INTEGRATION THROUGH META-COMMUNICATION

Branislav Meandzga

Meta Access Inc.
P. O. Box 21956
Santa Barbara, CA 93121-1956
&
Department of Mathematics and Computer Science
University of California, Riverside, CA 92521-1956

ABSTRACT

A variety of communications architectures, protocols, and management systems necessitates methods and tools for integration of these communications systems components on a variety of different integration levels. No general integration methods exist today which are applicable to the integration of architectures, protocols, and systems. In this article we argue that meta-communication, i.e., communication about communication rules, is such a general integration methodology. We summarize our efforts towards the development of an automated methodology for meta-communication.

We view meta-communication as a distributed design problem. Meta-communicating entities exchange partially specified communication rules. Each entity, or a meta-communication center, applies a standard composition principle on the individual partially specified rules in order to derive the complete protocol architecture. We report on an existing implementation of the meta-communication mechanism and on translators that transform the resulting specification into C code.

1. INTRODUCTION

The goal of automating meta-communication is the capability of designing, specifying, and implementing communication rules at run-time. This means, communication is enabled solely through the meta-communication mechanism. Initially, the only known common denominator between communication partners is the meta-communication mechanism. That mechanism defines the communication rules and therefore defines the way in which the components are to be integrated to a system. Thus, a meta-communication method is the basic tool for integration of communications systems. We have developed, [8], [9], and partially implemented, [11], [13], a methodology that automates meta-communication. In the following sections we summarize our results and report on the implementation.

Meta-communication can be defined in such a way to include negotiation about communications media and media access procedures. This can be done by using a different communications medium or by special types of physical communications interfaces. In the following, however, it is assumed that both, meta-communication and communication use the same communications equipment, medium, and media access procedures.

The meta-communication approach allows us to postpone the protocol design decisions until all communication characteristics are known. This means we can solve each communication task in the most efficient way known. Thus, the meta-communication system is flexible to environments and applications. This should result in improved performance. Another advantage of flexibility is the adjustability of constraints. This should result in longevity of the design and implementation. Finally, it should be noted that meta-communication as it is defined in this paper can be easily modified to solve protocol conversion problems as defined in [4], [5], and [7].

We view meta-communication as a distributed design problem. Meta-communicating entities exchange partially specified communication rules. Each entity, or a meta-communication center, apply a standard composition principle on the individual partially specified rules to derive the complete protocol architecture.

The protocol architecture (both, protocol specifications and system architectures of all communications sides, are represented) is specified with the abstract specification language of Archetype [10]. Partial specifications are given as predicates on the set of all Archetype specifications. Thus, each meta-communicating entity formulates constraints on the set of acceptable protocol architectures. Each entity meta-communicates its constraints to each other entity (or to a center). All entities (or the center) apply the same constraint composition principle on the accumulated constraints to deduce a set of constraints that specifies a single protocol architecture. All entities compile this specification into executable programs that subsequently automatically loaded to drive the protocols of the specified protocol architecture.

Abstract specifications of Archetype are not directly translated into executable programs. This wouldn't be practical from the software engineering standpoint because of the very high level of abstraction of those specifications which was necessitated by the meta-communication methodology. The abstract specification language is descriptive and can be used to express a "Natural Language"-like specification of the protocol architecture. An abstract specification is transformed into a specification in the second language of Archetype, the data-driven, concurrent, executable specification language. A translator for the abstract specification language is being developed. A translator for the executable specification language into C has been implemented.

Thus, the meta-communication system has successfully produced protocol drivers for UNIX 4.3 and SYSTEM V and has successfully produced protocol drivers for SUN workstations, the TI Explorer, and a VAX 11/780.

The crucial part in the meta-communication mechanism is the combination of constraints and the deduction of the protocol architecture. We have implemented this process using LISP and have successfully deduced protocol architecture specifications by combining sets of constraints. Figure 1 illustrates the entire meta-communication architecture.
In Section 2 we introduce the abstract specification language. In Section 3 we outline principles of constrained specifications, their composition, and the special purpose theorem prover that applies.

In Section 2 we introduce the abstract specification language. In Section 3 we outline principles of constrained specifications, their composition, and the special purpose theorem prover that applies.

The Archetype executable specification language is suitable for specifying protocols into a C program. Therefore, it is necessary to be able to define the actual representation of the archetype in the information unit. This is done by defining <code>. One example for the correspondence between archetypes and information units is the archetype PHASES. It defines the subdivision of a communication session into subsessions. Subsessions can be represented by the unit types which can be exchanged in the subsession. For instance the "connect" phase needs the explicit identification of the "call" and the "response". The complete syntax of the abstract specification language of Archetype is given in Appendix A.

We distinguish among three types of structural archetypes: abstractive, ordering, and classificational. The only abstractive archetype is the LAYERING archetype. There are two types of ordering archetypes: pattern archetypes (which define patterns in the stream of information units) and sequencing archetypes (which define a stream of information units as a sequence with certain characteristics). Patterns can be structured or simple. Structured patterns (phases) can consist of several subpatterns and can be nested and recursive. Simple patterns (transactions) are one-time patterns which can recur but not in any predictable fashion. Sequencing archetypes specify how information units are to be sequenced and, possibly, how the sequence is to be maintained by means of windows and acknowledgments.

There are four types of classificational archetypes. They are distinguished by classification criterion. The possible criteria are: communication function (send or receive), communication service, information abstraction (data or control), and communication partners. There are two ways of classifying by distinguishing among communication patterns: on the level of an information unit stream (logical channel) or on the level of the single information unit.

Relational archetypes define horizontal (with respect to the tree) mappings between subtrees. Blocking and segmenting are defined as mappings between units. Multiplexing, splitting, and routing are defined as mappings between identifications.

Several archetypes can be explicitly refined. An explicit refinement is the definition of a causal relationship (event -> effect).

The example specification in Figure 2 illustrates the principle and level of abstraction of an abstract specification with Archetype.

![Diagram](image-url)

**Figure 1: Meta-Communication Architecture**

In Section 2 we introduce the abstract specification language. In Section 3 we outline principles of constrained specifications, their composition, and the special purpose theorem prover that applies the composition on constraints to deduce the Archetype abstract specification. The Archetype executable specification language is introduced in Section 4. In Section 5 we give the principles of Yet Another Network Compiler (YANC) [13] that translates the executable specification into a C program.

2. PROTOCOL SPECIFICATION MODEL

The abstract specification language of Archetype is suitable for the automatic design process because of its abstract descriptive nature. A more conventional protocol specification technique such as the state-transition approaches [2] would make the specification of constraints for the design process complex because of the level of detail required.

An abstract specification is represented as a tree. Each node of the tree is labeled with an archetype, which dictates the shape of the subtree. The subtree of a node is the refinement of that node's archetype. The tree is specified as a list structure. Each node is represented as a list in which each child node is represented (recursively) as a sublist. Thus, the tree is structured as a nested list.

Each node is labeled by an archetype. The archetype can be refined by a subtree, by an explicit refinement specification (will be explicated later), or can be structured with the subarchetypes possibly refined.

```
<leaf> ::= (leaf)
<node> ::= <medium><archetype> | <leaf>
<medium> ::= <empty> | MEDIUM(<medkind>)
<medkind> ::= POINT_TO_POINT | INTERNAL
```

The label MEDIUM is used to indicate the axis along which exchange of information units is to take place. Everything below a node labeled MEDIUM is regarded as one, possibly distributed, information processing system.

Each archetype consists of a description and, optionally, an explicit refinement. Three types of descriptions are distinguished. The atomic description is of significance only to the node for which it is the label. Currently, we distinguish only between two types of functions of atomic descriptions. The purpose of DATA is to indicate information which, at that particular level of interpretation, is not to be analyzed any further. The purpose of NEGOTIATE(<id>) is to indicate a piece of information that is to be identified by <id> and dynamically negotiated among communication partners.

```
<archetype> ::= <description><refinement>
<description> ::= <atomic> | <relational><tree>
<atomic> ::= <empty> | <fnct><code>{<tree>}
<fnct> ::= DATA | NEGOTIATE(<id>)
```

Most archetypes can, in some way, be mapped onto the actual information units that will be communicated by the protocols they specify. Therefore, it is necessary to be able to define the actual representation of the archetype in the information unit. This is done by defining <code>. One example for the correspondence between archetypes and information units is the archetype PHASES. It defines the subdivision of a communication session into subsessions. Subsessions can be represented by the unit types which can be exchanged in the subsession. For instance the "connect" phase needs the explicit identification of the "call" and the "response". The complete syntax of the abstract specification language of Archetype is given in Appendix A.

We distinguish among three types of structural archetypes: abstractive, ordering, and classificational. The only abstractive archetype is the LAYERING archetype. There are two types of ordering archetypes: pattern archetypes (which define patterns in the stream of information units) and sequencing archetypes (which define a stream of information units as a sequence with certain characteristics). Patterns can be structured or simple. Structured patterns (phases) can consist of several subpatterns and can be nested and recursive. Simple patterns (transactions) are one-time patterns which can recur but not in any predictable fashion. Sequencing archetypes specify how information units are to be sequenced and, possibly, how the sequence is to be maintained by means of windows and acknowledgments.

There are four types of classificational archetypes. They are distinguished by classification criterion. The possible criteria are: communication function (send or receive), communication service, information abstraction (data or control), and communication partners. There are two ways of classifying by distinguishing among communication patterns: on the level of an information unit stream (logical channel) or on the level of the single information unit.

Relational archetypes define horizontal (with respect to the tree) mappings between subtrees. Blocking and segmenting are defined as mappings between units. Multiplexing, splitting, and routing are defined as mappings between identifications.

Several archetypes can be explicitly refined. An explicit refinement is the definition of a causal relationship (event -> effect).

The example specification in Figure 2 illustrates the principle and level of abstraction of an abstract specification with Archetype.
The system specified consist of one layer and two interfaces to this layer. The layer is based on a connection oriented protocol. The protocol is subdivided into three phases. The data-transfer phase is the same for sending and receiving. All information units carry an acknowledgement. All information units are subdivided into two types: data and control units. All data units are sequenced. Data units carry data.

layers(medium interface(1) : protocol : user interface)
  log-chan(1 to 256)
  phases(connect : data transfer : disconnect)
  comm_fct(send : receive)
  unit_seq(window(3)receive)
  interpretation(data : control)
  unit_seq(window(3)send)
    data
  leaf

Figure 2: Archetype Abstract Specification

3. DISTRIBUTED DESIGN OF A PROTOCOL ARCHITECTURE

A partial specification of a protocol architecture is given by defining a set of predicates on all the syntactic domains of the grammar of Archetype. We can have multiple predicates for the same domain, which could represent distinct points of view or subsequent changes of the same point of view. Those predicates are acted on by a composition function in attempting to construct a single specification tree. The general methodology has been formally defined using [1] (see [9]).

3.1. THE PROTOTYPE

The currently existing prototype is written in LISP and operates on predicates that are written in LISP (for a complete description see [11]). A predicate formulates the constraints on the usage of an archetype. A meta-communicating entity specifies constraints on the usage of all archetypes. The weakest constraints are those which are formulated as predicates always yielding the value TRUE. The strongest constraints are those which are formulated as predicates that return true to a unique value in a unique position within the tree.

The prototype consists of two main modules. The Candidate Value Extractor constructs the set of candidate values from the predicates and stores those values in the Candidate Value Base. The Synthesizer is the core of the prototype. It recursively constructs the protocol architecture by using the Candidate Value Base to satisfy the composed predicates and dynamically maintaining a run-time environment. Starting at the root, the following steps are performed for each node to be generated:

- pick a candidate value belonging to a particular syntactic domain and label the node temporarily with that value
- apply the composed predicate on that node
- if composed predicate holds
  then label the node permanently and proceed with next node
  else if there are other candidate values for the same domain,
  then make the candidate value unavailable for this node
  and start with step 1 again
  else make this domain unavailable for this node and start
  with step one for the next domain
- if all domains have been tried unsuccessfully, then no
  specification can be generated

The information explicitly passed to each node in generation is the set of archetype descriptions and the set of archetype subtypes which have been applied along the path prior to that node. Once a leaf has been reached, the control falls back through the recursion levels until a node is reached for which not all of the children have been generated.

3.2. PREDICATE FORMULATION

Each predicate specifies four types of constraints: the constraints on the values of the variable parts of the archetype, the constraints on the archetypes used before this archetype, the constraints on the archetypes used in other sub-trees, and the logical interrelations between those constraints.

All specified predicates are combined by the composer which uses the composition function to logically deduce a single protocol architecture. The composer does this by successively constructing nodes of the Archetype specification tree in a leftmost, depth first fashion. Therefore, predicates essentially express two types of context constraints on the usage of archetypes in the construction of the final specification tree. These are constraints with respect to predecessor nodes (vertical constraints) and constraints with respect to sibling nodes (horizontal constraints).

Vertical constraints are formulated based on a path history (used-at and used-st variables record the archetypes and their subtypes used in the nodes of a path). Figures 3 and 4 illustrate this.

(or (not (member 'PHASES used-at ) )
  (and (member 'PHASES used-at )
    (member 'DATA_TRANSFER used-st )
    constraint ) )

Figure 3: Vertical Constraint Example: restricting sequencing

The LISP function in Figure 3 defines a constraint in the specification (usage) of the data-transfer phase of a protocol. Graphically, this function defines two types of predecessor - successor relationships:

- \# phases(_,data-transfer,_) = phases(_,data-transfer,_)
Horizontal constraints formulations are independent of the sequence of node constructions by the composer. A dependency of the usage of archetypes in sibling nodes is expressed by allowing an explicit evaluation of the composition function on an artificially constructed node. This enables the out-of-sequence determination of the archetype that will be selected in any node. Therefore, any archetypes usage can be constrained dependent on the usage of archetypes in other nodes even though there is no predecessor-successor relationship between the nodes. Figure 5 defines a complete predicate for the PHASES archetype that formulates the horizontal dependency of usage of archetypes in layers and interfaces.

(defun is-wf-pha (node-id st)
  (or (and (member 'PROTOCOL st)
            (comp comp_opr '(is-wf-pha2 is-wf-pha3 node-id
                            (append (isab at '(LOG-_CHAN))
                                    '(PHASES ADDR_MAP STREAM_MAP))
                            (User_INTERFACE NORM_MODE)))
       (not (member 'PROTOCOL st))))

Figure 5: Horizontal Constraint Example: layer interdependency

A PHASES archetype can be used in the PROTOCOL subtree if and only if it is also used in the USER_INTERFACE subtree. The variable comp_opr is bound to the composition function (e.g. logical AND, OR, etc. or any other function returning a boolean value). The function comp is defined below. Figure 6 gives a graphic representation of the predicate of Figure 5.

layers(..., protocol, user_interface)

<table>
<thead>
<tr>
<th>IFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>phase(...)</td>
</tr>
</tbody>
</table>

Figure 6: Horizontal Constraint Example

The composition function itself is easily defined because of LISP's ability to use functions as data. Predicate functions of meta-communicating entities are given as arguments to the composition function which uses the LISP primitive apply to apply the composition principle (simple logical operator or any other type of LISP function) on those predicate functions to construct the composed predicate.

(defun comp (comp_opr pfunc node-id st)
  (setq pfunc (fcreat pfunc '(node-id at st))
        (apply comp_opr pfunc)
  )

Figure 7: Composition Function

4. EXECUTABLE SPECIFICATION

The executable specification tree is identical to the abstract specification tree except for the substitution of archetype specifications by instruction specifications, and the assignment of a data structure to each arc. The primary object of operation of a node is an information unit called signal. A signal is passed from node to node of a path (root to leaf) of the tree. The flow control principle is specified as follows:

- Everything starts at the root.
- Each node operates on a portion of the signal and passes the (possibly modified) signal to the son node(s).
- Each node can emit new signals which are queued at the 'root' (closest predecessor node labeled medium).
- All nodes operate only on node local data structures.
- All nodes can operate in parallel.

The instructions specified in nodes are executed in sequence. Each instruction can be an assignment, a conditional branch, a while loop, a timer manipulation, or a signal creation instruction. All instructions of a node operate only on local data structures. These are variables declared in the node and data structures associated with the outgoing arcs. These data structures contain an arc identification and type, the specification of a pattern associated with the arc, the state indication, and the identification of a follow node.

There are two types of signals that can be generated by the nodes. The EXEC signal is specified directly by means of its information contents and indirectly by means of the path to be traversed by the signal and patterns associated with the arcs in this path. The MARK signal contains a specification of a path and the future markings (states) of the arcs in this path. It is used to change the global state of the tree.

All messages coming from the outside (users, lines, etc.) are transformed into EXEC signals without any path specification. Those EXEC signals traverse the path that is permitted by the arc markings for which a successful pattern matching between the arc pattern and the message can be achieved.

The specification in Figure 8 is similar to the one in Figure 2. The only principal difference is that there isn't any sequencing. Node 3 is the node that performs all the processing necessary for maintaining information about logical channels. It has 257 virtual outgoing arcs that all lead to node 9. Each of the arcs has a data structure associated with it. This data structure contains a state indicator and a pattern. The state indicator is initially set to MM for all virtual subarcs. This means, the subarcs can be traversed by all signals. Incoming units can traverse a subarc if the matching between the subarc's pattern and byte 1 of the incoming unit is successful. The only instruction of node 3 is an explicit setting of the marker of the subarc traversed to MM. This is necessary since each time a subarc is chosen for traversal it is implicitly set to UI. This state indicator disables any further signals from traversing the subarc. The complete syntax of the executable specification language of Archetype can be found in [8].

Node 9 is used to classify the signals into three subgroups, each represented by a subtree: the connection establishment phase subtree, the data transfer phase subtree, and the disconnect phase subtree. Node 11 is used to classify signals into outgoing (send) and incoming (receive). Node 16 distinguishes between data and control units. Node 17 is the LEAF node. It generates a new signal by using the function EXEC. The new signal is queued at the root.
5. AUTOMATIC GENERATION OF PROTOCOL DRIVERS

The translator of the executable specification language of Archetype into C has been developed along the lines of traditional syntax-directed translation. Yet Another Compiler Compiler (YACC) was used to automatically generate Yet Another Network Compiler (YANC). Figure 5 illustrates the development principle.

YANC generates one C function for each node. Passing of signals among these functions is implemented as passing of a pointer to the signal structure. The function representing a node is written in one of five files generated by YANC. The other files contain: node data structures (local variables and arc data structures), the initialization code for node data structures (e.g. patterns, markers, etc.), the calls to initialization functions, node instructions in terms of interfaces to run-time routines, and interfaces to run-time routines.

YANC is generated using YACC. Consequently, its implementation language and the language of the code it generates is C. The first issue in ensuring the portability of YANC was the compatibility of C compilers in the environments chosen for implementation. The tactic followed here was the usage of "least common denominator" C.

The tactic followed in implementing the system interfaces used by YANC is the one of "code isolation". Each YANC environment provides an IPC and a Media library. The standard environment interface provided to YANC generated code consists conceptually of two routines: a read and a write routine. Both routines have only two arguments: an (from/to) address and a pointer to a linked list of buffers. From the perspective of the abstract protocol specification the read routine is invisible, i.e. it is automatic. The write routine addresses are integers whose range is defined at YANC installation time and which are used to name user processes and hardware ports.

YANC has been implemented on UNIX 4.3 and SYSTEM V and has produced protocol drivers for SUN workstations, the TI Explorer, and a VAX 11/780 (for a detailed account see [13]).
6. CONCLUSION

We have introduced a methodology for automating meta-communication for the purpose of designing, specifying, and implementing communication rules at run-time. The main purpose of the meta-communicating communications model is the integration of communications architectures, protocols, and management systems. All currently existing approaches to integration of communications systems are of an ad-hoc nature (see [12]).

We have implemented crucial substeps of the meta-communicating methodology and will demonstrate a complete system in the near future. The composition of partial specifications to a complete specification of a protocol architecture has been successfully implemented and tested [11]. The translator for the Archetype executable specification language has also been implemented and has been used to automatically generate protocol drivers [13]. We are implementing the translator for the Archetype abstract specification language, and will be porting both translators to operating systems other than UNIX.

Each of the steps in the meta-communication methodology is under continuing investigation. On the topmost level we would like to provide a user-friendly way of specifying constraints that represent the knowledge of a meta-communicating entity. This could be accomplished by a knowledge-based system that automatically constructs these constraints from formalized application requirements and environment constraints. A possible approach to this problem is given in [6]. On the level of combining constraints further research is needed to clarify the relationship: composition function $\leftarrow\rightarrow$ predicates. Also, both the abstract and the executable specification language of Archetype have only partially defined interfaces to surrounding environments. These and other issues are subject of currently ongoing research.

In summary, we believe that future communications systems will be based on automatic meta-communication. We have demonstrated the feasibility of the meta-communication approach through our methodology and through implementations.

REFERENCES

APPENDIX A

SYNTAX (ABSTRACT SPECIFICATIONS)

```
<abs-tree> ::= <node>

<node> ::= <medium>|<archetype>|<leaf>

<medium> ::= <empty>|<MEDIUM><medkind>

<medkind> ::= POINT_TO_POINT|BROADCAST|INTERNAL

<leaf> ::= LEAF<refinement>

<archetype> ::= <description>|<refinement>

<description> ::= <structural>|<relational>|<atomic>

<atomic> ::= <empty>|<function><code><abs-tree>

<function> ::= DATA|NEGOTIATE

<code> ::= <empty>|CODE<position><val><errorcnt>

<position> ::= DELIMITER<val>|LENGTH<int-spec>FROM<integer>

| FROM<int-spec>|

<val> ::= UND|<base>|<string>

<base> ::= #o|#h|#b|ascii

<string> ::= <char>|<empty>|CRC-12|CRC-16|CRC-CCITT

<structural> ::= <abstractn>|<ordering>|<classificn>

<abstractn> ::= LAYERS<module><abs-tree>{<module><abs-tree>}

<module> ::= PROTOCOL|INTERFACE|USER_INTERFACE|MEDIUM_INTERFACE

@interface> ::= <empty>|_INTERFACE

<ordering> ::= <sequencing><code><abs-tree>|<pattern>

<pattern> ::= <simple>|<recursive>

<simple> ::= TRANSACTION<CALL><subtree>{<RESPONSE><subtree>}

<recursive> ::= PHASES<phase><char><subtree>{<phase><char><subtree>}

<subtree> ::= <refinement><code><abs-tree>

<subphase> ::= CONNECT|CLEAR|RESTART|RESET|INTERRUPT|DATA_TRANSFER|NEGOTIATION

| NORM_MODE<intgr>|EXC_MODE<intgr>|ASSOCIATION|
```
<sequencing>
  ::= UNIT_SEQ(<window>,[<direction>], [<selective>], [confirm], [refinement])
<window>
  ::= <empty> | WINDOW(<int>)
<selective>
  ::= SELECTIVE | ACCUMULATIVE | SEL_ACC
<confirm>
  ::= POS_ACK | NEG_ACK | PN_ACK
<direction>
  ::= SEND | RECEIVE | SEND_RECEIVE
<classfctnl>
  ::= <cl-by-fct> | <cl-by-srv> | <cl-by-ab> | <cl-by-adr> <abs-tree>
<cl-by-fct>
  ::= COMM_FCT(<direction> <abs-tree> [ <direction> <abs-tree> ])
<cl-by-srv>
  ::= SERVICE(<srv-cl> <abs-tree> [ <srv-cl> <abs-tree> ])
<cl-by-ab>
  ::= NGT(<id>) [ ,MUX(<int-spec>) ] [ ,CAT(<int-spec>) ]
<cl-by-adr>
  ::= INTERPRETATION(<itype> <abs-tree> [ <itype> <abs-tree> ])
<itype>
  ::= DATA | CONTROL
<stream-lev>
  ::= <stream-lev> | <unit-lev>
<id-type>
  ::= SINGLE_ID | DOUBLE_ID | ADDRDEP_ID
<unit-lev>
  ::= ADDRESSING(FROM<code>, TO<code>), SELF<val>)
<relational>
  ::= <unit-map> | <id-map>
<unit-map>
  ::= UNT_MAP(<cat-spec>)
<cat-spec>
  ::= SRV-DEP <int-spec>
<id-map>
  ::= <stream-map> | <entity-map>
<stream-map>
  ::= STREAM_MAP(<mux-spec>)
<mux-spec>
  ::= SRV_DEP | <int-spec>
<entity-map>
  ::= ADDR_MAP(<val>, val)
<int-spec>
  ::= <int> TO <int>
<int>
  ::= <integer> | NGT(<id>)
<id>
  ::= <string>
<refinement>
  ::= <empty> | <causal>
<causal>
  ::= <event> <effect> [ ,<event> <effect> ]
<event>
  ::= TIMEOUT(<int>) | COLLISION | SEQ_ERR | <direction> | WDW_ERR | CNT(<int>) OVF | PROT_ERR | MSG_ERR | <direction> <call-resp> <phase>
<effect>
  ::= COUNT(<int>) | TAKE_ARC(<int>) | NOOP <time> | START <phase> <LAYER_IDX(<int>)
<time>
  ::= TIME(<int>, <int>, <event> [ ,<event> ])
<call-resp>
  ::= CALL | RESPONSE