Refinement

© Michael Butler

University of Southampton

November 19, 2012

Abstraction

- Abstraction can be viewed as a process of simplifying our understanding of a system.
- The simplification should
  - focus on the intended purpose of the system
  - ignore details of how that purpose is achieved.
- The modeller needs to make judgements about what they believe to be the key features of the system.

Abstraction

- Abstraction can be viewed as a process of simplifying our understanding of a system.
- The simplification should
  - focus on the intended purpose of the system
  - ignore details of how that purpose is achieved.
- The modeller needs to make judgements about what they believe to be the key features of the system.
- If the purpose is to provide some service, then
  - model what a system does from the perspective of the service users
  - ‘users’ might be computing agents as well as humans.

Abstraction

- Abstraction can be viewed as a process of simplifying our understanding of a system.
- The simplification should
  - focus on the intended purpose of the system
  - ignore details of how that purpose is achieved.
- The modeller needs to make judgements about what they believe to be the key features of the system.
- If the purpose is to provide some service, then
  - model what a system does from the perspective of the service users
  - ‘users’ might be computing agents as well as humans.
- If the purpose is to control, monitor or protect some phenomenon, then
  - the abstraction should focus on those phenomenon
  - in what way should they be controlled or protected?
  - why should they be monitored?

Refinement

- **Refinement** is a process of enriching or modifying a model in order to
  - augment the functionality being modelled, or
  - explain how some purpose is achieved

Refinement

- **Refinement** is a process of enriching or modifying a model in order to
  - augment the functionality being modelled, or
  - explain how some purpose is achieved
- In a refinement step we refine one model $M1$ to another model $M2$:
  - $M2$ is a refinement of $M1$
  - $M1$ is an abstraction of $M2$

Refinement

- **Refinement** is a process of enriching or modifying a model in order to
  - augment the functionality being modelled, or
  - explain how some purpose is achieved
- In a refinement step we refine one model $M1$ to another model $M2$:
  - $M2$ is a refinement of $M1$
  - $M1$ is an abstraction of $M2$
- We can perform a series of refinement steps to produce a series of models $M1$, $M2$, $M3$, ...
- Facilitates abstraction: we can postpone treatment of some system features to later refinement steps

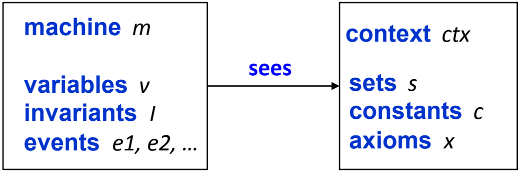
Refinement

- **Refinement** is a process of enriching or modifying a model in order to
  - augment the functionality being modelled, or
  - explain how some purpose is achieved
- In a refinement step we refine one model $M1$ to another model $M2$:
  - $M2$ is a refinement of $M1$
  - $M1$ is an abstraction of $M2$
- We can perform a series of refinement steps to produce a series of models $M1$, $M2$, $M3$, ...
- Facilitates abstraction: we can postpone treatment of some system features to later refinement steps
- Event-B provides a notion of consistency of a refinement:
  - We use proof to verify the consistency of a refinement step
  - Failing proof can help us identify inconsistencies in a refinement step

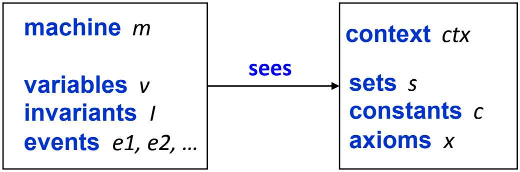
Refinement

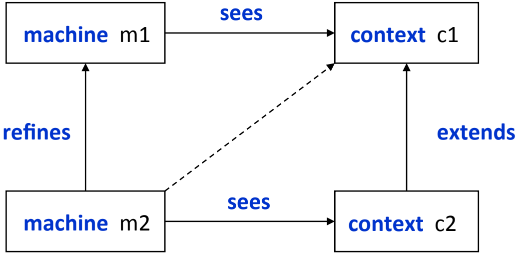
- **Refinement** is a process of enriching or modifying a model in order to
  - augment the functionality being modelled, or
  - explain how some purpose is achieved
- In a refinement step we refine one model $M1$ to another model $M2$:
  - $M2$ is a refinement of $M1$
  - $M1$ is an abstraction of $M2$
- We can perform a series of refinement steps to produce a series of models $M1$, $M2$, $M3$, ...
- Facilitates abstraction: we can postpone treatment of some system features to later refinement steps
- Event-B provides a notion of consistency of a refinement:
  - We use proof to verify the consistency of a refinement step
  - Failing proof can help us identify inconsistencies in a refinement step
- Abstraction and refinement together should allow us to manage system complexity in the design process

Modelling Components and Refinement



Modelling Components and Refinement





Extension Refinement in Event-B

A refined machine has the following form:

machine $M2$
refines $M1$
variables ...
invariants ...
events...

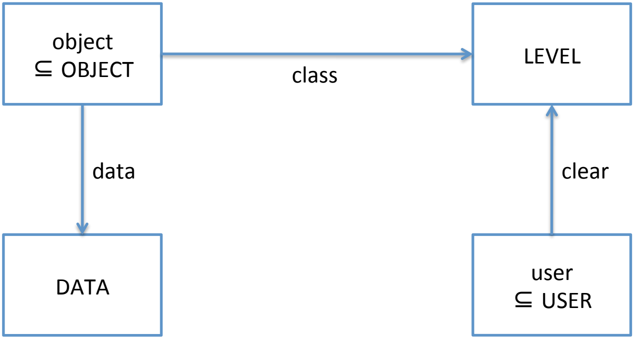
Extension refinement can be used to extend or add new features to a model.

- Add variables and invariants
- Extend existing events to act on additional variables
- Add new events to act on additional variables

All events must maintain the new invariants.

- Extension example: add ownership to secure database

Class diagram for secure database



Types and variables for Secure DB

context $c1$
sets $OBJECT \quad DATA \quad USER$
constants $LEVEL$
axioms $LEVEL = 1..10$

machine $SecureDB1$
sees $c1$
variables $object,\ user,\ data,\ class,\ clear$
invariants

$$\begin{aligned} object &\subseteq OBJECT \\ user &\subseteq USER \\ data &\in object \to DATA \\ class &\in object \to LEVEL \\ clear &\in user \to LEVEL \end{aligned}$$

Adding object ownership

Extend the database specification so that each object has an owner.

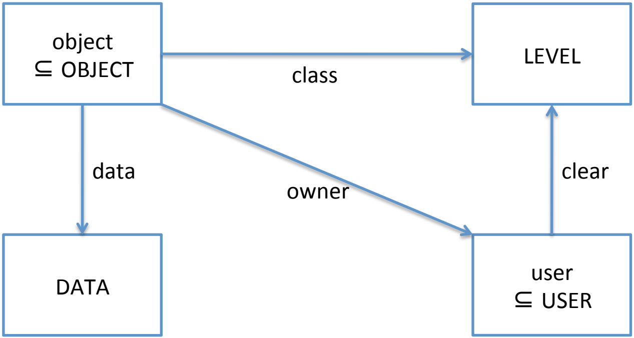
The clearance associated with that owner must be at least as high as the classification of the object.

Only the owner of an object is allowed to delete it.

What additional variables are required?

What events are affected?

Class diagram with ownership



Refinement

machine $\mathit{SecureDB2}$
refines $\mathit{SecureDB1}$
variables $\mathit{object},\ \mathit{user},\ \mathit{data},\ \mathit{class},\ \mathit{clear},\ \mathit{owner}$
invariants

$$\mathit{owner} \in \mathit{object} \to \mathit{user}$$

Note we must list all the variables: those from M1 that we wish to retain as well as new ones

Here *owner* is a new variable.

We do not repeat invariants of M1 in M2

Adding users

$$
\begin{array}{l}
\mathit{AddUser} \;\; \widehat{=} \\
\quad \textbf{any } u, c \textbf{ where} \\
\qquad u \in \mathit{USER} \\
\qquad u \notin \mathit{user} \\
\qquad c \in \mathit{LEVEL} \\
\quad \textbf{then} \\
\qquad \mathit{user} := \mathit{user} \cup \{u\} \\
\qquad \mathit{clear}(u) := c \\
\quad \textbf{end}
\end{array}
$$

Do we need to modify this?

Adding objects

$$
\begin{array}{l}
AddObject \ \widehat{=} \\
\quad \textbf{any}\ o, d, c\ \textbf{where} \\
\qquad o \in OBJECT \\
\qquad o \notin object \\
\qquad d \in DATA \\
\qquad c \in LEVEL \\
\quad \textbf{then} \\
\qquad object := object \cup \{o\} \\
\qquad data(o) := d \\
\qquad class(o) := c \\
\quad \textbf{end}
\end{array}
$$

Do we need to modify this?

Event Extension

$$
\begin{array}{l}
\mathit{AddObject} \;\; \textcolor{red}{\textbf{extends}} \;\; \mathit{AddObject} \;\; \widehat{=} \\
\quad \textbf{any} \;\; u \;\; \textbf{where} \\
\qquad u \;\; \in \;\; \mathit{user} \\
\qquad \mathit{clear}(u) \geq \mathit{class}(o) \\
\quad \textbf{then} \\
\qquad \mathit{owner}(o) \;\; := \;\; u \\
\quad \textbf{end}
\end{array}
$$

This is equivalent to

$$
\begin{array}{l}
\mathit{AddObject}\ \ \textbf{refines}\ \ \mathit{AddObject}\ \ \widehat{=} \\
\quad \textbf{any}\ \ o, d, c, \color{red}{u}\ \ \textbf{where} \\
\quad\quad o \ \in\ \mathit{OBJECT} \\
\quad\quad o \ \notin\ \mathit{object} \\
\quad\quad d \ \in\ \mathit{DATA} \\
\quad\quad c \ \in\ \mathit{LEVEL} \\
\quad\quad u \ \in\ \mathit{user} \\
\quad\quad \mathit{clear}(u) \geq \mathit{class}(o) \\
\quad \textbf{then} \\
\quad\quad \mathit{object} \ :=\ \mathit{object} \cup \{o\} \\
\quad\quad \mathit{data}(o) \ :=\ d \\
\quad\quad \mathit{class}(o) \ :=\ c \\
\quad\quad \color{red}{\mathit{owner}(o) \ :=\ u} \\
\quad \textbf{end}
\end{array}
$$

Other events to consider

- *Read*
- *Write*
- *ChangeClass*
- *ChangeClear*
- *RemoveUser*, *RemoveObject*

Do we need new events?

Forms of Event-B Refinement

1. Extension:
   - Add variables and invariants
   - Extend existing events to act on additional variables
   - Add new events to act on additional variables

Forms of Event-B Refinement

1. Extension:
    - Add variables and invariants
    - Extend existing events to act on additional variables
    - Add new events to act on additional variables
2. Extension with Guard Modification:
    - Similar to model extension, except that we modify guards of existing events

Forms of Event-B Refinement

1. Extension:
   - Add variables and invariants
   - Extend existing events to act on additional variables
   - Add new events to act on additional variables
2. Extension with Guard Modification:
   - Similar to model extension, except that we modify guards of existing events
3. Variable Replacement / Data Reification:
   - Replace some variables with other variables, i.e., replace abstract variables with concrete variables
   - Modify existing events, add new events

Forms of Event-B Refinement

1. Extension:
   - Add variables and invariants
   - Extend existing events to act on additional variables
   - Add new events to act on additional variables
2. Extension with Guard Modification:
   - Similar to model extension, except that we modify guards of existing events
3. Variable Replacement / Data Reification:
   - Replace some variables with other variables, i.e., replace abstract variables with concrete variables
   - Modify existing events, add new events
4. Variable Removal:
   - Remove variables that have become redundant through earlier introduction of other variables.

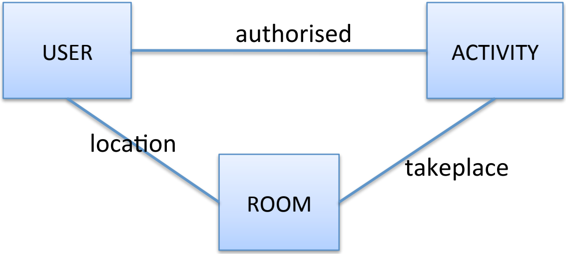
Forms of Event-B Refinement

1. Extension:
   - Add variables and invariants
   - Extend existing events to act on additional variables
   - Add new events to act on additional variables
2. Extension with Guard Modification:
   - Similar to model extension, except that we modify guards of existing events
3. Variable Replacement / Data Reification:
   - Replace some variables with other variables, i.e., replace abstract variables with concrete variables
   - Modify existing events, add new events
4. Variable Removal:
   - Remove variables that have become redundant through earlier introduction of other variables.

- Verification of 2, 3 and 4 requires gluing invariants that link abstract and concrete variables.

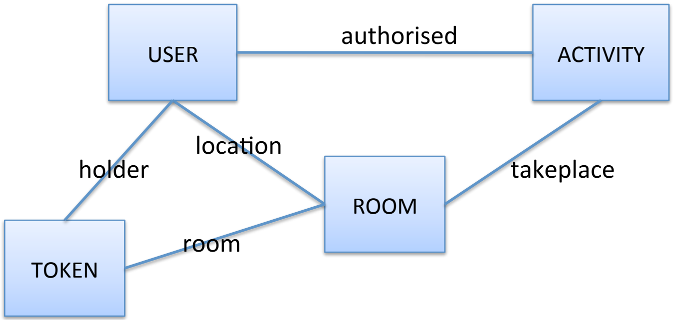
- Extension example: add ownership to secure database
- Extension with Guard Modification example: add tokens to access control system
- Variable replace example: simple data sampling system

Abstract model of access control to rooms



Access Control Policy: *Users may only be in a room if they are authorised to engage in all activities that may take place in that room*

Refine this by introducing a token mechanism



Tokens: *Users must acquire a token in order to enter a room*

Access control with tokens

$$
\begin{array}{lll}
AbstractEnter & \hat{=} & \textbf{any } u, r \textbf{ where} \\
& & \quad u \in USER \setminus dom(location) \\
& & \quad r \in ROOM \\
& & \quad takeplace[\{r\}] \subseteq authorised[\{u\}] \\
& & \textbf{then} \\
& & \quad location(u) := r \\
& & \textbf{end}
\end{array}
$$

$$
\begin{array}{lll}
RefinedEnter & \hat{=} & \textbf{any } u, r, t \textbf{ where} \\
& & \quad u \in USER \setminus dom(location) \\
& & \quad t \in valid \\
& & \quad r = room(t) \\
& & \quad u = holder(t) \\
& & \textbf{then} \\
& & \quad location(u) := r \\
& & \textbf{end}
\end{array}
$$

GRD Proof Obligations

We need to prove that the guard of a refined event is not weaker than the guard of the abstract event.
E.g., the refined enter event should not weaken the conditions under which a user may enter a room.

GRD Proof Obligations

We need to prove that the guard of a refined event is not weaker than the guard of the abstract event.
E.g., the refined enter event should not weaken the conditions under which a user may enter a room.

GRD Proof obligation:
Assume: guard(RefinedEnter) + invariants
Prove: guard(AbstractEnter)

GRD Proof Obligations

We need to prove that the guard of a refined event is not weaker than the guard of the abstract event.
E.g., the refined enter event should not weaken the conditions under which a user may enter a room.

GRD Proof obligation:
Assume: guard(RefinedEnter) + invariants
Prove: guard(AbstractEnter)

For the access control refinement, we need this invariant:

$$\forall t \cdot t \in valid \implies takeplace[\{room(t)\}] \subseteq authorised[\{holder(t)\}]$$

Simple data sampling system

$$
\begin{array}{ll}
\textbf{machine} & MaxSet1 \\
\textbf{variables} & samples \\
\textbf{invariants} & samples \subseteq \mathbb{N} \\
\textbf{initialisation} & samples := \{0\} \\
\textbf{events} &
\end{array}
$$

$$
\begin{array}{lcl}
Add & \hat{=} & \textbf{any } x \textbf{ where} \\
 & & \quad x \in \mathbb{N} \\
 & & \textbf{then} \\
 & & \quad samples := samples \cup \{x\} \\
 & & \textbf{end}
\end{array}
$$

$$
\begin{array}{lcl}
GetMax & \hat{=} & \textbf{any } result \textbf{ where} \\
 & & \quad result = max(samples) \\
 & & \textbf{end}
\end{array}
$$

Refine to a more optimal design

machine $MaxSet2$
refines $MaxSet1$
variables m we only need to store the maximum so far
invariants $m \in \mathbb{N}$
initialisation $m := 0$
events

$Add \;\widehat{=}\;$ **any** x **where**
$x \in \mathbb{N}$
then
$m := max(\{m, x\})$
end

$GetMax \;\widehat{=}\;$ **any** $result$ **where**
$result = m$
end

Gluing invariant

What is the relationship between m and *samples*?

Gluing invariant

What is the relationship between m and $samples$?

machine $MaxSet2$
refines $MaxSet1$
variables m
invariants $m = max(samples)$
events...

This is called a gluing invariant: it specifies the relationship between the abstract and refined variables.

Proving that the gluing invariant is maintained

Abstract *Add*: $samples := samples \cup \{x\}$

Refined *Add*: $m := max(\{m, x\})$

Proving that the gluing invariant is maintained

Abstract *Add*: $samples := samples \cup \{x\}$

Refined *Add*: $m := max(\{m, x\})$

Assume: $m = max(samples)$

Prove: $max(\{m, x\}) = max(samples \cup \{x\})$

Proving that the gluing invariant is maintained

Abstract *Add*: $samples := samples \cup \{x\}$

Refined *Add*: $m := max(\{m, x\})$

Assume: $m \;=\; max(samples)$

Prove: $max(\{m, x\}) \;=\; max(samples \cup \{x\})$

This is valid since:

$$max(s \cup \{x\}) \quad =$$

Proving that the gluing invariant is maintained

Abstract *Add*: $samples := samples \cup \{x\}$

Refined *Add*: $m := max(\{m, x\})$

Assume: $m = max(samples)$

Prove: $max(\{m, x\}) = max(samples \cup \{x\})$

This is valid since:

$$max(s \cup \{x\}) = max(\{max(s), x\})$$