A Lightweight Integration of Theorem Proving and Model Checking for System Verification

Weiqiang Kong, Takahiro Seino, Kazuhiro Ogata, and Kokichi Futatsugi

Graduate School of Information Science,
Japan Advanced Institute of Science and Technology

Outline of the talk

- Background and motivation
  - Comparison between theorem proving and model checking.
  - Target point in theorem proving that we focus on
  - Verification flow of the lightweight integration.
- The translator – Cafe2Maude
  - Data type module translation
  - OTS module translation
  - Invariant property defining module translation
  - Initial state generation
- Conclusion and Future work

Part I: Background and motivation

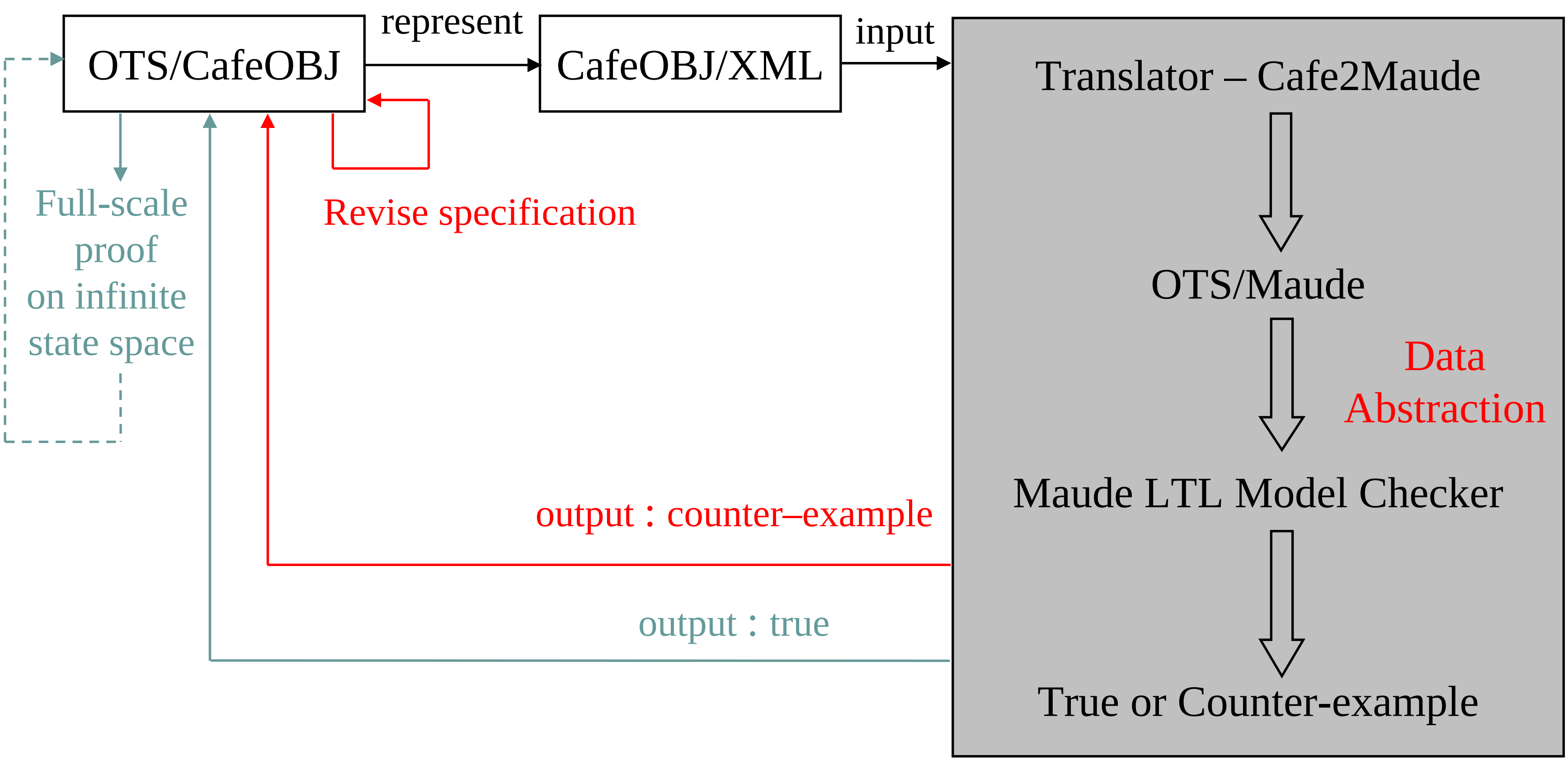
A general comparison of typical theorem proving and model checking:

	Theorem proving	Model Checking
State space	Infinite	Finite
Verification procedure	Limited automatic	Fully automatic
Counter-example	No automatic	Automatic
Obtaining insight of the system	Tell how the system is correct	Tell how the system is incorrect

Our target point in theorem proving

- In case that a property fails to hold
  - Difficult to extract enough information from the verification result
    - Errors exist in specifications? If so, where?
    - Need more guidance to complete the proof?
  - Considerable time is used to discover and prove auxiliary invariants.
- If counter-example can be generated automatically
  - Easier to find out the reason for the failure
  - Benefit from firstly model checking the newly founded invariant:
    - If counter-example, then revise specifications or discard the invariant
    - If true, then there might exist a proof for the invariant
- To able to find "bugs" in the early stage of verification (before we write proofs manually) and ease the hard-work of theorem proving.

Verification flow when using Cafe2Maude

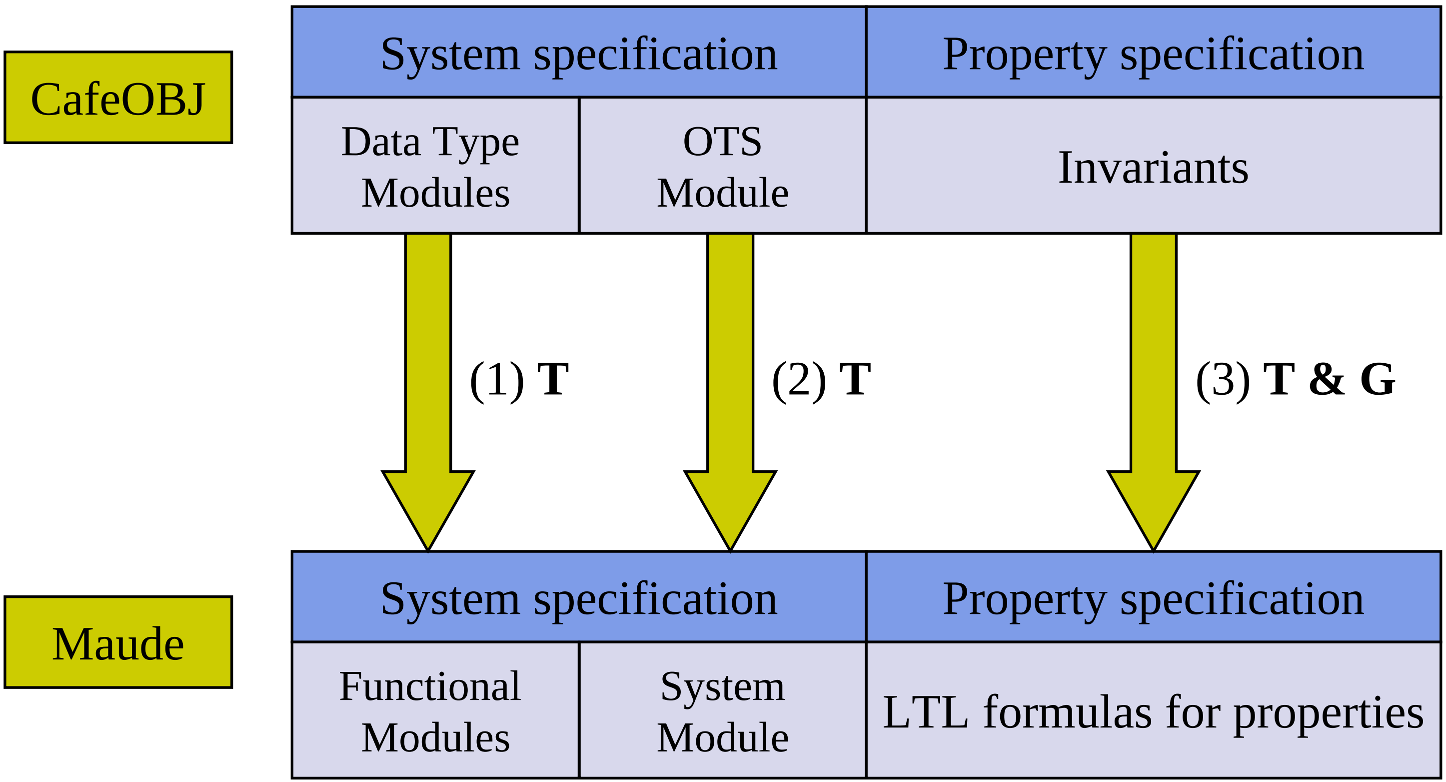


Why called “lightweight”

- Good aspects: the formalisms of OTS/CafeOBJ and OTS/Maude are quite similar (both based on equations).
  - Equations are easy to understand and use.
  - Similar formalisms can alleviate the burden for the users to learn two different formalisms.

- Bad aspects: the data abstraction method we used may not preserve soundness.
  - The abstracted model may has some property that does not hold in the original model.
    - But, this simple abstraction method is effective when aim to exposing bugs

Part II: Cafe2Maude introduction



Given a user's input of data abstraction:
T : Translation   G : (Initial State) Generation

A Mutual Exclusion Algorithm

Pseudo-code of the mutual exclusion algorithm:

l1 : *put(queue, i)*
l2 : repeat until *top(queue) = i*
Critical Section
cs : *get(queue)*

Initially, each process *i* is at label *l1* and *queue* is empty.

- The algorithm is modeled as an OTS <*O*, *I*, *T* >:
  - Observers: *queue* and *pc*
  - Transition rules: *wait*, *try* and *exit*

Data type module translation

CafeOBJ Data Type Module	Maude Functional Module

mod! LABEL {
[Label]
ops l1 l2 cs : -> Label
op _=_ : Label Label -> Bool {comm}
var L : Label
eq (L = L) = true .
eq (l1 = l2) = false .
eq (l1 = cs) = false .
eq (l2 = cs) = false .
}
	fmod LABEL is
	sort Label .
	ops l1 l2 cs : -> Label .
	endfm

- Other two data type module PID and QUEUE are translated similarly.

OTS module translation (1)

CafeOBJ OTS module – signature	Maude system module
-- hidden sort declaration *[Sys]*	subsort OValue TRule < Sys . op none : -> Sys . op _ _ : Sys Sys -> Sys [assoc comm id : none]
-- observer declaration bop o : Sys V_{i_1} … V_{i_m} -> V -- (m >= 1) bop o : Sys -> V -- otherwise	op (o[_,…,_] : _) : V_{i_1} … V_{i_m} V -> OValue . op (o : _) : V -> OValue .
-- transition rule declaration bop t : Sys V_{i_1} … V_{i_m} -> Sys	op t : V_{i_1} … V_{i_m} -> TRule .

OTS module translation (Example 1)

CafeOBJ operator declarations	Maude operator declarations
-- observers bop pc : Sys Pid -> Label bop queue : Sys -> Queue -- transition rules bop want : Sys Pid -> Sys bop try : Sys Pid -> Sys bop exit : Sys Pid -> Sys	*** Observers op pc[_] : _ : Pid Label -> OValue . op queue : _ : Queue -> OValue . *** transition rules op want : Pid -> TRule . op try : Pid -> TRule . op exit : Pid -> TRule .

OTS module translation (2)

CafeOBJ OTS module – equations	Maude system module – transition rule
-- equations defining state transition Given a transition rule $t_{j_1,\ldots,j_n}$ denoted by t, and the observers needed and affected (return value is changed) by this transition rule are $o_1,\ldots,o_l$, the equations are translated to one (conditional) rewrite law as follows:	*** Maude rewrite law crl [relaw] : $t(X_{j_1},\ldots,X_{j_n})$ $(o^1[X_{i_1}{}^1,\ldots,X_{i_{m_1}}{}^1] : X_1)\ \ldots\ (o^l\,[X_{i_1}{}^l,\ldots,X_{i_{m_1}}{}^l] : X_l\,)$ => $t(X_{j_1},\ldots,X_{j_n})$ $(o^1[X_{i_1}{}^1,\ldots,X_{i_{m_1}}{}^1] : X_1{}')\ \ldots\ (o^l\,[X_{i_1}{}^l,\ldots,X_{i_{m_1}}{}^l] : X_l{}')$ if $c\text{-}t(X_{j_1},\ldots,X_{j_n}, X_{i_1}{}^1,\ldots,X_{i_{m_1}}{}^1, X_1, X_{i_1}{}^l,\ldots,X_{i_{m_1}}{}^l, X_l)$.

OTS module translation (Example 2)

CafeOBJ equations defining action	Maude rewrite law defining action

op c-want : Sys Pid -> Bool
eq c-want(S,I) = (pc(S,I) = l1) .
ceq pc(want(S,I),J) =
        (if I = J then l2 else pc(S,J) fi)
        if c-want(S,I) .
ceq queue(want(S,I)) = put(queue(S),I)
        if c-want(S,I) .
ceq want(S,I) = S if not c-want(S,I) .
	crl [want] :
	want(I) (pc[I] : LABEL) (queue : QUEUE)
	  =>
	want(I) (pc[I] : l2) (queue : put(QUEUE,I))
	if LABEL == l1 .

- Equations defining transition rules *try* and *exit* are translated similarly.

Property translation (1)

Procedure of model checking OTS using Maude.

- Given a Maude system module, say M
  - Defining a new module, say M-PREDS that defines state predicates.
  - Defining a new module, say M-CHECK that defines LTL formulas for properties.
  - Given an initial state init, model check defined properties

```
Maude>  red modelCheck(init, property)
```

Property translation (2)

Properties to be proved for the mutual exclusion algorithm

predicates

```
mod INV {
Pr (QLOCK)
… -- constant, operator and variable declarations
eq inv1(S,I,J) = (pc(S,I) = cs and pc(S,J) = cs implies I = J) .
eq inv2(S,I) = (pc(S,I) = cs implies top(queue(S)) = I) .
eq inv3(S,I) = (pc(S,I) = l2 or pc(S,I) = cs implies not empty?(queue(S))) .
eq inv4(S,I) = (pc(S,I) = l2 implies I /in queue(S)) .
```

- An invariant consists of a set of predicates and logical connectives.
- What we need to do is to firstly extract these predicates and then define state predicates in the module M-PREDS

Property translation (3)

- Assumption: Each predicate has at most one observation operator. Predicates with two (or more) observation operators should be written separately. Such as pc(S,I) = pc(S,J), should be written as pc(S,I) = VAR and pc(S,J) = VAR.

- Predicates *without observation operator* (such as I = J):

$bool(V_1,\ldots,V_m) \Rightarrow S \models prop(V_1,\ldots V_m) = true \text{ if } bool(V_1,\ldots V_m)$.

  - Example
    - T => S |= prop(T) = true if T .
    - I = J => S |= prop(I,J) = true if I = J .

Property translation (4)

- Predicates *with observation operator*
  - In the form of normal observation equation

$o(S,V_1,\ldots,V_m) = term$

$=>$

$(o[V_1,\ldots,V_m] : term)\ S \mid= prop(V_1,\ldots,V_m , X_1,\ldots X_n) = true$.

* *term* contains no observation operator due to the assumption.

- Example:
  - pc(S,I) = cs => (pc[I] : cs) S |= prop(I) = true .

Property translation (5)

- Predicates *with observation operator*
  - Other non-normal ones

    $pred(\ldots, o(S, V_1, \ldots, V_m), \ldots)$
    =>
    $(o[V_1, \ldots, V_m] : VAR)\ S \models prop(V_1, \ldots, V_m, X_1, \ldots, X_n) = true$
    if $pred(\ldots, VAR, \ldots)$.

    - Example:
      - top(queue(S)) = I => (queue : VAR) S |= prop(I) = true
        if top(VAR) = I .
      - I /in queue(S) => (queue : VAR) S |= prop(I) = true
        if I /in VAR .

Property translation (Example)

- Translate the properties based on the declared *props*.

```
eq inv1(S,I,J) = (pc(S,I) = cs and pc(S,J) = cs implies I = J) .
```

```
eq (pc[I] : cs) S |= prop1(I) = true .
eq (pc[J] : cs) S |= prop2(J) = true .
eq S |= prop3(I,J) = true if I = J .
```

```
"and"      ---->   "/\"
"implies"  ---->   "->"
"[]"       ---->   "Always"
```

```
eq property1(I,J) = [] (prop1(I) /\ prop2(J) -> prop3(I,J)) .
```

Data abstraction for translated properties

- Simple data abstraction (reduction or valuation): reducing the infinite domain of each sort to some concrete values, where variables belonging to this sort occur in the formula for property.

```
[] (prop1(I) /\ prop2(J) -> prop3(I, J)) .
```

sort Pid ⇔ p1, p2

```
[] (((prop1(p1) /\ prop2(p1)) -> prop3(p1, p1))
  /\ ((prop1(p1) /\ prop2(p2)) -> prop3(p1, p2))
  /\ ((prop1(p2) /\ prop2(p2)) -> prop3(p2, p2))
  /\ ((prop1(p2) /\ prop2(p1)) -> prop3(p2, p1))) .
```

Initial state generation

CafeOBJ equations defining initial state, say init	Maude equations defining initial state

• eq pc(init,I) = l1 .
eq queue(init) = empty .
 	
• Information about
  • transition rules
  • data abstraction
	eq init = want(p1) try(p1) exit(p1)
	want(p2) try(p2) exit(p2)
	(pc[p1] : l1) (pc[p2] : l1)
	(queue : empty) .

Part III: Conclusion and future work

- Conclusion
  - Designed and implemented a translator from OTS/CafeOBJ to OTS/Maude. (using Java, currently about 4000 line codes)
  - Proposed a simple method to make theorem proving task easier by taking advantage of model checking.
- Future work
  - Doing more non-trivial case studies to convince people that our integration is useful
    - Secure workflow
    - Authentication and ecommerce protocols
  - Formally prove the correctness of the translation.

Thanks!