Binding Network Topologies to Specifications via Pronouns

Carlos Bazilio, Edward H. Haeusler, Markus Endler
Dept. of Informatics
PUC-RJ
{bazilio,hermann,endler}@inf.puc-rio.br

Abstract—This paper presents a high-level specification language (LEP) for designing protocols and distributed algorithms. Its high-level feature is due mainly to pronouns, which are an abstract and homogeneous way of accessing distinct nodes over a network. A couple of examples are given in order to illustrate LEP’s use. Since many different topologies may be used by protocols and distributed algorithms, we use the concept of graph grammars in order to specify those topologies. Moreover, we define the semantics of pronouns through attributes over a structure that we call attribute graph grammar. As a consequence, a user can define a new pronoun and specify its semantics by inserting a new attribute into the grammar. In order to take profit from this topology generation feature, some aspects concerning the generation of minimal models regarding a specification and a query (in a model logic language) are briefly discussed. This discussion intends to set a bound on the size of the generated networks.

Index Terms—specification languages, graph grammars, attribute grammars, protocol specification.

I. INTRODUCTION

Despite the widespread use of communication protocols and distributed algorithms for many sorts of networks and services, to date most protocols are designed and verified in an ad hoc way, and without the use of rigorous techniques. Regarding the design, the specification languages like [1]–[5] have a crucial role, since they are the first step to be taken when adopting a formal method for designing and validation. However, many of the available specification languages still do not provide high-level constructs in order to ease the task of specification.

In this paper we present the specification language LEP, which is a process-based language. It combines the concepts of guarded commands from CCS [6], overload of names from π-calculus [7] and pronouns (adapted from OO concerns [8]). Pronouns can be seen as a general means of referencing a set of elements, making the specification shorter, more legible and precise. For instance, to send a message to a neighbour or to every other node in a not fully connected network, it usually demands a couple of lines of specification code. Using LEP it is as simple as to send a message to a unique node.

Since protocols and distributed algorithms may adopt many distinct topologies, we use the concept of graph grammars [9] in order to generate them. In addition, we define attributes (inherited and synthesized) for each graph’s node that provide information for evaluating pronouns. This constitutes a structure that we call an attribute graph grammar.

Besides adding pronouns to the protocol and distributed algorithm specification, we also provide pronouns to specify temporal properties. The idea behind the use of pronouns in the specification language and temporal properties is to provide a high-level structure that will be combined with a formal method like a model checker in order to validate those specifications. LEP is going to be a front-end that will connect the user to a validation system like a model checker.

The remainder of the paper is organized as follows. Section II presents some basic definitions. Section III informally describes the domain-specific language LEP and its pronouns. Section IV presents a couple of examples that illustrate LEP’s use. Section V presents the semantic specification of pronouns through attribute network grammars. Section VI gives formal and informal insights about limiting the generated network’s size. Section VII comments on some related work. Finally, section VIII contains final remarks and future work.

II. DEFINITIONS

A. Graph Grammars

A graph grammar [9] is a 5-tuple G=(Σn, Σt, Δ, S, R) where the nonterminal node alphabet (Σn), the terminal node alphabet (Σt), and the edge alphabet (Δ) are finite nonempty mutually disjoint sets, S ∈ Σn is the start label, and R is a finite nonempty set of production rules. Each element in R is a quadruple r=(A, D, I, O), where A ∈ Σn, D=(N, φ, ψ) is a connected graph over Σ = Σn ∪ Σt, where N is a finite nonempty set of nodes, φ : N → Σ
is the node labeling function, and \( \psi \subseteq N \times \Delta \times N \) is the edge labeling function, and \( \Delta, I \in N \) is the input node and \( O \in N \) is the output node.

Given a graph whose nodes are labeled, a new graph is derived using one of the rules of the grammar. We start with a graph with a single node whose label is the start symbol \( S \). During a derivation, a node with label \( A \) is replaced by the graph \( D \) in some derivation rule \( (A, D, I, O) \). Every arc originally entering (exiting) the node labeled by \( A \) becomes an arc entering (exiting) the input node \( I \) (output node \( O \)).

B. Attribute Graph Grammars

An attribute graph grammar is a graph grammar decorated with attributes, which are used to calculate the values (the referred nodes in the network) of the pronouns at each node in the network.

Formally written, an attribute graph grammar is a 5-tuple \( G=\left( \Sigma_n, \Sigma_t, \Delta, S, R \right) \) where the nonterminal node alphabet (\( \Sigma_n \)), the terminal node alphabet (\( \Sigma_t \)), and the edge alphabet (\( \Delta \)) are finite nonempty mutually disjoint sets, \( S \in \Sigma_n \) is the start label, and \( R \) is a finite nonempty set of production rules. Each element in \( R \) is a quadruple \( r=(A, D, I, O) \), where \( A \in \Sigma_n \), \( D=(N, \phi, \psi) \) is a connected graph over \( \Sigma = \Sigma_n \cup \Sigma_t \) and \( \Delta, I \in N \) is the input node and \( O \in N \) is the output node. For each \( X \in \Sigma_n \cup \Sigma_t \), there are two finite disjoint sets \( I(X) \) and \( S(X) \) of inherited and synthesized attributes.

C. Pronouns

A pronoun over a graph \( G=\langle N, E \rangle \), where \( N \) is the set of nodes and \( E \) is the set of edges, is a function \( pr: \Id \times N \rightarrow \{N\}^* \), where \( \Id \) is the set of pronoun identifiers. I.e., when a pronoun \( id \) is evaluated in a node \( n \), the result is a set of nodes \( n \), where \( \Id \subseteq \Id \) and \( n \in N \).

III. LEP

LEP is a domain-specific language for the specification of protocols and distributed algorithms. It is a process-based language and combines the concepts of guarded commands from CCS [6], overload of names from \( \pi \)-calculus [7] and pronouns (adapted from OO concerns [8]). Pronouns can be seen as a general means of referencing a set of elements, making the specification shorter, more legible and precise.

A specification in LEP is composed of a topology statement, which says over which network configurations the specification works, and a set of declarations of modules. The topology statement can be the explicit definition of the network, an attribute grammar for defining a new topology or a macro for using pre-defined topologies (rings, stars, trees, sequences, arbitrary and fully connected networks).

**topology is (1 - (2), 2 - (3), 3 - (4), 4 - (1))**; (a)

**Ring \( \Rightarrow t \{ t.id \leftarrow 1 \} \Rightarrow S' \{ S'.id \leftarrow t.id+1 \} \)** (b)

**S' \( \Rightarrow \text{in}(t) \{ t.id \leftarrow S'.id \} \rightarrow \text{out}(S') \} \{ S'.id \leftarrow S'.id+1 \} \)**

**S' \( \Rightarrow \text{in(out(t))} \{ t.id \leftarrow S'.id \} \)**

**topology is Ring (*..5) directed**; (c)

Fig. 1. Topology statements in LEP

Figure 1 presents the three ways of specifying a topology in a specification. Statement (a) explicitly specifies the connections among the nodes in the network. Node labelled 1 has the neighbourhood (nodes directly connected) 2, node 2 has the neighbourhood 3, 4 has 3 and 4 has 1.

Statement (b) specifies an attribute graph grammar for a ring topology with the definition of the labelling attribute \( id \), and must be followed by statement (c), which instantiates the grammar. In the first rule, the first element’s \( id \) label is assigned to 1 and the next element \( (S') \) will receive by inheritance this label plus 1, and so on until the last element is added. Then, assignments between \{ and \} regard attributes updates while elements and oriented arrows (\( \Rightarrow \), \( \rightarrow \), \( \leftarrow \)) outside regard the graph grammar; **Ring**, \( S \) and \( S' \) are non-terminals; \( t \) is a labelled terminal; \( \Rightarrow \) connects a non-terminal with its right-side; **in** and **out** determines input and output nodes of the graph grammar in a rule.

Statement (c) alone assumes the existence of the pre-defined macro **Ring** and specifies rings with less than 5 nodes, which is not so restrictive as statement (a). It uses the reserved word **directed**, which is a topology parameter. The topology’s parameters can be \( \text{unreliable, (un)directed, (un)secure and (dis)connected} \). A topology is **reliable** when links and nodes do not fail, i.e., messages are not lost in the system. **Secure** means that messages are not corrupted. **Undirected** means that links in the network are bi-directional. **Disconnected** means that possibly there are unreachable nodes in the network. **Reliable, undirected, secure and disconnected** are the default values of the topology’s parameters.

Whenever we use statement (c) with a pre-defined topology or (b) combined with (b) for a new topology, we generate instances of the specified topology with the syntax of (a).

A module (figure 2) consists of a set of local variables and a set of transitions. Each transition is composed of a pre-condition (a boolean expression or a receive command) and a possibly empty sequence of actions that are taken when the pre-condition succeeds. The
module <module-name>
<local-variables-declaration>
<guards,> -> <actions,>
...
<guards,> -> <actions,>
endmodule

Fig. 2. Module declaration in LEP

pre-condition can also be either the reserved word init that indicates which actions are taken in the initial state of the process, the reserved word true, which means that its action can be executed whenever possible, or the reserved word else that is executed when none of the previous transitions are executed. When more than one transition is enabled, a non-deterministic choice is made. Execution of the process is a loop over transitions except the init transition. The actions can be commands of assignment, synchronization (send/receive), conditional and loop. Synchronization of processes is done through the operators send "!" and receive "?". start and stop commands launch and kill one or many processes. Other syntactical details about ordinary structures like assignments, conditionals and loops are given through the examples in section [LV].

Regarding the synchronization commands, the communication among processes in a language based on process calculus has the following syntax:

P1: c!x
P2: c?y

where c is the communication channel, x is the message’s identifier and y is the variable used to receive a message. If we want to send the message to several processes, we have to iterate through the set of elements:

P1: for i:=1 to N do c[i]!x

Using LEP, we simply write:

P1: everyone!x

Pronoun everyone also works with partially connected networks, and where broadcast communication happens through the flooding of messages. If this pronoun is used for receiving a message

P1: everyone?x

it behaves like a synchronization point that will receive the message x from every element of the network. Since we may have disconnections, processes that execute this command may wait indefinitely. In order to alleviate it we can use the pronoun any(k) in the receive clause that waits for messages from k hops.

In LEP, pronouns may appear in any place where an element’s identifier can appear. We consider the following pronouns:

this: a reference to the same module instance where it occurs; when used as a right-side value, refers to an internal (not visible to the user) value that identifies this element;

sender: in an action clause of a receive command, this pronoun refers to the element who sent the message; this is useful in a request-reply communication style;

any(t, k): this is a parametric and generic pronoun that refers to any k elements of the system of type t (excluding the element where it occurs); this pronoun adds non-determinism to the specification;

anyother(t, x, k): this is a parametric and generic pronoun that returns any k elements of type t in a network that differ from the given argument x;

everyone(t): in the sending, it refers to every element of type t that can be reached from the element where it occurs; it can be used in broadcast and the corresponding reply messages. In the receiving of a message, it waits for messages of every element in the network;

neighbours(t, k): refers to the set of elements of type t that are reachable in k steps (paths from this node to the target have a maximum of k nodes);

parent: refers to the creator of the element where the pronoun occurs regardless of whether they are directly connected;

children: refers to all of the elements created by the element where the pronoun occurs regardless of whether they are directly connected.

The parameter k when omitted has a default value of 1. The parameter t is also optional and defines the types of elements (module’s identifiers) referred to by the pronoun. When omitted, t has the type of the current module. In protocols for ad hoc networks this parameter is usually omitted.

As we said, LEP is being designed to be used as a front-end for a validation system. Then, we also use pronouns in the specification of the model’s properties. In addition, those properties can be based on the patterns of specification for temporal logics provided in [10]. These properties can be composed of boolean expressions, send commands (sender!msg(receiver)), receive commands (receiver?msg(sender)) and variables. When an identifier that occurs in a formula is not a pre-defined pronoun, it is a variable and unifies over the occurrences of the other variables with the same name, i.e., refers the same node. For instance, we could specify the following property:

Prop1: [] (p!alive(everyone) → <>p?ack(any(k)))

However the interpretation of pronouns can change a little when used in a property. In Prop1 expressed in LTL (Linear Time Logic), it asks whether there always exists a state where a process p sends a message alive to every other process and eventually receives k messages ack as the responses to the alive message. We consider
the following pronouns for specifying properties that can be used as send or receive commands:

class everyone: specifies whether there is some state where a process sends or receives a message from every other process in the network. In the send, everyone means every reachable process. In the receiving, it means every process of the network;

class none: specifies that there are no states where a process sends or receives a given message; p!msg(none) is semantically equivalent to not p!msg or none!msg if we do not care about the agent;

class any(k): the meaning is the same as everyone to a k-size subset of the processes.

IV. EXAMPLES OF SPECIFICATIONS IN LEP

In this section we present a couple of examples that illustrate LEP’s use. First, figure 3 shows how a Leader Election algorithm could be specified in an arbitrary topology.

topology is (1 - (2,3), 2 - (1,3,4), 3 - (1,2,4), 4 - (2,3)) reliable;

module candidate
vars my, p, count : int;
init -> count = 0; my = this; neighbours!msg(my, count);
this?win -> stop;
this?msg(p, count) ->
if ((p > my) or
  ((p == my) and (count < topology.size)))
  my = p; count = count + 1;
  neighbours!msg(my, count);
else
  if ((p == this) and (count > topology.size))
    everyone!win; stop;
endif
endif
endmodule

Fig. 3. Leader Election in a arbitrary network specified in LEP

In figure 3 we have the statement of the topology and the module definition of a Leader Election algorithm implementation specified in LEP. In bold we have the reserved words of LEP and in italics we have the pronouns. If we replace the explicit definition of the reserved words of LEP and in italics we have the implementation specified in LEP. In bold we have the module definition of a Leader Election algorithm specified in LEP.

As properties we could have the following:

[] (anyone!msg -> <> someone!win)

which verifies that whenever a message msg is sent, at any future moment the message win is also sent.

Figure 4 shows the structure of a consensus algorithm, like a coordinated attack with one commander and at most 5 soldiers, specified in LEP. In this specification the commander initially sends a message agree to everyone reachable from him. Then, he waits for at least 3 yes messages, replying with the message consensus, or waits for a no message, replying with a cancel message. The soldiers on their turn, wait for the messages agree, replying non-deterministically the message yes or no, and messages cancel and consensus, replying with nothing by now.

For this algorithm we could have the following properties:

<> anyone?no

which verifies whether a message no is eventually received;

[] anyone?yes(everyone)

which asks whether a process (in this specification, the commander) receives the message yes from everyone;

[] anyone?yes(any(3))

which is less restrictive than the previous property and asks whether at least 3 yes messages are received by the commander.

V. SPECIFYING TOPOLOGIES AND PRONOUNS THROUGH ATTRIBUTE GRAPH GRAMMARS

We use the notation described in table [I] in order to specify a topology in LEP for the attribute graph grammars:

| NG ::= { Nt { Atribs }* ’⇒’ Gr { Atribs }* ’;’ }+
| Gr ::= Gr Oper [ ’(‘Label’)’ ] Gr | Atom
| Atom ::= Term | Nt | ’in’(’Atom’) | ’out’(’Atom’)
| Oper ::= ’−’ | ’−’ | ’→’ | ’⇒’ | ’−’
| Label ::= ID
| Nt ::= ID ∈ { A, .., Z }+
| Term ::= ID ∈ { a, .., z }+

TABLE I

ATTRIBUTE GRAPH GRAMMAR’S NOTATION
In table I \textit{in(Atom)} and \textit{out(Atom)} are input and output nodes of the attribute graph grammar, respectively; \(\rightarrow, \leftarrow, \leftrightarrow\) are infix operators that connect two nodes as an arc from left to right, right to left and both ways, and an arc can be labelled through a \textit{Label}.

Regarding the list of the pre-defined pronouns, \textit{sender} cannot be expressed semantically through attribute network grammars, since it can only be evaluated after the initial configuration, i.e., at runtime. Comments about the use of attribute graph grammars at run-time, i.e., during the validation of the specification, are given in section VIII. Pronouns \textit{any} and \textit{anyother} can be specified by defining an attribute whose content is the set of generated nodes. Then, those pronouns can be a non-deterministic choice over this set. Pronoun \textit{everyone} is usually simulated through flooding over networks not fully connected. This can be seen as a transitive closure of pronoun \textit{neighbours} over the network. Pronouns \textit{parent} and \textit{children} are also evaluated at run-time since LEP allows the dynamic creation and destruction of processes. However, we could apply the following definition in order to define those pronouns:

\[
a \rightarrow b \{ b.paren\rightarrow a, a.children\rightarrow a.children \cup \{b\} \}
\]

We can exemplify the use of this definition in the specification of a star topology (table II).

\[
\begin{align*}
S & \Rightarrow S' \{ S'.children \leftarrow \{ t \} \} \Rightarrow t \{ t.parent \leftarrow S'.this \} \\
S' & \{ S'.1 \leftarrow S'.2 \} \Rightarrow t \{ t.parent \leftarrow S'.this \} \\
\text{inout}(S'2) & \{ S'.2.children \leftarrow S'.1.children \cup \{ t \} \} \Rightarrow \\
S & \{ S'.this \leftarrow t \} \Rightarrow \text{inout}(t) \{ t.children \leftarrow S'.children \}
\end{align*}
\]

\textbf{TABLE II}
\textit{Attribute graph grammar of a star with pronouns parent and children}

Although we are just using the lower-case letter \(t\) to represent any node in the table II nodes can be distinguishable through the use of the attribute \textit{id} given in the figure II. Note also in the previous definition that we specify the pronouns \textit{parent} and \textit{children} using a one-way arc (\(\rightarrow\)). However, in the specification of a star topology (table IV) we use undirected arcs. In fact, this is a great advantage of defining an attribute graph grammar. We could manipulate the semantics of pronouns in any way.

In table I[III we show a way of specifying the pronoun \textit{neighbours} in the ring topology given in the figure I[IV. We define the neighbours of a node through the use of the attributes \textit{s-neigh} and \textit{h-neigh}. The pronoun \textit{neighbours} is given by the attribute \textit{s-neigh} for each generated node.

\[
\begin{align*}
S & \Rightarrow t \{ t.s-neigh \leftarrow S.s-neigh \} \Rightarrow S' \{ S.h-neigh \leftarrow t \} \\
S' & \{ S'.1.s-neigh \leftarrow t \} \Rightarrow \\
\text{inout}(S'2) & \{ S'.2.s-neigh \leftarrow S'.2.h-neigh \} \\
S & \{ S'.s-neigh \leftarrow t \} \Rightarrow \text{inout}(t) \{ t.s-neigh \leftarrow S'.h-neigh \}
\end{align*}
\]

\textbf{TABLE III}
\textit{Attribute graph grammar of a ring topology with pronoun neighbours}

The synthesized attribute \textit{s-neigh} of a node is updated whenever another node is connected to this node. In graph grammar terms, whenever a terminal has an arc to a nonterminal and it can be replaced by a subgraph with input nodes. The inherited attribute \textit{h-neigh} is only important to link the last added node of the ring with the first one. Then, this attribute receives the first node in the first rule (\(S'.h-neigh\leftarrow t\)), and carries it until the application of the last rule, where it is used (\(t.s-neigh\leftarrow S'.h-neigh\)).

The other pre-defined topologies (fully connected, sequence, tree and arbitrary) are defined in a similar way.

\textbf{VI. LIMITING THE GENERATED NETWORK’S SIZE}

Since LEP is supposed to be used as a front-end for a validation system like a model-checker, the size of the networks generated by the attribute graph grammar play an important role due to the model-checkers inherent complexity, not to say intractability, of some models. The use of model-checkers in our context suggests that instead of checking a property against a model, we aim to check the property against a class of models, namely those that have the specified topology. Thus we are interested in checking whether \(M^* = p\), where \(M^*\) is the set of instances of a topology to be analyzed and \(p\) is a model’s property. In order to take care of the generation of those instances, we define the idea of minimal models as a lower-bound of the network’s size and we recall that the Finite Model Property [11] of some temporal logics provides us an upper-bound.

It is out of the scope of this paper to provide a comprehensive presentation of the filtration method to prove the FMP of some modal logics. In this case it suffices to know that the filtration of a model \(M\) relative to a formula \(\phi\) produces a finite model of at most \(2^{|S_I(\phi)|}\) worlds, where \(|S_I(\phi)|\) is the set of subformulae of \(\phi\). Then, for some temporal logics like LTL, CTL, etc, that have the FMP, truth regarding the filtered model is equivalent to truth in the original one.

Regarding the upper-bound, let \(S\) be a specification in LEP, \(\kappa\) the set of classes of models of \(S\), Top a kind of topology adopted by \(S\), \(\mu_{\phi}\) is the filtered model of all of the \(\kappa\) models of \(S\), \(S_I(\phi)\) the set of subformulae

\footnote{A flooding algorithm is an algorithm for distributing material to every part of a connected network.}
of φ, Φ the set of atomic formulae and Pr(k) the set of parameterized pronouns that require k instances of such pronouns:

a) Definitions:
- (i) \( \kappa_S(Top, φ) = \{ μ/μ is the Kripke model of S over the topology Top \} \)
- (ii) \( \mu_f(\kappa_S(Top, φ)) = \{ \mu_f/\mu_f is the filtered model of \mu \in \kappa_S(Top, φ) \} \)
- (iii) About the cardinality of formulae, \( φ \in Φ, |φ| = 1; \phi \subseteq Pr(k), |\phi| = k; \phi \notin Φ, |\phi| = \sum Sf(\phi) \)

With these definitions we can conclude that: \( \mu \in \mu_f(\kappa_S(Top, φ)) \rightarrow |\mu| \leq 2^{Sf(\phi)} \).

Since all of the models belong to \( \kappa \), it happens that some topologies are fully disconnected, empty, with only one node, and many others in \( \kappa \). As \( M^* \models p \) intends to prove a property \( p \) over all of the instances \( M^* \), we can get unexpected results. Moreover, if there are parameterized pronouns in the property, like \( new(\langle k \rangle) \), then instances of \( M^* \) must have at least \( k \) elements. Otherwise, the property would always be false. In order to avoid those tricks we propose the definition of minimal models, which indicates how many nodes a network must have to be able to validate a specification. Then, the generation of the instances of a topology can be constrained by \( |M| \leq |\mu_f|, \forall M \in M^* \).

Regarding the lower-bound, let \( M \) be a LEP module of \( S, Cmds \) a sequence of commands in \( M, P \) a process described by \( M, m \) a message in \( S \) and \( ch \) a message's channel (element's identifier in LEP):

b) Definitions:
- (iv) if \( P \) can execute a receive command \( ch?m \) and \( P' \) can execute the send command \( ch!m \), then \( P[ch?m]/P'(ch!m) \) represents a possible synchronization between \( P \) and \( P' \); a synchronization is seen as a static link of \( S \);
- (v) if there is a transition \( ch?m \rightarrow Cmds \) in a module \( M \) and \( ch!m' \in Cmds \), then \( ch?m \rightarrow ch!m' \) is also a static link;
- (vi) a path \( C \) is a possible sequence of static links;
- (vii) \( C_φ \) is a possible execution path for a property \( φ \) if it is valid at any point of \( C \); \( C_{r_φ} \) is the set of intersections of \( C_φ \) paths;
- (viii) \( S \) is in a state proccesslock regarding a property \( φ \) if \( \exists P \in S, \exists ch?m \in C_{r_φ}/P(ch?m) \land \beta Q \in S/Q(ch!m) \); i.e., \( P \) may wait indefinitely for a message \( m \) since there is no process \( Q \) that can send \( m \);
- (ix) \( \forall P \in S_f(φ), P \in Pr(k), \mu \text{suffices } φ \) if \{ \( p_1, ..., p_k \) \} \subseteq \mu;
- (x) a minimal model \( μ_m \) for a specification \( S \) and a property \( φ \) is the smallest model where the state proccesslock never occurs and \( μ_m \text{suffices } φ \).

Then, based on the definitions above, in order to verify whether a specification \( S \) satisfies a property \( p \) we have just to prove that \( M_s \models p,μ_m(S,p) \leq M_s \leq 2^{S_f(φ)}, \forall M_s = \{ μ/μ is a model of S \} \).

VII. RELATED WORK

SDL (Specification and Description Language) [2] is an ITU-T Recommendation language for telecommunication reactive systems. SDL systems consist of a structure of communicating agents. For each agent we can have a set of instances of such an agent. Representations in SDL can be presented in textual of graphical mode. These can be provided in a hierarchical way and in different levels. Regarding to LEP comparison, SDL has four anonymous variables that can be applied to the agents and are similar to LEP’s pronouns: self refers to the agent instance (pronoun this); parent refers to the creating instance; offspring refers to the most recent creating agent (similar to pronoun neighbours); and sender is exactly the same as pronoun sender in LEP. Comparing those constructions, LEP’s pronouns demand more computational effort (e.g., pronoun everyone could be implemented through flooding), while SDL anonymous variables of SDL agents can be obtained in a straightforward way. Roughly speaking, SDL is a well-know, established and really complete specification language, sometimes close to a programming language like Pascal, while LEP is supposed to be small, simple and with domain-specific constructions in order to be easy to use by protocol and distributed algorithm designers.

In [12] the authors give some insights about modelling considerations and types of ad hoc protocol properties that can be realistically verified. A case study is presented, whose verification is made with model checkers Spin [4] and UPPAAL [13], the latter for the timing requirements of the case study. Regarding our work, in [12] the authors argue that a general proof of correctness should perform all permutations of the topologies for any given number of nodes. It seems that attribute graph grammars are a step towards this direction.

In [1] it is described how the UML (Unified Modeling Language) can be used to describe agent interaction protocols. The basic UML diagram used in those specifications is the Activity Diagram. The example presented is an English Auction, whose diagram is depicted on a figure with two agents: the auctioneer and one bidder. Although clearly described, the diagram probably will increase many times as soon as we add more bidders. It suggests the use of a schema or an abstraction in order to reduce the size of the whole diagram.

In [5] a language called CLAIM is proposed for the designing of multi-agent systems. This language is supported by a multi-platform system called SyMPA, which is also described. Agent’s components of CLAIM
are similar to LEP’s pronouns. Pronouns _neighbours_, _any_ and _anyother_ from LEP are missing in CLAIM. It is not clear how the component _all_ of CLAIM is implemented (everyone is implemented through flooding, using the pronoun _neighbours_), or whether there exists an extension mechanism in order to specify a new component.

VIII. CONCLUSION

From this work we could conclude, as expected, that domain specific languages can greatly improve the task of protocol and distributed algorithm specification. In addition, LEP’s pronouns can really simplify intricate protocol behaviours such as broadcast, multicast and consensus algorithms, both in the specification and description of properties. Although not described here due to the lack of space, the specifications presented were compared with their versions in Promela (specification language of Spin) [4] and SMV [3], which are bigger and less legible.

Regarding specification language usability, LEP’s pronouns save a lot of lines of code when compared with languages like Promela and SMV, which do not have those high-level constructs. Moreover, properties in temporal logics with pronouns are more abstract and concise. In addition, the idea of building LEP using elements from well-known languages for specification and verification gives us an informal proof that it is feasible for specification and execution.

Attribute graph grammars are a helpful formalism for describing the topology of the network and expressing the semantic of the pronouns. It is helpful because, initially, LEP only accepted pre-defined topologies and there was no simple way of defining new pronouns. However, attribute graph grammars were just applied in the initial configuration, since the dynamic behaviour of the network is guided through the specification. Future work could be to use attribute context-sensitive graph grammars, whose rules would be applied during the validation of the specification in order to deal with the dynamic behaviour of a network.

With the aid of LEP we aim to create a high-level front-end for the formal verification of protocols and distributed algorithms. In this front-end we plan to provide a set of high-level constructs like pronouns and pre-defined topologies to encapsulate features commonly required in the specification of such systems.

REFERENCES


