Applying Reduction Techniques to Software Functional Requirement Specifications (Use Case Maps Slicing)

Jameleddine Hassine
Rachida Dssouli
Juergen Rilling
Concordia University, Montreal, Canada

SAM'04

Fourth SDL And MSC Workshop

Outline

- Part I: Traditional Program Slicing
  - Introduction
  - Program Slicing
  - Slicing Example
  - Generalized Slicing
- Part II: Use Case Maps
  - What is Use Case Maps
  - Design Pyramid
  - UCM Definition
  - Example
- Part III: UCM Slicing Approach
  - Need for Requirement Slicing
  - Slicing Criteria
  - UCM Slicing
  - Limitations
  - Conclusion & Future work

Part I
Traditional Program Slicing

Introduction

- Originally Introduced by Weiser in 1984
- Program Reduction Technique (Simplification Technique)
- Studied primarily in the context of conventional programming languages (C, ADA,..etc.)
- Application of program slicing:
  - Debugging
  - Differencing
  - Program Testing
  - Program Maintenance (Comprehension, Analysis, …etc.)
  - Reverse Engineering
  - Formal Verification

Program Slicing ?

Given:

- **Program** (in a conventional programming language such as C)
- Variable **V** at some point **P** in the program (Called a slicing **Criterion**)

Goal:

Find the part of the program that is responsible for the computation of variable v at point P

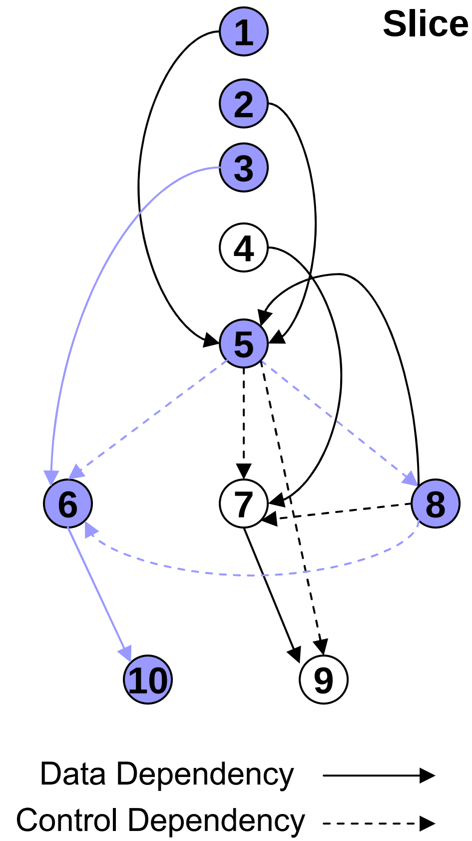
Output : Slice (Weiser's Definition 1984)

A Slice S is a Reduced, executable program obtained from program PG by removing statements such as S replicates parts of the behavior of PG.

Slicing Example

- **Data Dependency:** Represents data flow (definition-use chain).

- **Control Dependency:** The execution of a node depends on the outcome of a predicate node.



Program Dependency Graph

Slice w.r.t criterion <10, sum>:

begin
1 read(n)
2 i:=1;
3 sum:=0;
4 prod := 1
5 While (i<=n) do
6 sum:=sum+i;
7 prod:=prod*i;
8 i:=i+1;
end;
9 write(prod);
10 write(sum);
end;

Generalized Slicing

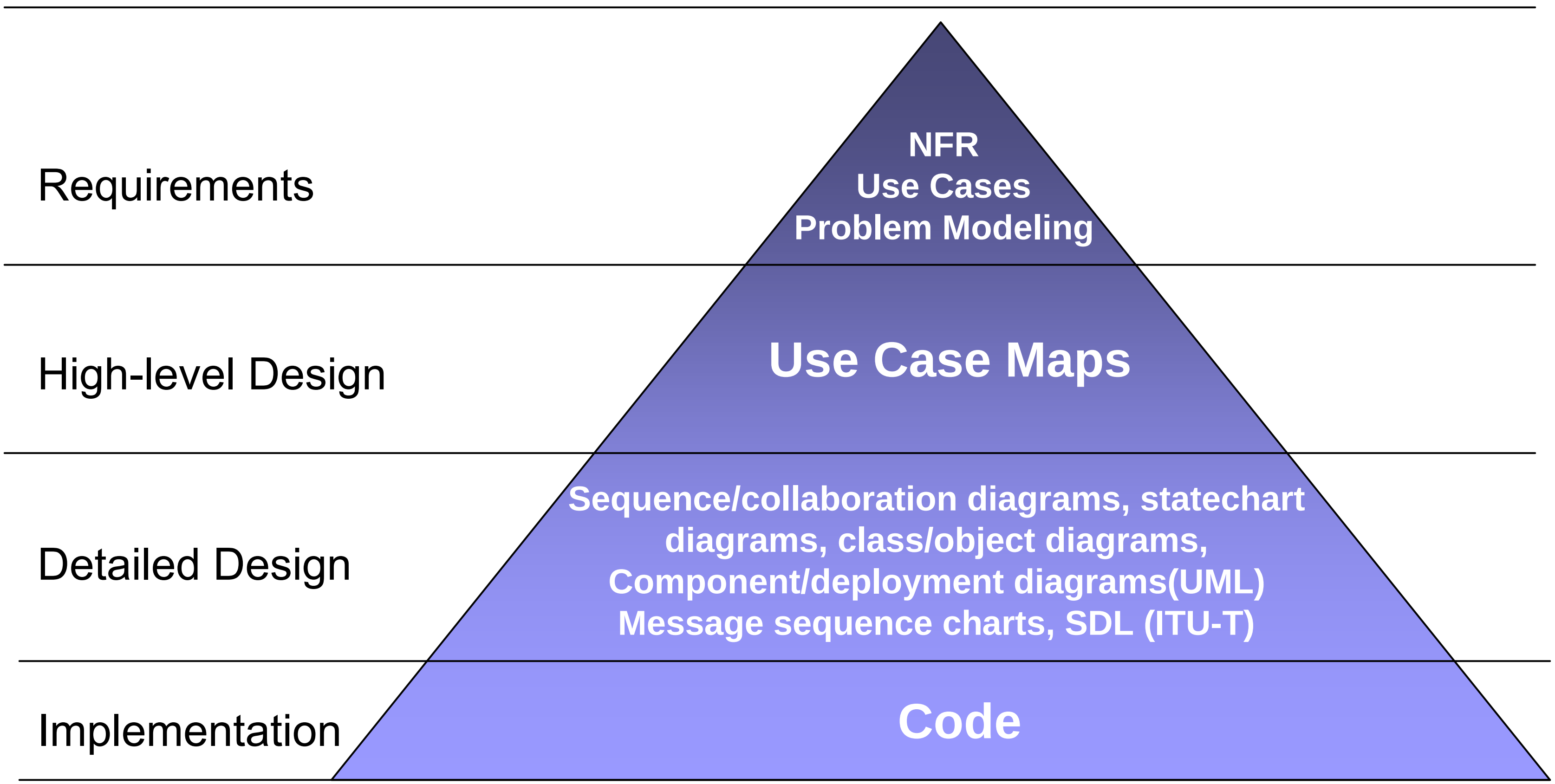
- Slicing has been generalized to other software artifacts including :
  - Requirement models: Requirement State Machine Lamguage (RSML), Extended Finite State Machine (EFSM)
  - Software Architecture (Language WRIGHT (ADL)).
  - Specification Languages (Z, VHDL)
  - Grammar
  - ..etc.

Part II
Use Case Maps

What is Use Case Maps (UCMs) ?

- A graphical **scenario** notation (map-like diagram)
- Describes system functional requirements
- Reason about the system at a high-abstraction level (without reference to message exchanges)
- Facilitate moving towards design
- UCM part of URN (User Requirement Notation, Being standardized by ITU-T in Z.15x)

The Design Pyramid



Strengths of UCM

- Bridge the modeling gap between requirements (use cases) and detailed design
- May be transformed (e.g. into MSC/sequence diagrams, performance models, test cases)
- Model dynamic (run-time) refinement for variations of behaviour and structure
- Visually integrate behaviour and structural components in a single view.

UCM Definition

- A UCM requirement specification is defined as a seventuple (D, C, V, λ, Bc, S, Bs)

Where:

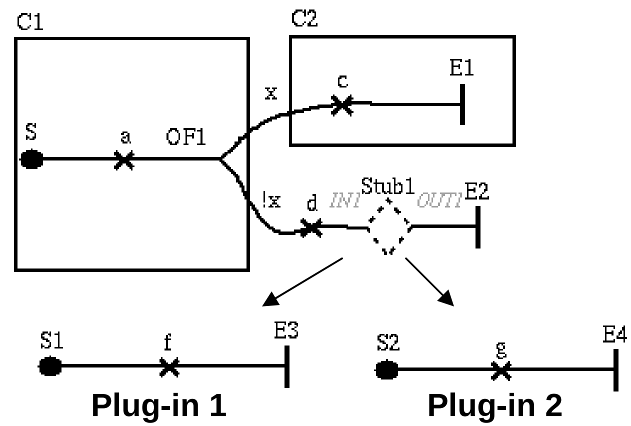
- D is the UCM domain, composed of sets of typed constructs.

  $D = R \cup SP \cup EP \cup AF \cup AJ \cup OF \cup OJ \cup AF \cup ST \cup Tm \cup ST \cup$…etc

  Where R: Responsibilities, SP: Start Points, EP: End points, AF: AND-fork, AJ: AND-join, OF:OR-fork, OJ : OR-Join, AF: AND-fork, ST: Stubs…etc.
- C is the set of components ($C = \emptyset$ for unbound UCM)
- V is the set of global variables,
- G is the set of guard expressions over V,
- λ is a transition relation (path connection) defined as: $\lambda = D \times D \times G$
- Bc is a component binding relation and is defined as $Bc = D \times C$.
- S is a Stub binding relation defined as $S = ST \times RS \times G$.
- Bs is a Plug-in binding relation defined as :

  $Bs = RS \times \{IN/OUT\} \times SP/EP$.

Example



- D = {S} ∪ {E1, E2} ∪ {a, c, d} ∪ {OF1} ∪ {Stub1}
- C = {C1, C2}
- V = {x, y}
- G = {x, !x, y, !y,…etc.}
- 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= {(Plug-in1,IN1, S1), (Plug-in1,OUT1, E3), (Plug-in2, IN1, S2), (Plug-in2, OUT1, E4)}

Part II
UCM Slicing Approach

Need For Requirement Specification Slicing

- Requirement Modeling and analysis represent a critical phase of complex system development
- Requirements are evolving ⇨ Complex and error-prone
- Extract only just enough information to perform the task at hand (focus on some parts and ignore others)
- Come up with Techniques and Tools to support requirement:
  - Analysis
  - Comprehension
  - Testing
  - Maintenance

Slicing Criteria & Reduced UCM

- UCM Slicing Criterion:
  - A responsibility or start/end point (A component may be part of the slicing criterion)

- Reduced UCM: RS'= (D', C', V', λ', Bc', S', Bs')
  - D' is a reduced set of D
  - C' is a reduced set of C (a component with reduced functionalities)
  - V' is a reduced set of V
  - λ' is a reduced transition relation
  - Bc' is a reduced component binding relation
  - S' is a reduced Stub binding relation
  - Bs' is a reduced Plug-in binding relation

UCM Slicing

- Input:
  - A UCM
  - Slicing criteria (SC)

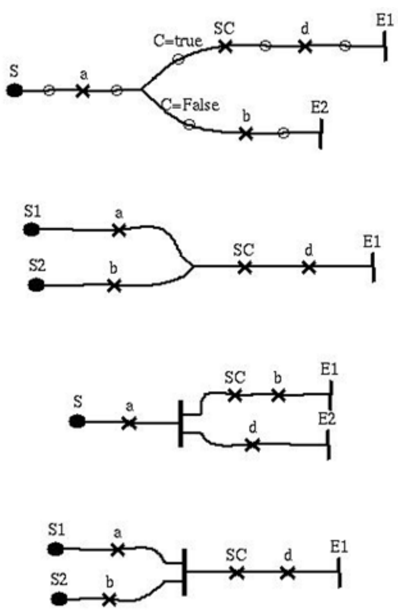
- Output:
  - Reduced UCM (Backward Slice)
  - Reachability expression: A logical expression combining guards (first-order logic predicates)

Note: In order to reach SC, the reachability expression should be satisfiable (i.e. evaluated to : *True*)

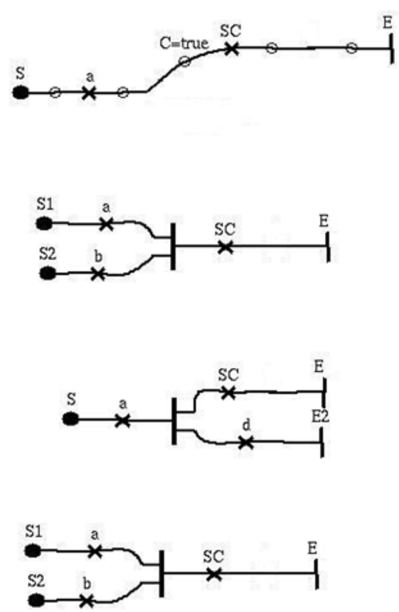
Solving the Reachability expression

- Is there some assignment of "true" and "false" values to the variables that will make the entire expression "true"?
- Satisfiability Problem (SAT) ⇨ NP-complete problem
- UCM Boolean variables ⇨ Boolean Satisfiability Problem
- Many approaches for solving instances of SAT in practice: Davis-Putnam, WALKSAT, GSAT...etc.

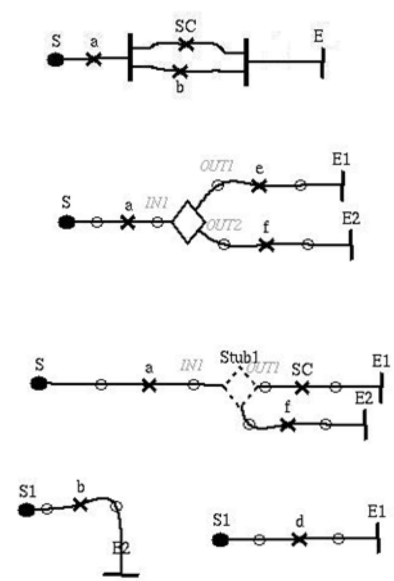
Slicing UCM Constructs



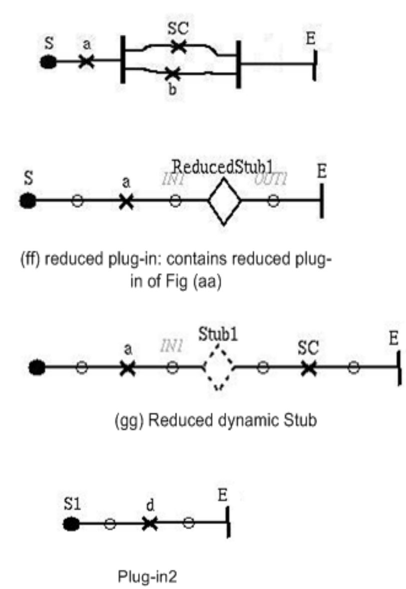
UCM construct



Reduced UCM construct

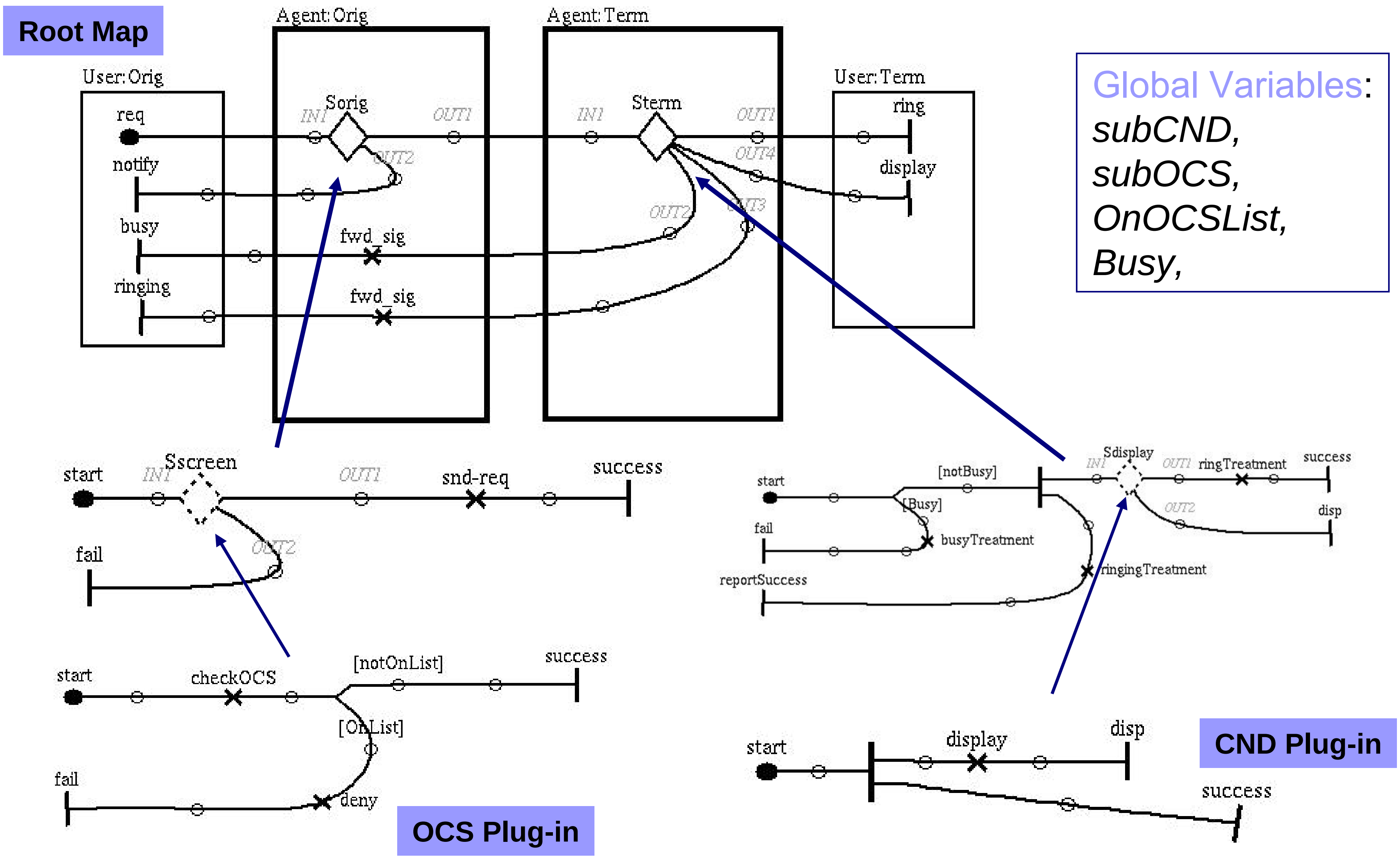


UCM construct



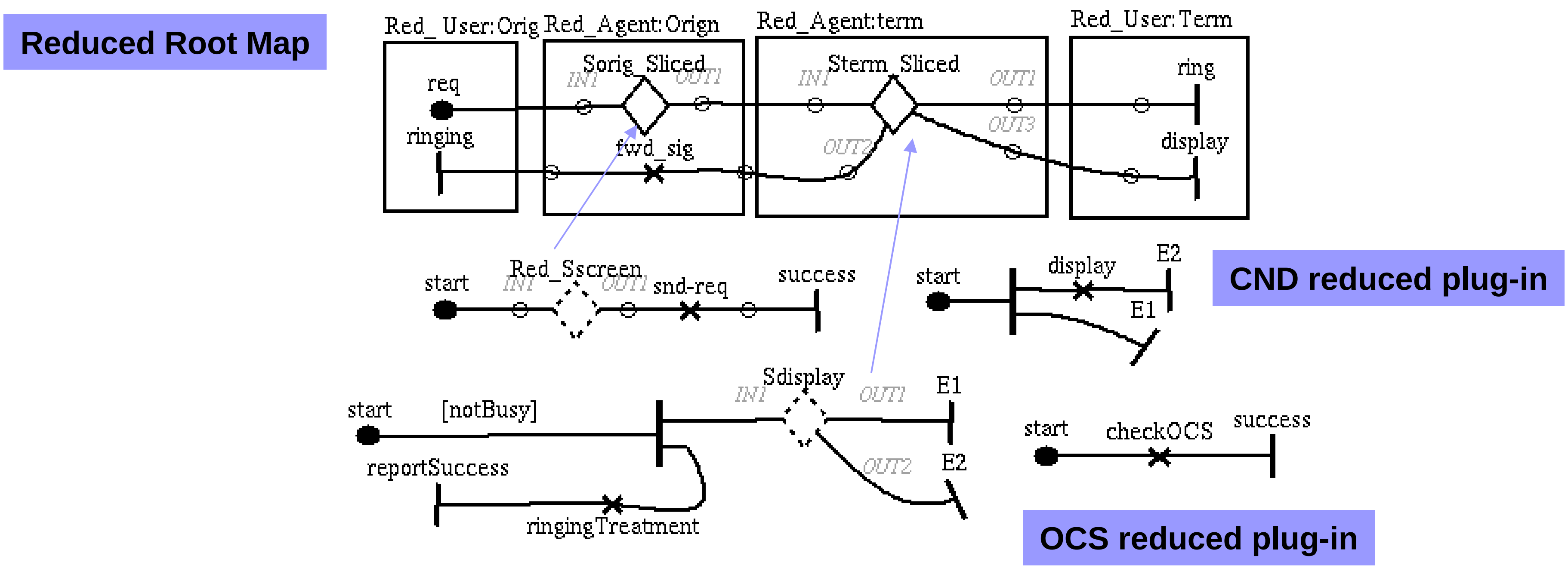
Reduced UCM construct

Case Study: A Simple Telephony System



Global Variables:
subCND,
subOCS,
OnOCSList,
Busy,

Example: SC = 'display' in the CND stub



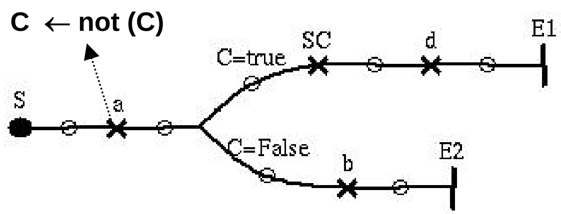
Reachability Expression:

((subCND = True) **AND** (Busy =False) **AND** (subOCS = False))

OR

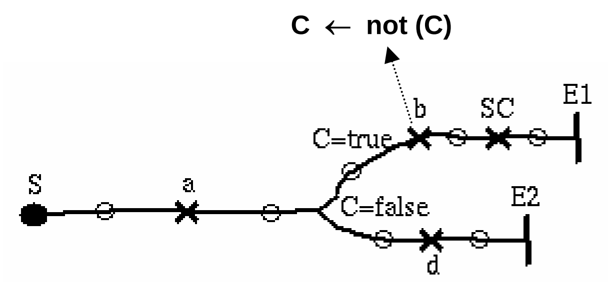
((subCND = True) **AND** (Busy =False) **AND** (subOCS = True) **AND** (OnOCSList = False))

Variable Assignment



Case1: the new definition of variable *C* should be considered in the reachability expression : {(C ← not(C)), (C= true)}
After Unification:
True = not(C)

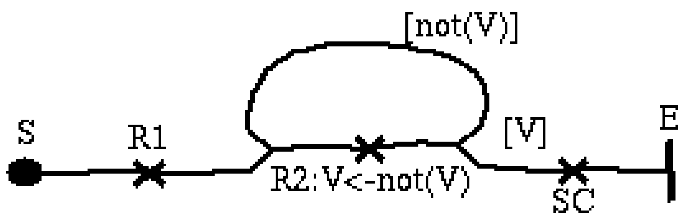
Rule1: v ← f(x1,..,xn) ; g(y1,..,yn,v)	⇨	g(y1,..,yn,f(x1,..,xn))



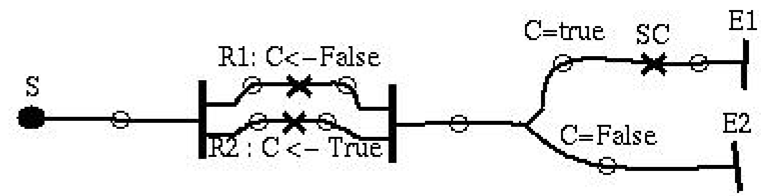
Case2: The update happened after a path has been taken. The reachability expression should not be affected and should remain: **C = *true***

Rule2: g(y1,..,yn,v) ; v ← f(x1,..,xn)	⇨	g(y1,..,yn,v)

Limitations



Loops



Non-determinism

- Loops: The number of times a loop is visited is known only at run time. Such information is needed in order to compute the slice and to solve the reachability expression.
- Non-determinism: SC is reached only when R2 is executed after R1. One possible option is to investigate both alternatives. Each alternative will be evaluated separately and taken as a slice if it is a consistent one.

Conclusion & Future work

- **Benefits**
  - Requirement understanding and analysis (Complexity reduction (search into a hierarchy of levels of abstraction (Stubs)), Feature extraction…etc.)
  - No state explosion, since UCM original semantics are preserved (Concurrency, non determinism)
  - Testing (Regression testing, development testing)
  - Maintenance (Corrective, perfective, Impact analysis…etc.)

- **Future Work**
  - Derive test suites based on slicing (Selective testing, Regression testing)
  - Dynamic Slicing (Reduces the size of a slice and simplifies the reachability expression)
  - Impact Analysis (Combine backward and forward slicing)