On Verifying Reactive Rules Using Rewriting Logic

Katerina Ksystra, Nikolaos Triantafyllou, and Petros Stefaneas

Roadmap

- Introduction
- Reactive rules and motivation
- Rewrite theory OTS
- Reactive Rules and Rewrite theory OTS
  - Detecting termination – confluence properties
  - Verifying safety properties
- Conclusions and Future Work

European Union
European Social Fund
OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS
MANAGING AUTHORITY
Co-financed by Greece and the European Union
NSRF 2007-2013
programme for development
EUROPEAN SOCIAL FUND

Introduction

- Central idea
  - Framework for formally expressing reactive rules
    - based on Rewrite Theory
- Why?
  - Formal analysis of reactive rule based systems
  - Verification of their behavior
- Benefits
  - Ability to detect structure errors and
  - prove safety properties in the same framework

Reactive Rules and Motivation

- Production rules, "If condition do action"
  - Specify the execution of an action in case some conditions are satisfied
  - reaction to states changes
- Event Condition Action (ECA) rules, "On event if condition do action"
- A rule reacts to detected events
  - by evaluating a (set of) condition(s) and
  - by executing a reaction whenever the event happens and the condition(s) is true.

Properties of interest

- Safety property:
  - a desirable property holds in all reachable states (i.e. is an invariant) of the rule-based system and is specific to the purpose of the specified application
- Confluence:
  - whether the result of executing a set of triggered rules depends on the execution order of the rules or not (all state paths lead to the same final state)
- Termination:
  - ensure that a set of rules will eventually terminate (i.e. reach a final state) and will not continue to trigger each other infinitely

Formal Specification of Reactive Rules

- Some first steps to formalize Reactive rules using algebraic specification techniques were presented in:
  - Ksystra, K., Triantafyllou, N., Stefaneas, P.: On the Algebraic Semantics of Reactive Rules. 6th International Symposium, RuleML 2012, Springer, (2012)
  - Ksystra, K., Stefaneas, P., Frangos, P.: An Algebraic Framework for Modeling of Reactive Rule-Based Intelligent Agents. SOFSEM: Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science, LNCS, 407-418 (2014).

Formal Specification of Reactive Rules

- In that work, we proposed OTS/CafeOBJ method to verify safety properties about reactive rule-based systems
  - *but* cannot express naturally structure properties
- In this paper we extend the previous approach by adopting the logical formalism of *rewrite theory specifications*
- So that *both* safety and structure properties can be formally checked in the same framework

CafeOBJ-Rewriting Logic Specification

- Algebraic specification language
  - executable by term rewriting
- Supports both *equational* theory and *rewrite* theory specifications
- State transitions are described in equations in the former and in rewriting rules in the latter.
  - Equational theory specification is used for interactive *theorem proving*
  - For rewrite theory specification CafeOBJ can conduct *exhaustive searches*
    - In a recent paper by CafeOBJ team, a way to theorem prove that rewrite theory specifications of OTSs have invariant properties is presented (combining the two approaches)!

Production rules and rewrite rules-CafeOBJ

- In a rewrite theory, states are expressed as collections of observable values
- Reactive rules can be formally defined as a set of rewrite theory specifications in CafeOBJ as follows:
- A production rule of the form *Ri = On Ci do Ai*
  - where Ai denotes a variable assignment can be naturally transformed into the rewrite rule
- `ctrans [Ri] (V: v0) D => (V: v1) D if Ci = true .`
- The above rewrite rule states that the observable value `V` will become `v1` if the condition of the rule is true.

Production rules and rewrite rules-CafeOBJ

- When the result of action Ai is the assertion of the fact ki to the knowledge base, its definition is the following;

```
ctrans [assert ki] (knowledge: K) D =>
knowledge : (ki U K) D if Ci /in K .
```

  - knowledge is the observable value corresponding to the knowledge base (KB) and it is defined as a set of boolean elements.
  - U is an operator for adding elements in a set
  - /in is an operator that returns true when an element belongs to a set, here the knowledge base
- This rule states that the fact *ki* will be added to the KB if the condition of the rule holds
- Similar is the definition when the action of the rule is *retract* or *update* the KB or another *generic action*

ECA rules and rewrite rules-CafeOBJ

- An Event Condition Action rule of the form *Ri := On Ei if Ci do Ai*, is defined in CafeOBJ terms as *two* transitions
  - The first one specifies the event *Ei*
  - the system after the detection of the event stores its "id" number in an special observable value called *event-memory* for remembering that event
- `ctrans [Ei] (event-memory: null) => (event-memory: i) if c-ei = true .`
- the value *null* of event-memory, denotes that no other event is detected at the pre state and `c-ei` is a boolean CafeOBJ term denoting the detection conditions for Ei.

ECA rules and rewrite rules-CafeOBJ

- The second transition rule specifies the action Ai
  - it defines that the system must respond to the detected event by performing the corresponding action
- The triggering of the action as a response to the event is simply defined by:
  - adding the condition that in the pre state the event memory will contain the index of the occurred event
- `ctrans [ai] (event-memory: m) (oi: vi) => (event-memory: null) (oi: vj) if Ci = true and (m = i) .`

ECA rules and rewrite rules-CafeOBJ

- This ensures that only the guard of this transition rule will hold at the pre state and thus
  - this will be the only applicable transition for that state of the system
- After the occurrence of the action, event-memory will become null again
  - denoting that the system is ready to detect another event

Running example

- A company's e-commerce web site
- (R1) If the Customer is Gold and his/her shipping cart is worth 2,000 or more then the discount is increased by 10 points.
- (R2) If the Customer is Platinum and his/her shipping cart is worth 1,000 or more then the discount is increased by 15 points.
- (R3) If the Customer is Gold and he is aged 60 or more he is promoted to the Platinum category

In CafeOBJ terms;

- ctrans [gold] : (category: G) (value: V) (age: N) (discount: M) => (category: G) (value: V) (age: N) (discount: (M + 10)) if ((V >= 2000) and (G = gold)) .
- ctrans [platinum] : (category: G) (value: V) (age: N) (discount: M) => (category: G) (value: V) (age: N) (discount: (M + 15)) if ((V >= 1000) and (G = platinum)) .
- ctrans [upgrade] : (category: G) (value: V) (age: N) (discount: M) => (category: platinum) (value: V) (age: N) (discount: M) if (G = gold) and (N >= 60) .

Detecting termination issues

- A state s is terminating if it leads to a state where no rules can be applied (final state)
  - We define the following predicate
- `op terminates? : State -> Bool`
- `terminates?(s) = s =(1,*)=>! (event-memory: M1) (value: V1) (o: N)`
- `red terminates?(s) .`
  - with the command red `t ==> p` CafeOBJ can traverse all the terms reachable from `t` wrt transitions in a breadth-first manner and find terms (called solutions) such that they are matched with `p`
- By reducing `terminates?`, we basically ask CafeOBJ to find a *final* state reachable from the state s (that's why `!` is used at the end of `==>`)

Detecting termination issues

- If true is returned (together with a final state) it means that the state s is terminating;
- if false is returned it means that in the state s, no transition can be applied;
- the CafeOBJ reduction may not terminate, indicating that s is not terminating.

Termination-example:

- Suppose we have the following state
- `op s : -> State .`
- `eq s = (category: gold) (value: 500) (age: 50) (discount: 0) .`
- `red terminates?(s) .`
- When we test this state, CafeOBJ returns false together with following message
- `** No more possible transitions.`
- `(false): Bool`
  - which is reasonable since no transition can be applied

Termination-example:

- `eq s = (category: gold) (value: 2000) (age: 50) (discount: 0)`
- `red terminates?(s) .`
- When we test the above case (where the gold rule can be applied on and on) the reduction of CafeOBJ does not halt indicating that this state is *not* terminating.
  - Having detected this issue we can add the constraint (discount: (M1 + 10) <= 100)) to the rule, since the discount cannot surpass this value.
- When the same case is tested after adding the constraint, CafeOBJ finds the final state;
- (category: gold) (value: 2000) (age: 50) (discount: 100) .

Detecting confluence issues

- A rule program's state s is non-confluent if there exist two traces from this state that lead to *distinct* (final) states.
  - We define the following predicate:
- `op notConfulent? : State -> Bool`
- `notConfluent?(s) =(2,*)=>! (event-memory: M1) (value: V1) (o: N) .`
- `red notConfluent?(s) .`
- The above reduction i.e. asks CafeOBJ to search if it can find starting from an arbitrary state s *two different final states* of the system
  - if two such solutions are found it means that the state s is not confluent
  - if false is returned and one solution is found, the state is confluent

Confluence-example

- Consider the following state:
- `eq s = (category: gold) (value: 500) (age: 60) (discount: 0)`
- `red notConfluent?(s) .`
- In this case where upgrade is the only applicable rule CafeOBJ returns *false*, as it finds one final state meaning that the state s is confluent.

Confluence-example

- Let us consider the state which is defined by the following observable values;
  - the value of the items of the cart is equal to 2000 dollars,
  - the age of the customer is 60 years old and
  - his category is gold.
- In this case:
- the customer is eligible to being granted the gold discount *and*
- being upgraded to the platinum category!

Confluence-example

- This state is defined as follows in CafeOBJ terms:
- `eq s = (category: gold) (value: 2000) (age: 60) (discount: 0)`
- `red notConfluent?(s) .`
- Indeed, when we test this case CafeOBJ returns true as it finds two solutions, denoting that s is not confluent, as we expected.
- In particular, it returns;
- `** Found [state 25] (category: platinum) (value: 2000) (age: 60)`
- `(discount: 90)`
- `** Found [state 27] (category: platinum) (value: 2000) (age: 60)`
- `(discount: 95)`

Confluence-example

- Then using the command `show path id` we can see the two transition paths that cause the problem (and then we can add constraints in the conditions of the rules to solve this issue, by letting for example the upgrade rule to be applied first).

Proving safety properties

- The built-in CafeOBJ search predicate can also be used to prove safety properties
- For the verification of such properties model checking and/or theorem proving can be used
- We will demonstrate through the running example how to use theorem proving in our framework

Safety property-example

- For our rule based system an invariant safety property could be the following;
  - A customer cannot belong to the platinum category if his/her age is less than 60 years.
- This property is expressed in CafeOBJ terms as;
- `op isSafe : State -> Bool .`
- `eq isSafe((category: G) (value: V) (age: N) (discount: M)) = not ((G == platinum) and (N < 60)) .`
- Then the proof is done by induction *on the number of transition rules of the system*

Safety property-example

- Starting from the initial state of our system
- `eq init = (category: gold) (value: 2000) (age: 50) (discount: 0) .`
- and using the above command:
- `red check(true, isSafe(init)) .`
- the base case is successfully discharged
  - operator *check* takes as input a conjunction of lemmas and/or induction hypotheses and a formula to prove and returns true if the proof is successful
- Ogata, K., Futatsugi, K.: Theorem Proving Based on Proof Scores for Rewrite Theory Specifications of OTSs

Safety property-example

- The inductive step consists of checking whether from an arbitrary state, say s, we can reach in one step a state, say s', where the desired property does not hold
- When *false* is returned
  - it means that CafeOBJ was unable to find a state s‘ such that the safety property holds in s and it does not hold in s’
- If a solution is found, i.e. the above term is reduced to *true*,
  - either the safety property is not preserved by the inductive step or we must provide additional input to the CafeOBJ machine (case analysis or lemma discovery)

Safety property-example

- Consider the inductive step where the *gold* transition rule is applied to s;
- `eq s = (category: gold) (value: 2000) (age: N) (discount: 0)`
- Using the above reduction
- `red s =(*, 1)=>+ s' suchThat(not check(isSafe(s),isSafe(s'))) .`
- CafeOBJ returns false, and thus this induction case is discharged.

Safety property-example

- Consider the case where the *platinum* rule is applied;
- `eq s = (category: platinum) (value: 2000) (age: N) (discount: 0) .`
- `red s =(*, 1)=>+ s' suchThat (not check(isSafe(s),isSafe(s'))) .`
- CafeOBJ returns false for this case, thus this induction case is aso discharged.
  - Following the same methodology the induction case for the upgrade rule was discharged as well, and thus the proof concludes.

Conclusions

- We have presented rewrite theory semantics for reactive rules
  - Goal:
- Detect termination and confluence errors
- Verify safety properties of reactive rule based systems
  - in the same framework

Future Work

- Future plans:
  - Conduct more case studies using the proposed methodology
  - Develop a tool for translating reactive to rewrite rules
- Extend the framework
  - To support reasoning about ontologies
  - Reasoning about reactive rules in combination with ontologies
- Verification of Rule engine implementation
  - Using design by contract methods

Thank you!

Questions?