Applying Reduction Techniques to Software Functional Requirement Specifications

Jameleddine Hassine[1], Rachida Dssouli[2], and Juergen Rilling[1]

[1] Department of Computer Science, Concordia University, Montreal, Canada
{j_hassin,rilling}@cs.concordia.ca
[2] Concordia Institute for Information Systems Engineering, Montreal, Canada
dssouli@ece.concordia.ca

Abstract. *Requirement Specification* is gaining increasingly attention as a critical phase of software systems development. As requirement descriptions evolve, they quickly become error-prone and difficult to understand. Therefore, the development of techniques and tools to support requirement specification development, understanding, testing, maintenance and reuse becomes an important issue. This paper extends the well-known technique of program slicing to *Functional Requirement Specification* based on the Use Case Map notation. This new application of slicing, called *UCM Requirement Slicing* is useful to aid requirement comprehension and maintenance. In contrast to traditional program slicing, requirement slicing is designed to operate on the requirement specification of a system, rather than the source code of a program. The resulting requirement slice provides knowledge about high-level structure of a system, rather than its low-level implementation details. In order to compute a UCM Requirement slice, we provide a three steps slicing algorithm.

Key words: Functional requirement specification, program slicing, Use Case Maps, comprehension, maintenance.

1 Introduction

Over the last several years, requirements specification and engineering is gaining in importance, as part of the ongoing trend towards improving the software development and maintenance process. Requirement analysis is the first step in the development process, capturing the functionalities of systems, often in the form of scenarios and use cases.

In the early stages of common development processes, system's functionalities are captured in terms of scenarios and use cases. Scenario-driven approaches are widely accepted based on their intuitive syntax and semantics (Amyot and Eberlein [1] provide an extensive survey of scenario notations). Use Case Maps (UCMs), can be applied to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a higher level of abstraction, to provide the stakeholders with guidance and reasoning about the

system-wide functionalities and behavior. Use Case Maps are part of a proposal to ITU−T for a *User Requirements Notation (URN)* [9]. However, a suitable description notation syntax and semantics alone cannot overcome the problems caused by inherent system complexity. There is a need for techniques and tools to simplify requirement specifications in order to support their comprehension, testing, maintenance and reuse.

In our research we address these issues by introducing a new approach to reduce the complexity of the requirement specifications. Our new approach is based on *slicing techniques* to guide requirement engineers, designers and programmers during the comprehension process of requirement specifications. Program slicing was originally introduced as a technique to simplify programs to provide support during debugging and program comprehension [18, 19] and has been applied to a wide variety of problems including: program understanding, maintenance [5], debugging, differencing, integration, testing [18] and model checking [13]. Program slicing, a program reduction technique, allows one to reduce the size of the source code of interest by identifying only those parts of the original program that are relevant to the computation of a particular function/output of interest [19]. It is crucial that slicing preserves the semantics of the original program with respect to the slicing criterion [19].

Our paper introduces a new form of slicing, referred to as *UCM (Use Case Maps) Requirement Slicing*, to aid UCM comprehension and maintenance.

The organization of the paper is as follow: in the next section, we briefly describe the traditional approaches to program slicing. Section 3 introduces the Use Case Maps notation and presents an UCM example. In section 4, the UCM slicing approach is presented and an example is given followed by a discussion on the application of UCM slicing in section 5. Section 6 discusses the UCM data flow and presents the limitations of the proposed approach. Section 7 presents related work. Finally the paper concludes with section 8.

2 Traditional Program Slicing

The notion of program slicing originated in the seminal paper by Weiser [19]. Weiser defined a slice S as a reduced, executable program P$'$ obtained from a program P by removing statements such that S replicates parts of the behavior of the program. Informally, a static program slice consists of those parts of a program that potentially could affect the value of a variable V at a point of interest. Korel and Laski introduced in [10] the notion of dynamic slicing that can be seen as a refinement of the static approach. The dynamic slice preserves the program behavior for a specific input, in contrast to the static approach, which preserves the program behavior for the set of all inputs for which a program terminates. Furthermore, different slicing techniques and criteria are required because various applications require different properties of slices. In recent years, the application of slicing has been extended to other software artifacts [16] including: software architecture [20], requirement models [7, 11] and formal specification [2,

13, 14]. A detailed survey of different slicing techniques and their applications can be found in [4, 18].

3 Describing Requirements Using Use Case Maps Notation

3.1 Use Case Maps

A UCM [8] describes a system in terms of causal relationships between responsibilities (e.g., operation, action, task, function, etc.) along paths allocated to a set of components. The relationships, representing the UCM control flow, are said to be causal because they involve concurrency, partial ordering of activities and they link causes (e.g., preconditions and triggering events) to effects (e.g., postconditions and resulting events). A responsibility can potentially be associated or allocated to a component. In UCMs, a component is generic and abstract enough to represent software entities (e.g., object, agent, process, etc.) as well as non software entities (e.g., actors or hardware). With the UCM notation, scenarios are abstracted above the message exchange level, and therefore are to some extend independent from the underlying implementation level. Path details can be hidden in sub-diagrams called plug-ins, contained in stubs (containers) on a path. A stub can be either static (represented as plain diamond) which contains only one plug-in, or dynamic (represented as dashed diamonds) which may contain several plug-ins whose selection can be determined at run time according to a selection policy. The main UCM constructs are: OR-Fork (alternative scenarios), OR-Join (merging scenarios), AND-Fork (concurrent scenarios), AND-Join (synchronizing scenarios). More details on the UCM semantics can be found in [8].

3.2 Case Study –A Simple Telephony System

Figure 1 shows a UCM model that was originally introduced in [12], describing the connection request phase in an agent based telephony system with user-subscribed features.

It contains four components (originating/terminating users and their agents) and two static stubs. Upon the request of an originating user (req), the originating agent will select the appropriate user feature (in stub Sorig) that could result in some feedback. This may also cause the terminating agent to select another feature (in stub Sterm) which in turn can cause a number of results in the originating and terminating users. Stub *Sorig* contains the originating plug-in whereas stub *Sterm* contains the Terminating plugin. These sub-UCMs have their own stubs, whose plug-ins are user-subscribed features.

1. Stub *Sscreen*:
   - OCS (Originating Call Screening): blocks calls to people on the OCS filtering list.
   - Default: used when not subscribed to any other originating feature.

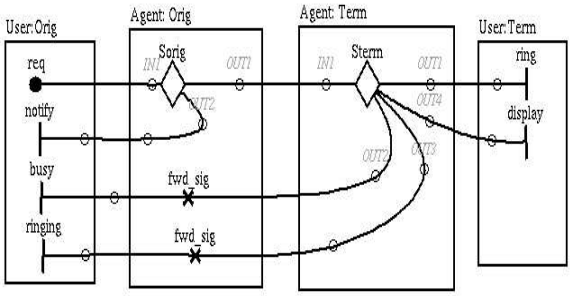
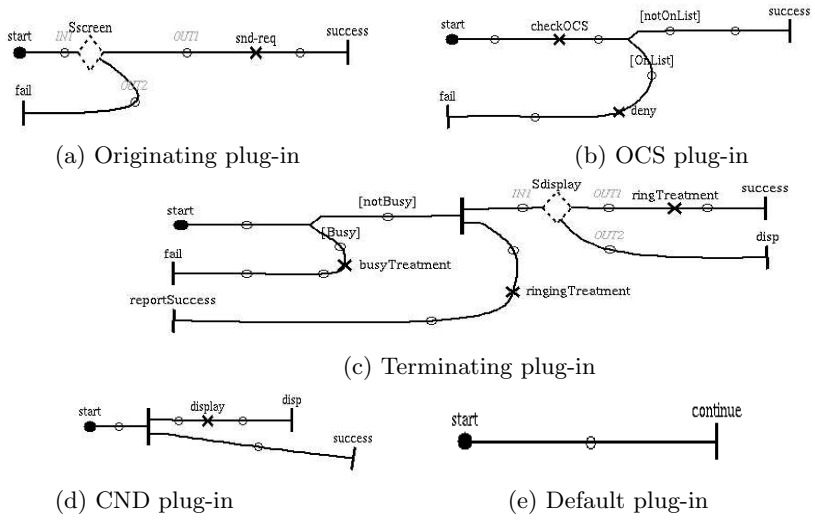


Fig. 1. UCM model (Root Map)

2. Stub *Sdisplay*:
   - CND (Call Name Delivery): displays the caller's number on the callee's device (display) concurrently with the rest of the scenario (ringing).
   - Default: used when not subscribed to any other terminating feature.

The set of global variables for the UCM map are: Busy (the callee is busy), OnOCSList (the callee on OCS list), subCND (the callee is subscribed to CND), subOCS (the caller is subscribed to OCS).



(a) Originating plug-in

(b) OCS plug-in

(c) Terminating plug-in

(d) CND plug-in

(e) Default plug-in

Fig. 2. Plug-ins

Each plug-in (Fig. 2) is bound to its parent stub, i.e., stub input/output segments (IN1, OUT1, etc.) are connected to the plug-ins start/end points, as follow:

1. Sorig Stub : Originating UCM. Condition: *true*.
   Binding:((IN1, start), (OUT1, success), (OUT2, fail))
   - Sscreen Stub
     - OCS UCM. Condition: subOCS. Binding: ((IN1, start), (OUT1, success), (OUT2, fail))
     - Default UCM. Condition: ¬ subOCS. Binding: ((IN1, start), (OUT1, continue))
2. Sterm Stub : Terminating UCM. Condition: True. Binding: ((IN1, start), (OUT1, success), (OUT2, fail), (OUT3, reportSuccess), (OUT2, disp))
   - Sdisplay Stub
     - CND UCM. Condition: subCND. Binding: ((IN1, start), (OUT1, success), (OUT2, disp))
     - Default UCM. Condition: ¬ subCND. Binding:((IN1, start), (OUT1, continue))

4 Use Case Map Slicing

Intuitively, a UCM slice may be viewed as a subset of the behavior of a global UCM. While a traditional slice intends to isolate the behavior of a specified set of program variables, a UCM slice intends to isolate a set of scenarios that lead to a specific behavior. When a UCM slicer is invoked, it takes as input:

1. A complete system requirement specification based on the UCM notation
2. A slicing criterion.
   Note: The choice of slicing criteria will be discussed in detail in section 4.2.

Depending on the user's interest, the UCM slicer computes a backward slice with respect to the selected slicing criterion. While performing the backward traversal, the slicer collects all the logical predicates, defined on UCM global variables, leading to the execution of the targeted criterion and produces what we refer to as *reachability expression*. The reachability expression is solved by finding the initial variable values and/or the sequence of inputs that the environment has to provide to be able to reach the slicing criterion.

4.1 Definitions

In order to focus on the key ideas of UCM slicing, we give the following definitions:

Definition 1 (Use Case Maps). *We assume that a UCM Requirement specification* RS *is denoted by (D, C, V, G, λ, Bc, S, Bs) where:*

- *D is the UCM domain, composed of sets of typed elements. D= SP ∪ EP ∪ R ∪ AF ∪ AJ ∪ OF ∪ OJ ∪ Tm ∪ ST ∪ Ab. Where SP is the set of Start Points, EP is the set of End Points, R is the set of Responsibilities, AF is the set of AND-Fork, AJ is the set of AND-Join, OF is the set of OR-Fork, OJ is the set of OR-Join, Tm is the set of Timers, Ab is the set of Aborts and ST is the set of Stubs.*

- *C is the set of components in RS (C = ∅ for unbound UCM)*
- *G is the set of guard expressions over V, where V is the set of global variables in RS*
- *λ is a transition relation defined as: λ=D×D×G*
- *Bc is a component binding relation defined as Bc =D×C. Bc specifies which element of D is associated with which component of C.*
- *S is a plug-in binding relation defined as S = ST×RS×G.*
- *Bs is a stub binding relation and is defined as Bs =ST×RS×{IN/OUT}×SP/EP. Bs specifies how the start and end points of the plug-in map would be connected to the path segments going into or out of the stub.*

Note: This definition represents our interpretation of Use Case Maps to provide the basis setting for our UCM slicing approach.

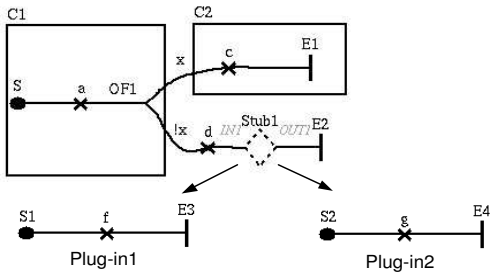


Fig. 3. A UCM example

The UCM of Figure 3 is described as follows:

- D={S}∪{E1, E2}∪{a, c, d}∪{OF1}∪{Stub1}, where OF1 is the OR-Fork.
- C={C1, C2}; V={x, y}; G={x, ¬x, y, ¬y }
- λ={(S, a, true),(a, OF1, true),(OF1, c, x),(OF1, d, ¬x),(d, Stub1, true),(Stub1, E2, true)}
- Bc={(S, C1),(a, C1),(OF1, C1),(c, C2),(E1, C2)}
- S={(Stub1, Plug-in1, y), (Stub1, Plug-in2, ¬y)}
- Bs = {(Stub1, Plug-in1,IN1, S1),(Stub1, Plug-in1,OUT1, E3),(Stub1, Plug-in2, IN1, S2),(Stub1, Plug-in2, OUT1, E4)}

Where Plug-in1 is defined as: D={S1, f, E3}; λ= {(S1, f, true),(f, E3, true)}; C=V=G=Bc=S=Bs=∅
And Plug-in2 is defined as: D={S2, g, E4}; λ= {(S2, g, true),(g, E4, true)}; C=V=G=Bc=S=Bs=∅

In order to define a *UCM slice*, we introduce the concept of: reduced domain, reduced stub, reduced component, reduced guard set, reduced transition relation and reduced binding relations.

Definition 2 (Reduced UCM elements).
Let RS = (D, C, V, G, λ, Bc, S, Bs) be an UCM Requirement Specification.

- *A reduced domain is a set D' that is derived from D by removing zero, or more elements (i.e., $D' \subseteq D$).*
- *Since a plug-in is also a stand alone UCM, a reduced plug-in can be defined in the same way as a reduced UCM(see definition 3).*
- *A reduced stub is a stub that contains reduced plug-ins and may have fewer plug-ins than the original stub.*
- *A reduced component c' is a component that has less functionalities than the original component.*
- *A reduced guard set G' is a set $G' \subseteq G$ that is derived from G by removing zero, or more expressions.*
- *A reduced transition relation λ' is a relation derived from λ by removing zero or more tuples (i.e., $(d_1,d_2,e) \in \lambda$ and $(d_1,d_2,e) \notin \lambda'$).*
- *A reduced component binding relation Bc' is a relation derived from Bc by removing zero or more couples (i.e., $(d,c) \in Bc$ and $(d,c) \notin Bc'$).*
- *A reduced plug-in binding relation S' is a relation derived from S by removing zero or more tuples.*
- *A reduced stub binding relation Bs' is a relation derived from Bs by removing zero or more tuples.*

Given a UCM, our goal is to compute a UCM slice which corresponds to a subset of the original UCM that preserves the semantics of the UCM with respect to chosen slicing criterion.
Note: We can have as a result a set of flat scenarios (i.e., sequential traces where no concurrency nor choices are involved. However the original UCM semantics will not be preserved.

Definition 3 (Reduced UCM). *Let RS = (D, C, V, G, λ, Bc, S, Bs) and $RS' = (D', C', V', G', \lambda', Bc', S', Bs')$ be two UCMs. RS' is a reduced UCM of RS if:*

- *D' is a reduced set of D*
- *$C' = c'1, c'2, \ldots, c'n$ is a subset of C such that for k=1, 2,...,n. $c'k$ is a reduced component of ck*
- *V' is a reduced set of V and G' is a reduced set of G*
- *λ' is a reduced transition relation of λ*
- *Bc' is a reduced component binding relation of Bc*
- *S' is a reduced plug-in binding relation of S*
- *Bs' is a reduced stub binding relation of Bs*

4.2 Use Case Map Slicing Criteria

The selection of a slicing criterion depends on the particular analysis task. The focus is frequently on the examining of the requirement with respect to particular functionality, e.g a particular system feature or a particular behavior.

Definition 4 (UCM Slicing Criteria). *Let RS be a Requirement Specification. A slicing criterion (SC) for RS may be:*

- *A responsibility*
  or
- *Start/end point*

The slicing criterion may eventually include a UCM component. Based on the task and the degree of system understanding a user may choose between specifying only a responsibility or a start/end point as a slicing criterion (Ignoring where the responsibility takes place) or a responsibility and a specific component (to focus the analysis on one specific component).

For the scope of this paper, we limit ourselves to *unbound UCMs* since the *reduced architecture* (set of reduced components) can be easily added (from the original UCM) once we obtain an unbound UCM slice.

4.3 Slicing UCM Constructs

Figures 4 and 5 show different UCM constructs and their potential reduced versions after applying program slicing. E is a generic end point which is added after the SC to form a valid reduced UCM. In the reduced OR-Fork (Figure 4 (aa)), only one path is included in the reduced UCM.In the reduced OR-Join (Figure 4 (bb)), the non-determinism is preserved. In the reduced AND-Fork (Figure 5 (cc)), the interleaving semantics is preserved, since concurrent responsibilities SC and d may occur in different order(SC;d or d;SC).

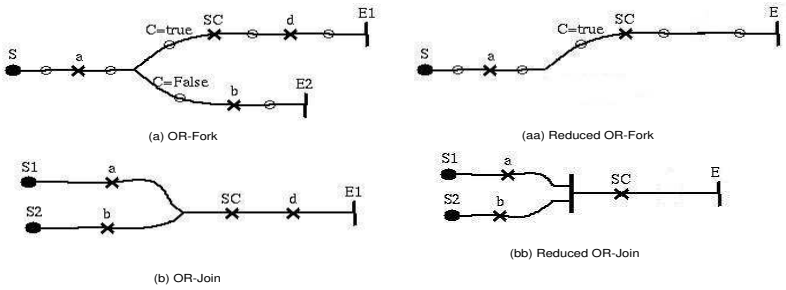


Fig. 4. UCM constructs and its reduced form(1) (SC is the slicing criterion)

Figure 5(gg) shows the slice obtained for a UCM with a dynamic stub. The selection policy between plug-in1 and plug-in 2 is based on the value of global

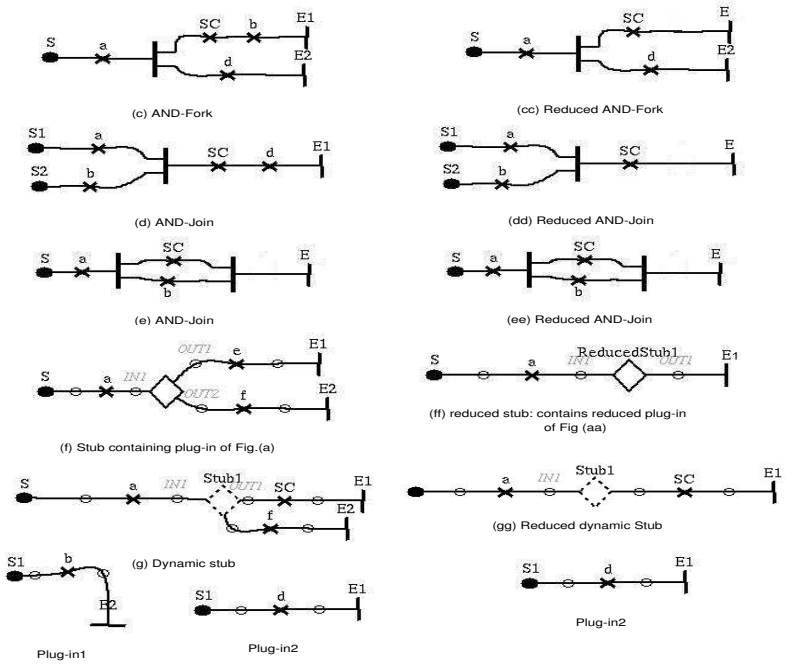


Fig. 5. UCM constructs and its reduced form(2) (SC is the slicing criterion)

variable C:(1) C=*true*→ Plug-in 1 (connects IN1 to OUT1)(2) C=*false* → Plug-in 2 (connects IN1 to OUT2). Plug-in 1 is sliced out because its end point is bound to end point E2. The resulting stub is a reduced stub with only one exit point *E1* containing plug-in 2. In this case, the *reachability expression* is reduced to the plug-in selection condition: *C=false*.

4.4 UCM Slicing Algorithm

In what follows, we present our UCM slicing algorithm, which is based on a backward traversal of the UCM specification. Figure 6 describes the high level schema of the UCM Slicing algorithm.

Logical conditions are collected as the traversal progresses. Each stub defines a level of abstraction and is treated separately. Therefore, we obtain reduced stubs at different abstraction levels. Since a plug-in can be installed in many stubs according to the chosen scenario, the user is asked to provide the targeted stub to which the SC belongs. This information is essential because of the '*many-to-many*' association between plug-ins and stubs.

It should be noted that the presented algorithm is not necessarily the most time and space efficient approach to compute UCM slices. The algorithm will terminate due to the backward traversal step and the fact that there is a finite number of responsibilities in the UCM.

Input:UCM + slicing criterion SC(Responsibility or start/end point)
Output: Reduced UCM, Reachability Expression
Step1:(*Searching SC*)

```
Traversal of the all UCM maps using a depth first algorithm
IF (SC not found) THEN notify the user;exit
ELSE IF (SC part of the root map) THEN Go to step 2
      ELSE Read(targetStub) (*User is asked to provide the targeted stub*)
           IF (plug in is part of Dynamic stub) THEN
             globalReachability := selectionCond (*Selection policy for dynamic stubs*) ENDIF
     Point to TargetedStub and start at SC
     Go to step 2 ENDIF
ENDIF
```

Step2:(*UCM Backward Traversal: executed for every single path (recursively); Read Previous element, collect conditions, etc. use access functions defined over the different sets of the UCM definition*)

```
WHILE (not(startPoint)) DO Read_previsous(element)
rootStack := rootStack + ((element) or (stub to which the SC belongs))
IF (OR-Fork) THEN reachabilityExpression:=reachabilityExpression AND (OR-Fork condition)ENDIF
IF (OR-JOIN) THEN  FOR (each alternative path i) create new stack (Stack_i)
                    to handle the alternative path and repeat step2 (recursive traversal)ENDIF
If (AND-Join) THEN FOR (each concurrent path) create new stack (stack_j)
                    to handle the concurrent path and repeat step2 (recursive traversal)ENDIF
IF (AND-Fork) THEN FOR (each concurrent path) create new stack (stack_k)
   perform a forward traversal and save elements in the stack til reaching the end points ENDIF
IF (Static Stub) THEN rootStack := rootStack + StubName ENDIF
IF (Dynamic Stub) THEN select only plugins bound to the exit point
                       from which the backward traversal entered the stub ENDIFENDWHILE
```

Step3:(*Construct the UCM slice: Convert stacks into sequences*)

```
UCMSlice:= (generic end point, SC) // Initialize Slice
FOR (each stack) DO
      stack.pop(element)
      UCMSlice := UCMSlice + (element.construct)
      IF (stack of dynamic plug-in) THEN
         Add for each plug-in a new alternative to the reachability expression
         global_ReachExp := globalReachExp OR (selection condition)
      ENDIF
ENDFOR
```

Fig. 6. Slicing algorithm for unbound UCMs

4.5 Solving the Reachability Expression

The Boolean Satisfiability Problem (SAT). The resulting slice is considered to be correct if and only if the set of computed conditions are satisfied. Given a reachability expression the question is: *Exist there any true/false assignments that will change the entire expression to true?*. Since UCM deals only with boolean variables, the reachability problem can be reduced to an instance of the boolean Satisfiability Problem (SAT) [3]. SAT is the first known NP-complete problem, as proved by *Stephen Cook* [3] in 1971. There are many approaches for solving instances of SAT in practice. Just to name few: Davis-Putnam, GRASP, WALKSAT, GSAT, CHAFF and SATO. Finding a solution to the reachability

expression is outside the scope of this paper. For a detailed coverage of this problem refer to [6].

Conflicting Conditions and Non-Determinism. We may obtain unsatisfiable reachability expressions in the following situations:

1. Conflicting conditions: unsatisfiable set of conditions in successive alternatives found in OR-Forks (For example: C1 and ¬C1), in selection policies of nested dynamic stubs, etc.
2. Non-determinism
   UCMs may contain some non-deterministic behavior due to overlapping conditions (For example: in an OR-Fork, conditions *Cond1:(C1=true)* and *Cond2:(C1=true and C2=true)* overlap when *C2=true*. This will result in a non-deterministic execution. Hence, the resulting initial condition does not guarantee the execution of the computed slice.

Note: Parnas tables can be applied at specification time to determine, if a collection of conditions is deterministic and complete [15].

5 Discussion on the Application of UCM Slices

UCM slices help analyse to what extent the behavior and/or architecture of the system might be affected by a specific maintenance task. For each slice, a maintainer can identify the part of the particular scenario that contributes to the slicing criterion (on both architectural and behavioral parts).

5.1 Applying UCM Slicing for the Simple Telephone System

In what follows, we apply the slicing algorithm for the Simple Telephone System presented in Section 3.2. Suppose that we want to perform an upgrade to the CND feature. The upgrade will involve the display not only of the Caller's name but also his/her service provider. This maintenance task cannot take place until the maintainer understands how the particular feature works and how it interacts with other system features. Knowing all the details of the requirement specification is almost never necessary; an experienced maintainer will try to extract only just enough information to perform the task at hand. The goal is to extract the scenarios leading to the display function (responsibility). Hence, the slicing criterion is the responsibility *display*.

Figure 7 describes the resulting UCM obtained from the original UCM of figure 1 with respect to the slicing criteria *display*. Figure 8 shows the corresponding *Reachability Expression*.The first part of the reachability expression ((1) in Fig 8) illustrates the fact that the default plug-in is selected *(subOCS = false)* and the second part of the expression ((2) in Fig 8) expresses the fact that the OCS plug-in was selected.

In our example the Reachability Expression itself provides the initial values of global variables leading to the slicing criterion and no further computation is needed.

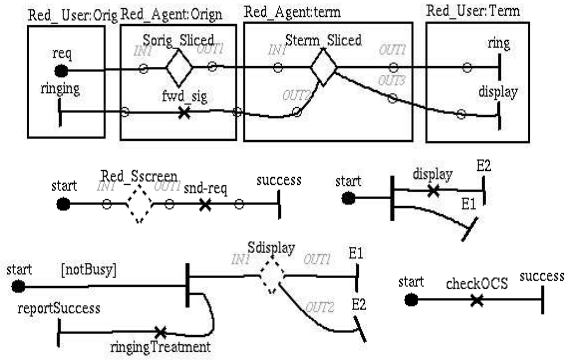


Fig. 7. Simple telephony system slice with respect to SC:*display*

((subCND = true) AND (Busy =false) AND (subOCS = false)) (1)
OR
((subCND=true) AND (Busy=false) AND (subOCS=true) AND
(OnOCSList=false)) (2)

Fig. 8. Reachability equation for responsibility *display*

6 UCM Data Flow

6.1 Variable Assignment

So far, global boolean variables were assigned values only at initialization time. However, UCM responsibilities and end points may affect the content of value identifiers ("⟵" denotes the assignment operator). As a result, the reachability expression may not hold and the correctness of the computed slice is affected.

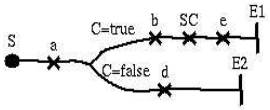


Fig. 9. Responsibilities updating boolean variables

Case1: Suppose that in the UCM of figure 9, responsibility a:C⟵ ¬C. Consequently, the new definition of variable C should be considered in the reachability expression : C = *true*, C ⟵ ¬C.

Case2: Suppose that in the UCM of figure 9, responsibility b:C⟵ ¬C. The update happened after a path has been taken. The reachability expression should not be affected and should remain: C=*true*.

This mixture of predicates and assignment statements should be eliminated before applying a satisfiability algorithm [6]. In order to obtain a reachability

expression containing only predicates, we substitute the affected variable of the assignment statement in the logical prediactes(also called *unification*). For example: C = *true*, C⟵ ¬C ⟹ *true*=¬C. This problem is formalised and solved by the two following rules:

Rule 1 *If a variable has been assigned a new value before participating in a choice condition, then the variables of the choice are are substituted with the new variable assignment.*

$$v \longleftarrow f(x_1,\ldots,x_n),\ g(y_1,\ldots,y_n,v) \Longrightarrow g(y_1,\ldots,y_n,f(x_1,\ldots,x_n))$$

where v is a boolean variable, f and g are logical expressions.

Rule 2 *If a variable has been assigned a new value after participating in a choice condition, the predicate condition is retained in the reachability expression and the assignment is ignored.*

$$g(y_1,\ldots,y_n,v), v \longleftarrow f(x_1,\ldots,x_n) \Longrightarrow g(y_1,\ldots,y_n,v)$$

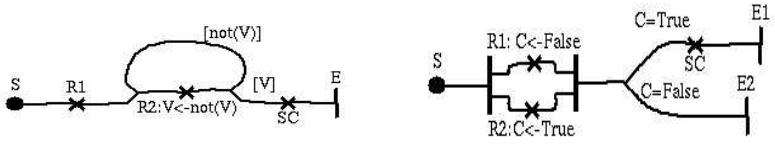
where v is a boolean variable, f and g are logical expressions.

6.2 Limitations

While the underlined rules are easy to apply and help reducing the reachability expression, they are not applicable in the following circumstances:

Loops. When a UCM contains loops, the number of times a loop is visited is known only at run time. Such information, which depends on the variable's initial values and guard's evaluation, is needed in order to compute the slice and to solve the reachability expression. For example, in the simple UCM of Figure 10(a), the number of times the loop is entered (zero or one time) is not available when the backward traversal is performed.

In a more complex situation, where instead of having only a responsibility like R2 of Figure 10(a) we have a dynamic stub where the selection of plug-ins depends on the values of the variables at run-time. Hence, non executable plug-ins may be part of the resulting slice whereas they should be left out.



(a) UCM with a loop (b)Non-deterministic UCM

Fig. 10. Special cases

Non-Determinism. Figure 10(b) shows a UCM with two interleaving responsibilities R1 and R2. SC is reached only when R2 is executed after R1. One possible option is to investigate all possible alternatives (i.e., execution paths). Each alternative will be evaluated separately and considered in the resulting slice if it is a consistent one. Therefore, the resulting slice will be the union of all consistent executions. Another option is to keep the non-determinism. Then the user can analyze the resulting slice and make the appropriate decision.

7 Related Work

Our work on UCM slicing builds on from prior work in the following two primary areas: functional requirement slicing and architectural slicing.

7.1 Slicing of Hierarchical State Machines

Heimdahl *et al.* [7] apply slicing to the requirement specification language RSML (Requirement State Machine Language). Their proposed method consists on reducing the requirement specification based on a specific scenario of interest. The reduced specification contains only the behaviors that are possible when the operating conditions defining the reduction scenario are satisfied. Such a reduced specification is called the *interpretation* of the specification under this scenario. Next, the produced interpretation is sliced based on different entities in the model to highlight the portions of the specification affecting an output variable or a specific transition. This is achieved through a data and control flow information analysis. The slices can be arbitrarily combined using standard set of operations to construct a combined slice containing the information of interests.

7.2 Slicing of State Based Models

Korel *et al.* [11] presented an approach of slicing EFSM (Extended Finite State Machines) models. Their approach produces an EFSM slice based on EFSM dependence analysis. The resulting slice may further be reduced by merging states and transitions to construct a non-deterministic EFSM. This is called non-deterministic slicing.

RSML and EFSM slicing emphasizes only the behavioral part of the requirement specification. The architectural part is left aside. Use Case Maps scenarios combine both aspects (i.e., behavioral and architectural) in a single representation. Our proposed technique took advantage of this dual representation.

7.3 Architectural Slicing

Zhao [20] introduced a new form of slicing called *the Architectural slicing* to aid architectural understanding and reuse. He applied slicing to an architectural specification of a software system written in WRIGHT, which is an Architectural

Description Language (ADL). A Wright architectural specification of a system is defined by a set of component and connector type definitions, a set of instantiations of specific objects of these types, and a set of attachments. Attachments specify which components are linked to which connectors. Each component has an interface defined by a set of ports and each connector has an interface defined by a set of roles. In order to compute an architectural slice, an Architecture information flow graph is constructed then a traversal algorithm is applied. The reduced architectural description contains only the lines of ADL code that could be associated with a particular slicing criterion. In [17, 20] the slicing criterion is either a set of ports of a component or a set of roles of a connector. Stafford *et al.* [17] presented a closely related method to Zhao's work. They introduced a software architecture dependency technique called chaining. Their work consists on extracting a chain of dependences (called links) between the specification's elements based on a set of ports of a component (slicing criterion).

8 Conclusion

In summary, our approach for slicing Use Case Maps allows an analyst to reduce a requirement specification based on a selected slicing criterion. Our approach is *two tiered*. First, we allow an analyst to reduce a UCM specification according to a slicing criterion. Second, a reachability expression is attached to the slice, which provides insight on the feasibility of the selected scenarios. We illustrated potential uses of UCM slicing in testing and requirement comprehension of complex specification, by reducing the complexity of the given specification. Furthermore, we see potential application domains for UCM slicing in feature extraction, impact analysis, and reuse of requirements. In fact, while reuse of code is important, more significant improvements in productivity and quality can be expected from reuse of software designs and requirement patterns. By slicing a UCM requirement, a system designer can extract reusable parts from it, and reuse them into new system designs for which they are appropriate. As part of our future work, we will investigate the use of dynamic slicing that may significantly reduce the size of a UCM slice. Providing inputs helps reducing the domain of the UCM and only the parts that comply with the input values are kept in the final slice. Consequently, the reachability expression is also reduced. We are currently investigating those research directions.

Finally the approach outlined in this paper is not limited to Use Case Maps specifications. The approach is general enough to be applied to all languages with guarded transitions such as activity diagrams part of UML.

References

1. Amyot, D., Eberlein, A.: An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. Telecommunications Systems Journal, 24:1, 61–94, September 2003.

2. Chang, J., Richardson, D.G.: Static and dynamic specification slicing. Proceedings of the Fourth Irvine Software Symposium, April 1994.
3. Cook, S.: The complexity of theorem proving procedures. Proc. 3rd ACM Symp. On Theory of Computing (1971) 151–158.
4. De Lucia, A.: Program slicing: Methods and applications. 1st IEEE International Workshop on Source Code Analysis and Manipulation (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142–149.
5. Gallagher, K.B., Lyle, J.R.: Using program slicing in software maintenance. IEEE Transactions on Software Engineering, SE-17(8): 751-761, August 1991.
6. Gu, J., Purdom, P.W., Franco, J., Wah, B.W.: Algorithms for satisfiability (SAT) problem: A survey. DIMACS Volume Series on Discrete Mathematics and Theoretical Computer Science, 1996.
7. Heimdahl, M.P.E., Whalen, M.W.: Reduction and Slicing of Hierarchical State Machines. Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, Zurich, Switzerland, 1997, pp. 450–467.
8. ITU-T, URN Focus Group (2002): Draft Rec. Z.152 - UCM: Use Case Map Notation (UCM). Geneva. http://www.UseCaseMaps.org/urn/
9. ITU-T: Recommendation Z.150 (02/03), User Requirements Notation (URN) - Language Requirements and Framework. International Telecommunication Union, Geneva.
10. Korel, B., Laski, J.: Dynamic program slicing. Process. Letters, 29(3), Oct. 1988, pp. 155–163.
11. Korel, B., Singh, I., Tahat, L., Vaysburg, B.: Slicing of State-Based Models. Proceedings of the International Conference on Software Maintenance 2003. IEEE Computer Society, Washington, USA.
12. Miga, A., Amyot, D., Bordeleau, F., Cameron, C., Woodside, M.: Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. Reed, R., Reed, J. (Eds) 10th SDL Forum (SDL'01), Copenhagen, 2001. Volume 2078 of Lecture Notes in Computer Science, 268–287.
13. Millett, L., Teitelbaum, T.: Slicing Promela and its applications to model checking. Proceedings of the 4th International SPIN Workshop, 1998.
14. Oda, T., Araki, K.: Specification slicing in formal methods of software engineering. Proceedings of the Seventeenth International Computer Software and Application Conference, November 1993.
15. Parnas, D.L., Madly, J., Iglewski, M.: Precise Documentations of Well-Structered Programs. IEEE Transactions on Software Engineering, Volume 20, Number 12 (December 1994), 948–976.
16. Sloane, A.M., Holdsworth, J.: Beyond traditional program slicing. Zeil, S.J. (Ed) Proceedings of the 1996 International Symposium on Software Testing and Analysis, 180–186, New York, January 1996. ACM Press.
17. Stafford, J.A., Wolf, A.L.: Architecture-Level Dependence Analysis in Support of Software Maintenance. Proceedings of the Third International Workshop on Software Architecture, November 1998, pp. 129–132.
18. Tip, T.: A survey of program slicing techniques. Journal of programming languages, 3:121–189, 1995.
19. Weiser, M.: Program slicing. IEEE Transactions on software Engineering, SE-10(4):352–357, July 1984.
20. Zhao, J.: Applying slicing techniques to software architectures. 4th IEEE Int. Conf on Engineering of Complex Computer Systems, 1998, Monterey, California, and 5th European Conf. on Software Maintenance and Reengineering (CSMR01).