Conducting Requirements Evolution by Replacing Components in the Current System

Haruhiko Kaiya & Kenji Kaijiri
Shinshu University, Japan

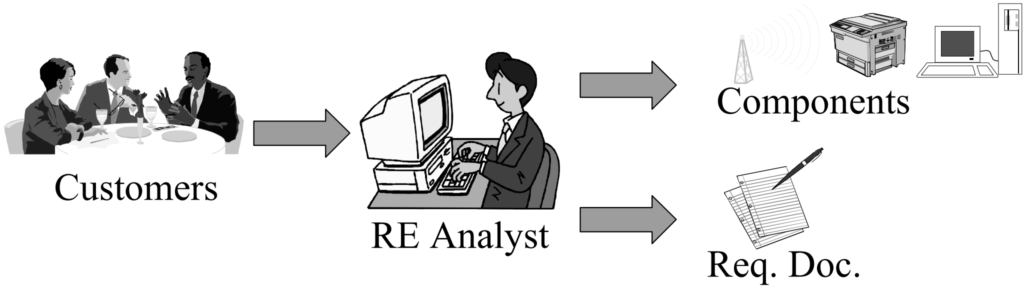
Dec. 8, 1999

Outline of this Presetation

- What kind of Requirements Acquisition?
- Basic techniques and concenpts:
  Activity Digaram, Design by Contract, Spec. Match.
- How to encourage requirements evolution by the Component Change?
- Requirements Evalution:
  Rules and Procedure.
- Example.
- Conclusion and Discussion.

Requirements Acquisition

Traditional Requirements Definition:



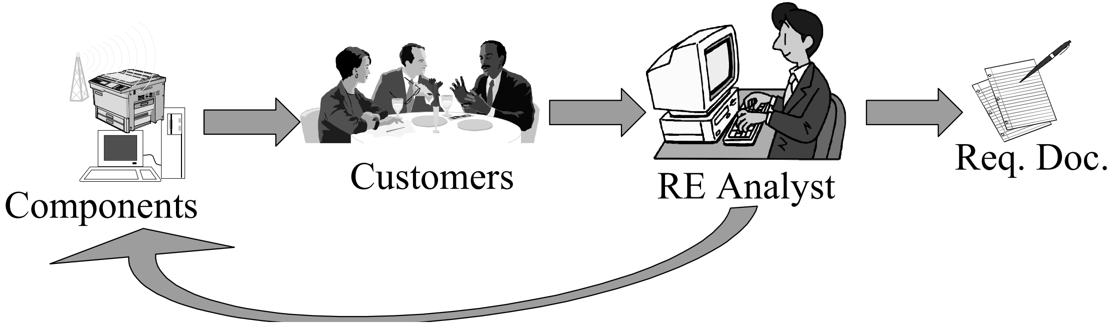
Customers

RE Analyst

Components

Req. Doc.

Requirements Definition We Intended:



Components

Customers

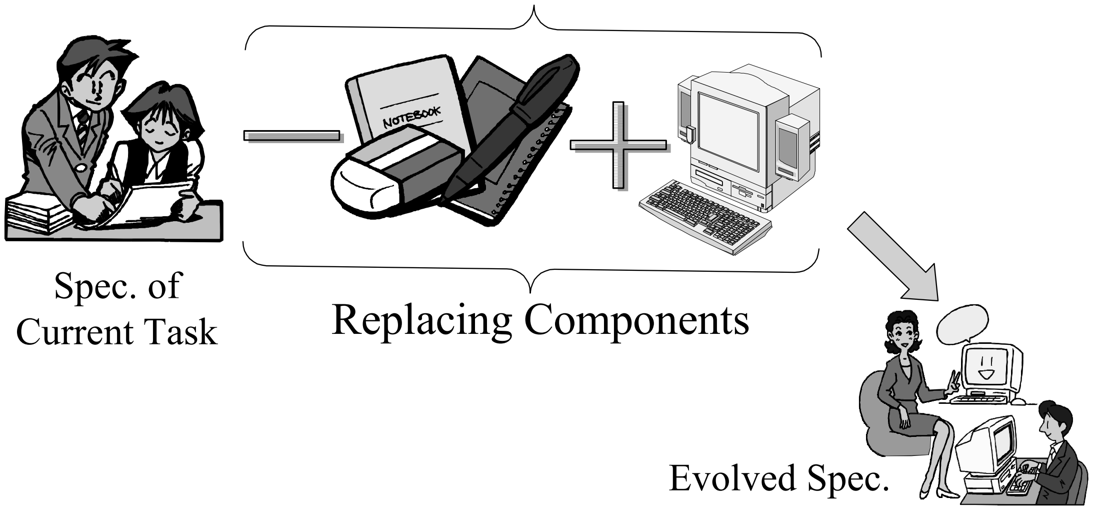
RE Analyst

Req. Doc.

New Technologies (Components)

can encourage the evolution of the Tasks (Requirements)!!

Requirements evolution by the Component Change



Spec. of
Current Task

Replacing Components

Evolved Spec.

- How to represent the task to specify.
- How to find alternatives of the current components.
- How to clarify the differences of them.
- How to exploare new possibilities of the task.

Basic Techniques and Concepts

- Activity Diagram in UML –

  Representing the structure of req. spec.
- Pre/Post specification –

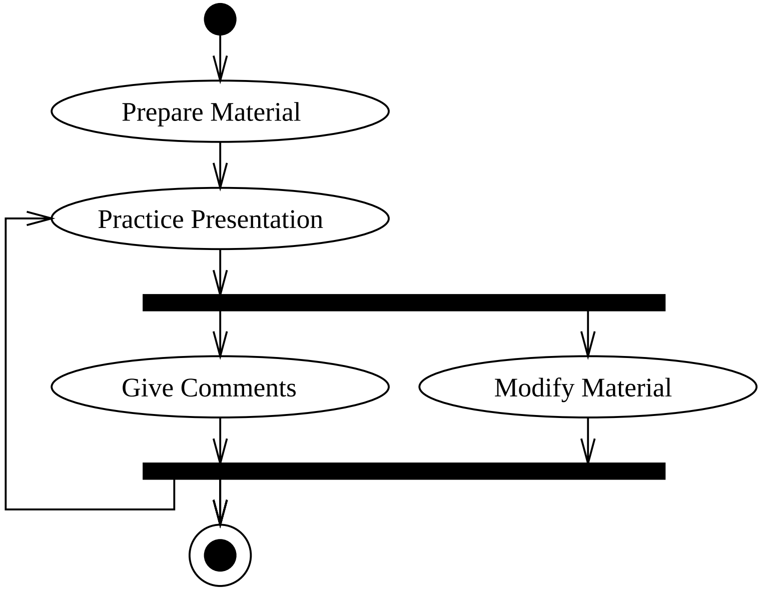
  Specifying each component.
- Design by Contract (DBC)–

  Invariant during evolution.
- Specification Matching –

  Finding the alternative components and clarify the difference.

Activity Diagram

- An activity diagram shows a sequential flow of activities.
- Similar to a flow-chart and a petri-net.
- It can be used for representing scenario of users and system.
- We regard each activity as replaceable component.



H. Eriksson and M. Penker. UML Toolkit.

Pre/Post Spec. for Components & Design by Contract(DBC)

- Component: *a funtion*.
- Pre/Post specification: traditional way to specify a function.
- Pre-condition: specify the responsibilities of the component users, i.e. caller's responsibilities.
- Post-condition: specify the responsibilities of the component itself.
- Non-Redundancy principle (of DBC):
  A component *should not* guarantee its pre-condition,
  and *only* the callers of the component *should* guarantee the pre-condition.

Bertrand Meyer. Object-oriented software construction, 2nd edition. Prentice Hall, 1997, p.412.

Specification Matching (1/2)

- Pre/Post Match is one of the matchings for components presented by Zaremski.
- $match(S, Q) = (Q_{pre}\mathcal{R}_1 S_{pre}) \wedge (\hat{S}\,\mathcal{R}_2 Q_{post})$
  .... predicate for deciding match or not.

Match	Predicate Symbol	$\mathscr{R}_1$	$\mathscr{R}_2$	$\hat{S}$
Exact pre/post	$match_{E\text{-}pre/post}$	$\Leftrightarrow$	$\Leftrightarrow$	S_{post}
Plug-in	$match_{plug\text{-}in}$	$\Rightarrow$	$\Rightarrow$	S_{post}
Plug-in post	$match_{plug\text{-}in\text{-}post}$	$*$	$\Rightarrow$	S_{post}
Guarded plug-in	$match_{guarded\text{-}plug\text{-}in}$	$\Rightarrow$	$\Rightarrow$	$S_{pre} \wedge S_{post}$
Guarded post	$match_{guarded\text{-}post}$	$*$	$\Rightarrow$	$S_{pre} \wedge S_{post}$
		$*$: dropped		

Q: query function. S: library function.

Amy Moormann Zaremski and Jeannette M. Wing. Specification Matching of Software Components. ACM TOSEM, Vol. 6, No. 4, pp. 333-369, Oct. 1997.

Specification Matching (2/2), Example

Plug-In Match: $match(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$

- BoundedBug's *add* operation (Query):
  $pre.add \mathrel{\widehat=} \left[\, s : \mathrm{seq}\, X \,\middle|\, \#s < 50 \,\right]$
  $post.add \mathrel{\widehat=} \left[\, \Delta s : \mathrm{seq}\, X;\ e? : X \,\middle|\, \#s' = \#s + 1 \,\right]$
- Stack's *push* operation (in Library):
  $pre.push \mathrel{\widehat=} \mathrm{true}$
  $post.push \mathrel{\widehat=} \left[\, \Delta s : \mathrm{seq}\, X;\ e? : X \,\middle|\, s' = s \frown \langle e? \rangle \,\right]$

Then

$$match(push, add)$$

is hold. i.e. ***add* is matched by *push***, because

- $pre.add \Rightarrow pre.push$
- $post.push \Rightarrow post.add$

Beyond the Matching – Requirements Evolution

- $Q_{pre} \Rightarrow S_{pre}$ and $S_{post} \Rightarrow Q_{post}$
  - Under guard and Over functionality of a component.
  - Redundant properties – against the mind of DBC.
- $Q_{pre} \Leftarrow S_{pre}$ and $S_{post} \Leftarrow Q_{post}$
  - These are NOT match.
  - Over guard and Under functionality.

Requirements Evolution

- by modifying the topology of the Activity Diagram
- or by replacing precedent and/or succeeding components.

We rename $match_{pre/post}(S, Q)$ as ***Evolutional Predicate, evolve(S,Q)***.

Rule for exploring new Possible Requirements

$$\mathit{evolve}(S, Q) = (Q_{\mathit{pre}}\ \mathcal{R}_1\ S_{\mathit{pre}}) \wedge (S_{\mathit{post}}\ \mathcal{R}_2\ Q_{\mathit{post}})$$

Rule1 [$\mathcal{R}_{1\mathrm{or}2}$ in $\mathit{evolve}(S, Q)$ = $\Rightarrow$]:

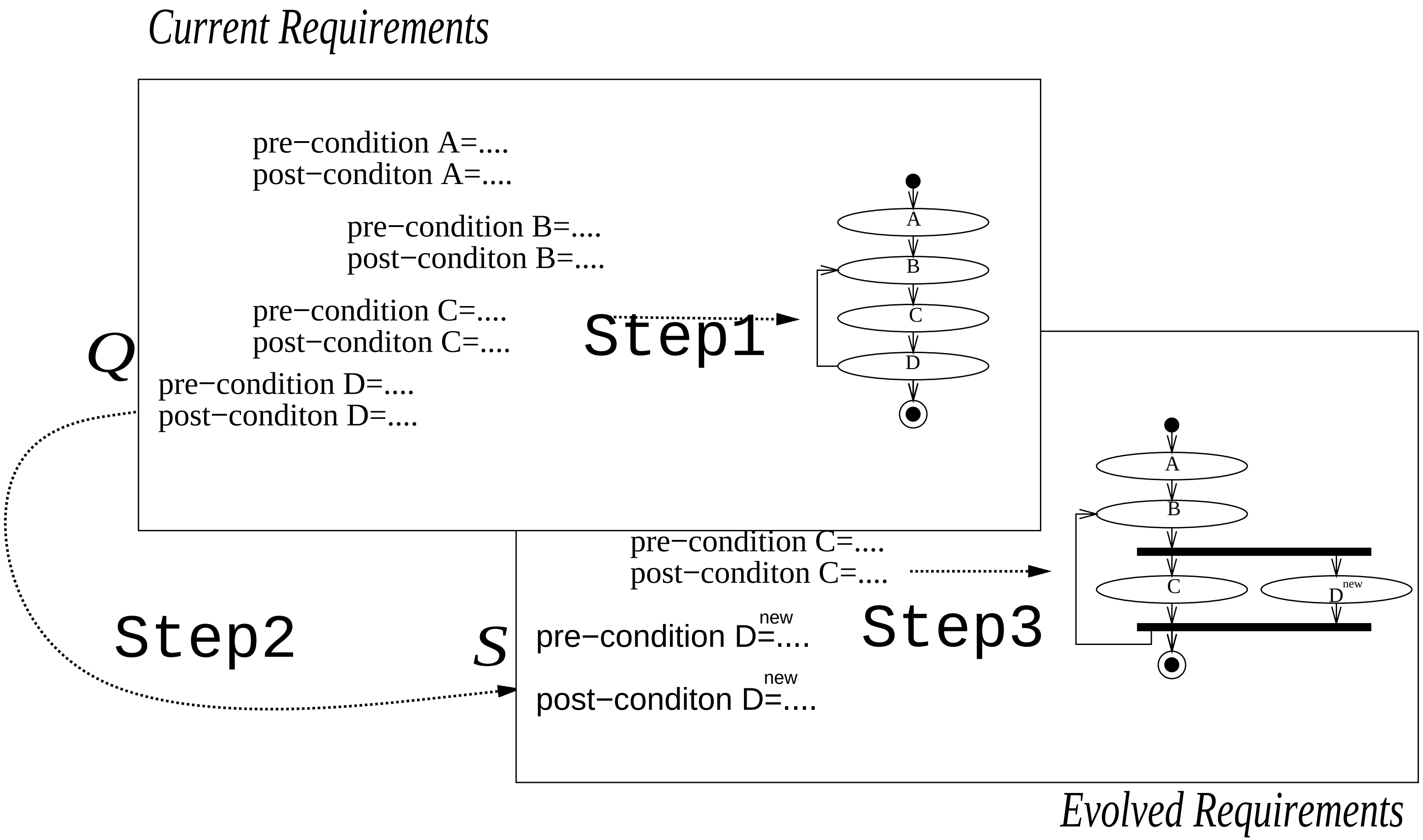
an activity of the new component is moved forward in the sequence of activities.

Rule2 [$\mathcal{R}_{1\mathrm{or}2}$ in $\mathit{evolve}(S, Q)$ = $\Leftarrow$]:

an activity of the new component is moved backward in the sequence of activities.

Note that this strategy is only valid when the conditions are gradually strengthened.

Summary of our Method



Current Requirements

pre−condition A=....
post−conditon A=....

pre−condition B=....
post−conditon B=....

pre−condition C=....
post−conditon C=....

Q

pre−condition D=....
post−conditon D=....

Step1

Step2

S

pre−condition C=....
post−conditon C=....

pre−condition D^{new}=....

post−conditon D^{new}=....

Step3

Evolved Requirements

Example: Assigning Reviewers of a Conference(1/4)

Tasks: You become a program chair of APSEC'99, you should

- Organize the committee from all over the world.
- Call for papers.
- Assign the reviewers of each submitted paper.

Circumstances: The committee members have

- Suitable ways to share and to read the papers
  – multicast distribution by PDF.
- A meeting easily even if they lives in the different countries
  – email.

Example: Assigning Reviewers of a Conference(2/4)

Current Component & Requirements:

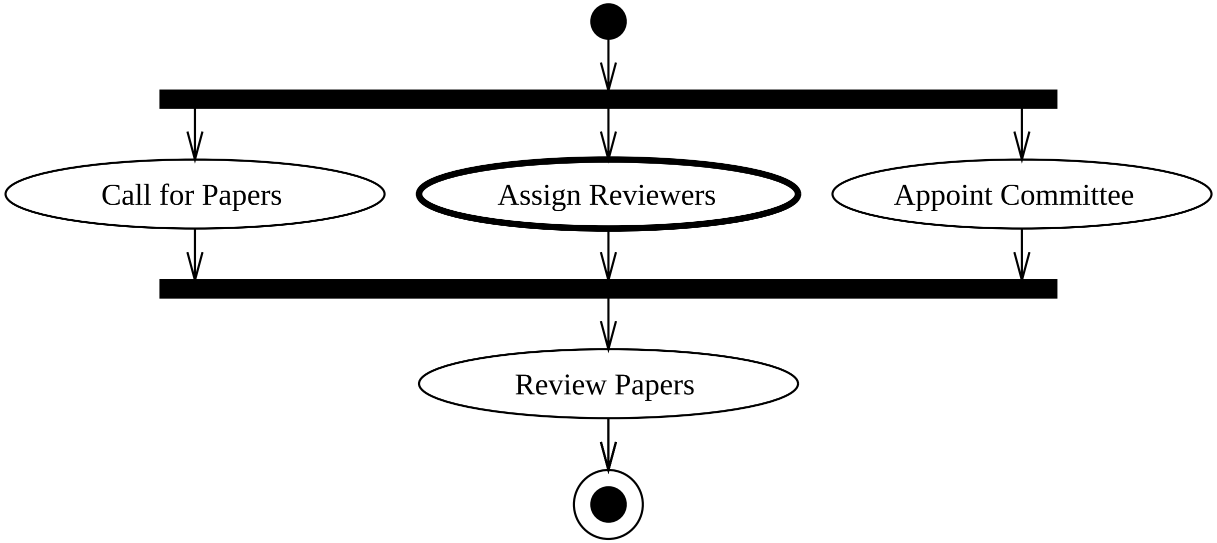
pre.Assign_Reviewers . . . . Q_{pre}

true

post.Assign_Reviewers

$p' : \mathbb{P}\, Paper;\ c' : \mathbb{P}\, Committee;\ m' : Paper \times Committee$ . . . . Q_{post}

$m' = assign(p', c')$



Example: Assigning Reviewers of a Conference(3/4)

Circumstances are changed → Spec.of Component is changed:

$pre.Assign\_Reviewers^{new}$

$p : \mathbb{P}\, Paper$ . . . . S_{pre}

$fixed(p)$

$post.Assign\_Reviewers^{new}$

$(\texttt{not changed})$ . . . . S_{post}

Then,

$$evolve(S, Q) = (Q_{pre} \Leftarrow S_{pre}) \wedge (S_{post} \Leftrightarrow Q_{post}).$$

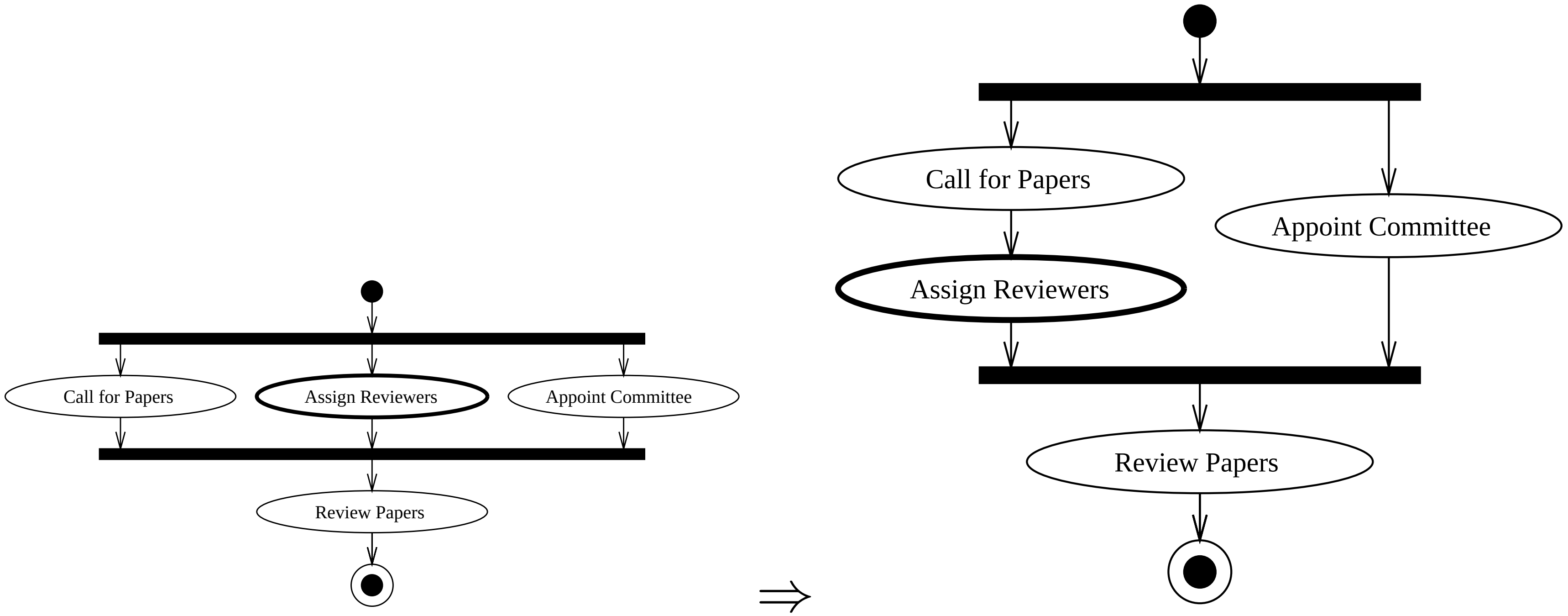
where

$$Q = Assign\_Reviewers$$

$$S = Assign\_Reviewers^{new}$$

Example: Assigning Reviewers of a Conference(4/4)

Applying rule2, the structure can be changed as follows;



Conclusion

- Define a concept of Requirements Evolution by Replacing Components.
- Present a method for the Evolution.
- Give an Example.

Discussion

- Introduce a class, i.e. set of functions for specifying a Component.
- Introduce more flexible Comparison Predicate: we do not always use *evolve*(and $match_{pre/post}$) predicate for comparison.
- Refine the rule of evolution: current rule is too limited.
- Build a natural and realistic example for this technique.