Algebraic Formalization and Verification of Behavioral Correctness of Dynamic Software Updating

Min Zhang, Kazuhiro Ogata, Kokichi Futatsugi

JAIST

Workshop on CafeOBJ and Specification Verification
2013.03.12

1 accepted by 2nd workshop on Validation Strategy of Software Evolution (VSSE), 2013
1. Dynamic software updating (DSU)

2. The correctness of DSU

3. Formalization by OTS (observational transition system)

4. Verification:
   ▶ theorem proving (CafeOBJ)
   ▶ model checking (Maude)
Dynamic software updating (DSU)

DSU

a technique for updating *running* software systems on the fly without incurring downtime.

Suitable for those *mission-critical* systems that provide 24x7 services, e.g.,

- Hosted email systems
- Consumer online banking system
- Traffic control systems
Dynamic updating is “dangerous”

It may cause errors that neither occur in the old program nor in the new program.

Example (A “bad” update)

```java
1  int i=2;
2  void main(){
3      while(1){
4          foo();
5      }
6  }
7  void foo(){
8      i--;  // old version
9  }
```

```java
1  int i=2;
2  void main(){
3      while(1){
4          foo();
5      }
6  }
7  void foo(){
8      i = 100/i+1;  // new version
9  }
```

Output: 2 1 0 EXCEPTION

We need to guarantee the correctness of updating
Contributions

Three research questions:

▶ What does correctness mean?
▶ How to formalize DSU
▶ How to verify

Our Contribution:

1. The notion of behavior-level correctness
2. An algebraic approach to formalizing dynamic software updating
3. Verifications by using theorem proving (in CafeOBJ) and model checking (in Maude)
Behavior-level correctness

Code-level correctness (existing work):
▶ whether functions refer to data of correct version.
▶ whether functions call sub-functions of correct version.

They focus on the correctness of the implementation of dynamic updating.

By behavior-level correctness, we mean:
▶ how system’s behaviors and states are changed by updating
▶ if desired behaviors can be eventually achieved
▶ no unexpected behaviors

We emphasize the correctness of the design of dynamic updating.
An example of dynamic updating

From a flawed mutual exclusion protocol (A) to a correct one (B)

Loop “remainder section”
rs: repeat until
locked = false;
ps: locked := true;
“critical section”
cs: locked := false;

Loop “remainder section”
rs: repeat while
fetch&store(locked, true);
“critical section”
cs: locked := false;

Protocol A

Protocol B
An example of dynamic updating

1. Update can be started any moment during the exclusion of the system

2. When updating starts, $locked_B$ is initialized with the value of $locked_A$.

3. During updating:
   - if a process is in the remainder section, it begins to execute new protocol next step.
   - if not, a process will continue to run until the current loop is done.

4. After updating, All processes execute the new system.
Three dynamic updating models

We need to know how system’s behavior is changed by update.

Invoke model

old system | notify | call update | return new system

Interrupt model

old system | interrupt | update conditions met | resume new system

Relaxed consistency model

old system | interrupt | update conditions met | resume new system

...
An abstract dynamic updating model

Three steps:

1. **prepare** *(notify in invoke model, and interrupt in the other two)*
2. **update** *(applying update: transform the state)*
3. **finish** *(finishing the update)*
Observational Transition System (OTS)

OTS is an algebraic way of formalizing computer systems. An OTS is a triple $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$

- $\mathcal{O}$: a set of functions (observers) that return values for given states
- $\mathcal{I}$: a set of initial states
- $\mathcal{T}$: a set of functions (transitions) that construct states for given states
An OTS of protocol A

The OTS of protocol (A): $S_A \triangleq \langle O_A, I_A, T_A \rangle$.

$O_A \triangleq \{pc_A : \Upsilon \times P \rightarrow L, \text{locked}_A : \Upsilon \rightarrow \text{Bool}\}$

$I_A \triangleq \{v_0 \mid pc_A(v, p) = rs \land \neg \text{locked}_A(v_0)\}$

$T_A \triangleq \{\text{wait}_A : \Upsilon \times P \rightarrow \Upsilon, \text{enter}_A : \Upsilon \times P \rightarrow \Upsilon, \text{exit}_A : \Upsilon \times P \rightarrow \Upsilon\}$

- wait...
- $c\text{-}\text{enter}_A : \Upsilon \times P \rightarrow \text{Bool}$.  
  $c\text{-}\text{enter}_A(v, p) = (pc_A(v, p) = ps)$.

When $c\text{-}\text{enter}_A(v, p)$ is true:

- $pc_A(\text{enter}_A(v, p), p') = \begin{cases} 
  \text{cs} & \text{if } p = p' \\
  pc_A(v, p') & \text{otherwise}
\end{cases}$
- $\text{locked}_A(\text{enter}_A(v, p)) = \text{true}$
- exit...
Formalizing the four-phase dynamic updating

Definitions of the three transitions \(\text{prepare}, \text{update}, \text{and} \space \text{finish} \):  

- **prepare \(\gamma \rightarrow \gamma\).**
  Effective condition: \(c\)-prepare \(\gamma \rightarrow \text{Bool}\),
  \(c\)-prepare\((v) = \text{phase}(v) = \text{pre-updating}\).
  When \(c\)-prepare\((v)\) is true, \(\text{phase}(\text{prepare}(v)) = \text{waiting}\).

- **update \(\gamma \rightarrow \gamma\).**
  Effective condition: \(c\)-update \(\gamma \rightarrow \text{Bool}\),
  \(c\)-update\((v) = (\text{phase}(v) = \text{waiting}) \land \text{UpdateCondition}\).
  When \(c\)-update\((v)\) is true:
  \(\text{phase}(\text{update}(v)) = \text{updated}\)
  \(\text{Obs}_{\text{new}}(\text{update}(v)) = \text{Obs}_{\text{old}}(v)\)

- **finish \(\gamma \rightarrow \gamma\).**
  ...

...
Formalizing the four-phase dynamic updating

Suppose that the old and new systems are formalized by two OTSs $S_o$ and $S_n$ such that $S_o = \langle \mathcal{O}_o, \mathcal{I}_o, \mathcal{T}_o \rangle$ and $S_n = \langle \mathcal{O}_n, \mathcal{I}_n, \mathcal{T}_n \rangle$. An update from $S_o$ to $S_n$ can be formalized as follows:

Modeling a dynamic update as an OTS: $S_u \triangleq \langle \mathcal{O}_u, \mathcal{I}_u, \mathcal{T}_u \rangle$.

\begin{align*}
\mathcal{O}_u & \triangleq \mathcal{O}_o \cup \mathcal{O}_n \cup \{ \text{phase} : \mathcal{Y} \rightarrow U \} \\
\mathcal{I}_u & \triangleq \{ \nu_0 | \nu_0 \in \mathcal{I}_o \wedge \text{phase}(\nu_0) = \text{pre-updating} \} \\
\mathcal{T}_u & \triangleq \mathcal{T}_o \cup \mathcal{T}_n \cup \{ \text{prepare} : \mathcal{Y} \rightarrow \mathcal{Y}, \text{update} : \mathcal{Y} \rightarrow \mathcal{Y}, \text{finish} : \mathcal{Y} \rightarrow \mathcal{Y} \}
\end{align*}

Note: $\mathcal{O}_o \cap \mathcal{O}_n = \emptyset$, and $\mathcal{T}_o \cap \mathcal{T}_n = \emptyset$ (easily achieved by renaming)

$U = \{ \text{pre-updating, waiting, updated, post-updating} \}$
An OTS of dynamic updating from protocol A to B

Assume that $S_A \triangleq \langle O_A, I_A, T_A \rangle$ for protocol A, $S_B \triangleq \langle O_B, I_B, T_B \rangle$ for protocol B, and $O_A \cap O_B = \emptyset$, and $T_A \cap T_B = \emptyset$.

The OTS of the dynamic update from protocol (A) to (B): $S_I = \langle O_I, I_I, T_I \rangle$

$O_I \triangleq O_A \cup O_B \cup \{\text{phase, evolved}\}$

$I_I \triangleq \{v_0 | v_0 \in I_A \land \text{phase}(v_0) = \text{pre-updating} \land \neg \text{evolved}(v_0, p)\}$

$T_I \triangleq T_A \cup T_B \cup \{\text{prepare, update, finish, evolve}\}$

$evolved : \Upsilon \times Pid \rightarrow \text{Bool}$:

to indicate whether a process has involved

$evolve : \Upsilon \times Pid \rightarrow \Upsilon$:

to specify the update of a process from protocol A to B
An OTS of dynamic updating from protocol A to B

Changes of the definition of effective conditions:

\[ c\text{-}wait_A(v, p) = pc_A(v, p) = rs \land \neg \text{locked}_A(v) \land \text{phase}(v) = \text{pre}\text{-}updating. \]
\[ c\text{-}evolve(v, p) = pc_A(v, p) = rs \land \neg \text{locked}_A(v) \land \text{phase}(v) = \text{post}\text{-}updating. \]
\[ c\text{-}enter_A(v, p) = pc_A(v, p) = ps \land (\text{phase}(v) = \text{pre}\text{-}updating \lor \text{phase}(v) = \text{post}\text{-}updating). \]
\[ \ldots c\text{-}exit_B(v, p) = pc_B(v, p) = cs \land \text{phase}(v) = \text{post}\text{-}updating. \]
Verification

Tools supporting the verification of OTSs:
- CafeOBJ an a proof assistant
- Maude’s model checking facilities
Desired properties of the updating

1. After updating, the system should enjoy the mutual exclusion property
2. After updating, no deadlock will occur

Definition

The update from protocol A to B is correct if the above two properties are satisfied.
Verification by proof score in CafeOBJ

Mutual exclusion property of the updated system

\[
\text{op } \mu : \text{Sys Pid Pid } \rightarrow \text{Bool} \\
\text{eq } \mu(S,I,J) = \\
(\text{pc-B}(S,I) = \text{cs} \text{ and } \text{pc-B}(S,J) = \text{cs}) \implies I = J.
\]

We can prove by theorem proving in CafeOBJ, with about 600 lines of code.
Verification by model checking in Maude

- The system after being updated should enjoy *deadlock free property*.
- After updating, all process will eventually execute the new protocol.

**Deadlock free property**

The system can never go to a *deadlock state* where each process is waiting for others to continue.

We consider the case where there are only two processes $p_1$ and $p_2$ in the system.

Two state predicates

- @rs?_ : Pid → Prop
- locked? : → Prop

Formula of deadlock free property:

\[
\Box \neg (\text{@rs?(p1)} \lor \text{@rs?(p2)} \lor \text{locked?})
\]
Verification by model checking in Maude


A counterexample:

What we can learn from the counterexample.

*When some processes are in critical section, updating at that point may cause deadlock.*
Possible solutions

1. update condition: \( \text{locked}_A(v) = false \)

2. set \( \text{locked}_B \) false when updating, i.e., \( \text{locked}_B(\text{update}(v)) = false \)

We can show no deadlock happens after updating.
Conclusion

The behavior-level correctness of DSU
An algebraic approach to modeling and verifying DSU.

Future work:
- case studies on practical dynamic updating
- to (semi-automate) the process of generating OTS for dynamic updating
- to formalize dynamic updating mechanism at the code level (by reflection of rewriting logic)
  - object-level: program + state
  - meta-level: mechanism
Questions?