How-To: Define a validation rule system

PEAR has been developed to be completely *parametric* with respect to the validation rule system to be used during the protocol analysis. In this section it will be shown how to define a new set of validation rules.

Base Syntax

Each system is ideally composed by two sections: in the first one you have to declare the identifiers that will be used in the rule definitions, in the second one you actually have to define the rules you wish to be tried during the protocol analysis. Each rule has a name, an associated identity, which represents the identity of the principal currently executing the corresponding instruction, a set of conditions that have to be satisfied in order to apply the rule, a set of actions to be executed if the conditions are satisfied. Each rule applies to a specified instruction with a given set of parameters. A generic rule has this shape:

```
rule A_RULE(I)
{
  { instr(p1, p2, p3) }

  conditions { /* condition definition */ }

  actions { /* action definition */ }
}
```

`A_RULE` represents the name of the rule, `I` the associated identity [1], `instr` is the name of the instruction which the rule is defined on, and `pi` represent the instruction parameters. Parameters and identities are identifiers, so they must have been declared in the apposite section; the assignable types [2] are those used in the following example:

```
declarations
{
  Principal I, J;
  Variable p, p1, p2; // there is difference between names and variables
  Name p3;
  Key sk, ak;
}
```

as you can see, it is possible to insert comments with the classic C/C++/Java syntax: the double slash `//` comments every character until an end-of-line character is found; moreover every character included between `/*` and `*/` is ignored.

Containers

Containers model both typing environments and effects: each action or condition is defined on containers. Each container has to be defined in the relative section, preceding the `declarations` one:

```
containers
{
  ContainerType c1;
  ContainerType c2;
}
```

`ContainerType` identifies the name of the container type, while `c1` is the name of the container itself; four container types are already provided with tool [3], while other may be defined as plug-ins.

Operations, conditions and actions

Each container has a standard operation set [4], whose elements may be invoked in the condition or action definitions [5]:

```
rule A_RULE(I)
{
  { instr(p1, p2, p3) }

  conditions { /* condition definition */ }

  actions
  {
    c1.add(p1);
    c1.add(p2);
    c1.delete(p3);
  }
}
```

The condition definition is a little bit more complex because a condition is made up of ipothesis-thesis couple in order to model the implication operator; the ipothesis is optional and if not defined is assumed to be true. Conditions may be defined as follows:

```
rule A_RULE (I)
```

```
{
  { instr(p1, p2, p3) }

  conditions
  {
    ipothesis(c1.contains(p1));
    thesis(c1.contains(p2) & !c1.contains(p3));
    thesis(c1.contains(p2));
  }

  actions { /* action definition */ }
}
```

the example shows that it is possible to combine the results of the single operations through the common boolean operators `&` (and), `|` (or), `!` (not); if all the conditions return `true` or there are no conditions defined, actions are performed.

Tags

Tags play a fundamental role in the validation algorithm: in fact both the rule systems provided with PEAR define a tag set. The tag definition is a *crucial* operation, because it has an impact on the behaviour of both protocol and rule parsers and, of course, on the tag inference algorithm. A message can not contain more than one tag for each type [6], moreover all the generic tags appearing in the same definition are mutually exclusive [7]:

```
tags
{
  IdentityTag id;
  Tag claim, verif;
}
```

the tag declaration is the last one and follows the identifier declaration. After declaring them, it is possible to tag identifiers as you wish:

```
rule A_RULE(I)
{
  { instr({p1:claim, p2:verif, p3}:sk) } // this is an error
}
```

the example above is wrong, because two tags defined together appear in the same message; it would be correct if the tag declaration had been:

```
tags
{
  IdentityTag id;
  Tag claim;
  Tag verif;
}
```

Parameter types

Instruction parameters may be simple as well as tagged identifiers, messages and instructions. The relative syntax states that a message is composed by an optional name, followed by a sequence of identifiers and an encryption or decryption key:

```
// p1 is a ciphertext encrypted with the symmetric key sk
p1 = {I:id, p2}:sk

// p2 is a ciphertext encrypted with the public part of key ak
p2 = {|p3|}:Pub(ak)

// a ciphertext decrypted with the private part of key ak
{|p3|}(:Priv(ak))
```

provided that operations are nothing else than instructions defined on containers, they have the same syntax; thus it is legal to write:

```
rule A_RULE(I)
{
  { instr({I:id, p1:claim, p2}:sk) }

  actions
  {
    c1.add(instr({I:id, p1:claim}:sk));
    c1.add(instr(p2));
    c2.add(p3);
  }
}
```

since parameters may be seen even as containers, it is possible, for example, to check the content of a message, treating it as a container:

```
rule A_RULE(I)
{
```

```
  { instr(p2 = {I:id, p1:claim, p3}:sk) }

  conditions
  {
    thesis(p2.contains(p3)); // true
    thesis(p3.contains(p2)); // false
    thesis(p3.contains(p3)); // true
  }
}
```

Eventually it is possible to assign a text value to an identifier using the following syntax:

```
rule A_RULE(I)
{
  { instr(p) }

  actions
  {
    c1.add(p="identifier")
  }
}
```

Sequential process validation

While defining a rule system it is useful to remember that a sequential process is characterized by a signature, possibly declaring identities and keys, and a closing instruction, the *null* instruction:

```
A > Initiator(B:Id, k:sym-key(A,B)) := … .0
```

if a validation rule for the null instruction is not provided, it is impossible to complete the validation process successfully; the recommended definition of the *null* rule is:

```
rule NIL(I)
{
  { 0 }
}
```

this definition instructs PEAR to do nothing in presence of the null instruction, in fact there are no conditions or actions defined: this rule always validates the null instruction; obviously you can always provide a definition that suits better your needs. It is also possible to define rules that validate the signature declarations:

```
rule IDENTITY_DECL(I)
{
  { id(J) }
}

rule SYMMETRIC_KEY_DECL(I)
{
  { let(k = sym-key(I, J)) }
}
```

the *pseudo-instructions* id and let have the function of providing an "access" to the signature declarations [8]; the rules defined in the above example do nothing and are not compulsory, because the signature is not an instruction of the protocol, but they may reveal useful in case one wishes to handle the containers' content to reflect the signature declarations. You should be aware that, once defined any condition in the above rules, the validation process may fail even at signature level.

Plug-ins

To provide an operational support to the widest range of definable rule systems, the ideal approach is allowing PEAR's functionalities to be extended by means of *plug-ins*. Extensibility works in two directions: new containers may be provided through classes overriding the standard containers' behaviour [9] or providing new operations; new single operations may be provided through classes implementing the `ExternalOperation` interface [10]; these may be applied to any container; Operations defined *by the container* are called `native`, while *stand-alone* operations are called `external`. You may use a new container `NewContainer` [11] simply inserting its name in the related declaration:

```
containers
{
  NewContainer nc;

  /* further declarations */
}
```

now you may use the operations defined on the new container exactly as explained previously (as a matter of fact the containers provided with PEAR *are* plug-ins):

```
nc.natop(p)
```

If you wish to invoke an external operation on a container you need to specify that the operation must not be looked for among the ones defined by the container; this may be done at three levels:

- at *operation level* you mark the single operation as external:

```
actions
{
  (external)c1.extop(p);
}
```

- at *condition-clause level* you mark all the operations appearing in the clause definition as external:

```
conditions
{
  ipothesis(c1.contains(p) & c2.contains(p));
  external thesis(c1.extop(p) & c2.extop(p));
}
```

- at *conditions/actions level* you mark all the operations appearing in the clause definitions below as external:

```
rule EXT_RULE(I)
{
  { instr(p, p1) }

  external conditions
  {
    ipothesis (c1.extop(p1));
    thesis(c1.extop(p) & c2.extop(p));
  }

  external actions
  {
    c1.extop(p2);
    c2.extop(p2);
    (native)c2.add(p1); // this operation is native
  }
}
```

moreover, everywhere you may use the `external` keyword, you may also use the `native` complementary keyword with opposite meaning.

Pattern-matching

The rules are associated to the instruction appearing in the protocol encoding trough *pattern-matching*. The rho-spi syntax is translated in the one used by the rules, thus letting an instruction to match exactly the shape defined by the associated rule:

```
decrypt xT as {nA:verif, B:id, kAB}:kAT.
```

is translated to:

```
decrypt(xT={nA:verif, B:id, kAB}(:kAT)).
```

so, by defining an appropriate rule, it is possible to validate the previous `decrypt` primitive:

```
rule A_DECRYPTION(I)
{
  { decrypt(p={J:id, p1:verif, p2}(:k1)) }

  actions { /* action definitions */ }
}
```

in this case, being `A` the principal executing the sequential process, the substitution map will have the following content:

```
[I=A, p=xT, p1=nA, J=B, p2=kAB, k1=kAT]
```

all the free identifiers appearing in the rule are mapped on the ones placed in the corresponding positions in the instruction, with an exception: the order of the components of a message is immaterial. Since it is unpredictable the number of identifiers with no tags appearing in a message, you may use the dots construct (…) to match them:

```
rule A_MORE_GENERAL_DECRYPTION(I)
{
  { decrypt(p={J:id, p1:verif, ...}(:k1)) }

  actions { /* action definitions */ }
}
```

this version handles a definitely wider range of cases:

```
decrypt xT as {nA:verif, B:id, kAB, z}:kAT.

[I=A, p=xT, p1=nA, J=B, ...=[kAB, z], k1=kAT]
```

the dots construct may be placed everywhere you may find a simple identifier. The colons construct matches both

tagged and untagged identifiers, and may be used in the same way as the dots construct:

```
rule THE_MOST_GENERAL_DECRYPTION(I)
{
  { decrypt(p={:::}(:k1)) }

  actions { /* action definitions */ }
}
```

this version handles all the cases:

```
decrypt xT as {nA:verif, B:id, kAB, z}:kAT.

[I=A, p=xT, :::=[nA:verif, B:id, ...], ...=[kAB, z], k1=kAT]
```

Every mapping is kept when passing to the condition evaluation; this means that all the identifiers that have already a mapping are overridden with their mapped value, while the free ones are subject to the *auto pattern-matching* algorithm, which provides a form of universal quantification: for each operation the related container content is checked, looking for new possible mappings to add to the substitution map: the process is strictly sequential and checks each element of the container. The algorithm makes successive attempts to find the right mapping, exploiting the underlying backtracking mechanism, until a correct mapping can be found or the whole container has been checked. Considering the previous example, the algorithm will work as follows:

```
rule THE_MOST_GENERAL_DECRYPTION(I)
{
  { decrypt(p={:::}(:k1)) }

  conditions { thesis(c1.contains(k1=sym-key(I,J))); }

  actions { c2.add(decrypt({:::}(:k1))); }
}
```

the previous substitution map will lead to an operation defined as follows:

```
c1.contains(kAT=sym-key(A,J))
```

supposing that the container `c1` has the following content:

```
c1=[kBT=sym-key(B,T), kAT=sym-key(A,T)]
```

To a human observer may be obvious that the missing mapping is `J=T`, but the validation core has no idea of the meaning of the `contains` operation. It tries then to match the first element of `c1`, failing because it should insert in the substitution map the element `kAT=kBT`, while the identifier `kAT` is already bound. Hence the validation core tries to match the second element obtaining the correct mapping `J=T`. If the condition evaluation is successful, then the actions are performed: in this case operations will be instantiated according to the substitution map content, but there will be no further pattern-matching. You may exploit conditions to perform a "void" auto pattern-matching through the external (hence defined on all the containers) operation `match`, which has no influence on the condition evaluation because it always returns `true`:

```
rule THE_MOST_GENERAL_DECRYPTION(I)
{
  { decrypt(p={:::}(:k1)) }

  conditions
  {
    thesis(c1.contains(k1=sym-key(I,J)));
    external thesis(c2.match(n="identifier"));
  }

  actions
  {
    c2.set(n="identified");
    c2.add(decrypt({:::}(:k1)));
  }
}
```

if `c2` does not contain an element that may be associated to `n`, then the action will be executed using `n` as actual parameter, else the substitution map will be updated as needed:

```
c2=[id="identifier"]

c2.set(id="identified")
c2.add(decrypt({nA:verif, B:id, kAB, z}(:kAT)))

c2=[id="identified", {nA:verif, B:id, kAB, z}(:kAT)]
```

Directives

To configure the validation process, three directives are available: the first one is directed to the auto pattern-matching algorithm, the other two to the backtracking handler.

- `@ra` is used to temporary reset the association of the dots construct in the substitution map, which may

result useful in certain situations; it must be placed before a condition clause and it *scopes* the whole clause:

```
rule A_RULE(I)
{
  { instr(...) }

  conditions
  {
    ipothesis(c1.contains({...}:k1));
    @ra thesis(c2.contains(p={...}:k2));
  }

  actions { /* action definitions */ }
}
```

the dots construct appearing in the ipothesis clause will be substituted with the value taken from the map, the one appearing in the thesis clause is subjected to auto pattern-matching.

- `@exist`, `@univ` are alternative and complementary directives: they are used to alter the standard behaviour of the backtracking handler; normally the validation core, in presence of an `ipothesis` clause, applies an *universal quantification* policy. In this case it tries to satisfy in every way the checks defined in the clause; instead, in presence of a `thesis` clause, the validation core applies an *existential quantification* policy: a single check's failure is enough to state the failure of the whole clause evaluation. This causes the evaluation of all the conditions to fail, because they are related by a *boolean and* operator. Usually this behaviour is correct and sensible; if you wish to modify it for any reason, then you may specify the desired directive before the interested clause:

  ```
  conditions
    {
      @exist ipothesis(c1.contains({...}:k1));
      @ra @univ thesis(c2.contains(p={...}:k2));
    }
  ```

  Obviously assigning to the clauses their default directives does not alter the behaviour of the backtracking handler in any way. As you may notice from the above example, directives have to be expressed in a precise order:

  ```
  [@ra] [@exist | @univ] condition_clause
  ```

^top

Notes:

1. This identity is associated with the actual one of the principal executing the sequential process during the pattern-matching process.
2. They have the same meaning as in standard *rho-spi*.
3. `SetContainer` models a simple set, `OrdContainer` models an ordered container, `MultisetContainer` models a multiset, `MapContainer` models a set whose elements have an associated value.
4. `add(p)` inserts the parameter `p` in the container, `add(p, val)` inserts the parameter `p` in the container, associating to it the value `val`, `contains(p)` checks if the element `p` is held in the container, while `delete(p)` removes the element `p` from the container.
5. Note that each operation that does not explicitly return a boolean value is assumed to be true.
6. `Tag` generic tag type, `KeyTag` used to tag keys, `TypeTag` used to type identifiers, `NonceTag` used to tag nonce, `MessageTag` used to tag messages, `IdentityTag` used to tag identities, `CiphertextTag` used to tag encrypted messages.
7. Ignoring these constraints will cause you to receive a syntax error notification from the parsers.
8. You should consider that `sym-key(A,B)` is equivalent to `sym-key(B,A)`.
9. The `Container` interface describes the set of standard operations.
10. For instance the the external operation `checkType` checks the type of a message.
11. New classes must be inserted into the package `it.unive.dsi.pear.plugins`.
