Abstract

The use case artifact is the de facto standard to represent functional requirements of a software and can be used to verify and analyze the requirements throughout the development process. Consistent and unambiguous requirements are documented using formal representation techniques. Non-technical stakeholders usually lack skills required to understand requirements documented using formal approaches. In this paper we propose an algorithm that transforms a use case description into a Kripke structure, which makes it suitable for formal representation of software requirements without requiring any special skills and facilitating non-technical stakeholders as well.

1 Introduction

A functionally acceptable software is the result of correctly gathered and written software requirements [26]. There are different approaches to gather and specify requirements. Two broad categories of software requirements [18] include: 1) functional requirements that describe the software behavior and 2) non-functional requirements which describe software acceptability in its intended operating environment. Requirements engineering and specification is a vast field and provides different approaches to document requirements [30]. The approaches to collect and represent requirements depend upon the nature of software to be developed and the type of stakeholders involved in the process. Depending upon these different factors requirements can be specified in a formal or an informal manner [26]. In an informal approach requirements are written in plain natural language. This makes it easier for the stakeholders to comprehend the requirements. Formal specification of requirements requires mathematical notation for its formulation. This entails special training in both requirements specification and their comprehension. This approach, however, provides the advantage of unambiguous and clear requirements. The informally written requirements are prone to ambiguity, inconsistency and incompleteness [27]. Therefore, both formal and informal specification of requirements have some advantages and disadvantages. It will be beneficial if we develop an approach that combines the benefits of both these approaches. In this paper we propose an approach that takes an informal requirements description and transforms it into a formal model that can later be used for both verification and software testing.

The use case artifact [13] is the de facto standard to represent software requirements. A use case artifact typically includes a use case diagram and textual description using natural language in a standard template. There are different templates available to document a use case description as the UML does not define a specific standard template for a use case description.

To create models of a software is another effective way of capturing software requirements. These models again can be formal, informal or semi-formal in nature depending upon the syntax and semantics of these models. In our proposed work discussed in Section 5, we use a Kripke structure [32] as a model. This relates to our earlier works in [28], [29] and [21] which require a Kripke structure and a formal specification. In this paper, we intend to bridge the gap between an informal requirements description and a formal specification that will be beneficial for those users in the industry that are not trained for writing formal specifications.

Since the use case artifact itself is quite flexible therefore, we propose a use case template which will be discussed in Section 4. This template will be used along with the proposed algorithm to transform a use case description into a Kripke structure. The motivation to convert a use case into a Kripke structure is that such a formalism can be used for formal verification of the system or it can be used for automatic test case generation as proposed in [28], [29], [6], [8], [22], [7] and [23]. The rest of the paper is organized as follows: Section 2 gives an account of related work on the subject, Section 3 describes the preliminaries of use cases and Kripke structures, Section 4 introduces the proposed use case template, Section 5 introduces the proposed algorithm for transforming a use case to a Kripke structure, Section 6 describes the working of the algorithm with an example and finally some conclusions and future directions of work are given in Section 7.

2 Related Work

Several previous works have investigated the possibility of using a combination of both informal and formal specifications or models (see [16], [5] and [9]). All these approaches
have some merits and demerits. Most of these require manual effort during all the transformation or during most part of transformation, e.g., a use case description can be transformed into a set of Boolean variables and a relation defined over them [16]. This will be useful for state exploration by model checking approach and enables symbolic reasoning using temporal logics [2]. This transformation of a use case into a Boolean expression requires manual effort and technical skills with mathematical training.

A GUI based tool Play-Go [11] requires to represent a use case by elaborating use case specifications along with user functionality. This requires a user to be an expert in programming concepts to clearly understand the semantics of specification for representation in PlayGo.

Another formalism used for informal to formal transformation is the X-machine [5]. This technique uses a supervised approach to transform a use case into an X-machine with the help of corresponding system sequence diagrams. Chunyan et. al. [20] and Kapeti et. al. [14] proposed approaches that facilitate the automated test case generation from X-machine.

Martin et. al. [9] suggested to use OCL language along with sequence diagrams has been used to verify the sequence of operations being performed. Sequence diagrams are created at later stage of analysis phase and correcting specification at this stage will include a reasonable cost in development process.

Several methods exist in the literature to specify and describe functional requirements. These can vary from formal to informal or from visual to textual approaches. Formal approaches to represent software behavior include abstract state machines [10], X-machines [12], Z notation [31]. Visual approaches consist of activity diagrams [3] and system sequence diagram [19]. Whereas textual description can be provided in structured or unstructured manner. Formalism in a requirement description brings in clarity and removes ambiguity but comes at the cost of requiring skilled and trained people for writing and understanding it. Textual approaches facilitate to document requirements description in natural language and are widely used in the industry because of no extra training required to read and write such requirements document.

3 Use Case Notation and Kripke structure

In this section we describe briefly the use case notation and the formalism of Kripke structures that will be required by our algorithm.

3.1 The Use Case Notation

A single use case represents a single software functionality and can be represented using a use case diagram along with its description. A use case diagram consists of all actors, all use cases and possibly the relationships between the use cases depending upon the taste of the requirements engineer modeling them. An actor exists outside the software boundary in a use case diagram and is responsible for initiating a use case. An actor is said to be a primary actor if it initiates a use case functionality whereas a secondary actor is the one that facilitates to complete a use case functionality and does not initiate one on its own. A primary actor initiates the use case and can receive output if required from the system. An actor can interact with one or several use cases. A use case description contains two distinct parts namely a success scenario and an alternate scenario. A success scenario describes normal software behavior corresponding to user’s actions/inputs. An alternate scenario, on the other hand, describes the potential system behavior of the software may exhibit if the user provides an invalid input or unspecified action. The structure of a typical use case diagram and its description is shown in Figure 1.

Structured textual descriptions encourage to document a use case description in a specified template. Jacobson [4], Toro et al. [24], Cockburn[1], RUP[17] and Leite et al.[19] have proposed templates that are being practiced in the industry but none of these is considered as a standard template by industry, academia or even OMG’s UML for documenting a use case description. However, it is suggested to use short sentences and to adopt SVDPI pattern (Subject, Verb, Direct-Object, Preposition, and Indirect-Object).

The template elements may include name, number, goal, scope, level, description, primary actor, secondary actor, precondition, post condition etc to be written clearly. There is no template element provided for input(s) given to the use case or output(s) produced by the use case. In Section 4 a template has been proposed that encourages to represent these distinctively along with some other adoption that may facilitate the transformation of a use case to Kripke structure.
3.2 Kripke Structure

As discussed in Section 2, several formalisms exist in the literature to model or specify software requirements. Some of these require a software to be represented in the form of some states along which the system can traverse on different inputs. Some others can describe behavior with the help of some kind of textual formal specification like temporal logic e.g LTL and CTL. In simple terms a state is a visible change in the behavior of the software. Some of the formalism used in the literature for software behavior modeling include finite state machines (FSMs) [33], deterministic finite automata (DFAs) including both its Moore [25] and Mealy [15] automata variants and the Kripke structure [32].

A finite state machine represents the possible state space of a software along with the inputs used from each state to transition to other states. Moore and Mealy automata have a similar state space concept but give just one-bit output in terms of an input being accepted by the machine or rejected by it. A Kripke structure in general, can be used for outputs in terms of atomic propositions.

A Kripke structure is a 5-tuple consisting of

\[ \mathcal{K} = (Q, \Sigma, \delta, q_0, \lambda) \]

where

- \( Q \) is a finite set of states,
- \( \Sigma = \{\sigma_1, \ldots, \sigma_n\} \) is finite set of input symbols,
- \( \delta \subseteq Q \times Q \) is the transition function,
- \( q_0 \in Q \) is the initial state,
- \( \lambda : Q \to 2^{AP} \) (elements of \( 2^{AP} \) are called state labels with \( AP \) number of atomic propositions) is a labeling function for states.

The labeling function of a Kripke structure labels different states by possible values of the atomic propositions in that state. The function \( \delta \) maps a state to another state and label of that particular state describes the software behavior as output labeled \( \lambda \). To reduce the unnecessary computation cost of evaluating atomic propositions in each state another flavor of a Kripke structure was introduced in [22] which labels the state with a binary bit-vector instead of another flavor of a Kripke structure was introduced in [22] which labels the state with a binary bit-vector instead of atomic propositions. This definition of a Kripke structure is isomorphic to a multi-bit Moore machine. A deterministic Kripke structure consists of a 5-tuple:

\[ \mathcal{K} = (Q, \Sigma, \delta, q_0, \lambda) \]

where

- \( Q \) is a finite set of states,
- \( \Sigma = \{\sigma_1, \ldots, \sigma_n\} \) is finite set of input symbols,
- \( \delta : Q \times \Sigma \to Q \) is the transition function,
- \( q_0 \in Q \) is the initial or start state,
- \( \lambda : Q \to B^k \) where \( (b_1 \ldots b_k) \in B^k \) is an enumeration or indexing of \( k \) atomic propositions.

The labeling function \( \delta \) maps a state to another state after reading an input symbol. Iterative \( \delta \) can be defined as \( \delta^* : \Sigma^* \to Q \), where \( \delta^*(q_0, \varepsilon) = q_0 \) and \( \delta^*(q_0, \sigma_1, \ldots, \sigma_n) = \delta(\delta^*(q_0, \sigma_1, \ldots, \sigma_{n-1}), \sigma_n) \) and \( \lambda^* : \Sigma^* \to B^K \) denotes iterated output function \( \lambda^*(\sigma_1, \ldots, \sigma_n) = \lambda(\delta^*(q_0, \sigma_1, \ldots, \sigma_n)) \).

4 Proposed Use Case Template

Data provided when an actor interacts with the use case can be considered an input, whereas a message produced can be treated as output. Each input and its corresponding output can be used to verify the behavior of the software and can be validated through software testing or formal verification.

It is proposed to describe input set and output set distinctively in use case description. A bit vector labeling a state represents the binary output of software on that state. Therefore, it is also proposed that output set elements are defined along with their binary representation and a set of bits named BitVector to represent output. Whereas elements of InputSet in proposed template can serve as input symbols used by transition function in the output Kripke structure.

This requires to mention system actor(s) in the actorlist, input(s) in the inputlist and output(s) in the outputlist and success use case scenario in scenario. It is also proposed to document corresponding alternate scenario along with success scenario. It is assumed that this representation will facilitate to automate the conversion procedure of use case to a Kripke structure. The proposed template for use case is shown in Figure 2.

<table>
<thead>
<tr>
<th>Use Case: Use Case Name [unique]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActorSet: List of Actor(s)</td>
</tr>
<tr>
<td>lengthBitVector: j [where j is a number]</td>
</tr>
<tr>
<td>BitVector: [bit assigned to appropriate Output]</td>
</tr>
<tr>
<td>InputSet: [Input1, Input2, Input3] [where k is a number]</td>
</tr>
<tr>
<td>OutputSet: [Output1, Output2, Output3] [where l is a number]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality 1</td>
</tr>
<tr>
<td>Alternate Scenario</td>
</tr>
<tr>
<td>Functionality</td>
</tr>
<tr>
<td>End of Use Case</td>
</tr>
<tr>
<td>Functionality 2</td>
</tr>
<tr>
<td>Alternate Scenario</td>
</tr>
<tr>
<td>Functionality</td>
</tr>
<tr>
<td>Continue</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Functionality m [where m is a number]</td>
</tr>
</tbody>
</table>

End of Use Case

Figure 2. Proposed Use Case Template
5 Proposed Algorithm

5.1 State

Kripke structure represents a state along with its label. We use a struct to represent state as in Figure 3. A state defined using this struct contains a caption and an array of bits named BitVector to store the particular value of output label at the corresponding state. The defineState function is used to define states and it accepts three arguments: cc a caption for state, bb[] bits values and aa that determine size of bits. This function assigns values to caption and BitVector of state.

5.2 Labeling Function

The labelState function in Figure 4 is used to update the value of a label represented by BitVector in Figure 3. This function accepts gg[] of type Boolean, OutputSet ff provided by user as represented in Figure 6 and output-Symbols ee parsed by algorithm from the use case. This function returns the updated BitVector gg.

5.3 Transition Function

A transition relation exists among states of a Kripke structure and represents different transitions from one state to another state after reading a particular input symbol. Figure 5 shows a struct that is being used to create and store relation between two states. A relation defined using this struct will take jj and ii state values in fromState and toState variables that are of type state and transitionInput is used to store the value of input symbol. The defineRelation function is used to relate two states and it accepts values of fromState as jj, to-state as ii and input-symbol as hh. The defineRelation will assign the values to variables of the relation struct.

5.4 Use Case

A use case is required to be defined in the proposed template given in Section 4. This again will be stored in a struct for computation and implementation simplicity. Figure 6 represents the struct used for this purpose. The defineUseCase accepts the values, details regarding assignment and data types are left for programming stage.

5.5 Proposed Algorithm

The proposed algorithm takes a use case description in the proposed template given in Section 2 and produces a Kripke structure as output. The proposed algorithm is as follows:

Require: Use Case in Proposed Template
Ensure: Kripke Structure

Data: tBitVector, tCaption, tInput, tOutput, currentState, bld_pointer, tReadLines, tActor, i, j, n
1: usecase=defineUseCase(name, actorlist, inputset, outputset, bitvector, size, scenario) //Read use case from file
2: n = usecase.lengthofBitVector
3: tBitVector[n]
4: i = 0
5: j = −1
6: Read BitVector Value into tBitVector for start state from user
7: \( S_0 = \text{defineState}(\text{Start\_State}, t\text{BitVector}) \)
8: currentState = \( S_0 \)
9: while \( \text{usecase.ScenarioList.current} \neq \text{EndofUseCase} \) do
10: \( t\text{ReadLines} = \text{usecase.ScenarioList.current} \)
11: if usecase.ActorSet \( \in \) tReadLines then
12: \( t\text{Actor} = \text{usecase.Actorfound} \)
13: end if
14: if usecase.InputSet \( \in \) tReadLines then
15: \( t\text{Input} = \text{usecaseInputfound} \)
16: end if
17: if tActor and tInput \( \neq \) null then
18: \( i = i + 1 \)
19: \( t\text{Caption} = \text{split}(\text{tReadLines}, t\text{Actor})/\text{split} \) function eliminates tActor value from tReadLines
20: \( t\text{BitVector} = \text{currentState.BitVector} \)
21: repeat
22: \( t\text{ReadLines} = \text{usecase.ScenarioList.NextLine} \)
23: if usecase.ActorSet \( \in \) tReadLines then
24: \( t\text{Actor} = \text{usecase.Actorfound} \)
25: end if
26: if usecase.InputSet \( \in \) tReadLines then
27: \( t\text{Input} = \text{usecaseInputfound} \)
28: end if
29: if usecase.OutputSet \( \in \) tReadLines then
30: \( t\text{Output} = \text{usecaseOutputfound} \)
31: end if
32: until ((AlternateScenario not found in tReadLines) or (t.Actor and tInput \( \neq \) null))
33: \( t\text{BitVector} = \text{labelState}(t\text{BitVector}, \text{usecase.OutputSet}, t\text{Output}) \)
34: \( S_i = \text{defineState}(t\text{Caption}, t\text{BitVector}) \)
35: \( j = j + 1 \)
36: \( r_j = \text{defineRelation}(\text{currentState}, S_i, t\text{Input}) \)
37: hld\_pointer = \( S_j \)
38: if Alternate \( \in \) tReadLines then
39: for all AlternateScenario do
40: repeat
41: \( t\text{ReadLines} = \text{usecase.ScenarioList.NextLine} \)
42: if usecase.OutputSet \( \in \) tReadLines then
43: \( t\text{Output} = \text{usecaseOutputfound} \)
44: end if
45: until (endofUseCase \( \in \) tReadLines) or (continue \( \in \) tReadLines)
46: \( t\text{BitVector} = \text{currentState.BitVector} \)
47: \( t\text{BitVector} = \text{labelState}(t\text{BitVector}, \text{usecase.OutputSet}, t\text{Output}) \)
48: \( i = i + 1 \)
49: \( S_i = \text{defineState}(t\text{Caption} + i, t\text{BitVector}) \)
50: \( j = j + 1 \)
51: \( r_j = \text{defineRelation}(\text{currentState}, S_i, t\text{Input}) \)
52: if endofUseCase \( \in \) tReadLines then
53: \( j = j + 1 \)
54: \( r_j = \text{defineRelation}(S_i, S_0, t\text{Input}) \)
55: else if continue \( \in \) tReadLines then
56: \( j = j + 1 \)
57: \( r_j = \text{defineRelation}(S_i, hld\_pointer, t\text{Input}) \)
58: end if
59: currentState = \( S_i \)
60: end for
61: end if
62: currentState = hld\_pointer
63: end if
64: usecase.ScenarioList.Next
65: end while
66: \( j = j + 1 \)
67: \( r_j = \text{defineRelation}(\text{currentState}, S_0, t\text{Input}) \)

6 Example

ATM cash withdrawal process can be used as an example to illustrate the proposed use case template and working of the proposed algorithm. The success scenario can be described as: customer presents a valid card, a valid pin and a valid amount to the ATM machine. The ATM machine issues cash if the value of amount is equal to or less than the
stored cash amount and finally ejects the card to the customer. Success scenario can have alternates: 1) The presented card is invalid. ATM rejects such a card and the use case finishes, 2) The customer fails to provide a valid pin. The ATM then allows the customer to make three attempts to provide a valid pin. If the customer fails to provide a valid pin the ATM ejects the card and the use case ends. and/or 3) The ATM allows the customer to enter a valid amount. After three failure attempts the system returns the card and the use case terminates. This functionality can be documented as use case description in proposed template shown in Figure 7.

```
Use Case: The ATM Cash Withdrawal
Actor Subsystem
lengthOfBitVector: 7

BitVector: 0000000

InputSet: Card, Balance, Pin

OutputSet: Valid_Card: 00, Invalid_Card: 10, Valid_Amount: 000, Invalid_Amount: 100, Valid_Pin: 00, Invalid_Pin: 01, Invalid_Pin: 11

Scenario:
Customer inserts Card
Software validates Valid Card
Alternate Scenario
Invalid_Card
Software notifies the Customer
Insert Card
End of Use Case
Customer enters Pin
System validates Valid_Pin
Alternate Scenario
Invalid_Pin
System notifies the Customer Requests Valid_Pin
Continue
Alternate Scenario
Invalid_Pin
System notifies the Customer Requests Valid_Pin
End of Use Case
Customer enters Amount
System validates Valid_Amount
Alternate Scenario
Invalid_Amount
System notifies the Customer Requests Valid_Amount
Continue
Alternate Scenario
Invalid_Amount
System notifies the Customer Requests Valid_Amount
End of Use Case
System ejects Cash
Customer takes Cash and Card
End of Use Case
```

Figure 7. The ATM cash withdrawal use case

The proposed algorithm in Section 5.5 is applied on ATM cash withdrawal example shown in Figure 7. The defineUseCase function initializes the object use case after reading the file containing use case. The tBitVector assigns value of the variable use case. The lengthofBitVector is copied into the variable n i.e. 7. The tBitVector is defined to the size of n. The variables i and j are initialized with values 0 and -1 respectively. The user provides the 0000000 value for the tBitVector of Start_State and is defined as the object S_0 is used to represent it. The currentState pointer is assigned the value of the S_0 for further processing. The while loop will execute till the use case scenario ends. The tReadLines is assigned the value of the current line of the use case scenario i.e. Customer inserts Card. The tReadLines contains Customer which is the actor of the use case ActorList and is assigned to tActor. The Card is member of InputList and is assigned to the tInput variable. The condition of if statement becomes true because value of the tActor and the tInput is not null. The value of variable i becomes 1 after it is incremented. The tCaption variable is assigned the value returned by the function split after eliminating the tActor value from the tReadLines value, i.e. inserts Card. The tBitVector is updated with value of the object currentState and the tBitVector value which is equal to 0000000. The scenario of the use case is read until the Alternate Scenario finishes. While reading these lines by algorithm Valid_Card is found which is contained in the OutputList and is stored in the variable tOutput. The return value 0000000 of the function labelState is assigned to the tBitVector. The function labelState updates and return the tBitVector value, for this purpose the function read the Boolean value of Valid_Card from the OutputList. The value of the variable j is incremented and updated value of j becomes 0. The S_1 state is created with the caption inserts Card and the Boolean label of 0000000. The defineRelation is used to create transition relation between S_0 and S_1 and the input symbol read for this transition is the value of the variable tInput. The value of tInput at this moment is Card. The S_1 is allocated to the hld_pointer to for further construction of the Kripke structure from this state after processing alternate scenario.

The variable tReadLines is updated with the read line from alternate scenario of use case until the end of use case or continue is read. The Continue reserved word specifies the use case to continue to normal execution after alternate scenario ends. While reading alternate scenario output symbol Invalid_Card is read and it is stored in the variable tOutput. The function labelState updates the tBitVector value with 0000100 representing the Boolean value of Invalid_Card at appropriate bit location. The function labelState reads Boolean value for the Invalid_Card from OutputSet of use case. The variable i’s value is incremented and its updated value is 2 at this stage. The defineState creates new state S_2 with the caption inserts Card2 and Boolean label of 0000100. The defineRelation creates transition relation between the S_0 and S_2. Value of the tInput variable is Card used as reading symbol for this transition. While reading alternate scenario the end of use case is found, and the defineRelation creates transition.
relation between \( S_2 \) and \( S_0 \) with input symbol \( Card \) that was read for this transition. The \texttt{currentState} pointer is updated with the value of \( S_2 \). There is no alternate scenario found for this success scenario. The pointer \texttt{currentState} is updated with the value of the pointer \texttt{hld_pointer} and the while loop begins its next iteration. The variable \texttt{tReadLines} is assigned next line of use case scenario i.e. Customer enters Pin. The remaining execution of algorithm is left to the reader. The algorithm produces the output Kripke structure shown in Figure 8.

7 Conclusions and Future Work

In this paper, we proposed a use case template and an algorithm which can be used to produce a Kripke structure as output from the use case input provided to it in the proposed use case template. This will enable the non technical stakeholders to write and understand software functionality more easily and will also provide a formalized model of the software system as a Kripke structure. This output Kripke structure can be used to validate and verify requirements and the software respectively using formal approaches. This will be specifically beneficial for automating software testing and software verification processes.

In the future, we intend to extend the proposed algorithm and the proposed template for use case relationships which consists of the include, the extend and the generalization relationships. The interaction of the secondary actor will also be considered in the future. This will make this algorithm more suitable for industrial applications.

References


