Dynamic Software Evolution – Model Checking with Abstraction

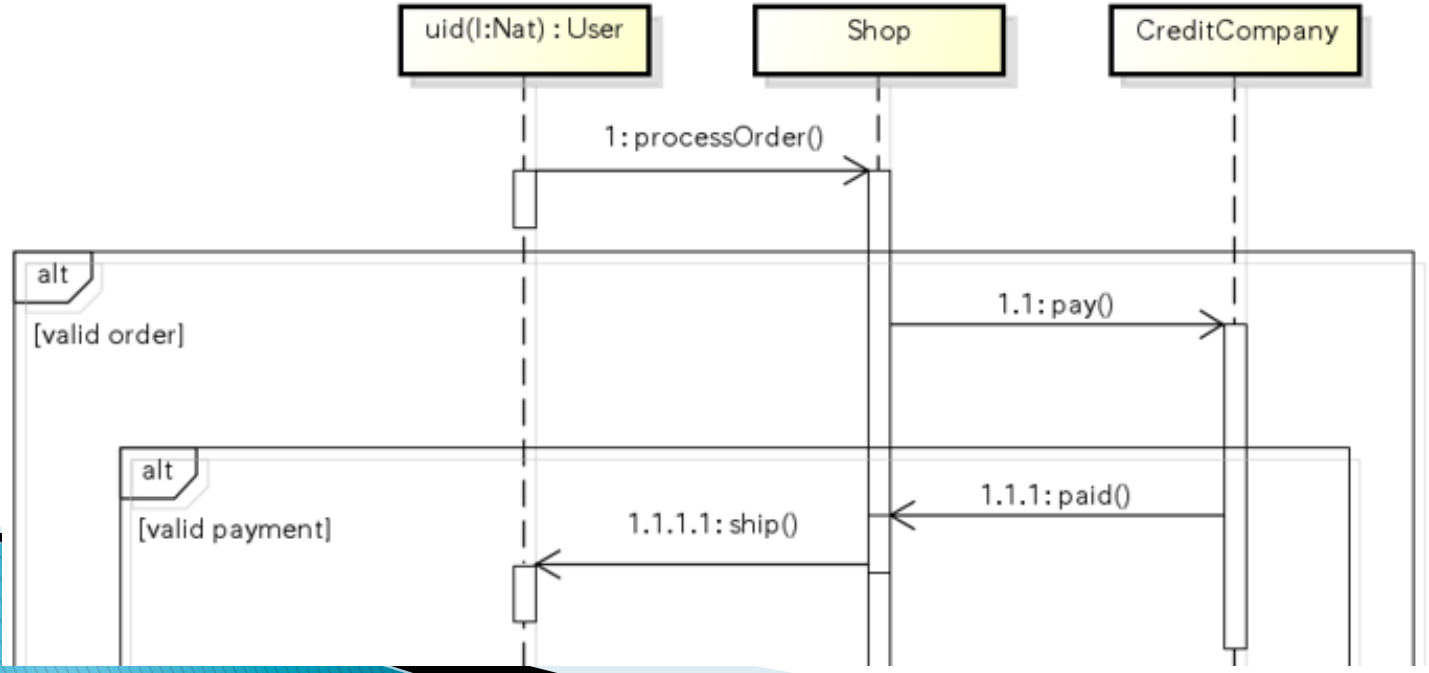
Yasuyuki Tahara, Shinichi Honiden

Contents

- Details of Motivating Example
- Backgrounds
- Details of Proposed Method
- Discussions
- Related Work
- Future work

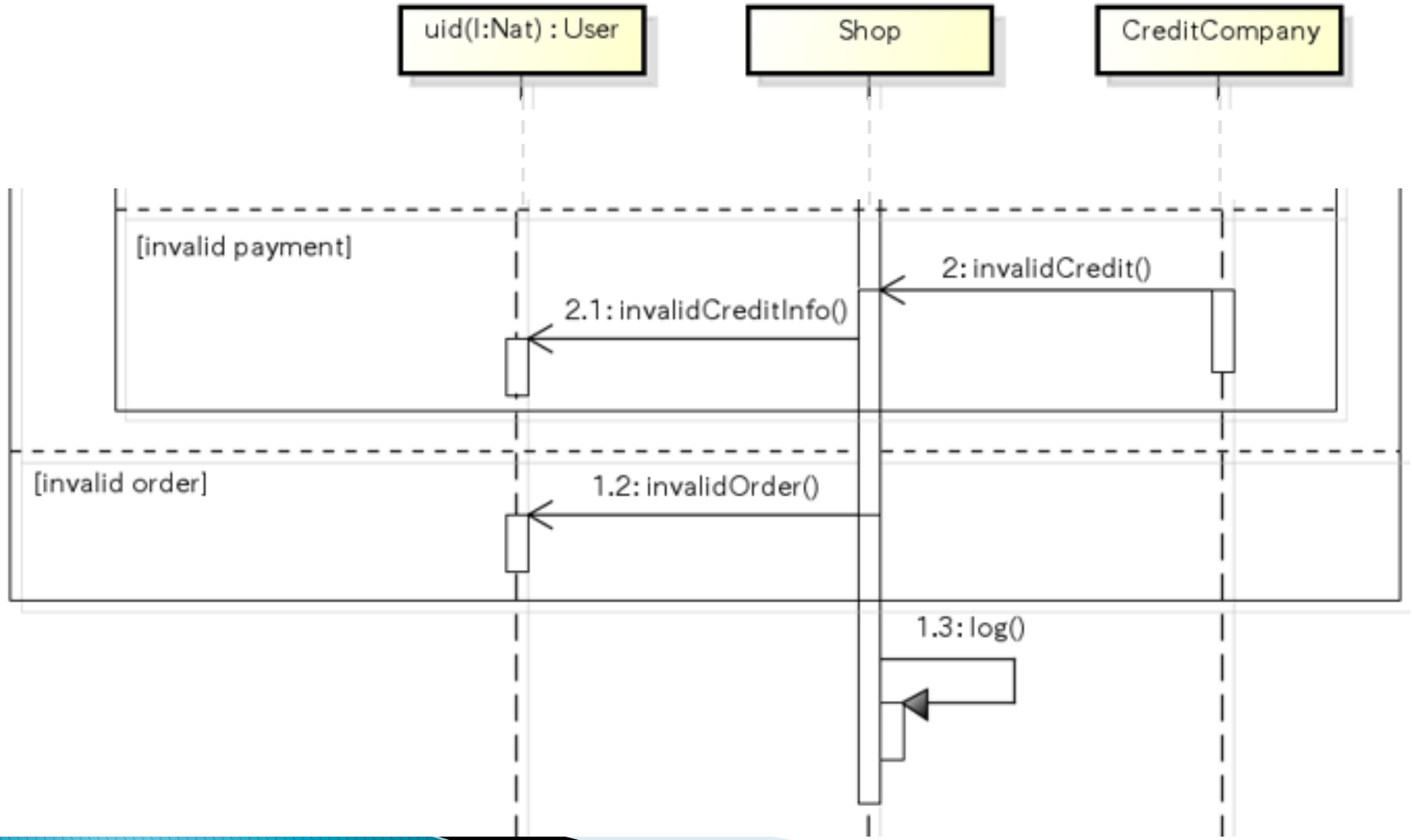
Details of Motivating Example

- Online shopping service
- Adopted from [Chen et al., 2014] and [Qian et al., 2014] and modified
- Scenario before evolution



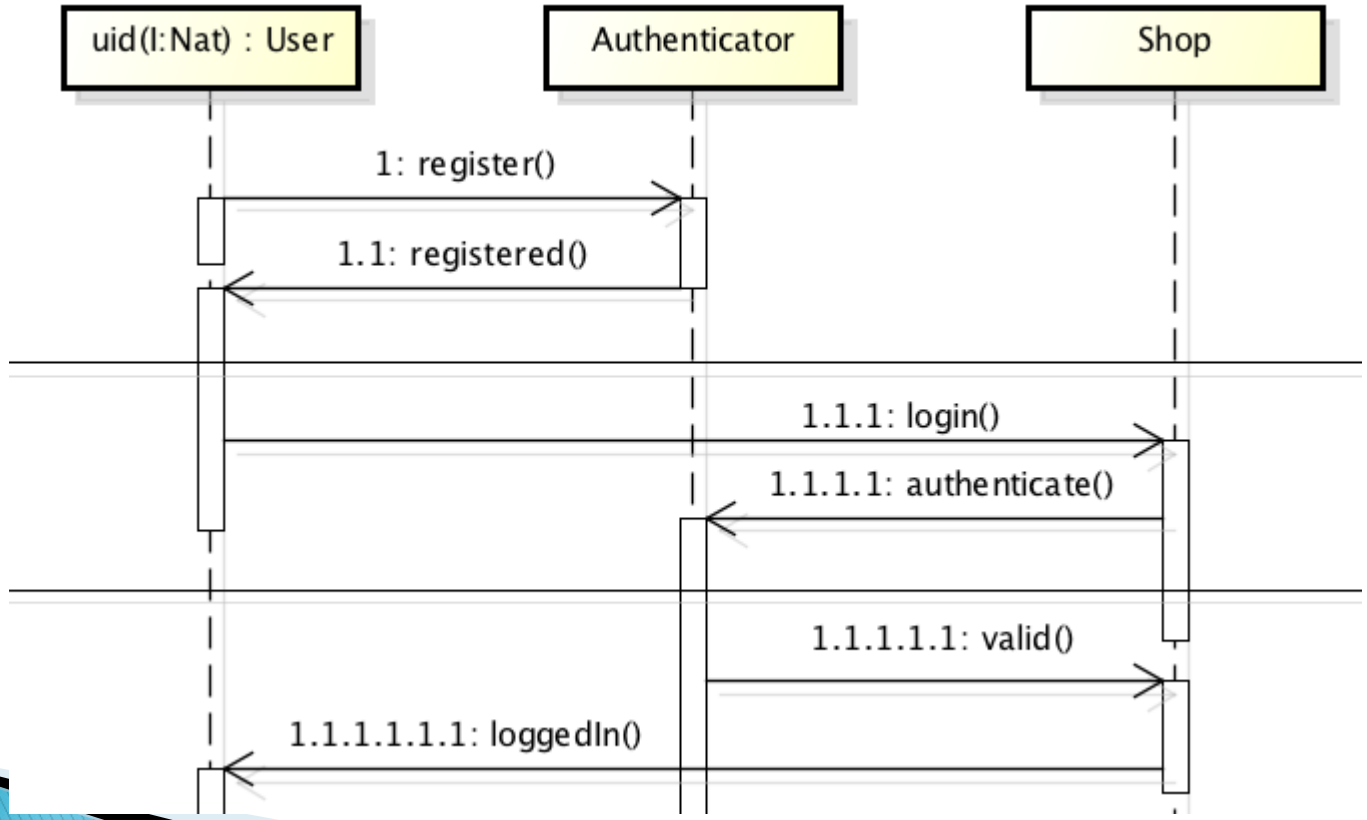
Details of Motivating Example

- Scenario before evolution (cont'd)



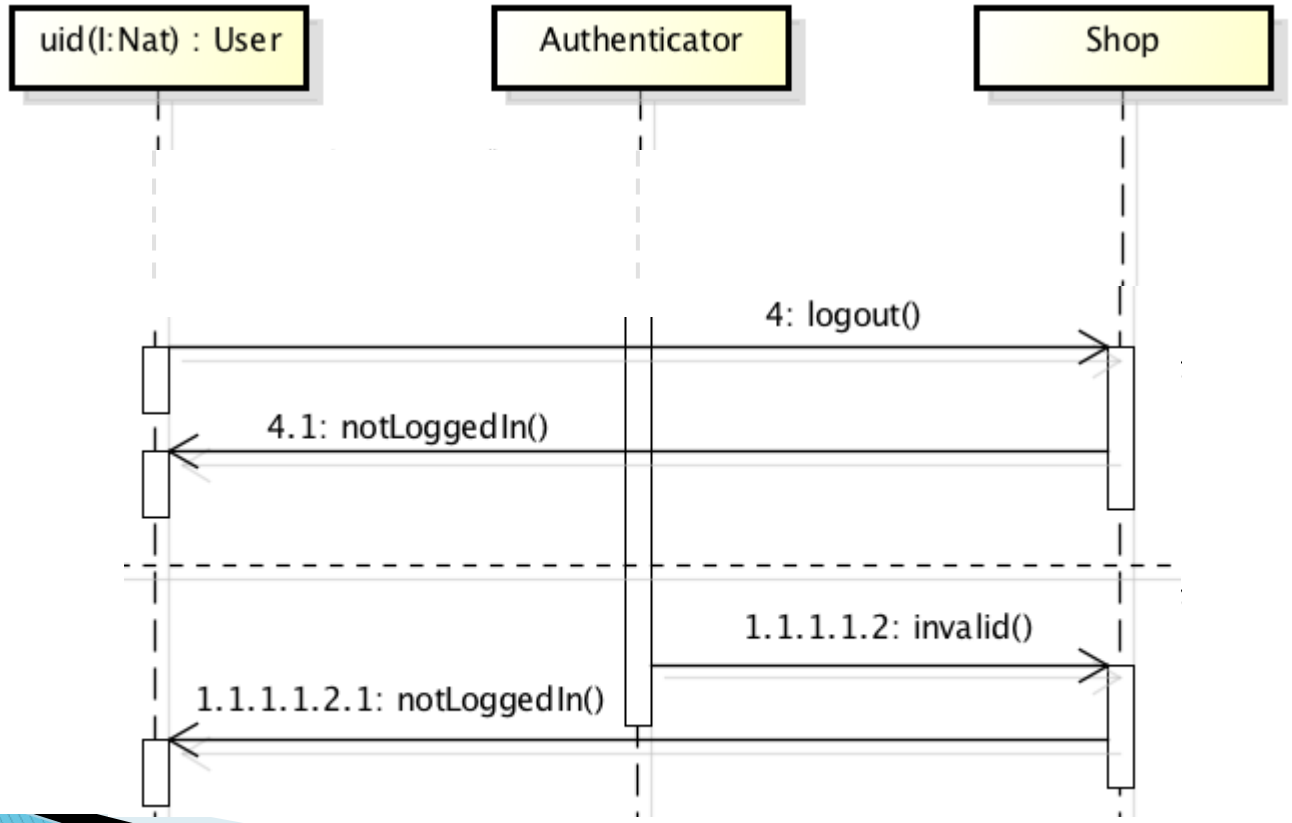
Details of Motivating Example

- Scenario after the first evolution (added parts)



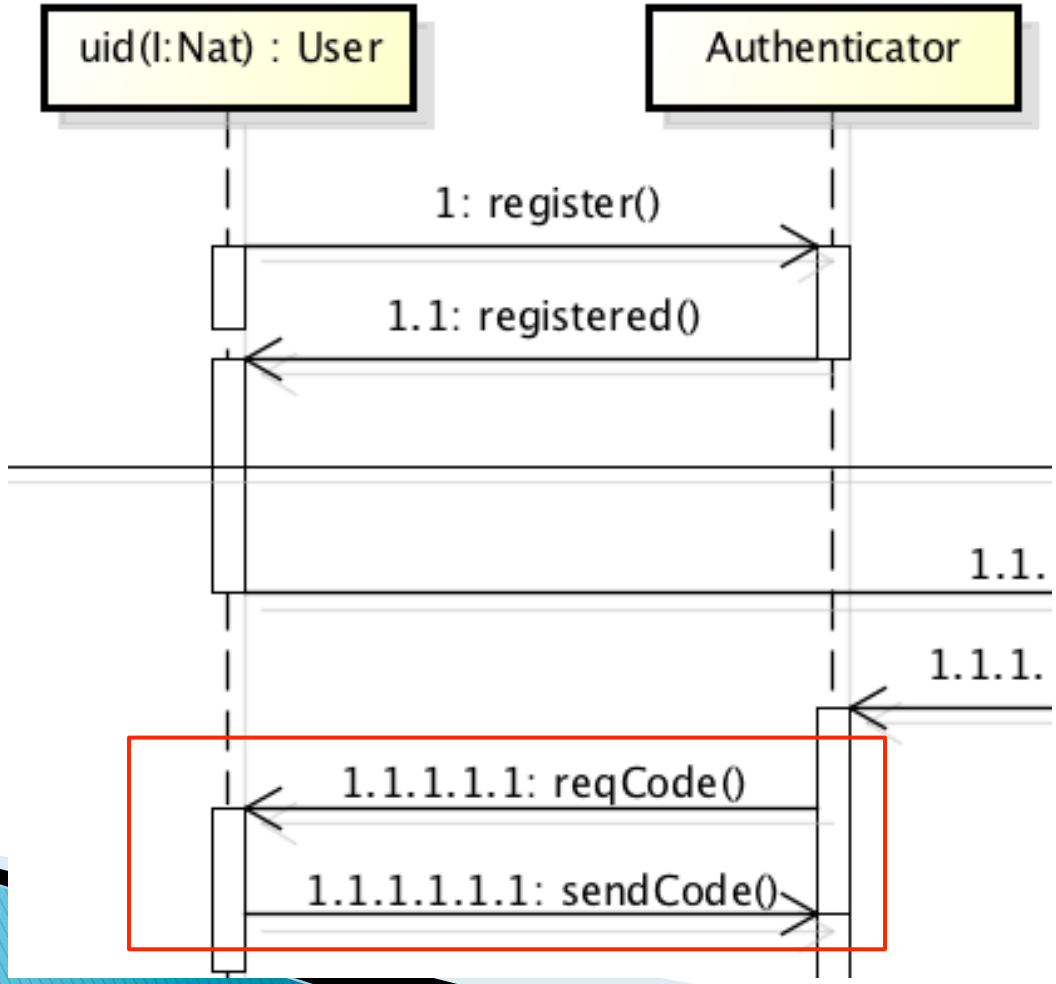
Details of Motivating Example

- Scenario after the first evolution (added parts, cont'd)



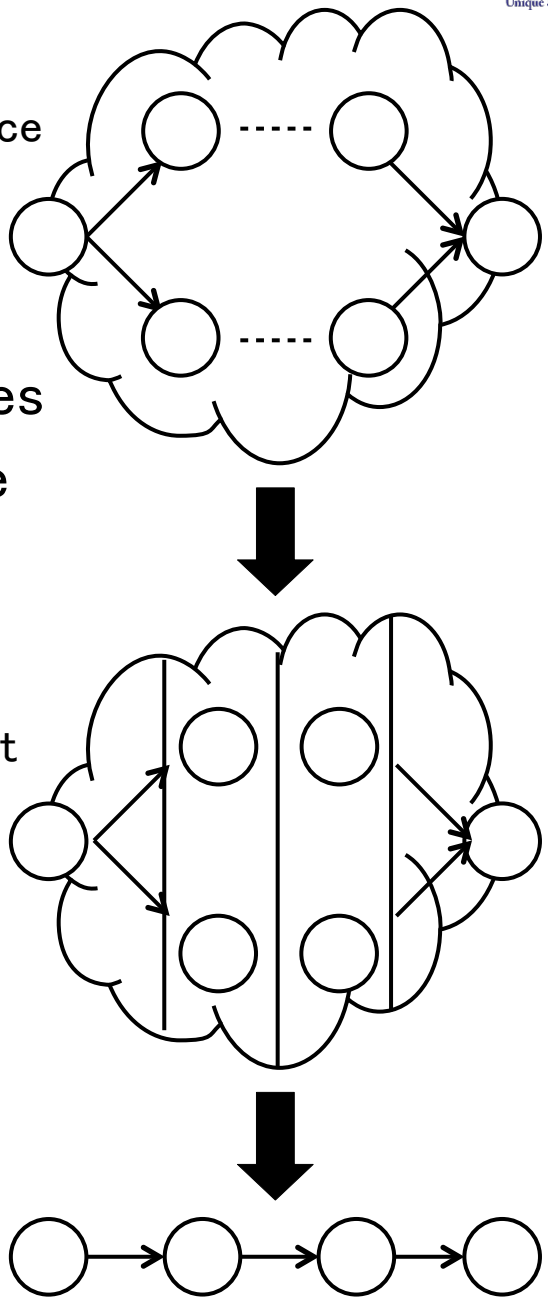
Details of Motivating Example

- Scenario after the second evolution (added parts)



Backgrounds

- Abstraction in model checking
  - Provide an equivalence relation of states
  - Make the quotient set as the new state space



Backgrounds

- Maude
  - Algebraic specification language
  - Behavior specifications represented by equations and rewrite rules between terms
    - A term represents a system state
    - A set of equations and rewrite rules with declarations is called a rewrite theory
  - Behaviors are represented by rewriting relations between terms inferred from the rewrite theory
    - For a rewrite theory R and terms t and t', we write $R \mid= t \Rightarrow t'$

Backgrounds

- Maude (cont'd)
  - Useful to write behavior specifications of distributed object-based systems
  - Support of reflection
  - Effective theoretical basis of abstraction called equational abstraction
    - Just adding equations to the rewrite theory
    - The newly added equations define the equivalence relation
  - Model checkers

Maude

- Rewrite rule representing the process of registering a user

Term

Object (authenticator)

```
rl [register(I) < A : Authenticator | none >]
   => < A : Authenticator | none > registered(I) .
```

Message

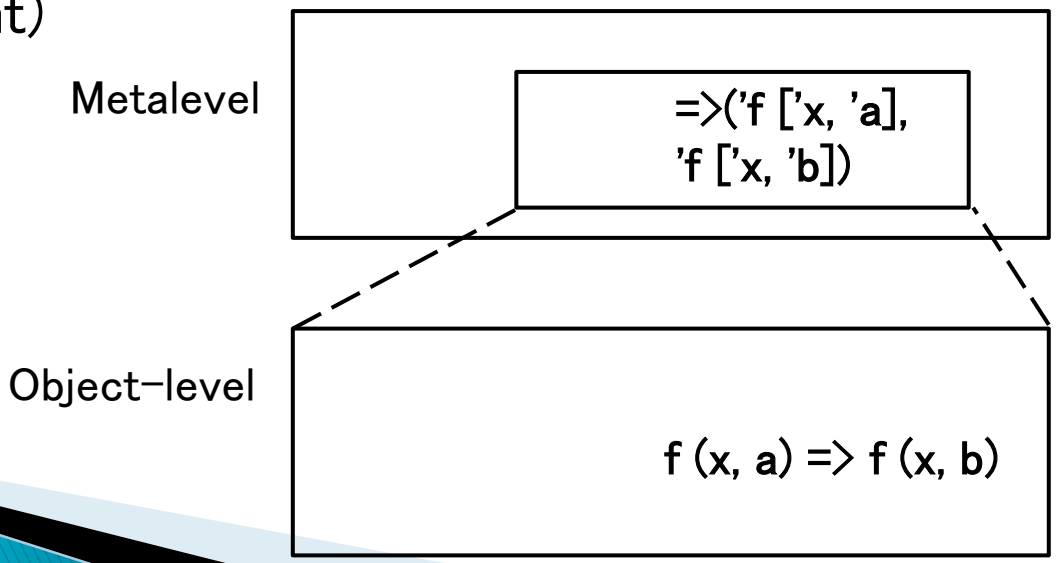
  - The authenticator receives the registration request of the user whose ID is “I” (variable for natural numbers) and informs the completion of the registration

Maude

- Modeling reflection
  - Terms that are metalevel representations of the object-level constructs such as terms, rewrite rules, and rewrite theories
  - E.g. The metalevel term for f(x, a)

    $\overline{f(x, a)}$ = 'f ['x, 'a] = _[_,_]('f, 'x, 'a)

    (Each underscore '_' is replaced by the corresponding argument)

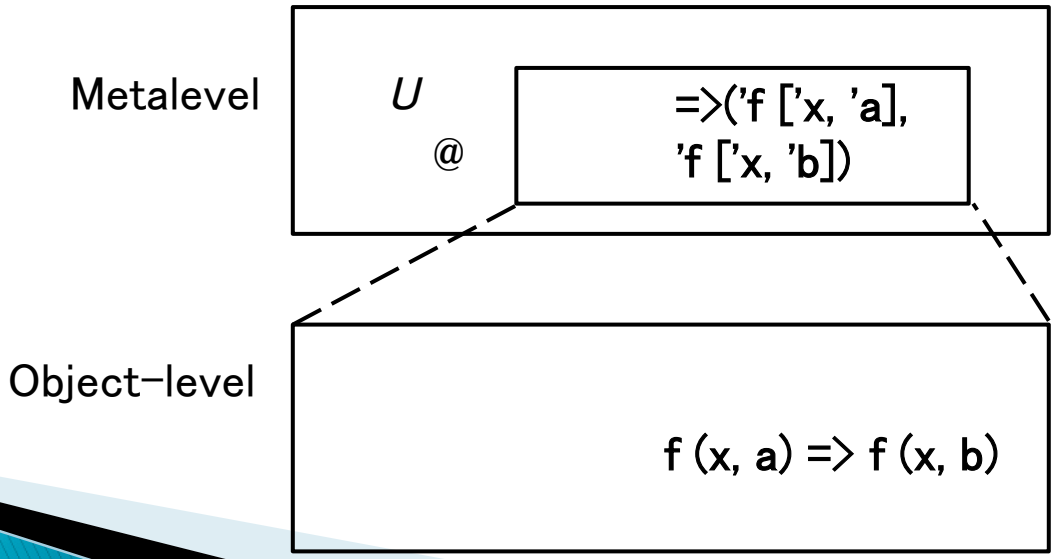


Maude

- Modeling reflection
  - A specific rewrite theory U and an operator @

    $R \models t \Rightarrow t'$ if and only if $U \models \overline{t} \,@\, \overline{R} \Rightarrow \overline{t'} \,@\, \overline{R}$

  indicating in the metalevel U simulates the object-level behaviors



Maude

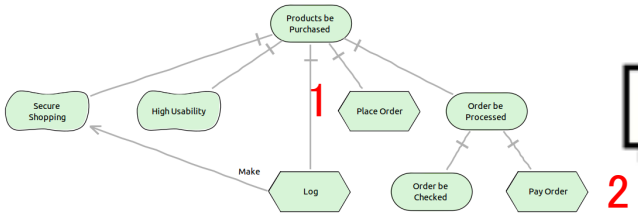
- Equational abstraction
  - Just adding equations to the rewrite theory
  - The newly added equations define the equivalence relation

Proposed Approach

- Assumptions
  - Existence of the goal model, the sequence diagram, and the program that is currently operating
  - Traceability from the goal model to the sequence diagram
    - Represented by correspondence between the functional requirements (leaf goals) and the parts of the sequence diagram

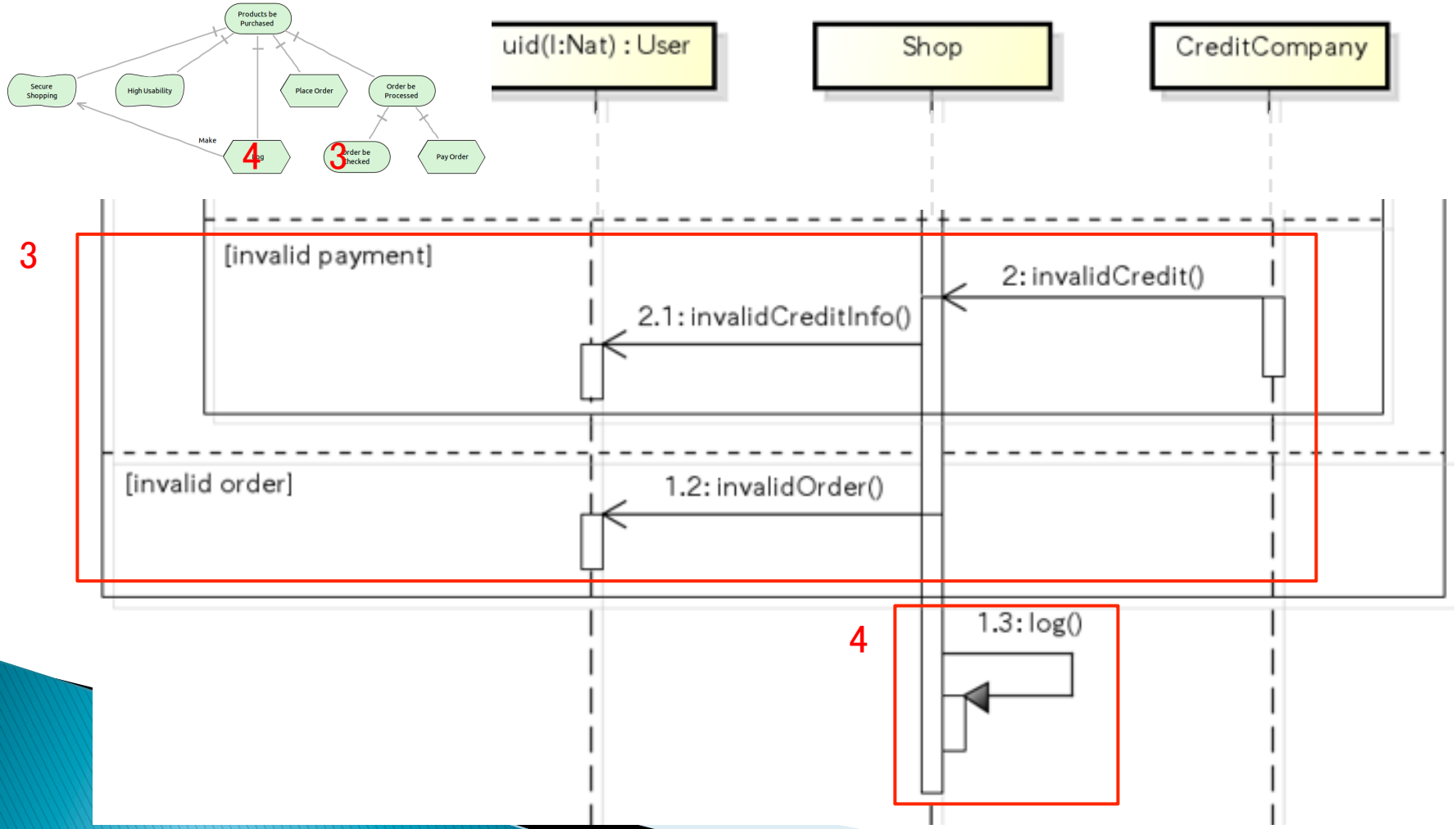
Proposed Approach

- Example



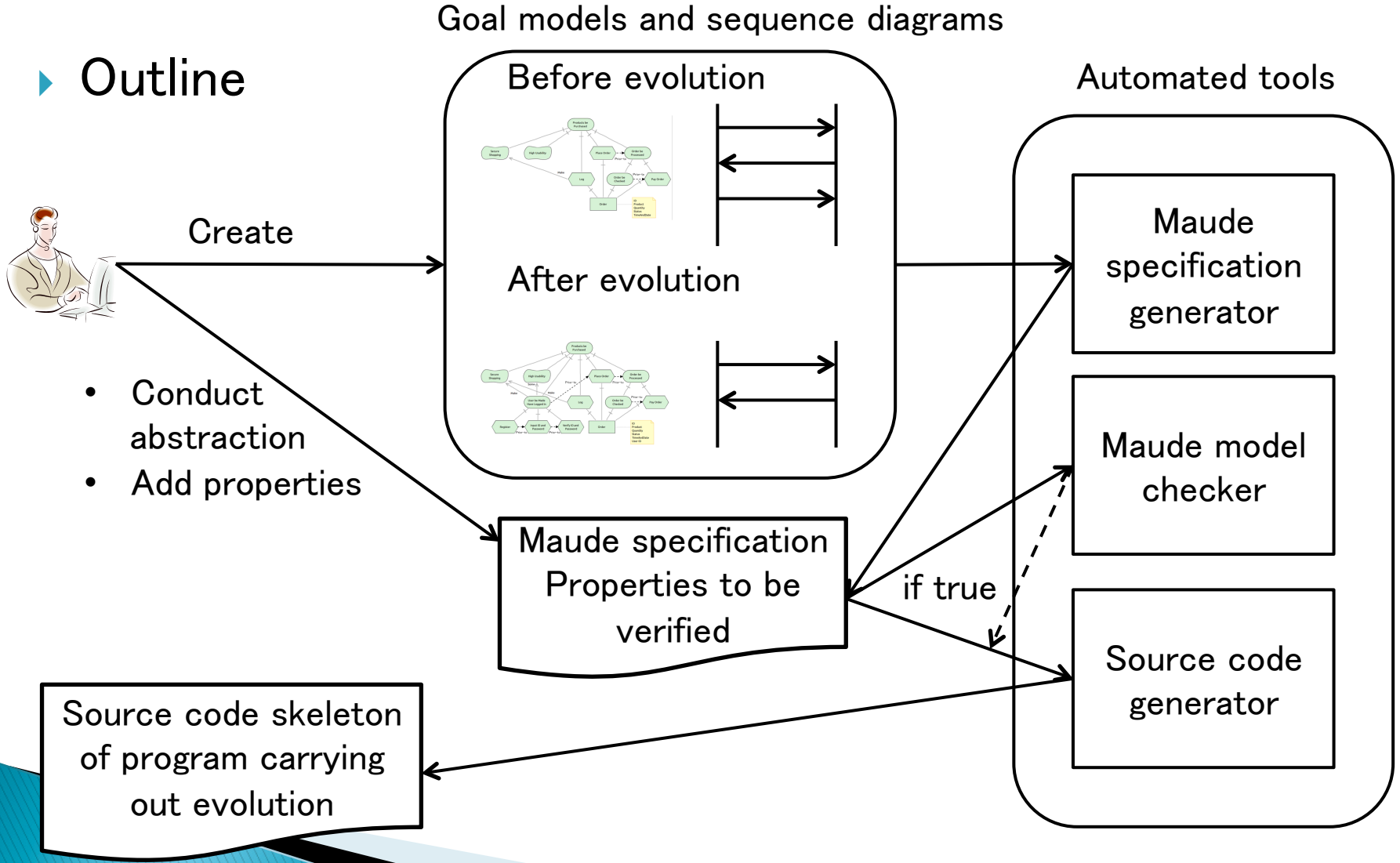
Proposed Approach

- Example (cont'd)



Proposed Approach

- Outline

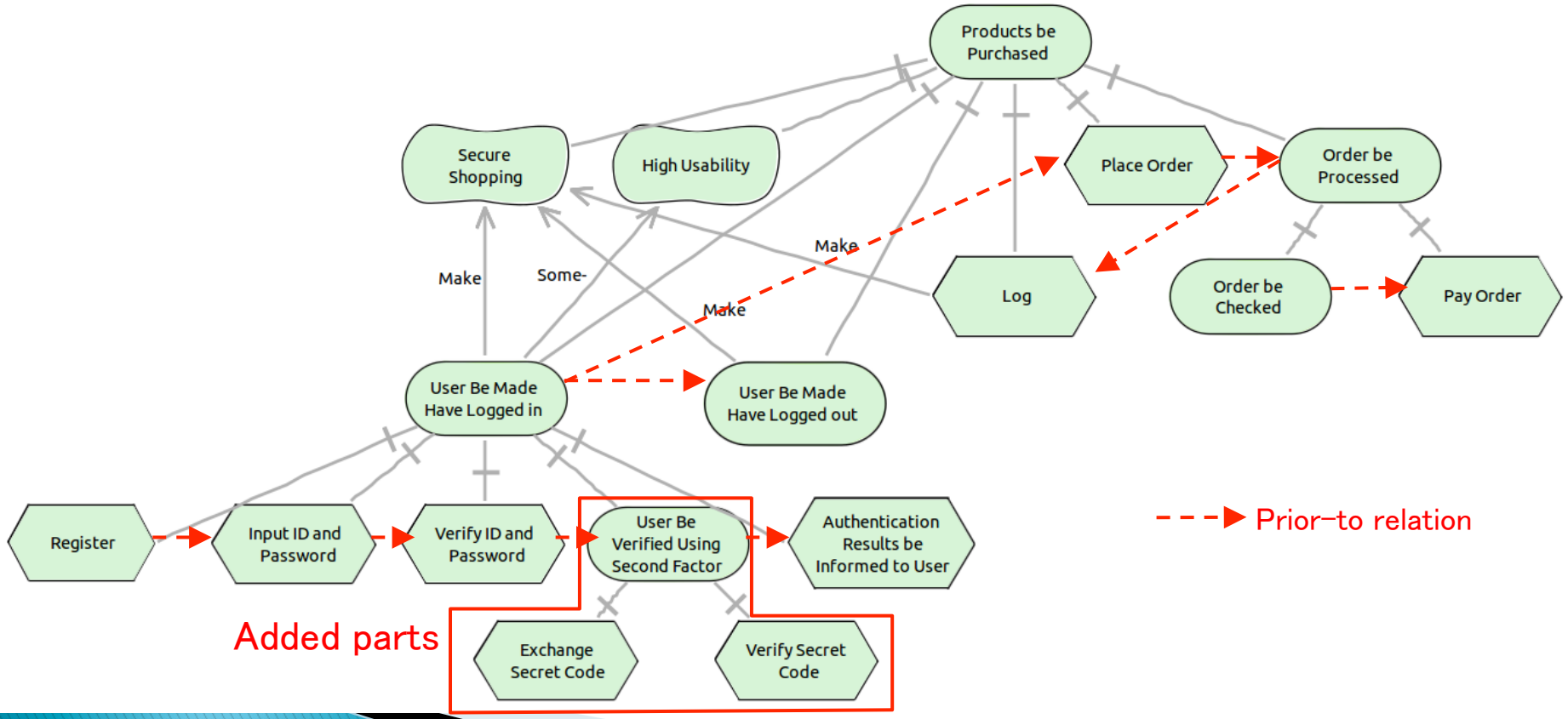


Proposed Approach

1. Identify the difference between the goal models before and after evolution
   - Express explicitly temporal orders of goal satisfaction as the "prior-to" relationship

Proposed Approach

- Goal model after the second evolution

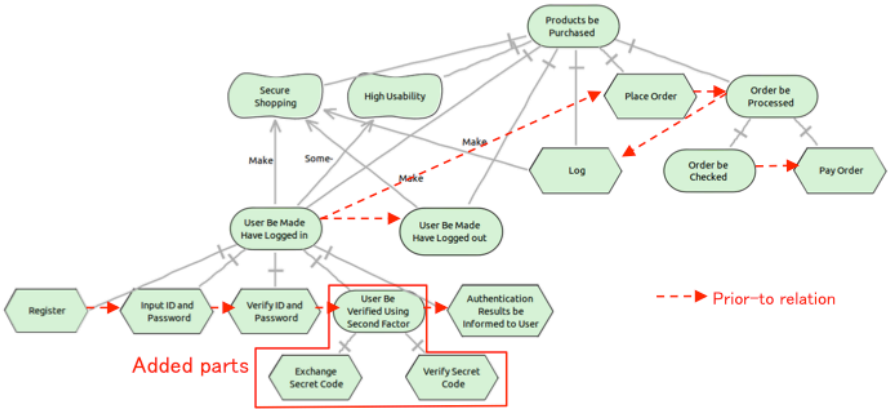


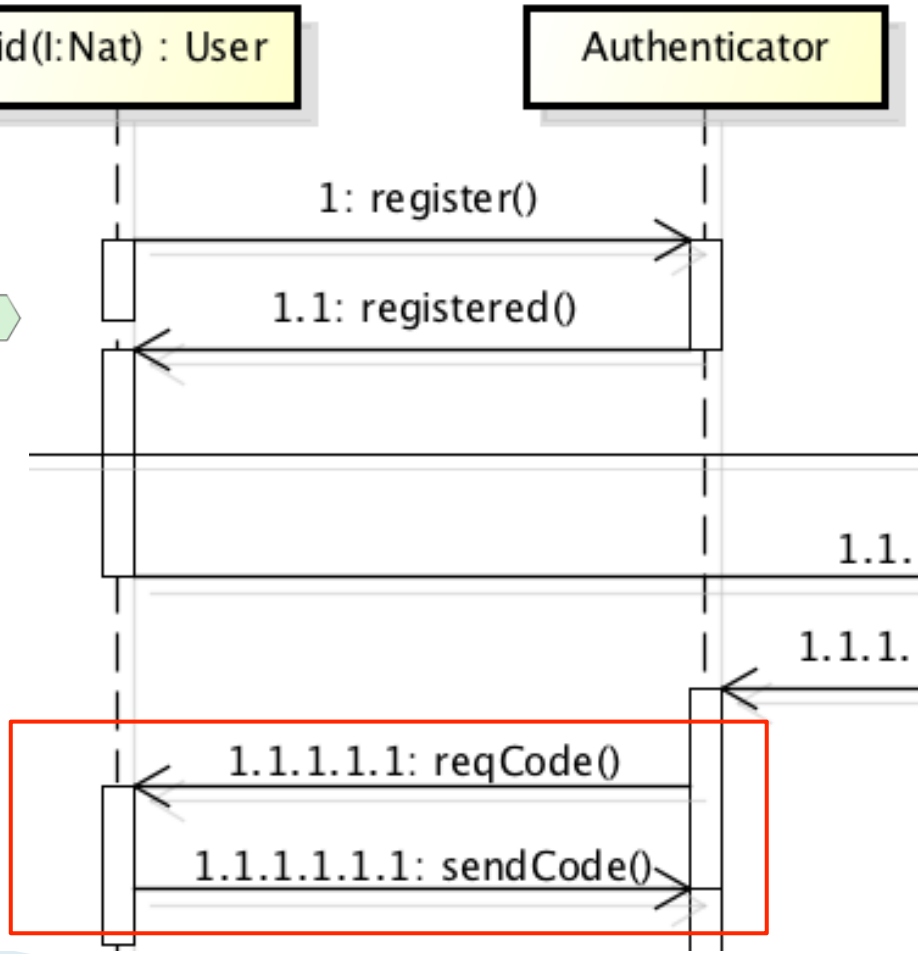
Proposed Approach

1. Identify the difference between the goal models before and after evolution
   - Express explicitly satisfaction orders between goals as the "prior–to" relationship
2. Create sequence diagram of the program after evolution, or sequence diagram fragments for each newly added functional requirements

Proposed Approach

- Example: Scenario after the second evolution (added parts)





Proposed Approach

3. After the Maude specification is automatically created, the engineer modifies the specification manually if needed
4. Model check the created specification
5. If model checking succeeds, create the new program from the sequence diagram and carry out evolution in practice, and if not, modify the goal model or the sequence diagram and return to 3

Proposed Approach

- Creating Maude specification
  - Create the Maude specifications from the sequence diagrams before and after evolution respectively


```
rl : register(I) < A : Authenticator | none >
   => < A : Authenticator | none > registered(I) .
```

Proposed Approach

- Creating Maude specification
  - Create the Maude specifications from the sequence diagrams before and after evolution respectively
  - Create the metalevel Maude specification that deals with the specifications created above as metalevel data and carries out the change
    - The rewrite rule to carry out the change:

      rl T @ r => T @ r'

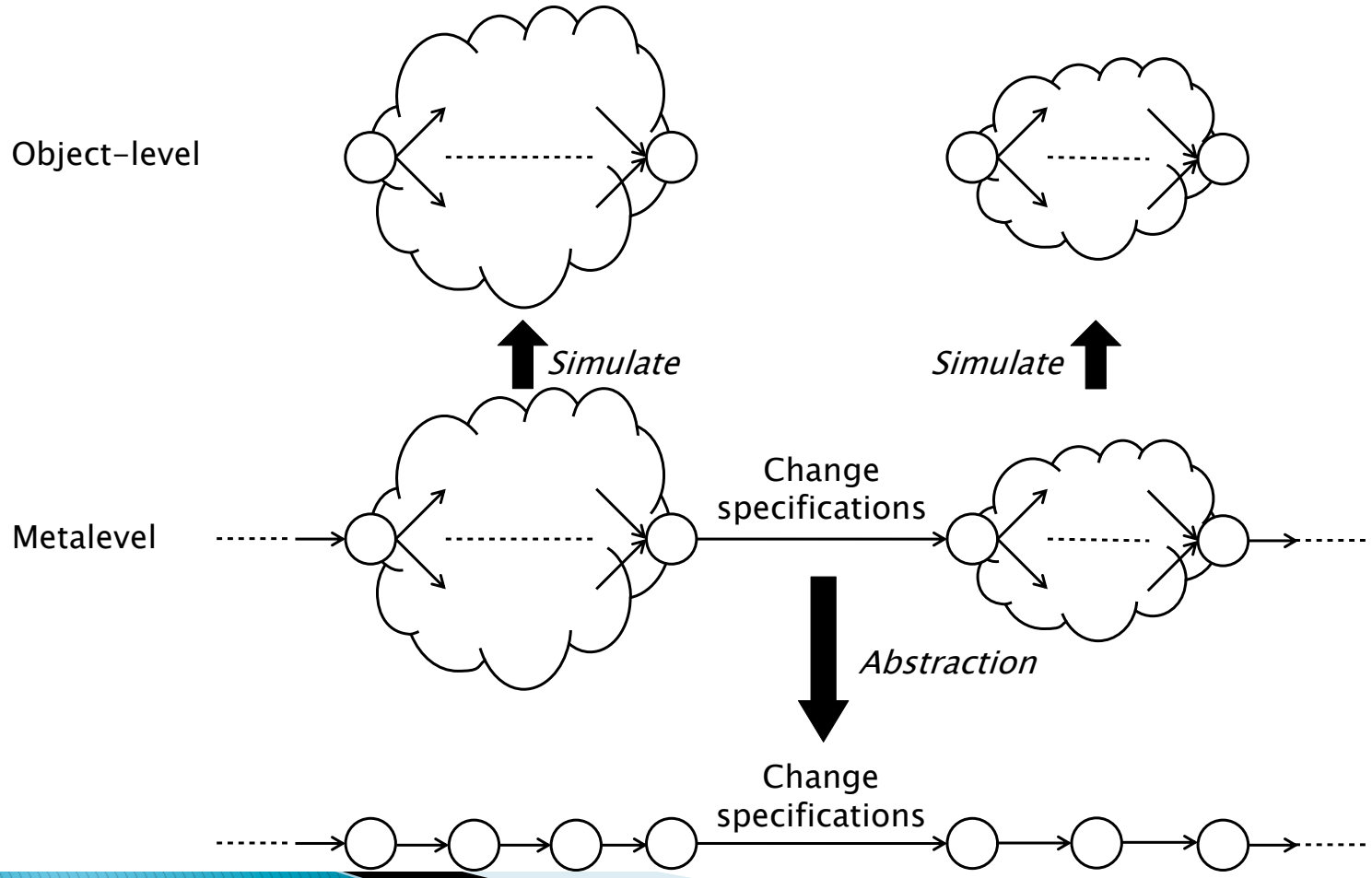
      where
      - T is a variable representing a term
      - r and r' are rewrite theories representing the specifications before and after the change

Proposed Approach

- Abstraction
  - Conduct equational abstraction on the object-level behaviors of both before and after evolution
  - Metalevel behaviors simulating the object-level are also abstracted
    - $R \models t = t'$ if and only if $U \models \overline{t} @ \overline{R} = \overline{t'} @ \overline{R}$
    - If a set of equations E is added to R (written as $R \cup E$), E also defines an equivalence relation for the terms $\overline{t} @ \overline{R \cup E}$
  - Action of changing object-level specifications remains

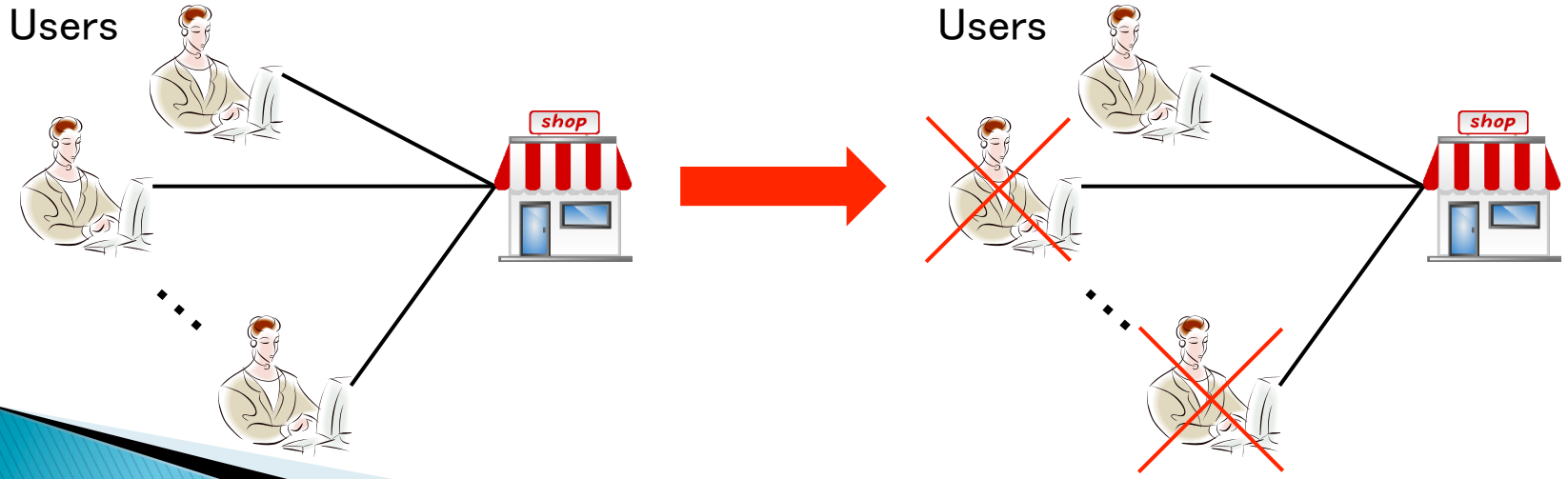
Proposed Approach

- Validation of Abstraction



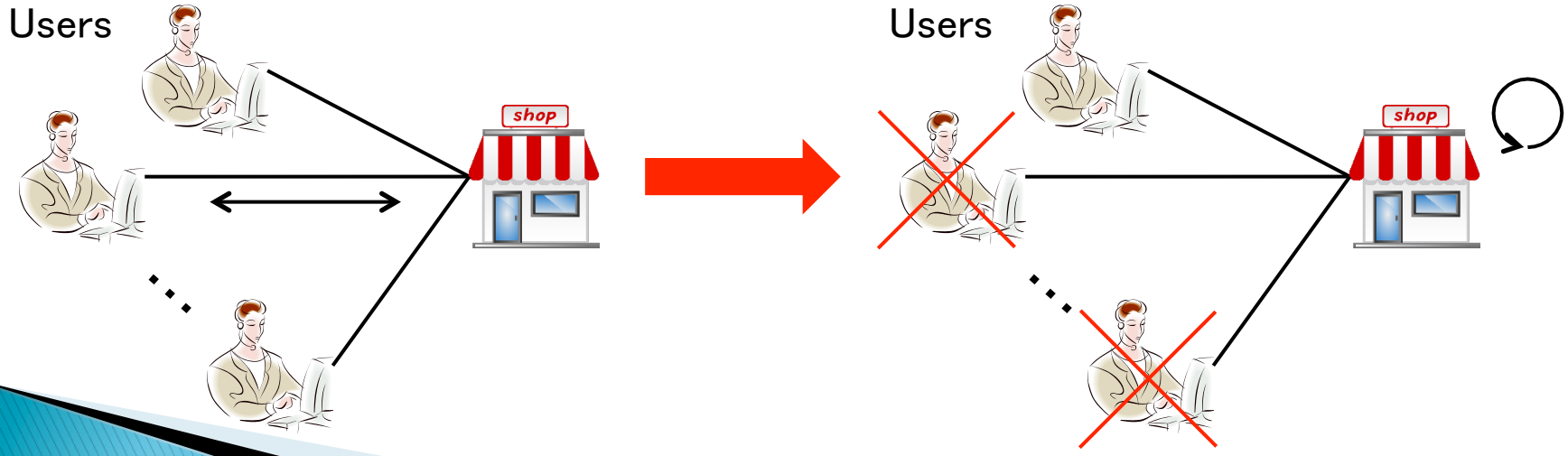
Proposed Approach

- Abstraction applied to our example
  - Model checking focuses on only one user
  - Behaviors of all of the other users are abstracted way
  - Specific expressions in the Maude specification carry out this abstraction



Proposed Approach

- Rationale of this abstraction
  - The behaviors involved with the abstracted users do not affect the behaviors involved with the remaining user
  - The former behaviors are abstracted to the self-loop transition
  - Instead, the fairness assumptions are required



Experiments

- First evolution: addition of the authentication functionality
  - Verified property: anytime the users can access the shop and the shop properly deals with the users' orders
    - Under the assumption that the system treats all the users fairly
  - Verification time without abstraction (in milliseconds)

No. of users	Before evolution	During evolution
1	80	120
2	200	1084
3	2432	42956

  - With abstraction: in all cases, verification take the same time as the case of one user

Experiments

- Second evolution: addition of the two-factor authentication functionality
  - Verified property: the same
  - Verification time without abstraction (in milliseconds)

No. of users	Before evolution	During evolution
1	644	696
2	1948	3124
3	43772	117252

  - With abstraction: in all cases, verification take the same time as the case of one user

Discussions

- Applicability of abstraction
  - The main reason of the success of abstraction: The behaviors involved with the abstracted users do not affect the behaviors involved with the remaining user
  - This situation does not hold in general
    - E.g. If the quantity of products is considered, a user's purchase may make the stock empty and other users cannot buy anymore
- Properties to be verified
  - Our example: only liveness properties
  - We must examine what properties need to be verified for dynamic evolution
    - E.g. [Zhang et al., 2009] proposed A-LTL (an adapt-operator extension to LTL)

Discussions

- Counterexample analysis
  - Currently the engineers need to analyze the Maude expressions
  - Supports such as the use the sequence diagrams are desired

Related Work

- Many formal models and verification approaches for dynamic evolution
- [Zhang et al., 2009]: The assume-guarantee style modular model checking
- [Filieri and Tamburrelli, 2013]: Efficient probabilistic model checking
- [Ghezzi and Sharifloo, 2013]: hybrid approach including both design-time and runtime verification

Related Work

- [Ghezzi et al., 2013]: Use of probabilistic model checking to adapt the system behaviors optimizing non-functional features
- [Bartels and Kleine, 2011]: Use of process algebra CSP for model checking of self-adaptive systems
- [Bruni et al., 2013]: Use of Maude for probabilistic model checking of MAPE-K loop
- [Tajalli et al., 2010]: Use of formal models of ADLs to produce correct configurations after evolution
- There are no studies of formal verification of dynamic evolution using reflection with abstraction

Future Work

- Solutions to the limitations
- Use of other verification techniques
  - Modular model checking
  - Probabilistic model checking
- Use of other dynamic evolution approaches
  - Architecture-based
  - Models @ run time
  - RE @ run time
  - DSPL
  - Aspects
  - Contexts