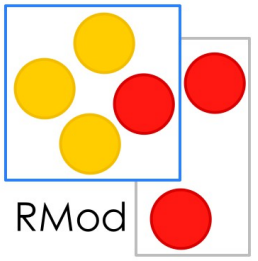
Instance Migration in Dynamic Software Update

P. Tesone[1,2] – G. Polito[1] – L. Fabresse[2]
N. Bouraqadi[2] – S. Ducasse[1]

[1]INRIA Lille–Nord Europe, France
[2]Mines Douai, IA, Univ. Lille, France



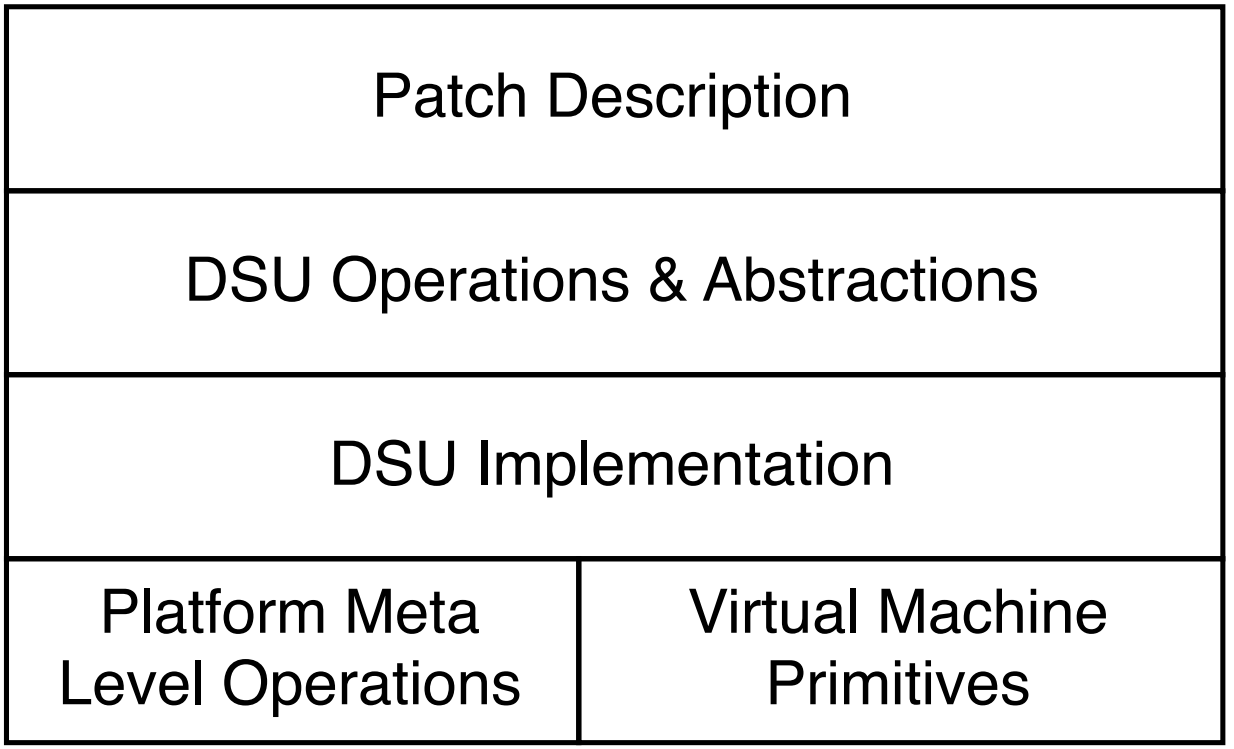
What is Dynamic Software Update?

- Updating a running application. Without needing to reinitialize the application.
- Guaranteeing the correct continuity of the execution.
- Evolving from one version to another (Code + State)
- Without losing state.
- Minimizing the downtime.

Two Use Cases

- Applications that can not be stopped (ex: global web applications, robotics, etc).
- Live Development Environments (ex: modifying data structures, complex refactors, self modification).
- Similar requirements:
  - Migration of state.
  - Atomicity in the changes.
  - Validation of the changes.
- But with some differences
  - Size of the changes.
  - Downtime.
  - Concurrence.

Generic Update Process Architecture.



Some challenges of Dynamic Software Update

- How to calculate the changes needed from one version to another.
- When to perform the changes.
- What to do with the running threads.
- How to migrate the data from one version to another.
- How to validate the changes.
- Rollback of the changes.

Instance Migration

- In a OO Environment the state of the application is represented in instances inside the environment.
- The current state has to be preserved between versions.
- Any change in the structure or the usage of the objects has to be updated.

Possible Changes

- Creating Instance Variables
- Removing Instance Variables
- Renaming Instance variables.
- Change in the value / usage of an instance variable.
- All this changes can be applied to a hierarchy, needing to be propagated.

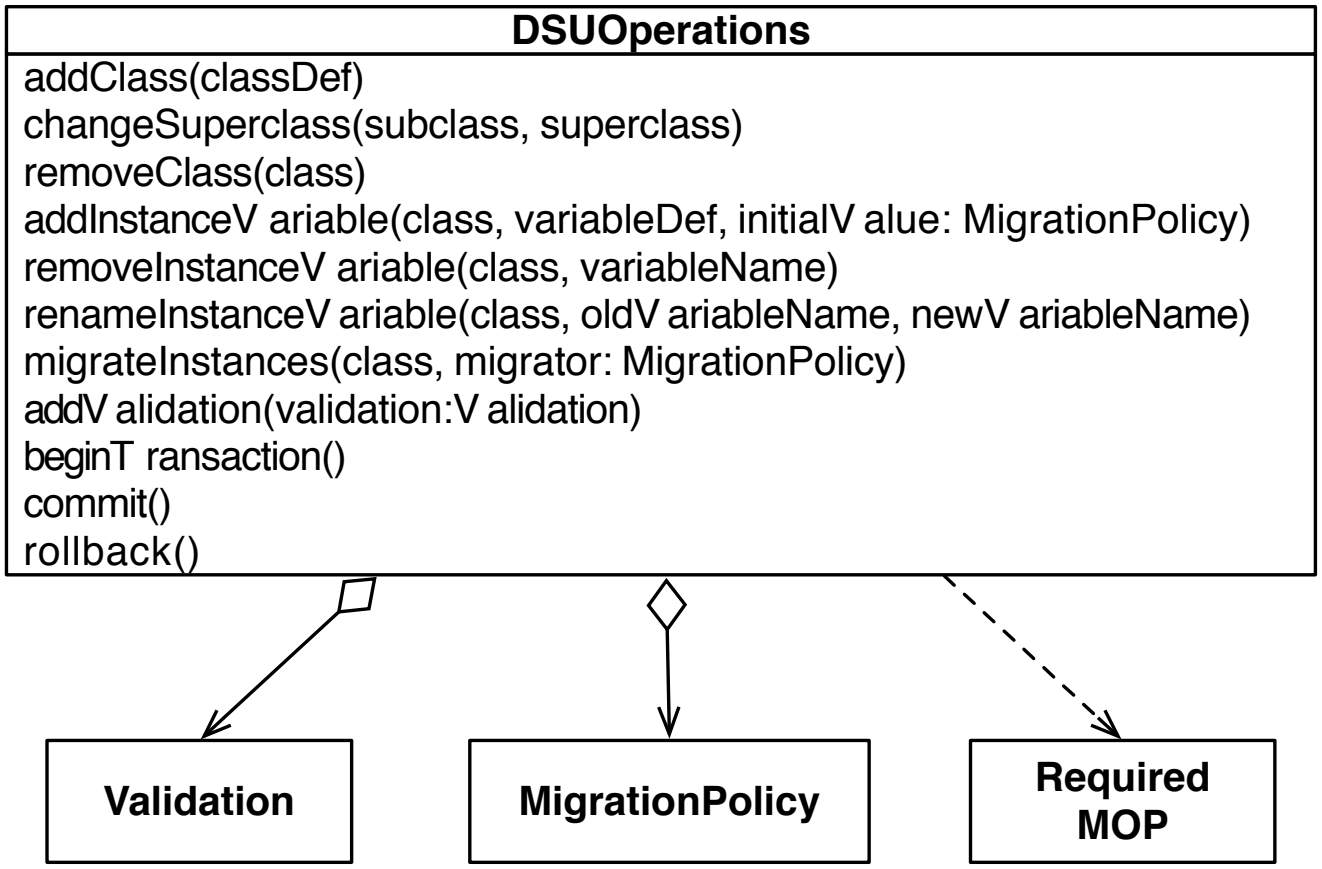
Instance Migration Requirements

- Atomicity
- Eager / Lazy
- Support of Application dependent migrations.
- Support of System and Application Validations
- Coherence
- Composability
- Support of both use cases.
- Handling global state.

Proposed Solution

1. The patch declares all the changes to perform.
2. The update process copy the namespace.
3. The update process performs all the changes in the new namespace.
4. The update process migrates all the instances to its new format (if needed).
5. The new namespace can be validated
6. If everything is correct all the old objects are replaced with the new objects, replacing the existent namespace with the new one (Bulk Replace).

Proposed Interface to Express the Changes.

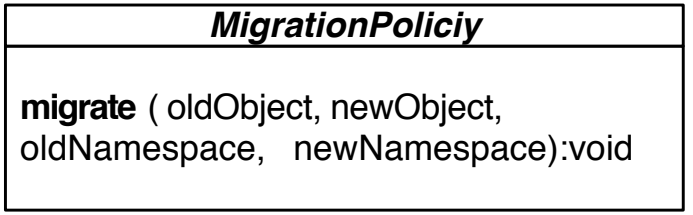


Required Operations

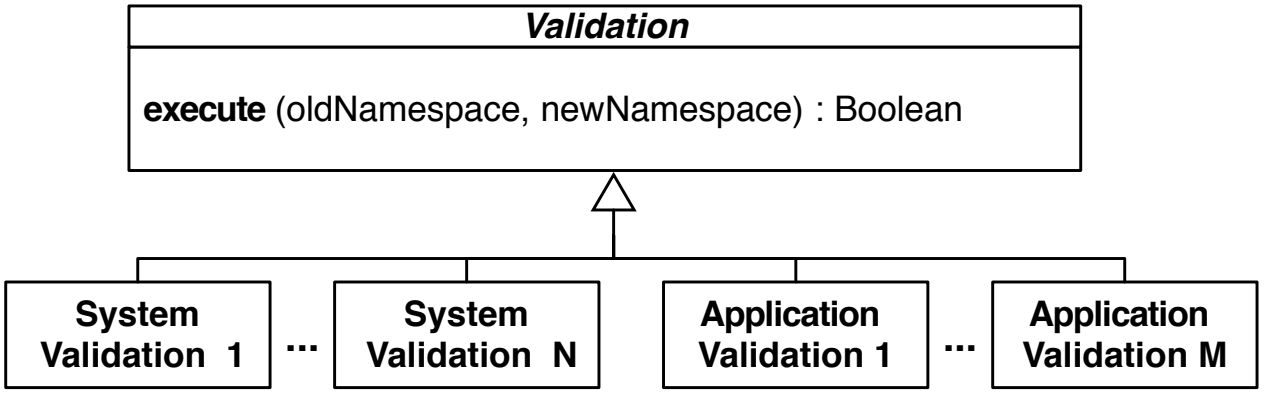
RequiredMOP
readClassDefi nition(Class):ClassDef
createClass(classDef):Class
readInstanceV ariable(instance, variableName): value
writeInstanceV ariable(instance, variableName, value)
allInstances(Class):List
cloneNamespace(namespaceName):Namespace
replaceNamespace(namespaceName,newNamespace:Namespace)
swapObjects(oldObjects:List, newObjects:List)

Detected Abstractions

- Migration Policies
  - General migration policies
  - Application Dependent



- Validations
  - System consistence validations
  - Application's Validations



Implementation Details

- Implemented as a Prototype in Pharo.
- Supporting the update in the two use cases.
- Implemented as Image side component, no VM modification needed.
- The full lifecycle of the update process is implemented.
- Allows research in different alternatives to the different aspects.
- Simple solutions for some aspects:
  - Patch creation
  - Quiescent point detection
  - Stack manipulation

Future work

- Studying the differences when using lazy migration.
- Restricting the number of objects to copy.
- Minimize the downtime (or maximize the operations that can be done without stopping the application)
- Improving the detection of changes and providing more automatic generated migration policies.
- Smarter detection of Quiescent points.

Thank you so much!