Option) which has one operand. On Strict CF, operands are executed in the graphical order. That's not the case for other CFs.

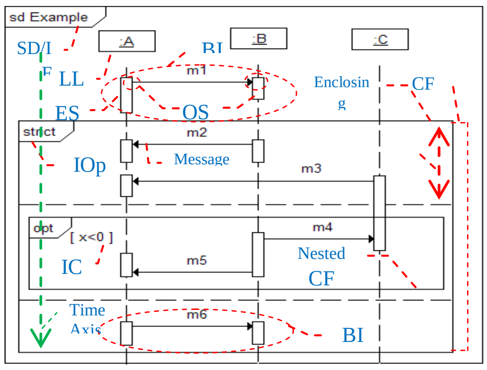

Figure 1. UML 2.0 SD.

There are two major difference between UML 1.x and UML 2.0 SDs: Syntactic, due the introduction of CFs for articulation constructs; and semantic, while UML 1.x's SD focuses on messages exchanging; UML 2.0's SD focuses on events generated from that exchange (Send/Receive). Additionally, CFs; therefore, they are responsible for changing the control flow in an execution trace. This generates the unobservable events that are the intersection between the CF and the incoming control flow [26].

Finally, interactions in SD are considered as collections of events instead of ordered collections of messages [14] which are based on traces semantics. These stimuli are partially ordered based on which execution thread LL they belong to. The partial order is defined by three conditions: Events on same LL are totally ordered, receiving event must appear after sending event for the same message and events on different LLs are concurrent and come in parallel or in an arbitrary order (interleaved).

There are other structured control constructs that have been introduced to express various control flows. General Ordering (GO) imposes (graphically) a binary relation to restrict the order between two OSs on different lifelines.

3. Model and Graph Transformation

Modeling enhances quality because it enhances communication. Through modeling, communication becomes efficient and effective. This is so because modeling raises abstraction to a level where only the core essentials matter. The resultant advantage is twofold: Easier understanding of the reality that exists and efficient creation of a new reality.

The trend among developers is to raise the abstraction level requiring programming technologies to improve continuously. From bit, assembler, procedural, object-oriented, component programming languages and finally models technology, a considerable experience is gained to reach the perspective of non-programming where, by non-programming, we mean mainly modeling. Hence, code is no longer a central element but derived or generated one.

Models technology or Model Driven Engineering (MDE) [11] is based on models and model transformation along life cycle of software production where many approaches have been adopted: Structural, relational, graphical and hybrid ones [25]. Graphical approaches are simply based on graph transformation techniques.

Basically, graph transformation is suitable for model transformation because of the following characteristics:

- Natural: Most of models are considered as graphs.
- Visual: Based on visual graphical notations.
- Declarative: Focus on “what to do” and not “How to do”.
- Intuitive: No standard methods to follow.
- High-level: deal with the model level.
- Formal: Mathematical foundations based on set and sets category theory.
- Supported by tools such as: AToM^3, PROGRES, GreAT, FUJABA and AGG [1, 8, 12, 13, 23].

3.1. Graph Transformation

In this section we present a short overview of the graph transformation approach, which is considered as the theoretical foundation of the proposed approach in section 4.

Graph grammars extend the generative grammars of Chomsky into the domain of graphs to deal with non linear structures. Different from string grammar expressing sentences in sequence of characters, graph grammars are suitable for specifying visual modeling in multi-dimensional fashions. The main idea of graph transformation is the rule based modification of graphs (rewriting or gluing) as shown in Figure 2.

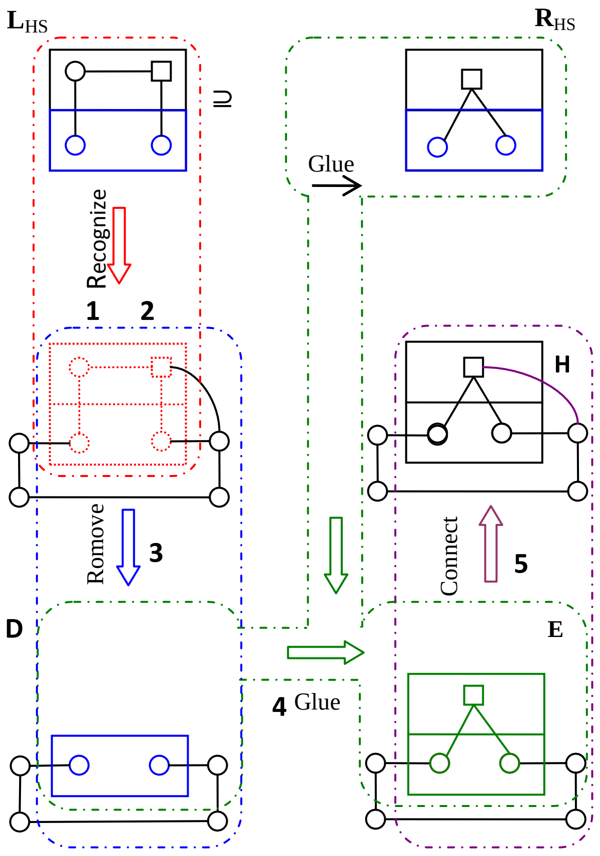

Figure 2. Graph transformation steps.

The core of a rule (production) p= (L, K, R) is pair of graphs (L, R) called Left Hand Side (LHS) and Right Hand Side (RHS) and a gluing or interface graph K expressing the shared part between L and R. Applying the rule p on a graph instance (called host graph) means to find a match of L in the host graph and to replace L by R leading to the target graph of the transformation. Any transformation of graphs can be realized by applying a sequence of production rules. Moreover, the transformation process termites when no more transformation rules can be applied. Briefly, graph transformation is a kind of programming by example [16].

SD has a well-known graphical form which is a graph (nodes, edges), thus it is natural to use graphical notation to depict and transform them through graph transformation. The transformation process produces a new graph from the input host graph after applying transformation rules. In our case, the host graph is an UML 2.0’s SD; the output graph is an UML 2.0’s SD adorned with control flow.

To illustrate graph transformation process, let L be the LHS of a grammar rule p, R be the RHS of the rule, and K the interface graph. Let G be a host graph. The transformation from graph G to graph H by rule p can be achieved through the following steps [4]:

1. Recognize sub-graph L in the host graph G.
2. Check if the transformation rule can be applied.
3. Remove the occurrence of L up to the occurrence of K from G. This yields the context graph D of L which still contains an occurrence of K.
4. Glue the graph D and R according to the occurrences of K in D and R. This yields the gluing graph E.
5. Connect the dangling edges on E if the node is marked to preserve the association surrounding of the replaced sub-graph L. This yields the modified graph H.

Steps 1 and 2 can be classified as a pattern matching process while steps 3, 4 and 5 are used to update the surrounding connections with the host graph. Figure 2 shows our vision of graph transformation process which illustrates, visually, the steps above where K is the graph in blue. The application of p to G yielding H is called a direct derivation from G to H through p.

3.2. AToM^3 Tool

AToM^3 is a tool for meta-modeling and multi-formalism, it can generate Domain Specific Visual Language (DSVL) [28, 29] for one or more desired formalisms to instantiate an infinity of valid models that conform to their meta-models. It can also perform models transformation by means of graph grammar using Push out approaches [9].

To define meta-models, the tool uses a meta-formalism in UML’s class diagram notation to describe concrete/abstract classes, associations, multiplicities, specialization and additional constraints described with Python code [21]. Also, for each concrete class, AToM^3 can associate a visual appearance to define the concrete syntax of the modeled domain. For each graph grammar, rules are modeled in concrete syntax with their execution priority and textual pre/post conditions that must be satisfied before application and finally an action to be performed when rule is executed. To execute one or more ordered graph grammars, in the same session, the graph rewriting system loops on rules until no one is applicable.

In addition, AToM^3 incorporates in its Kernel a generic meta-formalism called generic graph composed of two elements: Generic graph node and generic graph edge to make links, during transformation process, between models under transformation. We intensively use both elements in this paper to materialize graphically our control flow description.

Our choice of AToM^3 tool is motivated by the following reasons: AToM^3 is one of the few graph transformation tools using concrete syntax that users are familiar with, its looping execution mode is useful to deal with nested structures which are our case and its dedicated methods to perform hierarchical relationships (Father/Child). These reasons seem to be the major advantages of AToM^3 compared to other tools such as AGG [1] and others.

4. Our Approach

The proposed approach is based on the manipulation of control flow in the SD. The first step of the idea is to make appear explicitly (draw) control flow which is usually implicit between OEs (intersection between messages and lifelines) (CF. Section 2) as they appear in their chronologic order on the vertical axis without considering CF neither the interleaving of events. In other words, the control flow will play the role of GO on all events to define one execution trace. The second step is to update the control flow according to the apparition of the CF-which are responsible for changing the control flow by definition-to capture unobservable events (intersection between control flows, established in the first step and the CF). Due to the rectangular shape of the IO defined in section 4.1, we extend the same notion of unobservable events for them; hence, we update also the control flow according to the IO. The control flow in BI is kept unchanged.

Using model transformation taxonomy, we can say that we propose a meta-modeling approach which is a model to model transformation, endogenous, in place, horizontal and (1×1) cardinality in a declarative manner using graph transformation techniques and AToM^3 tool as shown below.

4.1. SD Meta-Model Description

In order to perform transformation of SD as source model into a decomposed equivalent one as target model, we have to propose meta-models to define the abstract and concrete syntax using UML class diagram notation for both. However, our source and target models are the same (endogenous) unless the presence of control flows on the target one. That's why we have to propose just one meta-model for SD. We use the elements for generic graph (section 3.2) to draw the control flow.

The proposed meta-model as shown in Figure 3 consists of five concrete classes, two visible relationships and four invisible and hierarchical relationships to represent the most useful aspects of this study. The entities whose icons have a hexagonal shape at the top are generated as relationships/edges. AToM^3 is designed to keep track of such hierarchical relationships, so finding parents and children is easy via dedicated methods.

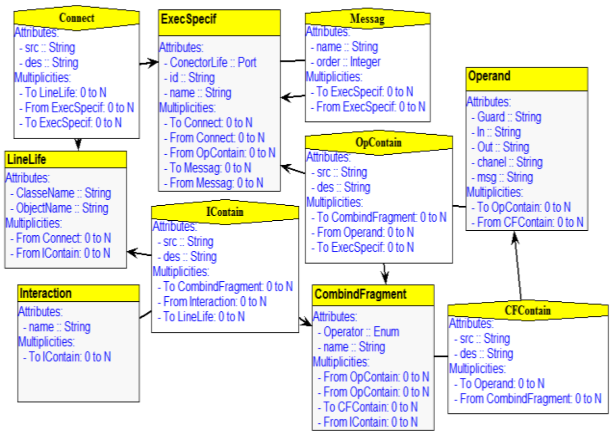

Figure 3. SD meta-model.

- *Class Interaction*: Is a representation of the entire model. All other entities will be contained by this entity.
- *Class LL*: Represents an individual participant in the interaction. A lifeline has two attributes which represent the name of the participant class (instanceName) and the name of an instance of that class (className) respectively.
- *Class ES*: Is used to hold the OEs at ends and beginnings of Messages (intersection between messages and Lifelines). It denotes the atomic actions for sending and receiving messages with the restriction of having a single point of connection on each side. The vertical order of each ES is significant chronologically according to its apparition along the same LL. This is the central class in the sense that it captures the elements of interest in this project (events) and it is the basic entity in any interaction diagram. Hence, it has relations with all other entities of the meta-model, among which two visible relationships: "Message" and "Connect" that bind together two ES in the horizontal and vertical direction respectively.
- *Class CF*: At this stage we introduce important and distinctive element for UML 2.0 interactions: CF to form the CF and hierarchies. This class has two attributes, its "Name" and "Operator" to designate

IO (enumerated type of 12 items). A CF can contain one or more IO by the invisible relationship CFContain according to the IO involved.
- *Class IO*: This class defines the content of a CF: IO as a nested element with the relationship CFContain. It has a "Guard" as a logical expression conditioning its execution C and a "Name" attributes to identify it. This class has also, invisible and hierarchical relationships with the classes CF and ES by the relation IOContain.

The visible relationships "Message" (with attribute name) and Connect (ES to LL) appear with their graphical forms in models. The hierarchical ones (IContain, CFContain, IOContain, LLContain) are invisible because they do not belong to UML's standard elements; they just have countenance effects in models.

Relationships CFContain and IOContain allow combinations or imbrications of CF so to define the hierarchy (Father/Child). A CF must contain at least one interaction operand. An IO can enclose a BI or/and one or more CF and vice versa. Hence, we can construct an arbitrary number of hierarchy layers.

Figure 4 shows the concrete syntax chosen for each visible element in the meta-model. All are conforming to UML standard, unless the IO where the dashed lines in Figure 1 is replaced by a rectangular form enclosing the whole interaction. This form is easy to match during the graph rewriting process.

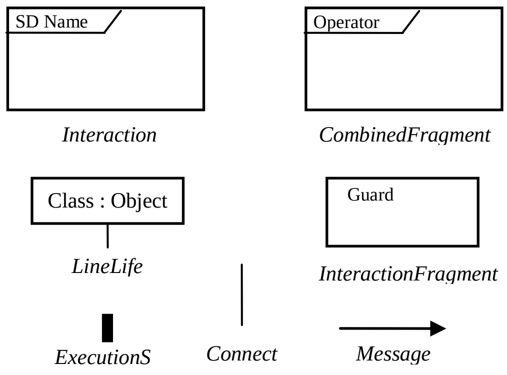

Figure 4. SD concrete syntax.

After meta-modeling phase, the DSVL for SD is instantaneously generated by the tool. Hence, we can instantiate an infinity of valid models expressed in concrete syntax.

4.2. Graph Grammar

Our graph grammar consists of fourteen rules where both LHS and RHS are graphs, centered on establishing and updating control flow. The updating of control flow does not concern all BIs in the model. Using priority mechanism, rules are designed in the five following categories:

1. The first category of rules deals with BI and consists of two rules to establish main control flow independently of CF and interleaving concept. This category captures the OEs.
2. Second category updates the CF's incoming control flow from outside ES. Composed of tree rules that returns the control flow to outside, with capturing unobservable events from IO to the enclosing CF and from CF to the enclosing IO and vice versa. This category starts on the deepest CF.
3. Third category is the same as the second one, but it deals with outgoing control flow from CF to outside ES.
4. Fourth category deals with CFs of the same imbrications level and contains three rules.
5. Fifth category deals with IOs of the same imbrications level and contains also three rules.

Table 1 summarizes the entire graph grammar proposed with a short description for each rule. Due to space constraints we present only the most important ones namely 1, 3 and 6.

Table 1. Graph grammar rules.

Cat	Name	Priority	Description
1	MainCtrFlow	1	Establish the main control flow
	TmpFLastES	2	Generalization of IFs
2	CFFromOutES	3	Link outside (top) ES to the deepest CF and IO
	CFIOFromOutES	4	Link CF to its enclosing IO
	CFFromOutES_2	5	Link CF to the nested IO(different layer)
3	CFForOutES	6	Link outside (bottom) ES to the deepest CF
	CFIOForOutES	7	Link CF (bottom) to its enclosing IO
	CFForOutES_2	8	Link CF (bottom) to the nested IO(different layer)
4	CFForCFInsideES	9	Link same level CFs (ES inside)
	CFForInsideES	10	Link CF to inside ES (with operand)
	CFForIOInsidES	11	Link CF to Enclosing IO
5	IOInESForIOInES	12	Link two contingent IO (ES inside)
	IOInESFromIO	13	Link two contingent IO (ES inside the first IO)
	IOForIOInES	14	Link two contingent IO (ES inside the second IO)

- *Rule* 1. MainCtrFlow as shown in Figure 5: Which is the central rule to establish the main control flow according the graphical position of OEs from up to down (time axis) and the direction of the message? The elements 10, 11 and 12 materialize the control flow using generic graph edge and generic graph node (section 3.2). As mentioned before, there is no CF in the rule. Also, the interleaving of events is delayed according to the semantic and syntax of the target model. Additionally, this rule has to determinate the first and the last event of the IF as a final action. It considers the entire IF as a BI with a GO on all events.

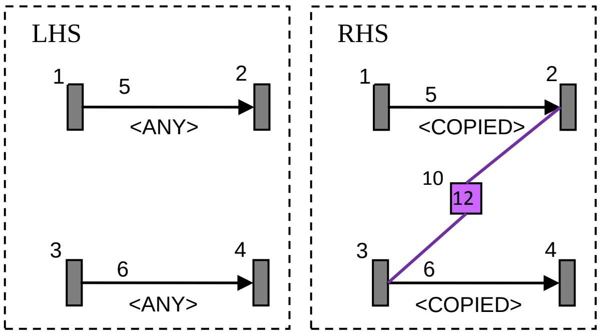

Figure 5. MainCtrFlow rule.

- *Rule 3.* CFFromOutES as shown in Figure 6: Updates the incoming control flow from an outside

OE to the deepest CF. In fact, the current CF and IO can themselves be enclosed in an enclosing IO; that justify the usage of two other rules in same category. As pre-condition, the OE must be outside of the current CF.

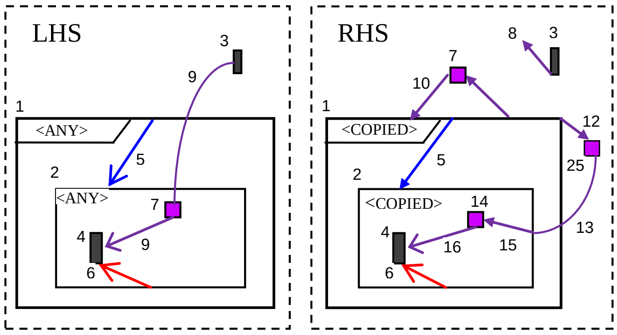

Figure 6. CFFromOutES rule.

- *Rule 6*. CFForOutES: Follows the same logic of rule 3 except that it acts on the CF's outgoing control Flow. As pre-condition, the OE must be outside (bottom) of the current CF. the green dashed circles on Figure 7 show the capture of non- OEs.

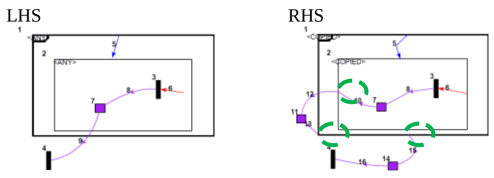

Figure 7. CFForOutES rule.

5. Example

We have tested our approach on numerous academic examples leading to correct results. To illustrate the concepts and benefits of our approach, let's consider the representative model depicted previously in Figure 1.

We use our DSVL for SD to model the scenarios as source model. After application of our graph grammar we obtain the result or (target model) shown in Figure 8.

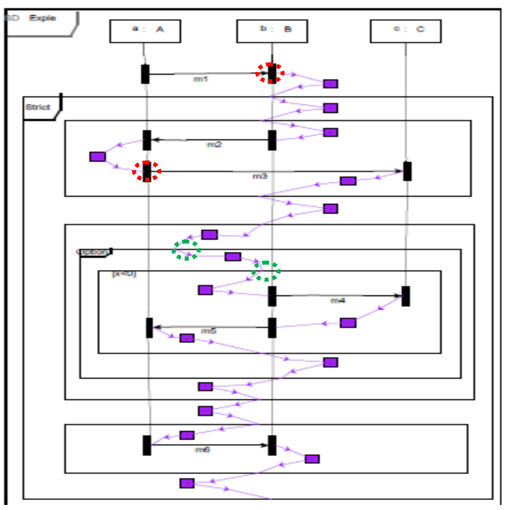

Figure 8. Decomposed SD corresponding to the Figure 1.

All initial elements and their structure are not changed. The control flow established in purple color show the single execution trace between observable and unobservable events from up to bottom updated according to CFs. OEs are shown in dashed red circle witch concern all endpoints of messages. Green dashed circles show the capture of all unobservable events on CFs and IOs of the interaction.

At this stage, one who wants to transform UML 2.0 SD to any other model must concentrate only on elementary elements and their mapping in syntax and semantic of that desired target model without matter about the content and the complexity of an arbitrary graphical representation of the initial interaction body. Hens, our obtained diagram will play the role of source model for further transformation.

6. Conclusions and Future Work

We have presented a reusable, visual, declarative, high-level and formal approach to decompose any UML 2.0 SD to an adorned one with control flow information. We proposed a meta-model, a DSVL and a graph grammar using AToM^3 graph transformation tool to identify and isolate elementary elements of an arbitrary SD while preserving the initial control flow and execution trace.

Our contribution is not an end in itself but a mandatory startup step for model transformations whose have SD as source model. This proposal (control flow approach) is based on materialization and manipulation of control flow which can be seen as a simple and visual alternative to the unwinding algorithm. We have experienced this graph grammar to produce the PROMELA code and interaction automata corresponding to any SD

We intend to integrate the remaining features specified by the UML 2.0 specification, such IU and gates. Also, we plan to extend our approach to deal with global interaction overview diagram that we can construct more complex interaction scenarios.

References

[1] AGG home page., available at: http://tfs.cs.tu-berlin.de/agg/, last visited 2014.

[2] AitOubelli M., Younsi N., Amirat A., and Menasria A., "From UML 2.0 Sequence Diagrams to PROMELA Code by Graph Transformation using AToM^3," available at: http://ceur-ws.org/Vol-825/paper_183.pdf, last visited Algeria 2011.

[3] Amirat A., Menasria A., Ait Oubelli M., and Younsi N., "Automatic Generation of PROMELA Code from Sequence Diagram with Imbricate Combined Fragments," *in Proceedings of the 2nd International Conference on Innovative Computing Technology*, Casablanca, Morocco, pp. 111-116, 2012.

[4] Andries M., Engels G., Habel A., Hoffmann B., Kreowski H., Kuske S., Plump D., Schürr A., and Taentzer G., "Graph Transformation for Specification and Programming," *Science of Computer Programming*, vol. 34, no. 1, pp. 1-54, 1999.

[5] Baier C. and Katoen J., *Principles of Model Checking*, MIT Press, UK, 2008.

[6] Brill M., Damm W., Klose J., Westphal B., and Wittke H., "Live Sequence Charts," *Integration of Software Specification Techniques for Applications in Engineering*, pp. 374-399, 2004.

[7] Damm W. and Harel D., "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45-80, 2001.

[8] DeLara J. and Vangheluwe H., "AToM3: A Tool for Multi-Formalism and Meta-Modelling," *in* proceedings of the 5th International Conference, FASE 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, France, pp. 174-188, 2002.

[9] Ehrig H., Ehrig K., Prange U., and Taentzer G., *Fundamentals of Algebraic Graph Transformation*, Heidelberg: Springer, 2006.

[10] Ehrig H., Engels G., and Rozenberg G., *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, world Scientific, 1999.

[11] France R. and Rumpe B., "Model-Driven Development of Complex Software: A Research Roadmap," *in Proceedings of Future of Software Engineering*, MN, USA, pp. 37-54, 2007.

[12] FUJABA Home Page., available at: http://www.fujaba.de, last visited 2012.

[13] GreAT Home Page., available at: http://www.escherinstitute.org/Plone/tools/, last visited 2004.

[14] Hammal Y., "Branching Time Semantics for UML 2.0 Sequence Diagrams," *in Proceedings of the 26th International Conference IFIP WG 6.1*, Paris, France, pp. 259-274, 2006.

[15] Haugen Ø., "Comparing UML 2.0 Interactions and MSC-2000," *in Proceedings of the 4th International SDL and MSC Workshop*, Ottawa, Canada, pp. 65-79, 2005.

[16] Heckel R., "Graph Transformation in a Nutshell," *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp.187-198, 2006.

[17] ITU-T.Z.120., available at: http://www.itu.int/ITU-T/studygroups/com10/ languages/Z.120_1199.pdf, last visited 1999.

[18] Jacobson I., Christerson M., and Övergaard G., *Object-Oriented Software Engineering, a Use Case Driven Approach*, Pearson Education India, 1992.

[19] Klose J. and Wittke H., "An Automata Based Interpretation of Live Sequence Charts," *in Proceedings of the 7th International Conference TACAS*, Genova, Italy, pp. 512-527, 2001.

[20] Knapp A. and Wuttke J., "Model checking of UML 2.0 interactions," *in Proceedings of Workshops and Symposia at MoDEL*, Genoa, Italy, pp. 42-51, 2007.

[21] Lutz M., *Programming Python*, O'Reilly Media, Inc., 2010.

[22] Micskei Z. and Waeselynck H., "The many meanings of UML 2 Sequence Diagrams: a Survey," *Software and Systems Modeling*, vol. 10, no. 4, pp. 489-514, 2011.

[23] Nagl M., Schurr A., and Munch M., "Applications of Graph Transformations with Industrial Relevance," *in Proceedings of International Workshop, AGTIVE'99 Kerkrade*, Netherlands, 2000.

[24] OMG., *Unified Modeling Language: Super-Structure 2.0*, 2003.

[25] Prakash N., Srivastava S., and Sabharwal S., "The Classification Framework for Model Transformation," *Journal of Computer Science*, vol. 2, no. 2, pp. 166-170, 2006.

[26] Rajabi B. and Lee S., "Consistent Integration between Object Oriented and Coloured Petri Nets Models," *the International Arab Journal of Information Technology*, vol. 11, no. 4, pp. 406-415, 2014.

[27] Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[28] Sprinkle J. and Karsai G, "A Domain-Specific Visual Language for Domain Model Evolution," *Journal of Visual Languages and Computing*, vol. 15, no. 3, pp. 291-307, 2004.

[29] Vangheluwe H. and de Lara J., "Computer Automated Multi-Paradigm Modelling for Analysis and Design of Traffic Networks," *in Proceedings of the 36th Conference on Winter Simulation*, pp. 249-258, 2004.

Abdelkrim Amirat received his PhD degree in computer science in 2007 and University Habilitation in 2010. Currently, he is a Professor at Computer Science department at the University of Souk-Ahras, Algeria. He is the Director of Mathematics and Computer Science Laboratory and the chief of the software architecture modelling team. His main research concerns is software architectures and their evolution, modelling and metamodeling. He worked on several national projects in software engeneering. Amirat has published several refereed journal and conference papers in the fields of software architecture, component-based and object oriented modelling. He has served on program committees of several international journals and conferences.

Ahcen Menasria is Assistant professor in the department of computer science, University of Souk-Ahras, Algeria. His research field is object-oriented, component, formal methods and modeling approaches.