


S-expressions
&
Maude + Pvs

Carolyn Talcott

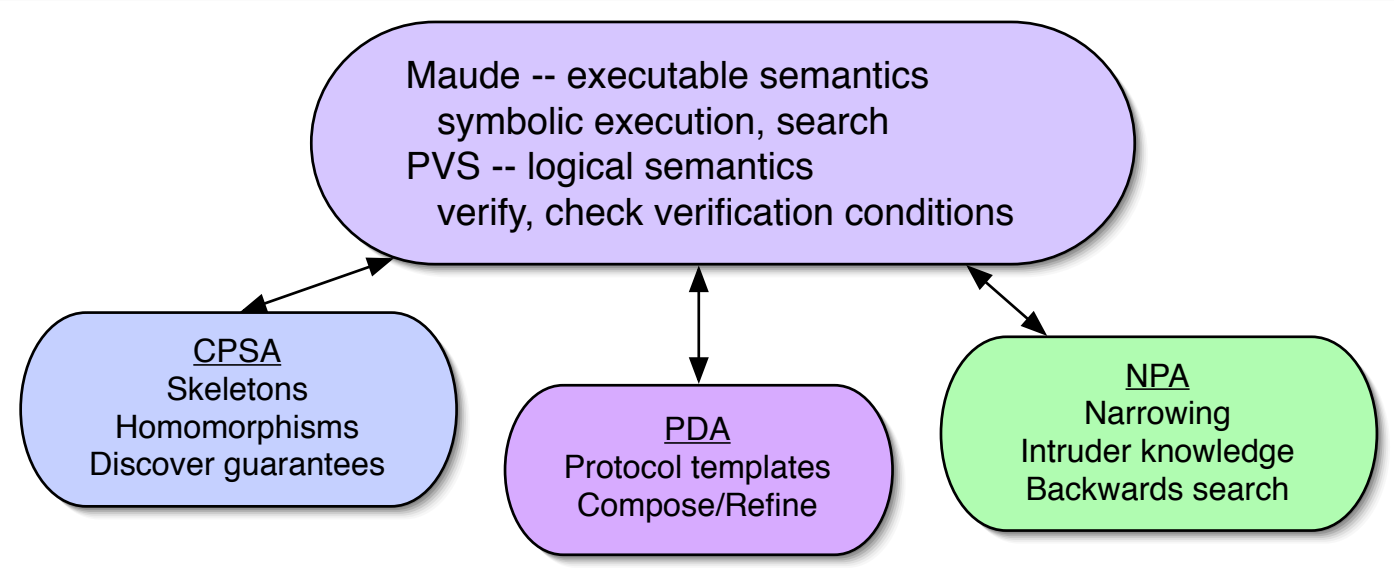
06 September Exchange

Outline

- Vision
- June S-expression grammar
- Issues
- Formalization Progress

Vision

Formal framework for semantically sound interoperation of tools for design and analysis of cryptographic protocols (building on the strand space model)



Tool Communication

- Shared S-expression grammar
  - captures basics concepts abstractly
  - sublanguage understood by each tool
  - version 0.1 defined
- Concepts
  - Term, Action / Signed Message
  - Program / Signed Message List, Agent/Role
  - Protocol
  - Process, Skeleton
  - Annotations

Tool Interoperation

The devil is in the details

- Choice of crypto primitives
  - CPSA currently fixed set of crypto
  - NPA allows user to introduce and define equationally
  - PDA allows arbitrary functions to be introduced, and axiomatized in the PDA logic
- Protocol structure
  - CPSA/NPA linear sequences of signed terms
  - PDA sequential/parallel composition of actions with intended run made explicit

Tool Interoperation II

- Role of intruder
  - CPSA / PDA -- implicit
  - NPA -- explicit
- Matching vs deconstuctors
  - {A,N}K := M, vs
  - A := 1st(decrypt(M,K)), N := 2nd(decrypt(M,K))

- CPSA -- implicit matching
- PDA -- explicit matching
- NPA -- uses deconstructors to analyze, but implicitly

Tool Interoperation III

- Expressing freshness
  - CSPA uses strand annotation
  - PDA has an explicit action
  - NPA-maude uses a `new' abstraction
- Confidentiality
  - CPSA: non-origination annotation, safe induction
  - PDA: specified in logic as assumptions
  - NPA: specified in facts about I (intruder knowledge)

Sexpression Grammar v 0.1

- The S-expression language supports both refinement and execution level specification.
- The rich notions of parameterization, principal and function require care.
- Execution level semantics is first order---operates on ground expressions
- This can be extended to do symbolic execution using inference rules (about what a player might be justified in concluding)
- Refinement level should be formalized as a meta theory
  - an algebra of specifications --- functions, roles .... are objects

Goals

- Simplest thing that works
- Union of PDA-needed and CPSA-needed information
- S-expression unit must be one of
  - common PDA/CPSA info, CPSA-only info, or PDA-only info
- S-expression output should be self-contained
- Policy: be liberal in what you accept, conservative in what you transmit


Grammar Highlights

Top Down

<Protocol> :=
  (protocol
    <Version>   // version number of the language
    <IdentDef>   // the protocol name. scope: whole protocol
    <Parameters>   // binding -- scope is whole protocol
    <Protocol-body>
    [<Annotations>]   // how to achieve uniformity
)

<Parameters> := (parameters <VariableDef>*)
<Protocol-body> := (roles <role>+)

(for pure syntax ignore Def/Ref endings)

<Annotations> := <Spec> <IntendedRuns> [<Skels>] [<Other>] [<Display>]

<Spec> := (spec [<String>])

<IntendedRuns> := (runs <IntendedRun>*)
<IntendedRun> := (run <Edge>*)

<Edge> := (edge <Ref> <Ref>)
<Ref> := (ref <Ident> <IdentRef>)
    // (ref foo bar) means (send...) or (recv...)
    // with label bar in role foo

<Other> := (other <String>)


```
<role> :=
  (role
    <IdentDef>              // unique in protocol
                  // parameters -- binding scope is role
    <PrincipalVar>            // identifies the active principal
    <InParameters>            // other parameters
    <Outputs>               // Output -- for composition of roles
    <Local-Constraints>
    <Code>                  // signed msglist / programs
    [<Display>]
   )

<Outputs>:= (outputs <Exp>*)

<Local-constraints> := (localconstraints <Local-Constraint>* )
<Local-constraint> := <Non> | <Spec>
<Non> := (non <AtomExp>*)
```


<Code> :=
  (action <Kind> <IdentDef> [<Display>])
 | (new <AtomExp>+ [<Display>]) // Defs
 | (match <Exp> <Pattern> [<Display>])
 | (cond <Bterm> <Code> [<Display>])
 | (send <SenderRef> <ReceiverRef> <Exp> <IdentDef> [<Display>])
 | (recv <SenderDef> <ReceiverDef> <Exp> <IdentDef> [<Display>])
      //<IdentDef> is a label for this specific action. Scope: role
 | (seq <Code>* [<Display>]) //CPSA Code uses only this
 | (par <Code>* [<Display>])

<Pattern> := <Exp>
<SenderRef> := (sender [<PrincipalRef>])
<ReceiverRef> := (receiver [<PrincipalRef>])

<Exp> := (term <Kind> <Ident>) | <AtomExp> | <TupExp> | <AppExp>

<TupExp> := (tuple <Exp>*)
<AppExp> := (app <Fun> <Exp>+)

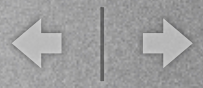
<Fun> := (fun (params <Exp>*) <Arity> <OutType> <Kind> <Ident>)
<Arity> := <Nat>
<OutType> := encr | hash | sig | other
  // example (fun (params A B) 1 encr "Encrypt")
  // for "Encrypt with the key shared by A and B"

<Principal> := (principal [(controls <Principal>+)] <Kind> <Ident>)
  // controls gives information about the partial order on principals


<AtomExp>
  := <Principal>
     | (nonce <Kind> <Ident>)
     | (stkey <Kind> <Ident>)
     | (text <Kind> <Ident>)
     | (ltkey <Principal>* <Kind> <Ident>)
     | (pubkey [<Principal>] <Kind> <Ident>)
     | (privkey [<Principal>] <Kind> <Ident>)

<Kind> := var | cons

To Do



- Specifications
- Skeletons
- Queries


Issues

Principals

- When should two principal expressions be considered the same?
- This is an issue for matching and substitution.
- Consider pc(id,pl) vs pc(id',pl') where pl are controlled by pc(id).
  - id =/= id' --- different principals
  - id == id' and pl =/= pl' --- ???
- For semantic purposes I assume that the partial order information is collected in the role specification or at the protocol level.
- Thus the controlees given explicitly in a protocol expression can be ignored. (May require a little preprocessing)
- (Should a principal playing one role care about the partial order of principals playing another role?

Functions

- To express PDA protocols at the refinement level, we need function variables, not just expression variables as parameters.
  - Consider the CR scheme where the C/R functions are parameters.
- How to axiomatize functions (fun params arity out kind id)
  - for each function used in a protocol, there should be axioms that can be used for simplification (reduction to canonical form) and matching.

Patterns -- what are they?

Pattern:	Matched by:	
stkv(id)	stkc(id')	
pv(id,pl)	pc(id',pl')	ignore controlees
ltkv(id,pl)	ltkc(id',pl')	if pl' ~ pl
pubkv(id,p)	pubkc(id',p')	if p' ~ p
[patl]	[expl]	if expl ~ patl
ctor(patl)	ctor(expl)	if expl ~ patl
invop(patl)	invop(expl)	if can invert (may bind) and expl ~ patl

Example: enc(K,pat) matched by enc(K',exp) if K = K' and

K = stk(id)	has K			
K = ltk(id,pl)	(exists p in pl) controls p			
K = pubk(id,p)	controls p	K = privk(id,p)		true

alternately exp matches enc(K,pat) if dec(inv(K'),exp) == exp' and exp' matches pat for some K' an instance of K and has K'

has is interpreted in the context of a role players knowledge kb: kb |- has K'

Progress on Semantics

- Representation of syntax data types
  - Exp
  - Code
  - Role
  - Protocol
- Reader: sexp2protocol
- Semantic data types --- in progress
- Environment -- with subst, match functions
- Player
- Msgs,
- Events
- Action rules
- KB

Example: sexp2role

```
(role A (principal var A) (parameters)  (outputs (term var out)) (localconstraints (spec ""))
   (seq (new (nonce var x0))
        (send (sender (principal var A)) (receiver (principal var B))
              (app (fun (parameters (principal var B)) 1 encr const  E)
                   (tuple (principal var A) (term var x0)))  p1)
        (recv (sender ...) (receiver ...) (term var z__0)  q2)
        (match (principal var Y__0) (principal var A))
        (match ...)  (send ...)))
 =>
role("A", pv("A", nil), nil, ev("out"), spec(""),
  seq(new(nv("x0")),
      send(pv("A",nil), pv("B", nil),  fc("E", 1, encr)[pv("B", nil)]([pv("A", nil),ev("x0")]), "p1"),
      recv(pv("X__0", nil), pv("Y__0", nil), ev("z__0"), "q2"),
      match(pv("Y__0", nil), pv("A", nil)),
      match(...), send(...)))
```


That's All Folks :-)