Relating the MSR Crypto-Protocol Specification Language to Rewriting Logic with Dependent Types

Mark-Oliver Stehr
University of Illinois at Urbana-Champaign

Overview

- Introduction
- Multiset Rewriting (MSR) Specification Language
- Rewriting Logic (RWL) with Dependent Types
- Representing Types and Protocol State of MSR
- Transforming MSR Rules into First-Order Rewrite Rules
- Optimizations of the Transformation
- Running Example
- Conclusion

MSR Specification Language

- Atomic Types:
  - princ, nonce, msg
- Dependent Types:
  - shK A B, pubK A, privK k, stK A B, ltK A B
- Subtyping:
  - princ <: msg, nonce <: msg,
  - pubK A <: msg, privK k <: msg, stK A B <: msg
  - stK AB, <: shK A B, ltK A B <: shK A B

◦ Data Access Specification

MSR Specification Language

- State is Multiset of Message Predicates:
  - N(t) represents info on the network
  - $L(t_1,\ldots,t_n)$ represents state of a role
  - $M_A(t_1,\ldots,t_n)$ represents memory of principal A
- Generic Roles:
  - $\forall A \;.\; \exists\, L : \tau_1,\ldots,\tau_n \;.\; \exists \ldots$

    $\forall\, x : \sigma \quad . \quad LHS \quad \Rightarrow \quad \exists\, n : \rho \;.\; RHS,$

    $\forall \quad \ldots \quad . \quad \ldots \quad \Rightarrow \quad \exists \quad \ldots \quad . \quad \ldots$
  - arbitrary number of $\exists\, L : \tau_1,\ldots,\tau_n$ and rules per role
  - arbitrary number of $\forall\, x : \sigma$ and $\exists\, n : \rho$ per rule
- Anchored Roles:
  - A is fixed instead of quantified

MSR Specification Language

$\forall A \;.\; \exists L : \tau_1,\ldots,\tau_n \;.$

$\forall\, x : \sigma \;\; . \;\; LHS \;\; \Rightarrow \;\; \exists\, n : \rho \;.\; RHS,$

$\forall \;\; \ldots \;\; . \;\; \ldots \;\; \Rightarrow \;\; \exists \;\; \ldots \;\; . \;\; \ldots$

Operational Semantics:

a. instantiate A with arbitrary principal (if not anchored)

b. instantiate L with fresh symbol

c. add resulting role instance to active role set

1. select role instance from active role set
2. select an arbitrary rule from this role instance
3. instantiate x with arbitrary term of type σ
4. instantiate n with fresh symbol
5. apply rule instance to current state
6. remove the rule from active role set

} can be skipped

Example

Otway-Rees Authentication Protocol:

1. A -> B : n A B $\{n_A\ n\ A\ B\}_{kAS}$
2. B -> S : n A B $\{n_A\ n\ A\ B\}_{kAS}$ $\{n_B\ n\ A\ B\}_{kBS}$
3. S -> B : n $\{n_A\ k_{AB}\}_{kAS}$ $\{n_B\ k_{AB}\}_{kBS}$
4. B -> A : n $\{n_A\ k_{AB}\}_{kAS}$

Note:

- A,B range over arbitrary principals
- S is a fixed principal, the key server
- k_{AS}, k_{BS} are long term keys
- k_{AB} is the new short term key

Specification in MSR

Generic Role B

$\forall$ B . $\exists$ L : princ$^{(B)}$ x nonce x nonce x ltK B S .

$\forall$ A : princ. $\forall$ n : nonce . $\forall$ k_{BS} : ltK B S . $\forall$ X : msg .

N(n A B X) =>

$\exists$ n_B : nonce . N(n A B X $\{n_B$ n A B$\}_{kBS}$)

L(B,A,n,n_B,k_{BS}),

$\forall$ A : princ. $\forall$ n,n_A : nonce.

$\forall$ k_{BS} : ltK B S . $\forall$ k_{AB} : stK A B . $\forall$ Y : msg .

N(n Y $\{n_B$ $k_{AB}\}_{kBS}$)

L(B,A,n,n_B,k_{BS}) =>

N(n Y)

RWL with Dependent Types

Combination of <u>two</u> key concepts:

- Conditional rewriting modulo equations:
  - (∀ x : S) A = B if C (generalizes equational logic)
  - (∀ x : S) A => B if C (generalizes rewriting logic)
- Lambda-Calculus with dependent function types
  - [x : S] M : { x : S } T (generalizes ordinary type theory)

Propositions-as-Types Interpretation:

(∀ x : S) P(x) is interpreted as { x : S } P(x)

=> provides an expressive higher-order logic

RWL with Dependent Types

- Operational Semantics:
  - open computation system
  - conditional rewriting modulo
  - automatic proof search for conditions
  - uniform for execution and type checking
  - equational/rewriting logic executable sublanguage
- Model-theoretic Semantics:
  - classical, set-theoretic
  - Zermelo-Fraenkel or Grothendiek-Tarski
- Current Implementation:
  - prototype in rewriting logic (on top of the Maude engine)
  - mapping of higher-order to first-order concepts

Example

```
fms : Type -> Type .

empty : {T | Type} (fms T) .
single : {T | Type} T -> (fms T) .
union : {T | Type} (fms T) -> (fms T) -> (fms T) .

comm : || {T : Type}{m1,m2 : (fms T)}
      (union m1 m2) = (union m2 m1) .
assoc : || {T : Type}{m1,m2,m3 : (fms T)}
      (union m1 (union m2 m3)) = (union (union m1 m2) m3) .
id : || {T : Type}{m : (fms T)}
      (union m empty) = m .
```

Example

```
select : {T | Type} (T -> Prop) -> (fms T) -> (fms T) .

select_empty : !! {T : Type}{P : (T -> Prop)}
 (select P empty) = empty .

select_single_1 : !! {T : Type}{P : (T -> Prop)}{x : T}
 (P x) -> (select P (single x)) = (single x) .

select_single_2 : !! {T : Type}{P : (T -> Prop)}{x : T}
 (Not (P x)) -> (select P (single x)) = empty .

select_union : !! {T : Type}{m,m' : (fms T)}{P : (T -> Prop)}
 (Not (empty? m)) -> (Not (empty? m')) ->
 (select P (union m m')) = (union (select P m) (select P m')) .
```

Example

```
fun : nat -> Type -> Type .
fun_eq_1 : !! {T : Type} (fun 0 T) = T .
fun_eq_2 : !! {T : Type}{i : nat}(fun (suc i) T) = (T -> (fun i T)) .

union* : { T | Type}{i : nat} (fun i (fms T)) .

union*_eq_1 : !! (union* 0) = (empty | T) .
union*_eq_2 : !! {T : Type}{s : (fms T)}
  (union* (suc 0) s) = s .
union*_eq_3 : !! {T : Type}{s,s' : (fms T)}
  (union* (suc 1) s s') = (union s s') .
union*_eq_4 : !! {T : Type}{i : nat}{s,s' : (fms T)}
  (union* (suc (suc (suc i))) s s') =(fun (suc (suc i)) (fms T))
  (union* (suc (suc i)) (union s s')) .
```

Representing MSR Types

```
princ : Type .
nonce : Type .
pubK : princ -> Type .
privK : {A | princ} (pubK A) -> Type .

shK : princ -> princ -> Type .
ltK : princ -> princ -> Type .
stK : princ -> princ -> Type .

subtype-ltK-shK : ?? {A,B : princ} (ltK A B) <: (shK A B) .
subtype-stK-shK : ?? {A,B : princ} (stK A B) <: (shK A B) .
```

Representing MSR Types

```
msg : Type .

subtype-nonce-msg : ?? nonce <: msg .
subtype-princ-msg : ?? princ <: msg .
subtype-stkey-msg : ?? {A, B : princ} (stK A B) <: msg .

nil : msg .
append : msg -> msg -> msg .
encrypt : {A,B | princ} msg -> (shK A B) -> msg .
```

Representing MSR State

```
state : Type .

empty : state .
union : state -> state -> state .

union_comm : || {s1,s2 : state}
  (union s1 s2) = (union s2 s1) .
union_assoc : || {s1,s2,s3 : state}
  (union s1 (union s2 s3)) = (union (union s1 s2) s3) .
union_id : || {s : state}
  (union s empty) = s .

N : msg -> state .
```

Representing MSR Roles

∀ ∃ over rules

∀ A . ∃ L . LHS => RHS (A : princ implicit)

becomes

P(A) F(L) LHS => P(A) F(suc(L)) RHS

Exception:

- P(A) not needed if LHS contains A

Finiteness Requirement:

- princ must be finite type, and
- all its elements are enumerated as P(A) in the state

Representing MSR Rules

∃ on the right-hand side

$\forall x : \sigma \,.\, LHS \Rightarrow \exists n \,.\, RHS$

becomes

$\sigma(x)\ F'(n)\ LHS \Rightarrow \sigma(x)\ F'(suc(n))\ RHS$

Exception:

- $\sigma(x)$ not needed if LHS contains x

Finiteness Requirement:

- σ must be finite type, and
- all its elements are enumerated as $\sigma(x)$ in the state

Representing MSR Rules

multiple ∃ on the right-hand side

LHS => ∃ n_1 . ∃ n_2 . RHS

becomes

F'(fresh) LHS => F'(fresh') RHS if n_1 := fresh /\
n_2 := (suc fresh) /\
fresh' := (suc (suc fresh)) .

Typed case:
add injections from the domain of fresh objects (e.g. nat)
into the required type (e.g. nonce, stK ...)

Representing Multiple Rules

$LHS_1 \Rightarrow RHS_1$,

$LHS_2 \Rightarrow RHS_2$

becomes

$\Rightarrow T_1\ T_2$

$T_1\ LHS_1 \Rightarrow RHS_1$

$T_2\ LHS_2 \Rightarrow RHS_2$

becomes

$LHS_1 \Rightarrow RHS_1\ T_2$

$T_1\ LHS_1 \Rightarrow RHS_1$

$LHS_2 \Rightarrow RHS_2\ T_1$

$T_2\ LHS_2 \Rightarrow RHS_2$

Motivation for 2nd step: elimination of intermediate states

Representing Multiple Rules

$\forall A . \exists L .$ $LHS_1 \Rightarrow RHS_1$,
$LHS_2 \Rightarrow RHS_2$

becomes

P(A) F(L) => P(A) F(suc(L)) T(A,L)
T(A,L) => T_1(A,L) T_2(A,L)
T_1(A,L) LHS_1 => RHS_1
T_2(A,L) LHS_2 => RHS_2

Typed case and multiple ∃:
similar to ∃ on right-hand side

Representing Multiple Rules

P(A) F(L) => P(A) F(suc(L)) T(A,L)

T(A,L) => T_1(A,L) T_2(A,L)

T_1(A,L) LHS_1 => RHS_1

T_2(A,L) LHS_2 => RHS_2

becomes

P(A) F(L) LHS_1 => P(A) F(suc(L)) RHS_1 T_2(A,L)

T_1(A,L) LHS_1 => RHS_1

P(A) F(L) LHS_2 => P(A) F(suc(L)) RHS_2 T_1(A,L)

T_2(A,L) LHS_2 => RHS_2

Optimization I

$\forall A \ . \exists L \ . \ LHS_1 \Rightarrow RHS_1$

becomes

R_{11}: P(A) F(L) $LHS_1 \Rightarrow$ P(A) F(suc(L)) RHS_1
R_{12}: $T_1(A,L)$ $LHS_1 \Rightarrow RHS_1$

Optimization:
$T_1(A,L)$ is never generated.
Therefore, R_{12} can be omitted.

(usually $\exists$ L is not present in this case)

Optimization II

$\forall A . \exists L .$ $LHS_1 \Rightarrow RHS_1$, (rule 1)
$LHS_2 \Rightarrow RHS_2$ (rule 2)

becomes

R_{11}: $P(A)\ F(L)\ LHS_1 \Rightarrow P(A)\ F(suc(L))\ RHS_1\ T_2(A,L)$
R_{12}: $T_1(A,L)\ LHS_1 \Rightarrow RHS_1$
R_{21}: $P(A)\ F(L)\ LHS_2 \Rightarrow P(A)\ F(suc(L))\ RHS_2\ T_1(A,L)$
R_{22}: $T_2(A,L)\ LHS_2 \Rightarrow RHS_2$

Optimization:
If LHS_i contains L then rule i cannot be applied first.
Therefore, R_{i1} can be omitted (and rules depending on it).

Optimization III

$\forall$ A . $\exists$ L . LHS_1 => $\exists$ n . RHS_1

becomes

R_{11}: P(A) F(L) F'(n) LHS_1 => P(A) F(suc(L)) F'(suc(n)) RHS_1
R_{12}: T_1(A,L) F'(n) LHS_1 => F'(suc(n)) RHS_1

Optimization of R_{11}: Use single source of fresh objects.

R_{11}: P(A) F(fresh) LHS_1 => P(A) F(fresh') RHS_1
if L := (suc fresh) /\
n := fresh /\
fresh' := (suc (suc fresh)).

Recall MSR Example

Generic Role B

$\forall B \;.\; \exists L : princ^{(B)} \times nonce \times nonce \times ltK\ B\ S \;.$

$\forall\ A : princ.\ \forall\ n : nonce\ .\ \forall\ k_{BS} : ltK\ B\ S\ .\ \forall\ X : msg\ .$

$N(n\ A\ B\ X) \Rightarrow$

$\exists\ n_B : nonce\ .\ N(n\ A\ B\ X\ \{n_B\ n\ A\ B\}_{kBS})$

$L(B,A,n,n_B,k_{BS}),$

$\forall\ A : princ.\ \forall\ n,n_A : nonce.$

$\forall\ k_{BS} : ltK\ B\ S\ .\ \forall\ k_{AB} : stK\ A\ B\ .\ \forall\ Y : msg\ .$

$N(n\ Y\ \{n_B\ k_{AB}\}_{kBS})$

$L(B,A,n,n_B,k_{BS}) \Rightarrow$

$N(n\ Y)$

Translation (Optimized)

```
B11: !! {B : princ}
        {L : {B : princ} princ -> nonce -> nonce -> (ltK B S) -> state}
        {A : princ}{kBS : ltK B S}{X : msg}
        {fresh,fresh' : nat}{n,nB : nonce}

           (L := (LB (suc fresh))) ->
           (nB := (nonce fresh)) ->
           (fresh' := (suc fresh)) ->

           (LB11 : (union* 3 (LTK B S kBS) (F fresh)
                             (N (append* 4 n A B X)))
            =>     (union* 5 (LTK B S kBS) (F fresh')
                             (N (append* 5 n A B X
                                         (encrypt (append* 4 nB n A B) kBS)))
                             (L B A n nB kBS)
                             (TB2 A B L)))
```

Translation (Optimized)

```
B12: !! {B : princ}
        {L : {B : princ} princ -> nonce -> nonce -> (ltK B S) -> state}
        {A : princ}{kBS : ltK B S}{X : msg}
        {fresh,fresh' : nat}{n,nB : nonce}

           (nB := (nonce fresh)) ->
           (fresh' := (suc fresh)) ->

           (LB12 : (union* 4 (LTK B S kBS) (F fresh)
                             (N (append* 4 n A B X))
                             (TB1 A B L))
            =>      (union* 4 (LTK B S kBS) (F fresh')
                              (N (append* 5 n A B X
                                            (encrypt (append* 4 nB n A B) kBS)))
                              (L B A n nB kBS)))
```

Translation (Optimized)

```
B21 : !! {B : princ}
        {L : {B : princ} princ -> nonce -> nonce -> (ltK B S) -> state}
        {A : princ}{kAB : stK A B}{kBS : ltK B S}{Y : msg}
        {fresh,fresh' : nat}{n,nB : nonce}

          (L := (LB fresh)) ->
          (fresh' := (suc fresh)) ->

          (LB21:  (union* 3 (F fresh)
                            (N (append* 3 n Y
                                        (encrypt (append* 2 nB kAB) kBS)))
                            (L B A n nB kBS))
            =>      (union* 3 (F fresh')
                            (N (append n Y))
                            (TB1 A B L)))
```

Translation (Optimized)

```
B22 : !! {B : princ}
        {L : {B : princ} princ -> nonce -> nonce -> (ltK B S) -> state}
        {A : princ}{kAB : stK A B}{kBS : ltK B S}{Y : msg}{n,nB : nonce}

          (LB22: (union* 3 (N (append* 3 n Y
                                       (encrypt (append* 2 nB kAB) kBS)))
                           (L B A n nB kBS)
                           (TB2 A B L))
           =>     (N (append n Y)))
```

Further Optimizations:

- Rule B21 can be eliminated by Opimization III.
- Rule B12 can be eliminated because it depends on B21
  (Token (TB1 A B L) can only be produced by B21)

Execution

To restrict and observe the protocol execution, we add START and TERMINATED Tokens to the rules of generic role A.

```
A : princ . B : princ .

rew (union* 6
        (F 0) (P A) (P B) (LTK A S kAS) (LTK B S kBS) START) .
```

result:

```
( union ( union ( union ( union ( union
TERMINATED
( F ( suc ( suc ( suc ( suc ( suc ( suc 0 ) ) ) ) ) ) ) )
( P A ) ) ( P B ) ) ( LTK A S kAS ) )  ( LTK B S kBS ) )
```

Conclusion

- We impose reasonable restrictions to achive executability:
  - finiteness of certain types
  - rules like · => N(...) are too nondeterminstic
- Embedding into RWL with Dependent Types
  provides an (alternative) logical semantics of MSR
  in addition to the operational semantics
- Possible Future Work:
  - Taking into account the data access specification
  - Implementation of the transformation
  - Specification & execution environment for MSR
- Here we have discussed MSR 2.0, but how
  to deal with rules as objects (as in MSR 3.0) ?