Interacting Abstract State Machines

Marcelo de Almeida Maia[1], Vladimir Oliveira Di Iorio[2], and Roberto da Silva Bigonha[3]

[1] Universidade Federal de Ouro Preto, Brazil, marcmaia@dcc.ufmg.br
[2] Universidade Federal de Viçosa, Brazil, vladimir@dcc.ufmg.br
[3] Universidade Federal de Minas Gerais, Brazil, bigonha@dcc.ufmg.br

Abstract. In this work we propose an extension to the original model of Abstract State Machines. We focus on the modularization support and on the explicit interaction abstraction between the modules (units of specification). We provide the new language syntax and semantics, and also the specification of the Alternating Bit Protocol using the proposed method.

1 Introduction

Much of the work being done in the software engineering area concerns the development of mechanisms that facilitate the reuse and flexibility of software components. The most powerful resource to achieve these goals is modularity, which is based upon abstraction and information hiding and it is the only effective way to break down the complexity of large systems. Even though Abstract State Machines[4] support abstraction and information hiding, we advocate more powerful abstraction mechanisms. If we consider the inherent methodology of producing ASMs specifications as a methodology that provides a vertical abstraction mechanism, in the sense that the ground model is successively refined until considered adequate, it is reasonable to think that it does lack some kind of horizontal abstraction to support the reuse of existent specifications. An argument to support this view can be found in [8], where are defined some desired characteristics for good modularization mechanisms such as modular composability, modular decomposability, modular understandability, modular continuity and modular protection. Considering these characteristics, a central theme that affects directly each one of them is the specification of how software modules interact with each other. So, our decision is in the direction of a formalism that explicitly enables the software engineer to write down how the interaction occurs between the modules. We adopt a message exchanging style because we believe it provides a natural abstraction of how objects interact in the real world. When we explicitly specify the interaction between modules, we are automatically inclined to think about the concurrency issues involved in the interaction process. In our view, modularization and concurrency concepts are interdependent and should not be addressed separately, and this has influenced our decision of putting them together in a unique framework.

In the context of ASMs, there is already some work in the direction of providing them with some kind of horizontal abstraction. Glavan and Rosenzweig developed a theory of concurrency [3] that enables the encoding of some traditional calculus as the $\pi-$calculus [9] and the Chemical Abstract Machine [2]. However, we can not see an explicit message passing mechanism and it does not support encapsulation and information hiding mechanisms, issues which will be directly treated in this work. May [7] has developed a work with the same aims as ours, and although it provides some form of encapsulation and information hiding, the usual modularization concepts must be further added to the model. The explicit message passing encoding is not considered too.

Instead of putting on the user the burden of providing the complete specification of the message interchanging between different specifications, our approach provides special constructions to help the explicit specification of how different pieces of specifications interact with each other. This idea can be thought as a better development of the concept of external functions [4], because the approach provides some environment behavior formalization. It is not necessary to know how the environment behaves internally, but it is necessary to know how it interacts with the system being specified. So, when we specify a system, we must have in hands a minimal formalization of the observed environment behavior that affects the system, what is a little diferent from the raw concept of external functions.

In the Section 2 we present the proposed new constructions. In the Section 3 we develop the semantics for that constructions. In the Section 4 we present an example of usage. And finally, in the Section 5 we conclude standing up for the suitability of the new constructions in the development of large scale specifications. In the Appendix A we explain the syntactic conventions used throughout the text.

2 The Interactive Abstract State Machine Language

A specification is defined as a set of unit definitions and unit instances. Units definitions are classified as *system units* and *environment units*. System units are those which will be completely specified, whereas environment units will be partially specified. We use the word *environment* not only referring to the external portion of the system, but also referring to some components of the system that had already been specified and are being reused.

The intention of specifying a system as a set of units is to encapsulate some portion of the state inside small pieces of specification. This leads to an isolation of the internal state of a unit. The information contained in the internal state of a unit only can be communicated to other units by explicitly specifying a pattern of interaction between the involved units. This interaction specification does not only specify the information flow but also the synchronization restrictions within the interaction.

In the sequel we present the abstract syntax of our proposed language.

A system unit definition U_s is composed of several parts and it is defined as:

U_s ::= `unit` *unit_name*
  `function names` *function_names*
  `interaction` *interaction*
  `rules` *rules*

where:

- *function_names* is a subset of the vocabulary that contain the names of the functions. It represents, together with the respective interpretations of the names into the super-universe, a local state alterable only by the local unit rules and interaction. Each function name may be optionally initialized with an arbitrary value. In order to make the ideas clear, we will define an abstract data type (ADT) *Stack*, as we explain the parts of a unit. For the *Stack* unit we may have the following function names:

```
function names
   max := 100       % Maximum length of Stack
   s                % The stack itself
   top := 0         % Index of the top elem
   topelem          % The top element
   ack              % Acknowlegdement of pushing
   c                % The client of the Stack
```

- *interaction* is defined as:

interaction ::= *internal_pub_name* `->` *u_name*
  | *buffered_var* `<--` *u_name*`.`*pub_name*
  | *var* `<-` *u_name*`.`*pub_name*
  | `connect` *unit* : *U.s* | `connect` *unit* : *U* | `connect` *u*
  | `new` *unit* : *U*
  | `destroy` *unit* : *U*
  | *interaction* + *interaction*
  | *interaction* | *interaction*
  | *interaction* ; *interaction*
  | *interaction* : *l*
  | `waiting`(*name*)
  | `if` *guard* `then` *interaction*
  | `extend` U `with` x I `endextend`
  | `choose` v `in` U `satisfying` e I `endchoose`
  | `var` v `ranges over` U I `endvar`

The basic operators for interaction are those that provide input and output within a unit. They are the `->` and `<-`, used to send a value to a unit and to receive a value from a unit into a variable, respectively. The operator `<--` denotes a buffered input that avoids an inconsistent update if two or more different inputs to same variable occur in the same step. Since we expect to define dynamically the communication topology, we provide the `connect` operator which binds a unit name to some unit instance. The operators *new* and *destroy* are used to create and destroy unit instances. Since units are mapped into agents, these operators update the corresponding enumerating set of agents derived from a module. In order to address complex interaction

patterns that may exist between units we provide the well-known composition operators "+" (non-deterministic choice), "|" (parallel composition), and ";" (sequential composition). As we will define soon, one cannot reason about the relative speed of execution of an atomic interaction compared to an internal rule of a unit. Thus in order to synchronize the interaction part with the computation part of a unit we introduce labeled interactions and the barrier `waiting`*(name)*. The label l uniquely identifies an interaction, and denotes how many times the interaction labeled with l has completely occurred. Its initial state is *zero*. We also inherit from the ASM notation the `if`, `extend`, `choose`, and the `var` rules.
Coming back to our example, the ADT *Stack* may be seen as a server and thus it must connect with the *Client* before performing any information exchanging. As we will see, the `connect` operator used below waits until there is an interested unit instance requesting the connection. After the connection, it may receive requests from the *Client* instance. The requests guide the sequel of the interaction, and the unit *Stack* interacts with its *Client* by sending to it the element on the top of the stack (popping it or not) or receiving from it an element to be pushed onto the stack.

```
interaction
   connect c;
   request <- c;
   if request = "top" then
      topelem -> c
   elseif request = "pop" then
      waiting(popped);
      topelem -> c
   elseif top < max then
      elem <- c.elem;
      waiting(pushed);
      ack -> c
   endif
```

– *rules* is defined as an element of ASM_RULES. These rules work by changing the internal state represented by $function_names$. In the abstract data type *Stack* we may define the rules as:

```
rules
   if waiting(popped) then
      top--;
      waiting(popped) := false;
   endif
   if waiting(pushed) then
      top++;
      s(top+1) := elem;
      waiting(pushed) := false;
   endif
```

Now, let us define an environment unit as a restriction on a system unit. It shows the public portion of a system unit that can be imported by other units and can be defined as:

u_e ::= `environment unit` *unit_name*
`interaction` *interaction*

Each system unit has a corresponding environment unit specifying what will be exported to other units.

3 Semantics

In this section we specify the IASM language semantics. Whenever readability is not impaired, we give a translational semantics that maps a syntactic domain corresponding to the IASM constructions into the original ASM language defined by Gurevich[4].

Whenever the translational semantics turns to be quite complicated we will prefer to give informal semantics, while giving precise definitions about the meaning of the IASM language. The interested reader may find the complete translational semantics in [6].

3.1 Unit Definition

The $\mathcal{U}$ compilation scheme translate unit definitions and is defined as:

$\mathcal{U}$, $\mathcal{U}_{\text{mod}}$: $IASM_CONSTRUCTIONS \rightarrow ASM_RULES$

$\mathcal{U}$ $[\![\ U_1; \ldots U_n\ ]\!] = \bigcup_{i=1}^{n} \mathcal{U}_{\text{mod}}[\![U_i]\!]$

$\mathcal{U}_{\text{mod}}$ $[\![U]\!] =$

`module` U

$\mathcal{I}$ $[\![U.interactions\ ]\!] \bigcup$

$\mathcal{R}$ $[\![\ U.rules\ ]\!]$

`end module`

The idea of this compilation scheme is to put together, inside a module, the rules corresponding to each construction of each unit definition. The source code of the module will be generated from the unit definition U. This compilation guarantees that each unit instance derived from this unit definition will have its own clock, and thus its execution will be independent from other instances. We will also make use of the function *Self* which allows an agent to identify itself between other agents.

3.2 Unit Instantiation

The instantiation of a unit means an extension of the universe that enumerates the instances of a unit definition. There are two ways of declaring units:

1. statically: the startup definition declares the initial unit instances. This declaration actually creates statically the unit instances which will live during the whole execution of the specification.
2. dynamically: the declaration of a unit as part of the internal state of another unit does not create an instance. Instead, the declaration produces a function name that will be dynamically bound to a unit instance, either a statically created, or a dynamically created one. A dynamic unit may be created and destroyed with the interaction instructions `new` and `destroy`, respectively.

Because unit instances are agents, creating and destroying units means to extend or retract the enumerating universe that contains the agent names of the specified module.

3.3 Output Interaction

The first case of interaction is when an internal value is sent to a unit. This sending means that the internal universe MSG is extended with a new message. This universe contains the messages exchanged between the units. A message carries its target, a label denoting the source of this value, and a value.

```
I [[internal_pub_name -> u_name ]] =
  extend MSG with x
    target(x) := ‘u_name;
    label(x) := ‘self.internal_pub_name;
    cont(x) := internal_pub_name;
  endextend;
```

3.4 Input Interaction

There are two possible semantics for receiving a value from another unit. The name responsible to store the received values may be buffered or not. In either case, it is done a query into the universe MSG to find a message that matches the required input interaction. If this message exists then the corresponding updates are done and the used message is discarded from the universe MSG.

In the case where received values are not buffered, the variable is translated into a function name and if there is a message that matches the input interaction then the corresponding updates take place.

```
I [[var <- u_name.pub_name ]] =
  if has_message(‘u_name.pub_name) then
    choose x in MSG satisfying match_msg(x, ‘u_name.pub_name)
      var := cont(x);
      MSG(x) := false;
    endchoose;
  endif;
```

In the other case, the buffered variable will be translated into a universe. Each time the variable receives a value, the corresponding universe will be extended with that value.

```
I [[buffered_var <-- u_name.pub_name ]] =
  if has_message(‘u_name.pub_name) then
    choose x in MSG satisfying match_msg(x, ‘u_name.pub_name)
      extend buffered_var with y
        cont(y) := cont(x);
      end extend;
      MSG(x) := false;
    endchoose;
```

```
endif;
```

In order to access the buffered values we will assume that a timestamp is assigned to each value received into a buffered variable. This timestamp is incremented with step one, and if there are many incoming values in the same step in the same variable, then the corresponding timestamps are assigned non-deterministically to each value. For example, suppose there is a buffered variable x that is receiving two values, for instance "v1" and "v2", in the same step. If the current timestamp to be assigned to the incoming value is, for example, 8, then the timestamps to be assigned non-deterministically to "v1" and "v2" will be 8 and 9, and the current timestamp will be set to 10.

3.5 Unit Connections

As stated before, a unit declaration inside a unit definition only produces a function name. Our intention is that this function name should be further bound to another unit instance which also has a function name bound to the former unit instance. This situation indicates an agreement between the two instances, and it is performed with the operator `connect`. The arguments for this operator are: 1) a function name u corresponding to a unit instance, 2) the name U corresponding to the unit definiton from which the instance u was derived, and 3) a function name s declared inside U that we expect to be bound to the current unit instance.

There are some possibilities when using `connect`:

- All arguments are defined. Then it must be checked that if there is another instance that attempted a connection that matches this one. If there is such attempt, then the connection is successfully performed, otherwise it is blocked until such attempt occurs.
- Some arguments are undefined. This possibility is necessary because when establishing a connection we may not know in advance which unit instance will be connected or even from which unit definition the unit to be connected was derived. We may write `"connect u: U"`, where `"u"` is undefined and we are not interested on which function name inside `"U"` will receive the name of the current instance. Alternatively, we may want more flexibility and write `"connect u"`, where `"u"` is undefined. In this case, any unit wanting to connect through the channel `"u"` can match this connection.

The semantics of this operator may be given defining a universe *Connections* that the operator `connect` can update and query in order to establish the connection. Each element of this universe is a pair representing the two connected instances. Each element of the pair is a triple (u, U, o), where u is the name of the instance involved, U is its corresponding unit definition name and o is the function name whose value is the name of the other instance belonging to the pair.

3.6 Interaction Composition and Runs

Since an IASM specification can be translated into the pure ASM notation, as a set of modules and agents, the notion of run for interactive ASMs is the same as that of pure ASMs. But, compared with the pure ASMs, the composed interaction portion of the specification has a different state transition granularity. So, the reasoning mechanism of pure ASMs should not be used for Interactive ASMs, which have a more elaborated notion of move. Thus, in the sequel, we define a special notion of *interaction cycle* that is independent from the notion of move of the internal rules. The latter obeys the partially-ordered semantics of distributed ASMs.

Definition 1. *Interaction tree is the abstract syntax tree derived from the interaction part of a unit definition.*

Definition 2. *Interaction cycle is the result of executing the moves of the rules corresponding to a node of the interaction tree until there are no more pending nodes waiting to be executed. It is executed according to the following definitions.*

Definition 3. *Parallelism: If there is an enabled interaction tree with the following form:* $(i_1|\ i_2|\ \cdots\ |\ i_n)$ *then all interactions* i_k $(1 \leq k \leq n)$ *are enabled and the execution of each* i_k *is cyclic and do not depend on each other.*

Definition 4. *Sequence: If there is an enabled interaction tree with the following form:* $(i_1; i_2; \cdots ; i_n)$ *then one and only one interaction* i_k $(1 \leq k \leq n)$ *is enabled at each time and all the sequence is executed in the cycle.*

Definition 5. *Non-determinism: If there is an enabled interaction tree with the following form:* $(i_1 + i_2 + \cdots + i_n)$*, then one and only one interaction* i_k $(1 \leq k \leq n)$ *will be effectively performed on each cycle of the non-deterministic interaction.*

Definition 6. *Blocking input: Suppose there is an enabled interaction tree with the following form:* $(a <- u.b; i_1)$*, where* a *is a function name, and* $u.b$ *is an incoming value from the function name* b *of the unit* u*. Then, the respective input blocks* i_1*, until the input effectively occurs.*

3.7 Synchronization of Internal Rules and Interactions

Unit internal rules are just like ASM rules and its semantics is exactly the same. But, there is no direct relation on the synchronism between the interaction rules and internal rules. In order to guarantee appropriated synchronization when executing these rules, the IASM method provides:

1. A waiting rule used in the interaction section. This rule is represented by a boolean function name. When executed the rule updates the function with `true` and freezes the execution of the current node in a interaction cycle until the function is updated with `false`.
2. All interactions may be labeled, for example:
   msgrec <- S.msgsend : nb_rcvd_msgs
   The corresponding label denotes an integer value which corresponds to how many times an interaction has been completed. This value can be used by the internal rules.

4 An example - The Alternating Bit Protocol

In this section we show a more elaborated example: the specification of the *Alternating Bit Protocol* [1]. The problem consists of transmitting messages through an unreliable channel. The channel delivers messages in the same order they were sent, but can occasionally loose some of them.

The specification is composed of three main unit definitions: *Sender*, *Receiver* and *Channel*. In addition, four other units are defined as environment units:

- unit *ClientSender*: Simulates the behaviour of a client that delivers messages; the messages are sent to the unit `Sender`, which sends copies of them to ensure the correct delivery.
- unit *ClientReceiver*: Simulates the behaviour of a client that receives messages from the unit `Receiver`.
- unit *Timer*: Sends periodically a message to the unit `Sender`, at a fixed rate, to indicate that a new copy of the current message must be sent. `Timer` has a behaviour which is similar to that of *external functions* in pure ASM.
- unit *Loose*: Sends messages to the unit `Channel` non-deterministically, indicating when a message will be lost. Like `Timer`, its behaviour is similar to that of external functions.

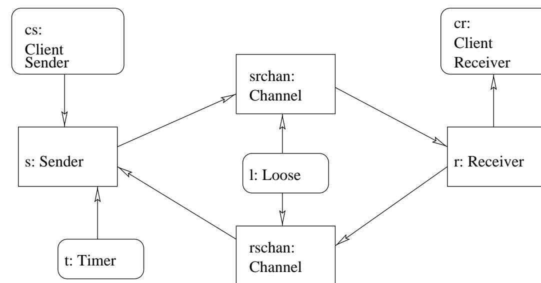

Fig. 1. Units and the flow of messages in the AB Protocol.

Figure 1 shows the relationship between these units. Note that two instances of the unit `Channel` are necessary.

After the unit `Sender` has performed the connections, it receives from the client a message which must be delivered to the receiver through an unreliable channel. Then it sends copies of the message through an output channel (srchan), together with a signal bit. This process is repeated until the unit `Sender` receives a correct acknowledgement message from the input channel. Then it waits for a new request from the client.

The timer is used to indicate that the sender has spent a certain amount of time waiting for the correct acknowledgement signal and must then deliver a new copy of the message. Note that after the message was sent via srchan, either the sender receives non-deterministically an acknowledgement bit or a timeout signal, as we should expect.

```
unit Sender
   function names
      timeout, connected := false;
      msg, recvbit;
      bit := 0;
      next := true;
      srchan := "srchan"; rschan := "rschan"; c := "c1"; t := "t";
   interaction
      if not connected then
         connect srchan: Channel.input, rschan: Channel.output,
                 c: Client, t: Timer;  : Connection
      else
         if next then
            msg <- c.msg;
         msg -> srchan ; bit -> srchan;
         (recvbit <- rschan.bit +
          timeout <- t.timeout);
         if not timeout and recvbit = bit then
            ack -> c;
            waiting(nextmsg);
         else
            waiting(samemsg);
      endif
   rules
      if Connection > 0 then
         connected := true;
      if waiting(nextmsg) then
         bit := toggle(bit); next := true;
         waiting(nextmsg) := false;
      endif
      if waiting(samemsg) then
         timeout := false; next := false;
         waiting(samemsg) := false;
      endif
end unit
```

After the unit `Receiver` has performed the connections, it receives a message from the input channel (*srchan*) and if it is not a copy of the last value received, it is sent to the client. It also sends an acknowledgement signal via an output channel (*rschan*). Note that *msg* is also sent via *rschan* in order to correctly match the interaction pattern of *Channel*.

```
unit Receiver
   function names
      connected := false;
      msg, bit;
      currbit := 0;
      srchan := "srchan"; rschan := "rschan"; c := "c2";
   interaction
      if not connected then
         connect srchan: Channel.output, rschan: Channel.input,
                 c: Client;  : Connection
      else
         msg <- srchan.msg ; bit <- srchan.bit;
         ((if bit = currbit then
              msg -> c;
              waiting(nextmsg);
           endif)
          |
          (msg -> rschan ; bit -> rschan))
      endif
   rules
      if Connection > 0 then
```

```
         connected := true;
      if waiting(nextmsg) then
         currbit := toggle(currbit); waiting(nextmsg) := false;
end unit
```

The unit `Channel` simulates a channel that delivers the messages in the order they are sent, but it may occasionally loose some of them. It connects to the units representing the *input* and *output* of the channel and also to a unit `Loose` that determines non-deterministically when a message will be lost. Note that the connection of the *input* and *output* will be blocked waiting the assignment that will be done in the units *Sender* and *Receiver*.

```
unit Channel
   function names
      input, output, msg, msg2, bit, bit2;
      loose: Loose;
      connected := false;
      queue := nil;
   interaction
      if not connected then
         connect input, output;
         connect loose: Loose;  : Connections
      endif ;
      ((msg2 <- input.msg ; bit2 <- input.bit ;
        waiting(buffering))
       |
       (if msg <> undef then
          msg -> output ; bit -> output ;
          waiting(cleanmsg);
        endif)
       |
       (loosemsg <- loose.loosemsg ;
        waiting(LoosingInQueue))
      )
   rules
      if Connection > 0 then
         connected := true;
      if waiting(buffering) then
         queue := append( (msg2, bit2), queue);
         waiting(buffering) := false;
      endif
      if waiting(cleanmsg) then
         msg := undef; waiting(cleanmsg) := false;
      if waiting(LoosingInQueue) then
         queue := tail(queue); waiting(LoosingInQueue) := false;
      if msg = undef and queue <> nil then
         msg = first(head(queue)); bit = second(head(queue));
         queue := tail(queue);
      endif;
end unit
```

This protocol has been previously formalized using the ASM method in [5]. In that work, the behaviour of the communication channel was not clearly defined. It was necessary to write identical code for both the communication sender-receiver and receiver-sender.

For the sake of space economy we will not specify the environment units *ClientSender*, *ClientReceiver*, *Timer*, and *Loose*. The startup specification that creates the initial unit instances may be written as:

```
specification ABP
   rschan, srchan: Channel;
   s: Sender; r: Receiver;
   t: Timer; l: Loose;
   c1: ClientSender; c2: ClientReceiver;
end specification
```

5 Conclusions

We have presented a proposal to promote the reuse of ASMs specifications while addressing important issues such as communication and concurrency. The idea of explicitly isolating the interaction between computing units with different purposes make clearer their interdependencies. This also provides a useful mechanism to formalize the environment in which a specification will work.

The approach was successfully used in the specification of the Alternating Bit protocol, where we have reused the specification of the unit *Channel*, which may be connected differently depending on its usage. We have shown that the explicit message passing mechanism composed with well-known operators provides a powerful and natural specification mechanism.

The dynamic configuration of the communication topology presented is an essential feature that may be used to specify mobile systems which are being increasily used but still lacks more suitable formal approaches.

There are some aspects that must be further studied:

- study of static checking mechanism, e.g., type systems;
- study of more powerful reuse mechanisms, e.g., inheritance;
- encoding of procedure and/or function calls and/or method invocations;
- use of the approach in large scale specifications.

References

1. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A Note on Reliable Full-Duplex transmission over Half-Duplex Links. *Communications of the ACM*, 12(5):260–261, May 1969.
2. G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
3. P. Glavan and D. Rosenzweig. Communicating Evolving Algebras. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 182–215. Springer, 1993.
4. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
5. J. Huggins and R. Mani. The evolving algebra interpreter version 2.0. Manual of the interpreter (http://www.eecs.umich.edu/gasm).

6. M. Maia and R. Bigonha. The Formal Specification of the Interactive Abstract State Machine Language. Technical Report 005/98, Universidade Federal de Minas Gerais, Brazil, 1998. http://www.dcc.ufmg.br/~marcmaia/iasmformal.ps.gz.
7. W. May. Specifying Complex and Structured Systems with Evolving Algebras. In *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, number 1214 in LNCS, pages 535–549. Springer, 1997.
8. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.
9. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

A Syntactic Conventions

We have used the following conventions:

- Unit definition is the specification of the code of a unit.
- Unit declaration creates a function name or a unit instance.
- Italic letters: u_i, $u_i.interactions$ denote syntactic elements.
- Typewriter letters: denote reserved words of the original ASM language and of the constructions proposed here.
- 'a : denotes a label that uniquely identifies the syntactic element a, i.e., is the element of the vocabulary without being interpreted.
- $[\![\ldots]\!]$: denotes syntactic elements.
- `a += b;` $\equiv$ `a := a + b;`
- `a -= b;` $\equiv$ `a := a - b;`
- has_message(*origin*) =($\exists$ `x` $\in$ MSG) *target(x)=self* $\wedge$ *label(x)=origin*
- match_msg(x, *origin*) = $target(x) = self \wedge label(x) = origin$