1 Introduction to the QoP Modelling Language

The goals of the presented QoP-ML are as follows.

1. Model the cryptographic protocol maintaining processes and communication steps consistency.
2. Abstract all security operations/mechanisms which are executed in the process of cryptographic protocols - multilevel security analysis.
3. Model the operations which refer to all security attributes/services.
4. Introduce quality of protection evaluation for the modelled security operations/mechanisms.
5. Enable the scalable analysis where different versions of the same protocol can be defined.
6. Enable the analysis of temporal properties of the cryptographic protocol steps.

1.1 General view

The structures used in the QoP-ML represent high level of abstraction which allows concentrating on the quality of protection analysis. The QoP-ML consists of processes, functions, message channels, variables, and QoP metrics. Processes are global objects grouped into the main process which represents the single computer (host). The functions, message channels and variables can be declared either globally or locally within a process. The process specifies behaviour, functions represent a single operation or a group of operations, channels and global variables define the environment in which the process is executed. The QoP metrics define the functions and channels according to their influence on the quality of protection.

1.2 Data Types

We assume an infinite set of variables, to be used for describing communication channels, processes and functions. The variables are used to store information about the system or specific process. The QoP-ML is an abstract modelling language so there are no special data types, sizes and value ranges. The variables do not have to be declared before they are used. The variables are automatically declared when they are used.

The scope of the variables declared inside high hierarchy process (`host`) is global for all processes defined inside `host`.

1.3 Functions

The system behaviour is changed by the functions which modify the states of the variables and pass the objects by communication channels. The function is declared as follows:

```
fun new_sk(id)[Availability:bitlength,algorithm; Integrity:alg2]
```

The function declaration is proceeded by the phrase `fun`. This function is named `new_sk`. After defining the name of the function one has to set the arguments of this function. These arguments should describe two types of factors. The functional parameters, which are written in round brackets, are necessary for the execution of the function. The additional parameters, which are written in square brackets, influence the system quality of protection. The name of the arguments is unrestricted.

In the above example, the function `new_sk` defines generation of a secret key. Six parameters are set: `id` - the id of the secret key owner (functional parameter); `Availability` - the name of the security attribute which describes the qop parameters defined after colon (qop parameter); `bitlength` - the bit length of the secret key (qop parameter); `algorithm` - the algorithm name (qop parameter); `Integrity` - the name of the next security attribute which describes the qop parameters defined after colon (qop parameter) and `alg2` - the algorithm name (qop parameter).

1.4 Equational rules

Equational rules play an important role in the quality of protection protocol analysis. Equational rules for a specific protocol consist of a set of equations asserting the equality of cryptographic primitives and other security functions. The asymmetric cryptography can be modelled as follows:

```
dec(enc(data,PKid),SKid) = data
```

In this example, the `data` represents the plaintext, `SKid` is a secret key of the site id and `Pkid` is a public key for the same site. The `enc` symbol defines the encryption operation and the `dec` symbol defines the decryption operation.

1.5 Process Types

The processes are the main objects in the QoP-ML. The objects which describe the system behaviour (functions, message passing) are grouped into processes. The processes are declared as follows:

```
process A(ch1, ch2)
{
  key = 2048;

  subprocess A2(ch2)
    {
       key2 = 1024;
    }
}
```

The process is named A. This process can communicate with another one by two channels named ch1 and ch2. These channels must be previously defined (communication channels will be described later). The body of the declarations is enclosed in curly braces. The process A consists of two simultaneously proceeded operations, the declaration of a variable key and assigning to it the value 2048. The statements inside the declaration body are separated by the semicolon.

Inside the processes (process operator) one can defined sub-processes. The sub-processes (subprocess operator) can be defined only inside the previously defined processes. In the presented example, the sub-process is named A2. This process can communicate with another one by channel ch2. The body of the declarations is enclosed in curly braces. The subprocess A2 consists of one operation, the declaration of a variable key2 and assigning to it the value 1024. The statements inside the declaration body are separated by the semicolon.

In the real system, the processes are executed and maintained by a single computer. In the QoP-ML the sets of processes are grouped into the higher hierarchy process named host. This process is defined below:

```
host A(fifo)(ch1,ch2)
{
  process A(ch1)
  {
    key1 = 2048;
  }
  process B(ch2)
  {
    key2 = 1024;
  }
}
```

The process A and the process B are grouped into the higher hierarchy of process defined by the operator host and the unique name A. The host operator has two parameters, the first one refers to the CPU scheduler algorithm which will set the process queue. The algorithm could be fifo or round robin where the quantum of time is defined as the single operation. The operations inside the process are executed one by one including those defined in the subprocesses (subprocess operator). The second parameter refers to the channels which will be available for the processes grouped into the host process. When there is the star in the second round brackets, it means that all channels are available. When the second bracket is empty then it means that no channels are available.

In this part it is important to discuss the scope of the variables inside the processes. All of the variables used in the high hierarchy process (host) have a global scope for all processes which are grouped by this processes. Normally, the variables used inside the host process can not be applied for other high hierarchy process. This operation is possible only when the variable is sent by the communication channel.

1.6 Process instantiation

The process defined by the operator `process` describes its behaviour but is not automatically executed. In the QoP-ML, the first executed process is the one named `init`. In this process one defined the version of the protocol which will be executed. The `init` process can be defined as follows:

```
init
{
  version 1
}
```

In the `init` process one can used operator `version`. This operator uses one argument, which refers to the specification of the protocol version and executes it. This specification is defined by the structure of the same name - **version**. It initializes global variables and other processes. The `version` structure can be defined as follows:

```
version 1
{
  run host A
   {
     run A1(A1A,A1B)
     run A2() -> run A3(*)
    }

  run host B {3}
    {
     run B1(*)
    }

  run host C {2}[ch1,ch5.3]
    {
     run C1(*)
     run C2()
    }

  run host D
    {
  run B1(*){5}
    }

}
```

The main operator, used in the `version` structure, which executes the process is `run`. This operator takes the name of any type of process and executes it, but normal processes can be performed only inside the high hierarchy process `host` (`run host A`). The processes executed by the `run` operator can be replicated. The number of replication one can defined inside curly brackets (`run host B {3}`). When the process is replicated, then to its name are added additional identification. In the presented example, the high hierarchy process `host B` is tripled and these process will be identified as `B, B.2, B.3`. When the normal process will be replicated, the naming scheme is the same and the process `B1` which is executed by the high hierarchy process `run host D` will be identified as `B1, B1.2, B1.3, B1.4, B1.5`.

When the processes are replicated, then the communication channels, used in these processes, are replicated, too. The channel names are expanded with additional information following the same naming scheme as in the case of processes. The processes replicated in this way will realize communication between other processes independently, because the channels are replicated too. In the QoP-ML there is possibility to replicate the process with manual definition of the channels, which allows you to assign the same channels for the replicated processes. This specification is defined in the square brackets `run host C {2}[ch1,ch5.3]`. In this situation, the high hierarchy process is replicated twice, but for the all processes defined in this high hierarchy process, the channel `ch1` is not replicated. The channel `ch5` is replicated with static identification `.3` and will be the same for all replicated high hierarchy processes. It means that all replicated processes from `host C` will communicate on channels `ch1` and `ch5.3`. Other channels are replicated in the same way as the processes are replicated. When the channels are defined manually for high hierarchy process then this definition applies to all channels used in it.

Other parameters which are used in `run` operator are round brackets. In round brackets one can select the sub-processes which will be executed as the child of the parent process. If the main process has to run all child processes then in the round bracket one has to write a star (*). When the brackets are empty, it means that no child process will be run.

A running process disappears when it terminates but not before all child processes are terminated. The process can be terminated when it reaches the end of the body of its process or by operators: `stop` or `end` (this operators will be described later).

Another operator is an arrow (`->`) which means that the instruction written after the operator will be executed only if the process which proceeds the operator is terminated without an error.

1.7 Message passing

The communication between processes is modelled by means of channels. Any type of data can be passed through the channels. The channels must be declared before the data is passed through. Below we present the declaration of the channel:

```
channel ch1 (10)[500kbits];
```

This declares a channel named `ch1` which can store 10 messages (buffer size) and the bandwidth of the channel is 500kbps. When there is a star in round brackets it means that there is no limit for message passing by this channel (buffer is not set). If the channels have the same parameters then they can be declared in one statement:

```
channel ch2,ch3 (*)[1mbits];
```

The data can be sent or received by the channels. The data sending can be modelled as follows:

```
out(ch1:M1,M2);
```

This statement sends the value of the expressions `M1` and `M2` through the channel `ch1`. That data can be received by the process in which the statement presented below is modelled:

```
in(ch1:X,Y);
```

The values of the expressions `M1` and `M2` will be assigned to the variables `X` and `Y`. The channels pass the message in the FIFO order.

Synchronous communication When the channels are declared with the non-zero buffer size, the communication is asynchronous. When the buffer size is equal to zero then the communication is synchronous. It means that the sender will transmit the data through the synchronous channel only when if the receiver is listening on this channel. When the size of the buffer channel equals at least 1 then the message can be sent through this channel even when there is no process of listening on this channel. This message will be transmitted to the receiver when the listening process in this channel is executed.

1.8 Control operators

The objects which describe the system behaviour can be controlled by two types of constructions. These are: condition statement and repetition.

Condition statement The `if` statement evaluates the expression. If that expression is true, then a statement is executed. The example is presented below:

```
if (A==B)
{
  out(ch1:M1,M2);
}
else
{
  out(ch1:M3);
}
```

In this example if the `(A==B)` conditional expression is true, then the messages `M1` and `M2` will be sent through the channel `ch1`. The `else` statement is given so if the condition expression is false, then the else's statement is executed.

Repetition The iterative loop is provided by the `do-while` structure:

```
do
{
 out(ch1:M1,M2);
}while(A==B);
```

In this example the messages `M1` and `M2` will be sent through the channel `ch1` as long as the expression `(A==B)` is true. The conditional expression takes place after the `while` operator.

Other structures In the QoP-ML one can use two types of structure which can control loops flow. These are `break` and `continue`. The `break` operator escapes from the nearest outer loop. The `continue` in the `do-while` loop will switch the program execution to the test condition. The usage of `break` and `continue` operators outside the loops makes no sense.

```
do
{
  if (A==Z)
  {
    break
  }

  if (A==T)
  {
    out(ch1:M1);
    continue
  }
  out(ch1:M1,M2);
}while(A==B);
out(ch2:M5);
```

In this example when the expression `(A==Z)` is true, then the `do-while` loop will be stopped. After this the `out(ch2:M5)` expression will be proceeded. Another possibility is when the expression `(A==T)` is true, then the `out(ch1:M1)` will be executed. After that, there is the `continue` statement so the loop will stop and the `do-while` condition will be checked `while(A==B)`.

Another operator which is very useful in the protocol modelling is a `stop` operator. This structure will stop further execution of modelled protocol which means protocol failure and an error is returned. The protocol can be stopped by the `end` operator as well. When this operator is used, it means that the protocol is executed successfully and no errors are returned.

```
do
{
  if (A==Z)
  {
    stop
  }
  out(ch1:M1,M2);
}while(A==B);
end
```

In this example when the expression `(A==Z)` is true, the protocol will cancel further execution and the error will be returned. But when the `do-while` condition will be true, the `end` operator will finish the protocol successfully.

In the QoP-ML one can use more operators. The one is a `true/false` operator which indicates that an expression is true or false. The next one is `#` which proceeds the expression in the process and it means that this expression was executed in the past (before the given process runs). Thanks to the `#` operator one can define the variables which can be used by the processes but would not be taken into consideration during the quality of protection analysis.

In the QoP-ML one can use other operators which are the same as in the language C. They are: `&&`, `||`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, `==`, `!=`.

2 Security Metrics

The system behaviour which is formally described by the cryptographic protocol can be modelled by the proposed QoP-ML. One of the main aims of this language is to abstract the quality of protection of particular version of the analysed cryptographic protocol. In the QoP-ML, the influence of the system protection is represented by means of the functions. During the function declaration the quality of protection parameters is defined and the details about this function are described. These factors do not influence the flow of the protocol but they are crucial for the quality of protection analysis. During that analysis the function qop parameters are combined with the next structure of QoP-ML which is called security metrics. In this structure one can abstract the functions time performance, their influence on the security attributes required for the cryptographic protocol or other important factors during the qop analysis.

The security metrics are started by the operator `metrics` and the body of the metrics is closed in the curly brackets. The metrics are defined by the five operators:

1. `conf` - defining the hardware and software host configuration;
2. `data` - defining the security metrics which can be measured and depends only on the host configuration;
3. `data+` - defining the security metrics which can be measured but depend on the randomness factors which can not be clearly defined by host configuration;
4. `data*` - defining the security metrics which can not be measured but can be modelled;
5. `set` - defining the set of security metrics for the processes which are grouped into high hierarchy process named `host`.

In the next part of this section the exemplary declaration of this structure will be presented. Afterwards, this structure will be described.

```
metrics
{
  conf(host1)
  {
    CPU = Intel Core 2 1.83 GHz;
    CryptoLibrary = Crypto++5.6.0;
    OS = Windows Vista in 32-bit;
  }

  conf(host2)
  {
    CPU = AMD Opteron 8354 2.2 GHz;
    CryptoLibrary = Crypto++5.6.0;
    OS = Linux 64-bit;
  }

  data(host1)
  {
    primhead[nr][bit length][algorithm][function][Availability:time];
    primitive[1][2048][RSA][encryption][0.16ms];
    #
    primhead[nr][bit length][algorithm][mode][function][time];
    primitive[2][256][AES][encryption/decryption][CBC][21.7 cycles/byte];
  }

  data(host2)
  {
    primhead[nr][bit length][algorithm][function][Availability:time];
    primitive[1][2048][RSA][encryption][0.08ms];
    #
    primhead[nr][bit length][algorithm][mode][function][Availability:time];
    primitive[2][256][AES][encryption/decryption][CBC][18.5 cycles/byte];
  }

  data+(host1:1)
  {
    primhead[nr][bit length][algorithm][function][Availability:time];
    primitive[1][2048][RSA][private key generation][10ms];
  }

  set host A(host1:1);
  set host B(host2:);

}
```

In this example, two types of the host configuration (`conf()`) were defined. The first one named `host1` and the second one named `host2`. In both cases, the hosts were declared by means of the three parameters: the processor type (CPU), Crypto Library and Operating System (OS). The number of parameters and the description details depend on the analysis requirements.

In the next step, the security metrics which can be measured and depend only on the host configuration were defined (`data()`). The `data()` operator requires one argument which automatically links that data with the previously defined name of the host configuration. In the example, security metrics were defined for host1 (`data(host1)`) and host2 (`data(host2)`). The body of the `data()` structure contains the two operators: `primhead` and `primitive`; which define security metrics for security algorithms or mechanisms. The `primhead` operator defines the required parameters for security mechanisms description. In the presented example for `data(host1)` the following parameters were defined: the unique (for the `data(host1)` structure) number of the primitive (`[nr]`), the bit length of the algorithm (`[bit length]`), the algorithm name (`[algorithm]`), the type of operation (`[operation]`) and the execution time of this operation (`[time]`). After that, the `primitive` operator is used which defines the details about previously defined security mechanisms. In the example the primitive indexed as 1 (`[1]`) defines the execution time for 2048 key bit RSA encryption. (The execution time for the cryptographic algorithm implemented in Crypto++ library was taken from the official benchmarks [http://www.cryptopp.com/]; in the literature one can find many benchmarks of the cryptographic primitives which can be used as security metrics. In the same `data(host1)` structure another primitive was defined but for its description one more parameter was required. Therefore the new `primhead` operator defines a new set of parameters. These two types of parameters are separated by `#`. The second primitive (`primitive [2]`) is declared and defines the execution time of 256 key bit AES encryption/decryption in the CBC mode of operation. Similar security metrics were defined for `host2` (`data(host2)`).

Afterwards, the security metrics are defined. They can be measured but depend on the random factors. This structure was defined by the operator `data+(host1:1)`. This operator has one argument which links the data with the host configuration named `host1` and after the colon the version of the data is indicated (`:1`). The version of data must be set because the security metrics which depend on the randomness factors can be different for the hosts which have the same parameters defined by the `conf` structure. The usage of the `primhead` and `primitive` operators which are used inside the body of the `data+()` structure is the same as in the `data()` structure.

In the last step the defined security metrics must be linked with the high hierarchy system processes which are declared by the `host` operator. These high hierarchy processes group together its child processes which are executed in one computer. These metrics are linked by means of the `set()` operator. In the presented example, the defined metrics are linked with two processes named `host A` and `host B`. The `set()` operator requires one more parameter and it is the name of the host configuration which is declared by the `conf` structure. In the presented example there are `host1` for the `host A` process and `host2` for the `host B` process. After declaring the name of the host configuration one can indicate other defined security metrics. In the presented example (`set host A(host1:1)`), after the colon the version of the defined security metrics is indicated.

After linking security metrics with the `host` high hierarchy processes, one has to link concrete primitives defined in metrics with specific security functions used during modelling the system behaviour. In the QoP-ML, the declaration function requires defining the quality of protection parameters which are written in square brackets. These factors refer to the primitives defined in security metrics.