IET Software

Research Article

Domain-specific language for automatic generation of UML models

ISSN 1751-8806
Received on 4th November 2016
Revised 5th September 2017
Accepted on 4th October 2017
E-First on 14th December 2017
doi: 10.1049/iet-sen.2016.0279
www.ietdl.org

Márcio Assis Miranda[1,2] ✉, Marcos Guilherme Ribeiro[3], Humberto Torres Marques-Neto[1], Mark Alan Junho Song[1]

[1]Department of Computer Science, Pontifical Catholic University of Minas Gerais, Belo Horizonte, MG, Brazil
[2]Department of Computer Science, Federal Institute of Minas Gerais, Ouro Branco, MG, Brazil
[3]Department of Computer Engineering, Federal Center of Technological Education of Minas Gerais, Timóteo, MG, Brazil
✉ *E-mail: marcio.assis@ifmg.edu.br*

Abstract: The majority of flaws found in software originates in the system requirements specification stage. The use of domain-specific languages has shown to be a valuable resource in this part of the process, since they help to establish communication standards, enable automation and bring productivity and quality gains, in spite of their limited vocabulary. This study proposes the implementation of *language of use case to automate models* (LUCAM), a domain-specific language that allows specification of textual use cases and semi-automated generation of use case diagrams, class diagrams and sequence diagrams through LUCAMTool. To verify the feasibility of the proposed solution, tests were performed in both simulated and real environments so as to comprise a variety of scenarios observed in systems development. The approach assists in the requirement analysis and modelling, minimising existing problems in natural language specification, such as the dependence on the knowledge of specialists, uncertainty, ambiguity and complexity.

1 Introduction

Constructing a high-quality software requirements specification is a key to a project's success. In addition to contributing to the quality of the software artefacts generated, the software specification facilitates effective communication between the client and the development team. Therefore, various methods have been developed, such as user stories, use case models, diagrams and formal language.

Most textual requirements are written in natural language because it can be easily understood by both the client and the developers. However, redundant, ambiguous, inconsistent, or difficult-to-interpret specifications can compromise the productivity of the developers and the quality of the final product. To minimise these problems, we need methods and tools that increase the formality while preserving as much of the understandability of natural language descriptions as possible.

According to [1], there is considerable distance between requirements specification and software development. Most development teams still rely upon manual processes and specialised expertise to generate a system model from the requirements specification. From [2], requirements should be presented to project stakeholders in an understandable way, offering a certain level of automation in part of the process.

Some advocate domain-specific languages (DSLs) as a viable solution to this problem because their use enhances productivity, promotes communication between engineers and domain specialists, and enables the removal of ambiguities and inconsistencies observed in the natural language texts [3]. The need for legibility, understandability, and productivity also justifies the use of DSLs to describe use cases [4, 5].

In this paper, we define an external DSL called *language of use cases to automate models* (LUCAM), which defines language standards to specify textual use cases. We also describe an implementation of LUCAMTool, a tool that automatically generates use case diagrams, class diagrams and sequence diagrams from use case specifications.

To demonstrate the LUCAM features and expressive capabilities, we apply the LUCAM approach to a banking software system, focusing on the '*SB_BankAccount*' use case. The bank manager, a primary actor, is responsible for managing user and accounts data. The customer, as a secondary actor, is responsible to provide personal data to the manager. Through the LUCAM approach, they can be registered in the system to make, among other things, financial transactions such as withdrawals and transfers.

Our tests encompass both real and simulated environments, with the former being a proof of concept trial in projects picked by collaborating companies. Our solution has features that contribute to the standardisation of important stages in software development, such as use case specification and modelling, bringing productivity gains and generating diagrams consistent with a specification and mitigating problems stemming from the use of natural language.

2 Proposed solution

We have proposed a DSL named *language of use case to automate models* (LUCAM), which contemplates textual use case detailing. Also, the use of the LUCAMTool enables the automatic generation of unified modelling language (UML) diagrams in requirement-oriented software development processes.

The LUCAMTool was implemented to enable users to specify functional software requirements using our LUCAM DSL – a parser maps relevant features of the textual use cases to UML use case, class and sequence diagrams. The LUCAMTool source code as well as the language grammar and all conducted tests are available at https://github.com/assismiranda/LUCAMTool.

The transition from use case specifications to models begins with the requirement analysis and discovery. After that, use cases are textually specified in the LUCAM language, each sentence is mapped by LUCAMTool's parser core and thus the elements necessary to generate the models are identified, as demonstrated by the process presented in Fig. 1.

The elements that compound use case specifications in LUCAM, the standards defined for each sentence in the specification, and the main stages in the mapping and artefact generation processes are presented below.

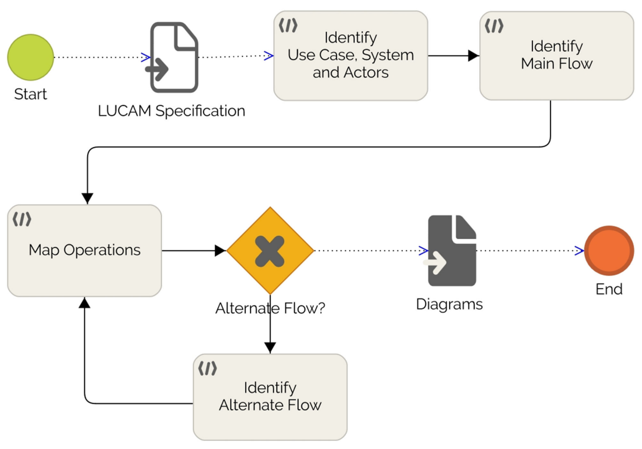


Fig. 1 *LUCAM general flow*

2.1 LUCAM specification

LUCAM use case specifications have the following attributes: use case name, brief use case description, system name, actors, basic and alternate flows, main scenarios, pre-conditions, post-conditions, special requirements and extension points.

We based the LUCAM templates on the templates in the *open unified process* (OpenUP) [6]. Since OpenUP uses free-form, natural language specifications that lack a precise syntax or semantics, it is difficult to automate the processing of use cases.

To achieve the purpose of LUCAM, we extend the OpenUP template in various ways. For example, we make the attributes and class methods explicit, identify the system with which the primary and secondary actors interact, define explicit control flow (*if, loop* and *concurrency*) for sequence diagrams and register the nouns and verbs before their use.

Use case details do not usually contain these elements, therefore, language expressive capabilities are increased by adding them, making language vocabulary richer and providing a bigger variety of written ways to specify the majority of scenarios present in software development. This formal syntax and semantics is also essential for the automatic generation of diagrams; it helps ensure consistency among the specification and generated models.

2.2 LUCAM grammar rules

In the first part of the grammar, we have the rules that compose the main structure (skeleton) of the use case detail. The DSL rules presented in this paper are in the extended Backus-Naur form (EBNF)

- *LUCAM :: = UseCaseHeader UseCaseFlows UseCaseFooter*
- *UseCaseHeader :: = UseCaseName UseCaseBriefDescription SystemName PrimarySecondaryActors*
- *UseCaseFlows :: = MainFlowName MainFlowScope [AlternateFlows]*
- *UseCaseFooter :: = Key Scenario {KeyScenario} PreConditions PosConditions SpecialRequirements ExtensionPoints*

Use case specifications in the LUCAM language have the following header elements: use case name, brief description, system name, primary actors and secondary actors, according to DSL rules presented below

- *UseCaseName :: = "**Use Case:** " ControlClassID POINT*
- *UseCaseBriefDescription :: = "**Brief Description**" [Text] POINT*
- *SystemName :: = "**System:** "SystemID POINT*
- *PrimarySecondaryActors :: = "**Primary and Secondary Actors** " PrimaryActorName [SecondaryActorName]*
- *PrimaryActorName :: = "**Primary Actors:** " ActorID {"," ActorID} POINT*
- *SecondaryActorName :: = '**Secondary Actors:** " ActorID {"," ActorID} POINT*

The control class names are identified from the use case names. The main goal of this class is to control the sequence of the use case behaviour. In other words, it represents the dynamics of a functional requirement. For each flow (main and alternatives), a control class method is generated, and for each control class method, a sequence diagram is generated automatically. Initially, the names of the primary actors, secondary actors and the system are also indentified.

The most important rules of LUCAM are presented below. Note that they have the main elements to represent the main flow of the use case specification

- *MainFlowName :: = "**Main Flow:** " MethodControlClassID POINT*
- *MainFlowScope :: = [ActorID " **starts Use Case**" POINT] MainFlow [ActorID "**finishes Use Case**" POINT]*
- *MainFlow :: = MainFlowElements {MainFlowElements}*
- *MainFlowElements :: = MainFlowCore | FlowIf | FlowLoop | FlowConcurrency*
- *MainFlowCore :: = ((ActorID | SystemID) (MainFlowTabTransVerb | **TABINTRANSVERB**) POINT) | ReturnMessage | InteractionUseCase POINT*
- *MainFlowTabTransVerb :: = **TABTRANSVERB** ["MethodBoundaryID"] (MainFlowAttibutes | "**on**" MainFlowBoundaryClass)*
- *MainFlowAttibutes :: = [["**the**"] **TABNOUN** ["(" [AttributeTypeID] AttributeID {, [AttributeTypeID] AttributeID}")"]] (("**on**" MainFlowBoundaryClass | ("**of**") MainFlowEntityClass)) | MainFlowsActorClass)*
- *MainFlowBoundaryClass :: = BoundaryClassID*
- *MainFlowActorClass :: = ("**for**" | "**to**") ["**the**"] ActorID*
- *MainFlowEntityClass :: = EntityClassID ["**on**" MainFlowBoundaryClass | "**by**" MainFlowCommunication | MainFlowsActorClass)]*
- *MainFlowCommunication :: = CommunicationID*
- *ReturnMessage :: = (SystemID **TABTRANSVERB** SimpleReturnMessage ("**to**" | "**for**") ["**the**"] ActorID | SystemID)*
- *FlowIf :: = "**If** " Condition MainFlow ["**Else** " MainFlow]"**EndIf**"*
- *FlowLoop :: = "**Loop** " Condition MainFlow "**EndLoop**"*
- *FlowConcurrency :: = "**StartConcurrency** " MainFlow "**concurrent**" MainFlow "**EndConcurrency**"*
- *InteractionUseCase :: = ActorID "**executes**" ("**use**" "**case**" | "**alternate flow**") "useCaseID" ("," | " **and**" | " **or**") "useCaseID"*

The part of the grammar that contemplates alternate flows is quite simple, because its main rule (*AlternateFlowCore*) derives from the most important main flow rule (*MainFlow*). This means that the same specification pattern defined for the basic flow is also accepted for alternate flows

- *AlternateFlows :: = "**Alternate Flows** " AlternateFlowscope { AlternateFlowscope}*
- *AlternateFlowscope :: = "**Alternate Flow** " NUM "**:** " MethodControlClassID POINT AlternateFlowCore*
- *AlternativeFlowCore :: = MainFlow*

In order to increase language expressiveness and fulfil requirements such as clarity, simplicity and interactivity, we created three tables of reference words previously recorded in the LUCAMTool. The term ***TABTRANSVERB*** refers to the set of transitive verbs, while ***TABINTRANSVERB*** refers to intransitive verbs. The third table is ***TABNOUN***, which contains a set of preregistered LUCAMTool nouns that make up the vocabulary of our proposed language.

2.3 Map operations

The *map operations* module is responsible for recognising sentences in the use case specification and validating them against defined patterns and grammar rules. The module contains the

LUCAMTool parser core, implemented by a class which contains two fundamental methods: *identifiesSentence()* and *identifiesMessage(Sentence)* (see Table 1).

Before describing these methods, we provide some examples of sentences written in LUCAM, which we use to demonstrate how to map and to generate the models. Each sentence follows the patterns presented in Table 2, in the same order.

1. ***Customer** informs the attributes to the **Manager**.*
2. ***Manager** selects "InsertAccount" on **MainForm**.*
3. ***Manager** enters attributes (ID, Name, DateBirth,...) of **Customer**.*
4. ***System** returns "Input mode screen" to **Manager**.*
5. ***System** sends the notification by **e-mail**.*
6. ***System searches** for the **Customer**.*
7. ***System returns** the attributes of **Customer**.*
8. ***System saves** the attributes of **Customer**.*
9. ***System validates** attributes of **Customer**.*
10. ***System verifies** the attribute (condition) of the **Customer**.*
11. ***System displays** the attributes of **Customer** on **MainForm**.*
12. ***System cancels** the operation.*

The first method is *identifiesSentence()*, which analyses every word in the sentences of the textual specification, identifying their grammatical class (subject, verb, method, noun, attribute, preposition, classes and actors) and returning an object with these terms. Table 1 presents an example of such object for the sentences presented above. Note that sentences are divided according to their terms, and object attributes are filled accordingly. After *identifiesSentence()* has mapped the words, the returned object is sent as a parameter to the *identifiesMessage(Sentence)* method. Based on the rules presented in Table 2, the method maps objects in the sequence diagram and returns another object containing a message label and origin and destination classes.

Rules 6 and 7 are identical, therefore, we have a type of pattern that generates two operations with two different behaviours. For example: when the user searches for something (verb *searches*), the message must be relayed from the *Boundary* class to the *Entity* class. The same process occurs when the system retrieves the data being searched: a message is sent from the *Entity* class and is replicated until it gets to the *Controller* class. In some cases, messages are originated from controller classes, which is why we defined the last receiver class (LRC) time – it is necessary to control, during the flow, in which class a method was last invoked.

Other class types must be identified in addition to the control class. We then need to define rules based on the prepositions that precede the name of each class in the sentence. Preposition ***Of*** precedes the names of classes of type *Entity*, ***On*** precedes the *Boundary* class, ***To/For*** precede the names of *Actors* and ***By*** precedes classes of type *Communication*, like the examples provided in Table 1.

Some verbs only make sense in the context of a phrase if they are related to the system. Verbs such as *validates* and *verifies* validate input data restrictions or business rules, and must have *self call* behaviours in sequence diagrams. Conversely, verbs such as (*searches* and *retrieves*) trigger a return message to the control class.

2.4 Diagram generation

This section uses examples to demonstrate the artefact generation process, taking into consideration the rules and patterns detailed in the previous section.

Fig. 2 shows the main steps executed by the LUCAMTool parser core to map elements present in the specification.

The algorithm takes a use case specification in LUCAM as input and returns the elements mapped into class and sequence diagrams. The *IdentifiesSentence()* and *IdentifiesMessage(Sentence)* methods are responsible for doing the mapping, as mentioned in Section 2.3. Elements are mapped with respect to the terms and behaviour of each sentence. Elements are stored in a data structure and relayed to a class that generates the diagram.

From the mapping, the parser stores in an intermediate structure all the information necessary (actors, methods, attributes, messages etc) to create the class and sequence diagrams. To generate the diagrams, based on the identified elements, the libraries *'astah-pro.jar and astah-api.jar'*, available in the Astah [7] tool, were used.

Suppose we apply the algorithm to the example use case '*SB_BankAccount*', where Figs. 3 and 4 give two fragments of the specification. The first shows the main flow *'AddAccount'*. The second shows an alternate flow specification *'CloseAccount'*. Figs. 5 and 6 show the outputs.

Fig. 5 depicts the class diagram generated from the elements of the textual specification. Note that the class diagram generated by the LUCAMTool accurately represents the provided specification. This is a very important feature of our proposed approach, because most manually written specifications contain some disparity between models and textual specification, thus propagating inconsistencies.

Fig. 6 depicts the sequence diagram generated from the elements of the textual specification and the class diagram. Note that through existing elements in the sequence diagram, we can clearly visualise the flow of the process described in the use case details, as well as identify mapped classes and methods, thus ensuring another level of consistency between the generated models. This diagram shows resources such as self calls, return messages, validations and combined fragments.

In addition to the features previously described, sequence diagrams generated by the LUCAMTool permit users to validate class diagrams because they have more details and, therefore, precisely reflecting characteristics and elements present in textual specifications.

3 Results and viability study

Tests were conducted in two stages. The first stage was a simulated environment, using classic scenarios (academic, banking, E-Commerce and medical) and the second stage used real software systems in three collaborating companies, as a proof of concept. The files containing our tests and respective results (specifications and models) are available at https://github.com/assismiranda/LUCAMTool/.

To conduct our proof of concept experiment, we selected three software systems whose original requirements had been specified in natural language and that had diagrams and source code manually written by specialists who have participated in the tests with LUCAM.

The first software is used to manage the concrete production and delivery in a mining company. The software has 115 features, 53 of which are of the type 'Manage Simple Entity' (which group subfeatures of type CRUD-Create, Read, Update and Delete actions), 25 are of the type 'Manage Composite Entity' (each of them grouping four to nine subfeatures of the type 'Manage Entity', not counted on the first group), seven are of the type 'Workflow Process' and 30 are of the type 'Report Execution'. The collaborating team was composed of three software developers and one manager.

The second system is used by a large tourism company that manages events. The software selected for our tests has 57 features, 21 of which are of the type 'Manage Entity', 16 are of the type 'Manage Composite Entity', five are of the type 'Workflow Process' and 15 are of the type 'Report Execution'. The team was composed of a software developer and a manager.

Finally, we picked a system used to control and standardise forms generated in a nationwide cellulose company. The software has 109 features, 42 of which are of the type 'Manage Entity', 33 are of the type 'Manage Composite Entity', six are of the type 'Workflow Process' and 28 are of the type 'Report Execution'. The project involved four developers, a business analyst and a manager.

The group of features covered by our tests is presented in Table 3.

Table 1 Examples of objects returned by *identifiesSentence()*

No.	Subject	Verb	Method	Noun or attributes	Prep1	Prep2	Entity class	Boundary class	Communication class	Actor
1	customer	informs		Attributes		to				manager
2	manager	selects	insert Account			on		mainform		
3	manager	enters		attributes (…)	of		customer			
4	system	returns	input … screen			to				manager
5	system	sends		notification		by			e-mail	
6	system	searches				for	customer			
7	system	returns		Attributes	of		customer			
8	system	saves		Attributes	of		customer			
9	system	validates		Attributes	of		customer			
10	system	verifies		Attributes	of		customer			
11	system	displays		Attributes	of	on	customer	mainform		
12	system	cancels	operation							

Table 2 LUCAM sentence patterns

No	Syntactic structure	Sender	Receiver	Operation
1	ActorS verb noun preposition ActorR	—	—	—
2	actor verb noun/method preposition boundary	actor	boundary	verb + noun/method
3	actor verb noun preposition entity	actor	boundary	verb + noun + preposition + entity
4	system verb noun/message preposition actor	controller	boundary	verb + noun/message
5	system verb noun preposition communication	controller	controller	verb + noun + preposition + commun.
6	system VerbEntityWithReturn noun preposition entity	last receiver class (LRC)	entity	verb + noun + preposition + entity
7	system VerbEntityWithReturn noun preposition entity	entity	controller	return + noun + preposition + entity
8	system VerbEntityWithoutReturn noun preposition entity	controller	entity	return + noun + preposition + entity
9	system VerbValidation noun preposition entity	controller	controller	verbv. + noun + preposition + entity
10	system VerbProcessing noun preposition entity	controller	controller	verbp. + noun + preposition + entity
11	system VerbReturnInBoundary noun preposition boundary	LRC	boundary	verbreturn + noun
12	actor/system verb noun	controller	controller	verb + noun

```
Require: Use case specification in LUCAM
Ensure: Diagrams elements
  $Control\_Class\_Name \leftarrow Use\_Case\_Name$
  $Actors[\ \ ] \leftarrow Primary\_Secondary\_Actors\_Names$
  $System \leftarrow System\_Name$
  for Each use case flow do
    while Sentence in flow do
      Read Sentence
      Execute the method IdentifiesSentence( )
      Execute the method IdentifiesMessage( Sentence[ ] )
      $Diagrams\_Elements[\ \ ] \leftarrow$
      $Mapped\_Elements\_Sentence$
    end while
  end for
  return $Diagrams\_Elements[\ \ ]$
```

Fig. 2 *Algorithm 1: Main steps executed by the parser*

The identities of collaborating companies and the original documents provided by them will be kept private, following compliance rules agreed between the parties.

It was noticeable in our tests that both class and sequence diagrams remained in accordance with what was specified in LUCAM language. This is a very important factor in stimulating the use of formal methods, because in practice we observed that most models that are handcrafted by specialists partially diverge from textual specifications, failing to accurately portray what was specified.

The cellulose company provided us with their TimeSheets records, from which we selected a set of 60 functional requirements (FR) for analysis, 30 of the type 'Simple Entity' and 30 of the type 'Composite Entity'. Our analysis stored the time each analyst took to specify (both textually and graphically) each use case, manually and using the LUCAMTool.

The first chart in Fig. 7 represents the time (in minutes) needed to create specifications for 30 features classified as 'Simple Entity'.

Analysing the overall results and considering all use cases, there was a productivity increase of about 30%. From the 15th functional requirement (FR15) onwards, we started to see a more stable behaviour, and result analysis from this point on revealed that productivity gains reached up to 50% in comparison with the manually crafted specification.

In the second chart, presented in Fig. 8, we can see how long analysts took to complete the specification of 30 'Composite Entity' use cases.

For use cases classified as 'Composite Entity', an overall result analysis, including language learning time, tells us that the effective productivity was ~20%. Beginning with the 20th functional requirement (FR20), the productivity increased for the remaining requirements by nearly 40%.

Even though the results presented in the chart were obtained in tests with a single team, we can already perceive significant productivity gains. We observe that initially analysts take a significantly longer time to complete a specification due to their unfamiliarity with the LUCAM features. The fact that LUCAM is a DSL with limited scope makes learning it very fast, especially in comparison with a general-purpose language. When users acquire a certain level of familiarity with the language, productivity gains ensue.

We also surveyed with the teams working in the collaborating companies. Below we detailed a few key points assessed from the answers gathered in the survey.

Note that in the biggest of our three collaborators, software developed was appropriately documented. The company mentioned, however, that quite often documentation becomes outdated with time, no longer accurately reflecting what was implemented.

However, the smaller companies do not have a culture of documenting the software they develop. When asked why they do not document the software, they told us: '[…] documentation is too

```
Use Case: SB_BankAccount.
Brief Description
"Brief description of the use case.".
System: SystemName.
Primary and Secondary Actors
Primary Actors: Manager.
Main Flow: AddAccount.
Manager starts Use Case.
   Customer informs the attributes to the Manager.
   Manager enters attributes (Int ID, String name, String
      socialSecurity, Date  birthDate) of Customer on MainForm.
   Manager enters attributes (String agency, String numAccount) of
      Account on MainForm.
   Manager selects "InsertAccount" on MainForm.
   System validates attributes of Customer.
   System validates attributes of Account.
   If ["Inconsistency"]
      System returns "Incorrect data" to Manager.
   Else
      System saves attributes of Customer.
      System saves attributes of Account.
      System returns "Account successfully inserted" to Manager.
   EndIf
Manager finishes Use Case.
```

Fig. 3 *Main flow – AddAccount*

```
Alternate Flow 01: CloseAccount.
   Customer informs the attributes (ID) for the Manager.
   Manager selects "CloseAccount" on MainForm.
   System validates the attributes (ID) of Customer.

   If ["Invalid ID"]
      System returns "Invalid ID" to Manager.
   Else
      System retrieves the informations of Customer.
   EndIf

   If ["Customer  nonexistent"]
      System returns "Customer nonexistent" to Manager.
   Else
      System displays the attributes of customer on MainForm.
   EndIf

   Customer informs the attributes (Password) for Manager.
   Manager enters the attributes (Password) of Customer on MainForm.
   System validates the attributes (Password) of Account.

   If ["Invalid Password"]
      System returns "Invalid password" to Manager.
   Else
      Manager selects "ConfirmAction" on MainForm.
      System disables the informations of Account.
      System returns "Account Closed Successfully" to Manager.
   EndIf
```

Fig. 4 *Alternate flow – CloseAccount*

time-consuming, and it is not worth it to our team to write it. […] UML diagrams are too complex and require too much effort from our analysts'.

Most of the interviewed staffs are versed in one or more programming languages and were interested in a solution to help automate some of the stages in software development. They are also willing to learn a new language to write software specifications.

Regarding results obtained from the use of the LUCAMTool, most of the respondents consider its application in a real scenario viable, according to a few answers: '[…] the language used by the tool is very intuitive and easy to learn. […] using the proposed language gives us more precision when building the generated diagrams and partly removes the reliance on specific software to generate diagrams …'.

When asked whether diagrams generated by the LUCAMTool accurately reflect what was specified, and whether specifications translated to LUCAM remain authentic to the natural language models provided, the majority of respondents confirmed that they do, validating the conducted tests.

After receiving training on the LUCAM language and the LUCAMTool, and when presented with the results obtained during testing, the teams identified some positive points: '[…] the language is objective, clear, and easy to learn. […] the models accurately reflect the specification. […] artefact generation in XMI format'; and a few negative ones: '[…] specifications of more specific processes, such as SAP, are not supported. […] the IDE could use some improvements in its code editor and include a feature to view generated diagrams'.

4 Related work

Early last decade, Li [8] proposed a set of rules to formalise the specification of textual use cases, enabling the analyst to infer what classes, objects, associations, attributes and operations compose a particular use case. However, it is a case of purely mechanical and non-automated activity, since a testing tool for the proposed standard was not developed.

Williams *et al.* [9] proposed an approach that allowed users to explore the most common problems observed in use case modelling, unveiling inconsistencies and misalignment between models and their textual specifications. Hoffmann *et al.* [10] later presented a meta model for textual use case description that defines a representation of use case behaviour, being easily understandable for readers that do not master the subject. For modelling narrative use cases, the authors developed the *NaUTiluS* tool.

In [2], a language for use case specification named SilabReq is proposed. From use cases, it generates domain models, the system operations list, the use case model and activity and state diagrams. In the following year, the same author proposed the separation of use case specifications into different layers of abstraction, since use cases are used by people with different roles and needs during software development, from end-users, requirement engineers and business analysts, to designers, developers and testers. Differently from the aforementioned approach, and in addition to use case and sequence diagrams, the LUCAMTool is also able to generate class diagrams, one of the most important diagrams in UML. The tests conducted with the LUCAMTool in real software development environments and the results derived from them also set LUCAM apart from SilabReq.

Thakur and Gupta [11] presented a tool that enables automatic generation of sequence diagrams from use case specifications written in natural language (English). In this approach, the natural language analyser developed by *Stanford NLP Group* [12] was used to identify the objects and interactions among them from the specification. The parser analyses sentences and classifies each word of the text as an adjective, adverb, article, pronoun, noun, verb etc. Therefore, the approach does not consider situations in the sequence diagram such as combined fragments, self calls and it does not generate other *Unified Modelling Language* (UML) models.

In the next year, Yue *et al.* [13] presented a tool denominated *aToucan*. The tool is able to perform automatic generation of a UML analysis model contemplating class, sequence and activity diagrams, from a previously defined use case model. Researchers also adopted a specification in natural language (English) and similarly to the previously cited work, *Stanford Parser* was used to classify words in sentences written in natural language.

Despite the evolution in automatic model generation techniques, we observed that adopting natural language implies embracing the risk of recurring problems, since natural language is inherently non-structured, ambiguous and does not establish a communication standard for the team. This hinders the automation of artefact generation and, consequently, the traceability among specifications, model and source code.

On the other hand, domain-specific languages (DSLs) establish a common language to be used by all team members. Its formality standardises communication among those involved in the project, paving the way towards more productivity, increased reliability and enhanced generated artefact quality [14, 15]. DSLs also permit solutions in the application domain level, enabling the analysts to understand, validate, modify and develop features utilising the defined language. Domain specialists and software engineers are already familiar with the formality of programming languages, which reduces the time for learning the DSL and optimises resource utilisation [3, 15, 16].

5 Conclusion

Features present in the LUCAM standardise important stages in system development, such as use case specification and modelling.

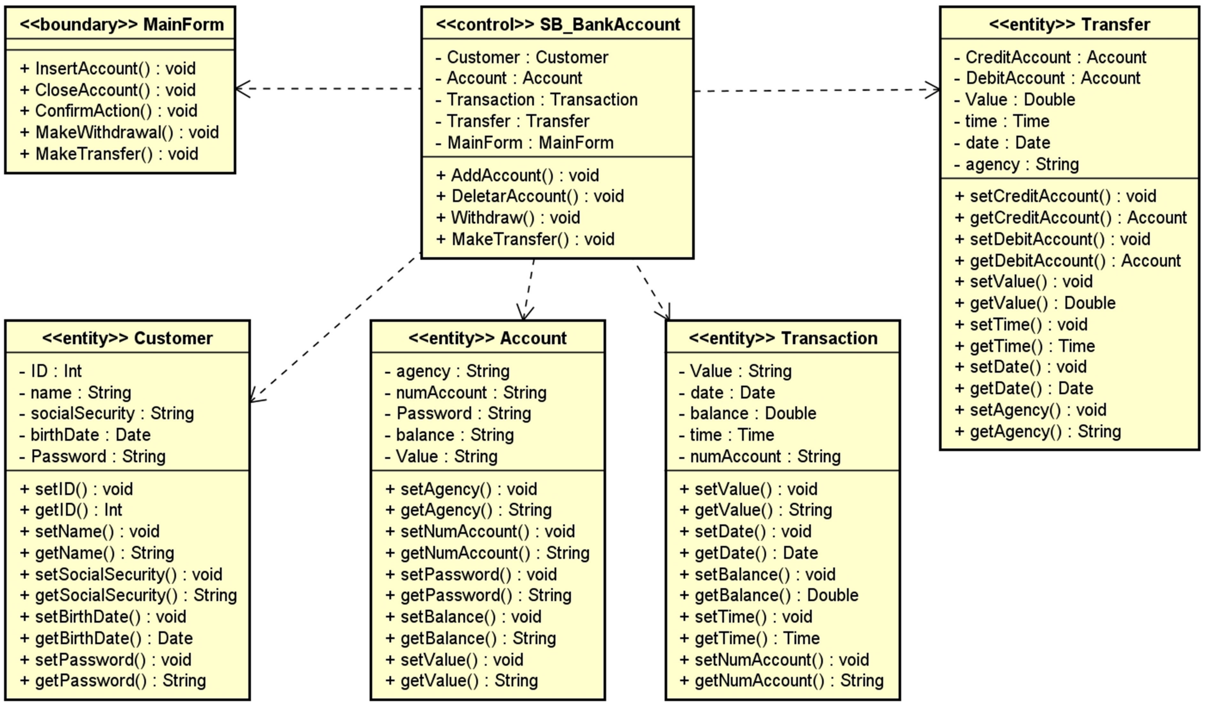
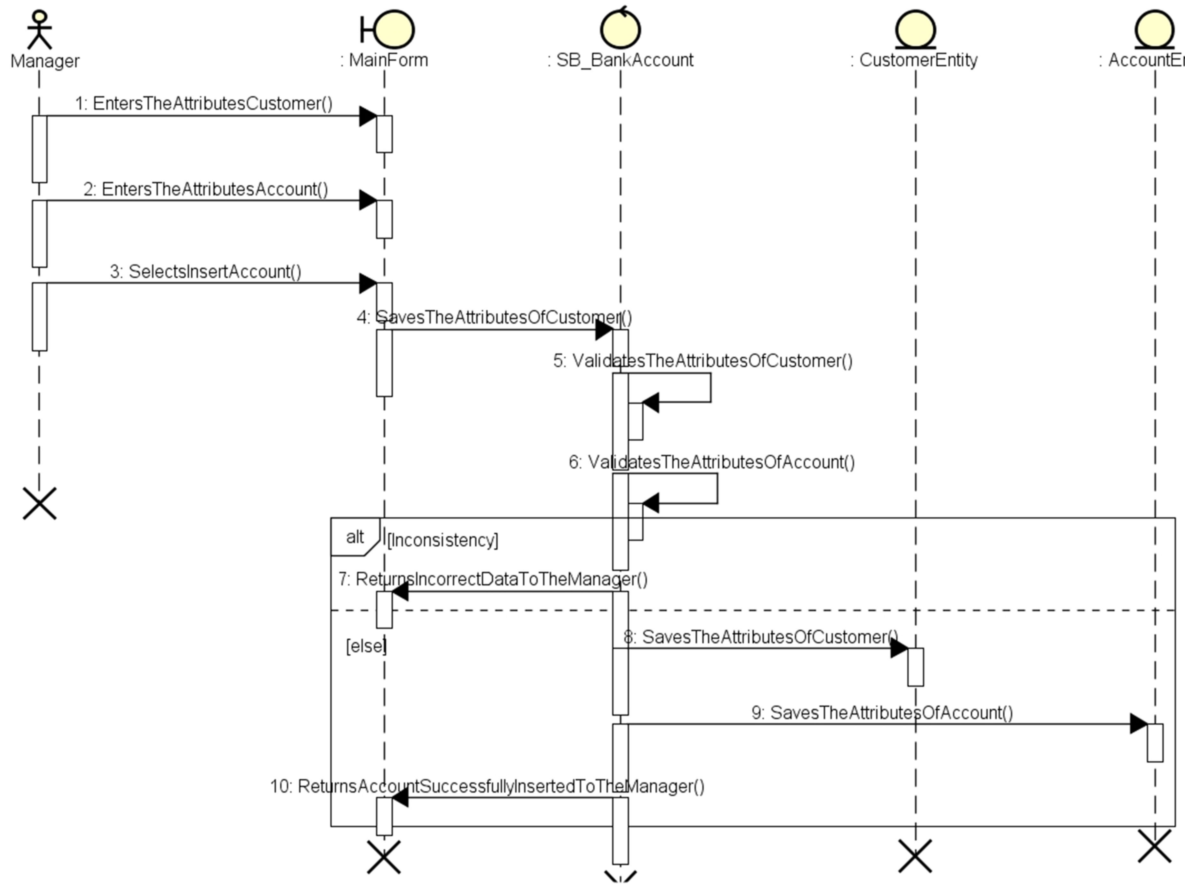


Fig. 5 *Class diagram*



Manager
: MainForm
: SB_BankAccount
: CustomerEntity
: AccountEntity
1: EntersTheAttributesCustomer()
2: EntersTheAttributesAccount()
3: SelectsInsertAccount()
4: SavesTheAttributesOfCustomer()
5: ValidatesTheAttributesOfCustomer()
6: ValidatesTheAttributesOfAccount()
alt
[Inconsistency]
7: ReturnsIncorrectDataToTheManager()
[else]
8: SavesTheAttributesOfCustomer()
9: SavesTheAttributesOfAccount()
10: ReturnsAccountSuccessfullyInsertedToTheManager()

Fig. 6 *Sequence diagram*

Table 3 Feature list for the tested systems

Systems	Single entity	Composite entity	Workflow	Reports	Total
concrete management	53	25	7	30	115
event management	21	16	5	15	57
form management	42	33	6	28	109

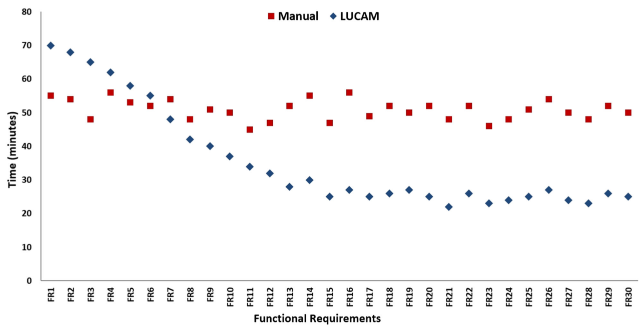


Fig. 7 *Time specification and modelling – single entity*

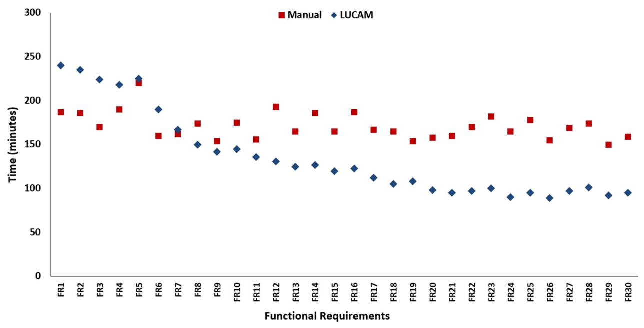


Fig. 8 *Time specification and modelling – composite entity*

This standardisation increases productivity and establishes a clear and precise team communication standard. It alleviates frequent problems arising from natural language use, such as ambiguity, uncertainty and complexity. The LUCAMTool also enables the automatic and consistent generation of use case, class and sequence diagrams, which can be used to generate source code prototypes.

Every DSL is defined by syntactic and semantic rules, which makes user training sessions an essential activity. As a result of our tests, we realised that we can mitigate productivity losses resulting from the use of an unfamiliar language by training users in the use of the tools [3, 16]. We learned that DSL designers must clearly understand the DSL's purpose and how its potential users work. The DSL's syntax and semantics should be simple and built straightforwardly on the concepts and vocabulary that users already understand. This makes the DSL easier to learn.

We concluded that it is possible to automatically generate software artefacts from a DSL. This has some advantages: It enables the reuse of existing software artefacts. It also produces standardised artefacts based on consistent models, thus potentially increasing productivity, decreasing errors and improving quality. By providing a consistent set of representations, it facilitates communication among analysts, software developers and others.

Although simulated environments are quite useful in debugging software tools, they often do not capture all the relevant characteristics of real-world software development environments. Some situations only arise on large projects in production environments. Thus, the opportunity to conduct tests in real-world environments was invaluable. It enabled us to identify potential features of the LUCAMTool and helped us plan future steps of the project.

What are the possible future steps? Possible extensions to LUCAMTool include:

- support for multiple natural languages, not just English;
- version control for the generated artefacts;
- generation of additional software artefacts such as other UML diagrams, screen prototypes, automated test cases and source code;
- import and export of specifications to enable interoperation with other tools;
- specification and management of non-functional requirements;
- support the user story format;
- automatic propagation of changes in any of the formats (DSL, diagrams and source code) to all the others to better support iterative refinement and validation of software.

The LUCAMTool has been tested in selected environments, but testing of the tool and its extensions should be expanded to a larger range of corporate environments.

6 Acknowledgment

The authors acknowledge the financial support received from IFMG, FAPEMIG and PUCMinas Brazil.

7 References

[1] El-Attar, M.: ‘A systematic approach to assemble sequence diagrams from use case scenarios’. 2011 3rd Int. Conf. Computer Research and Development (ICCRD), March 2011, vol. 4, pp. 171–175

[2] Savic, D., Antovic, I., Vlajic, S., *et al.*: ‘Language for use case specification’. 2011 34th IEEE Software Engineering Workshop (SEW), June 2011, pp. 19–26

[3] Fowler, M., Parsons, R.: *‘Domain-specific languages’* ‘(The Addison-Wesley Signature Series) (Addison-Wesley, Upper Saddle River (NJ), Boston, Paris, 2011)

[4] Heering, J., Mernik, M.: ‘Domain-specific languages for software engineering’. Proc. 35th Annual Hawaii Int. Conf. System Sciences, 2002, HICSS, January 2002, pp. 3649–3650

[5] Tiwari, S., Gupta, A.: ‘A systematic literature review of use case specifications research’, *Inf. Softw. Technol.*, 2015, **67**, pp. 128–158

[6] ‘Openup’. Available at http://epf.eclipse.org/wikis/openup/index.htm, accessed 2 August 2 2015

[7] ‘Astah’. Available at http://astah.net/editions/professional, accessed 30 August 2015

[8] Li, L.: ‘Translating use cases to sequence diagrams’. Proc. 15th IEEE Int. Conf. Automated Software Engineering, ser. ASE ‘00, Washington, DC, USA, 2000, p. 293

[9] Williams, C., Kaplan, M., Klinger, T., *et al.*: ‘Toward engineered, useful use cases’, *J. Object Technol.*, 2005, **4**, (6), pp. 45–57

[10] Hoffmann, V., Lichter, H., Nyßen, A., *et al.*: ‘Towards the integration of UML-and textual use case modeling’, *J. Object Technol.*, 2009, **8**, (3), pp. 85–100

[11] Thakur, J.S., Gupta, A.: ‘Automatic generation of sequence diagram from use case specification’. Proc. 7th India Software Engineering Conf., ser. ISEC ’14, New York, NY, USA, 2014, pp. 20:1–20:6

[12] ‘Stanford parser’. Available at http://nlp.stanford.edu/, accessed 5 August 2015

[13] Yue, T., Briand, L.C., Labiche, Y.: ‘Atoucan: an automated framework to derive UML analysis models from use case models’, *ACM Trans. Softw. Eng. Methodol., May* 2015, **24**, (3), pp. 13:1–13:52

[14] Heijstek, W., Chaudron, M.: ‘The impact of model driven development on the software architecture process’. 2010 36th EUROMICRO Conf. Software Engineering and Advanced Applications (SEAA), September 2010, pp. 333–341

[15] Ghosh, D.: ‘DSL for the uninitiated’, *Commun. ACM*, 2011, **54**, (7), pp. 44–50

[16] Gupta, G.: ‘Language-based software engineering’, *Sci. Comput. Program.*, 2015, **97**, (Part 1), pp. 37–40, special issue on new ideas and emerging results in understanding software