Language-Oriented Formal Analysis: a Case Study on Protocols and Distributed Systems

Carlos Bazilio [1]

Departamento de Informática
PUC-Rio
Rio de Janeiro, Brazil

Edward Hermann Haeusler [2]

Departamento de Informática
PUC-Rio
Rio de Janeiro, Brazil

Markus Endler [3]

Departamento de Informática
PUC-Rio
Rio de Janeiro, Brazil

Abstract

The main motivation of this paper is to describe an architecture that intends to ease the verification of distributed algorithms and protocols (possibly mobile) through model checking. The core of the architecture is the protocol specification language (LEP), which has constructions, called pronouns, that allows for high-level specification. This means a much less verbose specification, when compared with the general-purpose specification language of the model checker used in our experiments. Through a two-step process, LEP specifications are translated into the language of a model checker and the result is translated back to LEP. A formal communication model is used in the translation process in order to allow the use of different model checkers. Currently the prototype of the architecture uses the model checkers Spin and SMV.

Key words: Protocol Specification, Formal Verification, Model Checking.

[1] Email: bazilio@inf.puc-rio.br
[2] Email: hermann@inf.puc-rio.br
[3] Email: endler@inf.puc-rio.br

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Nowadays the task of validating a system is getting harder due to the huge complexity, which is inherent to most of the current systems. These difficulties are even bigger when the systems are distributed or have mobile elements. For those systems, techniques and automated tools that support validation, like simulation, equivalence checking, theorem proving and model checking are even more crucial. The application of above mentioned tools and techniques in the development and validation of systems is called Formal Analysis of the systems. This article focuses on the Formal Analysis of distributed algorithms and protocols. In the sequel we discuss, on a epistemological basis, the essential use of Formal Methods (mainly Model Checking **MC**) as the main extension of software testing.

Language oriented software development (LOSD) is a formal technique for the development of software systems based on the application of tools built from the (formal) semantics of a language designed to be strong enough to describe the specific domain associated with it. Successful and representative examples of LOSD cases are Database systems and Compiler Construction . The former is based on Relational Calculus and SQL variants while the latter is based on formal semantics like SOS, Action Semantics and Denotational Semantics, besides Grammars for the parsers construction. By providing an adequate language and its formal semantics one can automatically develop a debugger or even a visual environment for a programming language. During the 70's and 80's compiler compilers and DBML's were strongly developed.

One of the main advantages advocated by the formal methods community is the correctness of the software developed in this way. The general argument is that from a (correct) specification one derives (automatically) correct running code. We will not follow this line of argumentation. Instead, we argue that the main advantage of using formal methods is that, in general, they are based on a language or languages with precise semantics and most of the times this semantics is well-suited to the specific domain.

We adopt the analogy between the development of scientific theories and software systems which is nicely approached by Haeberer and Maibaum (the reader can read [12] as an example of a quite useful view for software engineers). According the main thoughts in Epistemology and Theory of Science, the validation of a scientific theory is unfeasible. Following Popper [17], Science evolves by means of the conjectures and refutations cycle. This cycle's necessary existence follows from the falsifiability principle that is inherent to Scientific Theories. Concerning its structure, a scientific theory is formed by theoretical and empirical terms, built from, respectively, theoretical and empirical languages that forms the linguistic apparatus of the theory. Carnap [4] has a linguistic analysis of Scientific Theories that is extremely useful when taking the analogy between Scientific Theory development and Software Systems development. Roughly speaking, his analysis concludes that while the

Theoretical Laws of a theory, stated in terms of the theoretical language, are essentially universals, the Observable Experiments are essentially existentials. So to say, the testing of a Theoretical Law by means of experimentation cannot "entails that the hypothesis is true on the basis of the Laws of the Theory". The honest statement is "the experiment did not refute the hypothesis made on the basis of the Laws of the Theory". Formally speaking, Carnap advocates that the analysis of a Scientific Theory is performed in a context with the Laws of the Theory, expressed by the Theoretical Language, a Hypothesis which is formulated with the help of the Empirical (Observable) Language of the Theory, and, with a background theory, that supports the evaluation of the Hypothesis. Carnap's analysis, together with Popper's view of Science, supports the dictum that is feasible to perform the analysis of a theory, aiming its utility, not its truth.

From the above discussed and already established analogy between Theory of Science and Software Development, we advocate that the process of software development should be based on Carnap's linguistic framework aiming to reach the utility of the systems. The correctness of a software is unreachable, according the above analysis. From a naive point of view, unless a formal specification is evidently truth regarding the background theory of the world, there is no way to, ontologically, support the truth of universals (the laws of a particular software) on an existential basis (the possible experiments). The task of formally theorem proving cannot be argue as a way of escaping from this situation, since, the very statements to be proved about the systems are not the whole systems set of properties, that, according Leibniz principle identifies it.

Aiming to provide a platform to perform the (Formal) Analysis of Software Systems that is able to develop useful software, we propose, following the Carnapian linguistic view of science, the Language Oriented Software Development. In fact this approach cannot be stated as a novelty by its own, since it has been applied, before us, in many well-established domains. The main purpose of this article is to show a case study, a language based architecture for distributed algorithms and protocols Specification and Formal Analysis. Our main contribution is a clear statement of this issue and the application of a uniform language to provide semantics for the theoretical as well the empirical terms of the specification. Uniformness is assured by providing smaller semantical gaps between the empirical and the theoretical (sub) languages. This task is particularly interesting with regard to our case study, since the linguistic concept of pronoun will be mapped to universals, from the distributed algorithm/protocol point of view, as well to existential, from a particular network-topology point of view. Other universals, as the time evolving, will be described theoretically as properties (Universals) and empirically as counter-examples. This last case is not different from the formal methods community; however, our architecture will provide an uniform correspondence between specifications universals concepts and the very counter-examples supporting

theirs falsifiability. A mapping from traces (counter-examples), provided by a model-checker or some other formal tool, to the very universal linguistic concept representing the time evolving in the property that produced the trace, induced by the rest of the specification, is provided by the architecture. There is plethora of formal techniques and languages that help SEs to perform (Formal) Analysis. Among the mentioned Formal Analysis techniques, **MC** is an automatic formal method that validates models by exhaustively exploring their computational trees. Those models are finite transition systems and are strongly related to Kripke models. Thus, the properties to be validated are usually expressed using temporal logics. A model checker then typically verifies whether or not a given set of properties holds over a model of the system. It has a complementary role when compared to Theorem Proving. The later aims to establish that a certain property is a logical consequence of the specification of the system/protocol/distributed algorithm. Of course, when such property is not a logical/deductive consequence of the specification the Theorem Prover is of no help at all. We advocate that **MC** is a better tool for Formal Analysis than Theorem Provers, being the later excellent tools for providing certificates of functionality, usually produced after the Formal Analysis phase. Of course one can use **TP**s in order to carry out Formal Analysis, however, the task of refinement of the specification provided by error-finding is harder by means of **TP**s.

Our language based architecture intends to ease the specification and verification of distributed algorithms and protocols, possibly mobile, through model checking. The core of the architecture is the protocol specification language (LEP), which has constructions, called pronouns, that allows for high-level specification. This means a much less verbose specification, when compared with the general-purpose specification language of the model checker used in our experiments. Through a two-step process, LEP specifications are translated into the language of a model checker and the result is translated back to LEP. An intermediate specification, at the level of the communication model, is used in the translation process in order to allow the use of different model checkers. Currently, the prototype of the architecture, in development, has the model checkers Spin [10] and SMV [6] as alternative back-ends. The communication model, formally specified in operational semantics, aims to be general enough to be able to represent any protocol, mobile or not. It is based on a (dynamically) configurable connection graph that represents the relative localization of each functional element of the protocol regarded to the whole net. Each element has an internal behaviour, regulated by a transition system, and, queues to manage any form of communication among the network's elements. This will be better explained in the respective section of the article.

Among the experiments performed with the prototype, DSR (Dynamic Source Routing - a protocol for ad-hoc networks) [11] is presented here. Some aspects such as the size and complexity of the specifications found in the manual specification of this protocol, motivated us to propose the architecture.

The analysis of the experiments performed with the architecture shows, in an evident way, that the use of pronouns as a bridge between the universals of the theoretical language, in the Carnapian sense, and the existentials of the empirical language is a strong linguistic component in achieving a useful platform for distributed algorithms and protocols analysis. It is worth mentioning that the use of Net-Grammar [7] together with filtrations of modal temporal-logic to improve the search-space exploration, in terms of time, proceeding the analysis of the (temporal) properties in a theoretically suitable way. So to say, the verification of a property of a large class of instances of a network-topology is performed in a sound and complete way on a finite set of instances. For reasons of lack of space, this is not detailed in this article, however, it is mentioned with the purpose of pointing out an uniformity issue concerning the case study.

The architecture does not intend to solve the so-called "state explosion" problem of verification of systems by means of the model checking technique. The problem is a hard one. Apart from the fact that the decidability problem for the usual temporal propositional logics used in **MC**, namely CTL, LTL and μ-Calculus, are known to be hard, from PSPACE-complete to EXPTime-Complete, the existence of exponential-size specifications (compared to the size of the valid properties) is strongly connected with the CoNP complexity class, by means of a mapping from classical propositional proofs into temporal properties over transition systems, as well as strongly related to Sat solvers (NP complexity class). Thus, feasibility of general schemas for compositional validation of systems seems to be as hard as to solve the main conjectures about the classes NP, CoNP and PSPace. Of course, it seems that, from the intractability of the temporal logics one could conclude that it is probably an unfeasible task, since the class EXPTime is independent of the main conjectures already mentioned. Thus, our architecture aims a better using of **MC** technique in a broader way, by allowing several tools at the backend, which is of course a feasible task.

2 Proposed Architecture

This architecture can be seen as an additional layer on the top of the usual model checking process. It is supposed to simplify the verification process since its domain-specific language becomes transparent to the user the details of the chosen model checker's specification language.

2.1 General description

Figure 1 depicts the proposed architecture. The top layer consists of LEP (protocol description plus properties to be checked) and the model checker's results at the level of the original LEP specification of the protocol. The bottom layer contains the traditional model checking process. The input of

this layer is the output of the intermediate layer, where code translations are made in order to transform the specification provided in LEP into the specification accepted by the model checker. These translations to the language of the model checker are made by the modules that are specific to the chosen model checker (CI2SMV and CI2Spin in figure 1). The intermediate layer also translates the model checker's result back to the level of LEP. This intermediate layer is important since different model checking tools may be used interchangeably. In out prototype all translator modules were implemented using the TXL transformation language [8], whose description is not on the scope of this text. However, the formal semantic of the translations presented in sections 2.4 and 2.5 defines how it works.

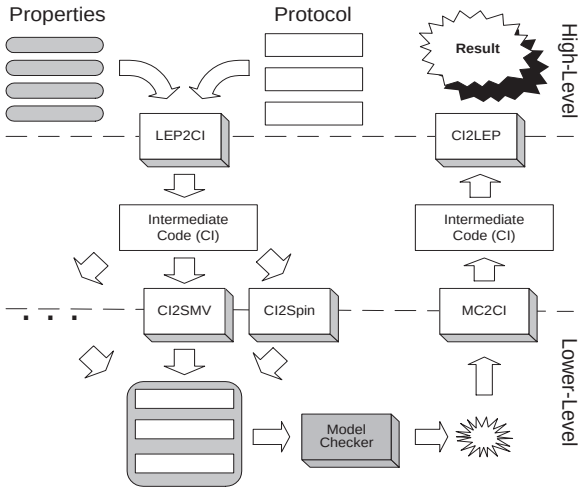

Fig. 1. Architecture's Description

2.2 LEP

LEP is a process-based language such as CCS [14] for the specification of mobile protocols and distributed algorithms. It combines the concepts of guarded commands from CCS, overload of names from Pi-calculus [15] and pronouns (adapted from OO concerns [9]). Pronouns can be seen as a general means of referencing a set of elements, making the specification shorter, more legible and precise. For example, in a ordinary specification language, if we want to send a message to several processes, we have to iterate through the set of elements. Using LEP, we simply write: *everyone!msg*

Pronoun **everyone** also works with partially connected networks, and where broadcast communication happens through the *flooding* [4] of messages. If this pronoun is used for receiving a message (*everyone?x*), it behaves like a synchronization point that will receive the message x from every element

[4] A message is sent through *flooding* when the hosts who receive it, pass to their neighbourhood in order to reach the whole network.

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 right-side value, regards an internal (not visible to the user) value that identifies this element; **sender**: in an action clause of a receive command, this pronoun regards 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 regards 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**: regards the creator of the element where the pronoun occurs; **children**: regards all of the elements created by the element where the pronoun occurs. The parameter k when omitted has a default value equals to 1. The parameter t is also optional and define the types of elements (module's identifiers) regarded by the pronoun. When omitted, t has the type of the current module.

Regarding the topology, we extend the concept of graph grammars [7] by adding attributes that are used to define pronouns. The grammar defines the topology adopted by a specification and the strings generated by the grammar are instances of the topology in the initial state of the validation. Pronouns are initialised through synthesized and inherited attributes, which are calculated during the generation of a network i.e., the application of the production rules of an *attribute graph grammar*. Inserting new attributes into an attribute graph grammar is a way of creating user-defined pronouns.

In table 1 we show a way of specifying a ring topology using attribute graph grammars. Here we define the pronoun *neighbours* through the use of the attributes *s-neigh* and *h-neigh* for synthesized and inherited attributes, respectively. At the end of the process the pronoun *neighbours* is given by the attribute *s-neigh* for each generated node. At this specification: **S** and **S'** are non-terminals; **t** is a terminal; $\Rightarrow$ connects a non-terminal with its right-side; **in** and **out** determines input and output nodes of a graph grammar in a rule; assignments between { and } regards attributes while elements and oriented arrows ($\leftarrow$, $\rightarrow$, $\leftrightarrow$) outside regards the graph grammar.

In the architecture we provide pre-defined grammars for rings, stars, trees, sequences, arbitraries and complete graphs (networks). Moreover, we can specify the initial network explicitly. In order to exemplify a specification in

$\mathbf{S} \Rightarrow \mathbf{t}\ \{\ t.s\text{-}neigh \leftarrow S'.s\text{-}neigh\ \} \leftrightarrow \mathbf{S'}\ \{\ S'.h\text{-}neigh \leftarrow t\ \}$
$\mathbf{S'}_1\ \{\ S'_1.s\text{-}neigh \leftarrow t\ \} \Rightarrow \mathbf{in(t)}\ \{\ t.s\text{-}neigh \leftarrow S'_2.s\text{-}neigh\ \} \rightarrow$
$\mathbf{out(S'}_2\mathbf{)}\ \{\ S'_2.h\text{-}neigh \leftarrow S'_1.h\text{-}neigh\ \}$
$\mathbf{S'}\ \{\ S'.s\text{-}neigh \leftarrow t\ \} \Rightarrow \mathbf{in(out(t))}\ \{\ t.s\text{-}neigh \leftarrow S'.h\text{-}neigh\ \}$

Table 1
Production rules for a ring using attribute graph grammars

LEP, figure 2 shows how a Leader Election algorithm could be specified in an arbitrary topology. In bold we have the reserved words of LEP and in italic we have the pronouns. A specification unit of LEP has a *topology* declaration that defines the initial structure of the network, and the declaration of the modules. The topology can be a pre-defined one or can be given by the user. In the figure 2, node labelled 1 has the neighbourhood (nodes directly connected) 2 and 3, node 2 has 1, 3 and 4, and so on. About the topology's parameters, we can have: *(un)reliable*, *(un)directed*, *(un)secure*, *static* or *dynamic*. 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. *Directed* means that links in the network are bi-directional. When a topology is *dynamic*, the movements of hosts are made automatically and transparent to the user. It is useful when modelling ad hoc networks, since normally mobile hosts move without notice. *Reliable*, *directed*, *secure* and *static* are the default values of the topology's parameters.

```
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))) then
                        my = p;  count = count + 1;
                        neighbours!msg(my, count);
                else
                        if ((p == this) and (count > topology.size)) then
                                everyone!win; stop;
                        endif
                endif
endmodule
```

Fig. 2. Leader Election in a arbitrary network specified in LEP

The module *candidate* defines the state-machine of the nodes. *init* and *stop* marks are the commands that will be executed in the initial and the final states of the module, respectively. Operator "->" defines a transition with the pre-condition before the operator and the action after it. Except *init* transition, which is executed once, in the beginning, the execution of a process is a looping on the module's transitions. The word *true* can also be a pre-condition, which says that its action can be executed whenever possible (non-deterministically).

Synchronization of processes are done through the operators send "!" and receive "?". If we replace the explicit definition of the topology in the figure 2 by "*topology is ring(<6) direct reliable*", the specification still works. It allows the reuse of specifications for distinct topologies, which shows a interesting feature of LEP.

We also use pronouns in the specification of model's properties. Moreover, when a identifier that occurs in a formulae is not a pre-defined pronoun, it unifies over the occurrences of the variables. In addition, send and receive commands can also be used as the subformulae: *[](p!alive(everyone)→ <>p?ack(any(k)))*. However, the interpretation of pronouns can change a little while used in a property. The property above expressed in LTL (Linear Time Logic) asks whether always exists a state where a process p sends a message *alive* to every other process and eventually receives at least k messages *ack*. We consider the following pronouns for specifying properties seen as commands of sending or receiving messages: **everyone**: specifies whether there is some state where a process sends or receives a message from every other process in the network; in the sending, everyone means every reachable process; in the receiving, it means every process of the network; **none**: specifies that there are no states where a process that sends or receives a given message; *p!msg(none)* is semantically equivalent to *not p!msg*; **any(k)**: the meaning is the same as *everyone* to a k-size subset of the processes.

2.3 Communication Model of LEP

In this section we present a formal description of a computation model for distributed algorithms, that is the basis of LEP's execution. In fact, in this part we describe the formal semantics of LEP by means of a translation from LEP into its communication model. For the sake of a brief and meaningful description, we focus on a essential fragment of LEP.

In order to specify the computation model of LEP in SOS [16] we use a structure that we call environment (figure 3). Its structure is a graph where nodes v_i represent processes and edges a_{ij} defines the available connections among processes. Each node is associated to a logical element that contains two buffers b_{in} and b_{out} for input and output exchange of messages, respectively. Channels interconnect these elements in pairs.

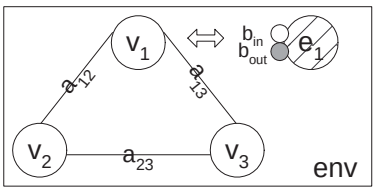

Fig. 3. Environment of the Computation Model of LEP

When sending a message, the sender process puts the message in its output buffer in order to be passed to the input buffer of the receiver process. Buffers'

size and the way of storing messages indicate how the system behaves. Buffer's size equals to zero implies rendezvous communication. Messages are stored in the buffers, which may behave as a queue, stack or even a simple set.

Besides external communication among processes, each process has internal transitions that may change its state. The state of a process is composed by a set of local variables and two buffers for storing input and output messages. These states are important in the specification of properties about the system. We consider the following syntactic categories: $e \in Elem = \{Id \times Buf \times Buf \times Loc \times Trans\}, v \in Vrt = Set\ of\ vertexes, a \in Edge = \{Vrt \times Vrt\}, gr \in Grf = Vrt \cup Edge, ch \in Chan = Set\ of\ channels, env \in Env = Environment.$

The abstract syntax of the computation model of LEP is the following:

$$\frac{v : Vrt}{v : Grf} \qquad \frac{v_1 : Vrt \;\; v_2 : Vrt \;\; a : Edge}{v_1 \xrightarrow{a} v_2 : Grf}$$

$$\frac{gr_1 : Grf \;\; gr_2 : Grf}{gr_1 \;\; gr_2 : Grf} \qquad \frac{gr : Grf \;\; f : Vrt(gr) \rightarrow Elem \;\; g : Chan \rightarrow Edge}{< gr, f, g >: Env}$$

The semantic rules of the computation model of LEP may be described as following:

$$(1)\ \frac{< id_a, b_{in}, b_{out}, \alpha, t > \rightarrow < id_{a\prime}, b_{in}\prime, b_{out}\prime, \alpha\prime, t\prime >}{< gr, f, g > \rightarrow < gr, f\prime, g >}$$

$$(2)\ < id, b_{in}, b_{out}, \alpha, ch!m(val) | t > \rightarrow < id, b_{in}, b_{out}.ch!m(val), \alpha, t >$$

$$< id, ch?m(val) | b_{in}, b_{out}, \alpha, t[x] > \rightarrow < id, b_{in}, b_{out}, \alpha, t[m(val)/x] >$$

$$(3)\ \frac{\begin{array}{c} < id_a, b_{in}, ch!m(val) | b_{out}, \alpha_1, t > \rightarrow < id_{a\prime}, b_{in}, b_{out}, \alpha_1, t > \\ < id_b, ch?x | b_{in_2}, b_{out_2}, \alpha_2, t_2 > \rightarrow < id_{b\prime}, ch?m(val) | b_{in_2}, b_{out_2}, \alpha_2, t_2 > \end{array}}{< gr, f, g > \rightarrow < gr, f\prime, g >}$$

$$(4)\ \frac{< id, b_{in}, b_{out}, \alpha, insert(a, v_2) | t > \rightarrow < id, b_{in}, b_{out}, \alpha, t >}{< gr, f, g > \rightarrow < gr\prime, f, g >}$$

$$(5)\ \frac{< id, b_{in}, b_{out}, \alpha, remove(a) | t > \rightarrow < id, b_{in}, b_{out}, \alpha, t >}{< gr, f, g > \rightarrow < gr\prime, f, g >}$$

$$(6)\ \frac{< id, b_{in}, b_{out}, \alpha, insert(v_1) | t > \rightarrow < id, b_{in}, b_{out}, \alpha, t >}{< gr, f, g > \rightarrow < gr\prime, f\prime, g >}$$

$$(7)\ \frac{< id, b_{in}, b_{out}, \alpha, remove(v_1) | t > \rightarrow < id, b_{in}, b_{out}, \alpha, t >}{< gr, f, g > \rightarrow < gr\prime, f\prime, g >}$$

The rules and the respective conditions and contexts are: (1) Evolution of the system $f(v) = A, f\prime(v) = A\prime, \{\forall v_1, v_1 \neq v, f(v_1) = f\prime(v_1)\}$; (2) Internal transitions $v_1, v_2 \in gr,\ f(v_1) = A,\ f(v_2) = B,\ v_1 \xrightarrow{a} v_2 \in gr,\ a \in g(ch)$; (3) Synchronization rule $\{v_1, v_2\} \subset gr,\ f(v_1) =< id, b_{in}, b_{out}, \alpha, insert(a) | t >$, $v_1 \xrightarrow{a} v_2 \notin gr,\ gr\prime = gr \cup \{v_1 \xrightarrow{a} v_2\}$; (4) Arrow insertion $\{v_1, v_2\} \subset gr,\ f(v_1) =< id, b_{in}, b_{out}, \alpha, remove(a) | t >,\ v_1 \xrightarrow{a} v_2 \in gr,\ gr\prime = gr - \{v_1 \xrightarrow{a} v_2\}$; (5) Arrow removal $f(v) =< id, b_{in}, b_{out}, \alpha, insert(v_1) | t >,\ v \in gr,\ /$

$\exists e, f(v_1) = e,\ v_1 \notin gr,\ f\prime(v_1) = e,\ e \in Elem,\ gr\prime = gr \cup \{v_1\}$; (6) Vertex insertion $\{v, v_1\} \subset gr,\ v \xrightarrow{a} v_1 \in gr,\ f(v) =< id, b_{in}, b_{out}, \alpha, remove(v_1)|t >$, $f\prime(v_1) = \phi,\ gr\prime = gr - \{v \xrightarrow{a} v_1\}$; (7) Vertex removal;

On these rules, operator '|' separates the first and the rest of the messages in a buffer, '.' concatenates a message with a buffer, a t[m(val)/x] is a λ-abstraction that will evolve the state of the element taking into account that a message m(val) was received.

2.4 Translation from LEP into its Computation Model

In this section we present a formal description in SOS [16] of how LEP specifications are translated through rewriting rules to the specification of its computation model. In the translation, we map the pronouns to their respective elements based on the topology and context (place where the pronoun appears) given. Exchange of messages are described only in the intermediate code since it is simpler to specify a one to one communication than a one to many that can be done by the pronouns. Due to the lack of space, we present part of the formal description of LEP. The rest may be described in a similar way. Considering the syntactic categories of the previous section, we can add the following: mid ∈ MId, t ∈ Trans = CExp × Cmd, ce ∈ CExp, m ∈ Msg, cm ∈ Cmd, ch ∈ Chan, top ∈ Top, bool ∈ Bool, connec ∈ MId → { MId }.

In the semantic model, *top* is the topology of the network, *st* is the state of the translation, which contains the current module's identifier plus the topology. Functions f:Pron x MId x Top→{Chan} and g:Chan→ MId x MId provides information about the topology of the network. Messages in LEP are translated to messages with two additional arguments that can be seen as messages' labels: the sender's identifier and a boolean value that indicates whether the message must be forwarded like in pronoun *everyone*. The semantics rules are the following:

$<topology\ is\ connections\ params;\ mod\ prop> \triangleright$ $[\![mod, [\![\text{processTopology(connections,params,mod)}]\!]]\!]$
$<module\ mid\ t\ endmodule> \triangleright$ $< b_{in}, b_{out}, assoc, [\![t, st(mid, top)]\!] >,$ $onde:\ \|b_{in}\| = \|b_{out}\| = 0, \{\forall v\ assoc(v) = 0\}$
$< t_{lep_1}\ t_{lep_2}, st(mid, top) > \triangleright$ $[\![t_{lep_1}, st(mid, top)]\!][\![t_{lep_2}, st(mid, top)]\!]$
$< ce_{lep} \rightarrow cm_{lep}, st(mid, top) > \triangleright$ $[\![ce_{lep}, st(mid, top)]\!]$ -> $[\![\text{generateConditionEveryone}(ce_{lep}, mid, top)]\!]$ $[\![cm_{lep}, st(mid, top)]\!]$
$< cm_{lep_1}; cm_{lep_2}, st(mid, top) > \triangleright$ $[\![cm_{lep_1}, st(mid, top)]\!]\ ;\ [\![cm_{lep_2}, st(mid, top)]\!]$
$< if\ ce\ then\ cm\ endif, st(mid, top) > \triangleright$ $if\ [\![ce, st(mid, top)]\!]\ \{\ [\![cm, st(mid, top)]\!]\ \}$
$< neighbours!m, st(mid, top) > \triangleright$ $local_1 = 0;$ $while\ (local_1 <= k)\ \{$ $ch[local_1]!m(mid, false);$ $local_1 + +;\ \}$ $,\ onde: \text{processPronoun}(neighbours, mid, top) = \{ch[1], ..., ch[k]\},$ $g(ch[1]) = mid_1, g(ch[k]) = mid_k, \{mid_1, mid_k\} \cup MId,$ $k = \|\text{processPronoun}(neighbours, mid, top)\|$

The conditions on the translation rules are respectively: (1) $|b_{in}| = |b_{out}| = 0, \{\forall v\ assoc(v) = 0\}$; (2)none; (3) none; (4) If the network is unreliable i.e., nodes may discard message (5) $g(ch) = (m, mid), \forall m \in MId$ (7) $f(neighbours, mid, top) = \{ch_1, ..., ch_k\}, g(ch_1) = (mid, id_1), g(ch_k) = (mid, id_k) id_{1..k} \in MId, 1 \leq k \leq |f(neighbours, mid, top)|$. Function *processTopology* generates topology information in order to be used in the translation, and function *generateConditionEveryone* generates commands that tests whether the arrived message must be forwarded or it was already received, and forwards according to these conditions.

2.5 Translation from the Computation Model into Promela

In this section we show how the intermediate code of the architecture is mapped into the constructions of Promela (Spin's input language)[10]. The translation is almost straightforward since pronouns were already treated in the translation from LEP to the intermediate code. Again the description is not complete due to the lack of space. Considering the syntactic categories of the previous sections, we have the following rewriting rules:

$<< mid, b_{in}, b_{out}, assoc, t >\ e_2, top, prop > \triangleright$ $[\![$declare-channel-msgs$(mid, t)]\!]$ $[\![$declare-vars$(prop)]\!]$ $[\![$declare-runs$(mid, top)]\!]$ $[\![< mid, b_{in}, b_{out}, assoc, t >, top, prop]\!]$ $[\![e_2, top, prop]\!]$
$<< mid, b_{in}, b_{out}, assoc, t >, top, prop > \triangleright$ $proctype\ mid\ \{$ $[\![$declare-locals$(assoc)]\!]$ $[\![$declare-init$(t)]\!]$ $do\ [\![t, top, prop]\!]\ od\ \}$
$< ce \rightarrow cm, top, prop > \triangleright$ $:: [\![ce]\!] \rightarrow [\![$update-vars$(ce, prop)]\!][\![cm, top, prop]\!]$
$< if\ ce\{cm\}, top, prop > \triangleright$ $if\ :: [\![ce]\!] \rightarrow [\![$update-vars$(ce, prop)]\!][\![cm, top, prop]\!]$ $:: else\ fi$

In these rules, function *declare-channel-msgs* defines the channel of messages and the messages' type exchanged by the given process, which are necessary in the communication of processes in Promela; function *declare-locals* declares the local variables of the process; *declare-init* inserts into the specification the commands that must be executed in beginning of the process and function *increment-count-variables* increments the global variables that will be used by the properties in Spin.

2.6 Translation from the Communication Model into SMV

Similarly to the section 2.5, we present here the translation of the Communication Model to SMV [6]. The translation is not so straightforward as the translation to Promela, since SMV does not have primitives for communication via channels in its basic implementation.

$< e_1\ e_2, top, prop > \triangleright$ $MODULE\ main$ VAR ⟦declare-globals(e_1, e_2, top)⟧ $ASSIGN$ ⟦assign-globals(e_1, e_2, top)⟧ $[\![e_1, top]\!][\![e_2, top]\!]$
$<< mid, b_{in}, b_{out}, assoc, t >, top > \triangleright$ $MODULE$ mod_mid$(mid, processes, matrix)$ VAR ⟦declare-locals$(assoc)$⟧⟦declare-states(t)⟧ $ASSIGN$ ⟦assign-initial-states(t)⟧$[\![t, top]\!]$ $FAIRNESS\ running$
$<< ce \rightarrow id = expr >,\ cm, top, pos > \triangleright$ $next(id) :=$ $case\ order = pos$ & ⟦generate-pre-conds(ce, id, top)⟧ : $[\![expr]\!]$; $esac$; $[\![ce \rightarrow cm, top, inc(pos)]\!]$
$<< ce_1 \rightarrow if(ce_2)\{cm_1\} >,\ cm_2, top, pos > \triangleright$ $[\![[\![ce_1]\!]$ & $[\![ce_2]\!] \rightarrow cm_1, top, pos]\!]$ $[\![ce_1 \rightarrow cm_2, top, inc(pos)]\!]$

In the translation rules, function *declare-main* declares the instances of the network and passes modules id's as parameters among the instances in order to perform the communication; function *declare-locals* declares the local variables like *partner* that stores the identifier of a partner in a message exchange; function *declare-states* declares a variable that stores one of the possible states of the instances based on the ingoing and outgoing arcs of the related node in the intermediate code; *assign-initial-states* assigns the initial state of the variables; and finally function *extract-state-transitions* maps commands of synchronization and assignment to states and state transitions.

3 Example of Use

In this section we describe informally the behaviour of the protocol DSR. In addition we describe how we model this protocol, the assumptions and how it was specified in LEP. For a not valid property over this model, we show the counter-example returned by Spin and how it is converted to another one at the abstract level of LEP.

3.1 Specification of DSR

Dynamic Source Routing [11] is a simple and efficient routing protocol for MANETs (Mobile Ad hoc NETworks). Each package sent across the network carries its route by adding node's identifiers to its header while it visits them. Each node keeps a cache of routes, which are learnt by the packages sent via this node. DSR is composed of two sub-protocols: one for finding out routes and another for maintaining routes.

When a node wishes to send a package to another node, it searches for a route to this destiny in its local cache. If this search succeeds, the package is sent to the first node of the route. This is repeated at every node until the package reaches its destination. If the search fails, the sub-protocol for finding routes is activated. In the cases when a routing error occurs, a route error packet is sent and the nodes which receive it update their cache. This is only the basic version of DSR, and many other optimizations have been proposed [11].

3.2 Modelling DSR

We specify the protocol DSR based on the following assumptions: (i) We assume that all nodes wishing to communicate are willing to participate fully in the protocol; (ii) Nodes do not suffer from interference (e.g. a host receives two distinct messages at the same time, which could cause the loss of the messages); (iii) Nodes within the ad hoc network may move at any time without notice; (iv) Nodes are distinguishable.

In the figure 4 we can see part of a manual DSR specification in LEP. Due to lack of space, we do not present the specification of the whole protocol. For instance, we do not include the cache of routes. Instead each mobile host has to recalculate a route whenever he needs to send a message.

```
1 topology is Arbitrary(7) undirected dynamic;
2 module hop
3           (seq:hop)#int packet;
4           int order=0;
5           true ->       packet#1.clean; packet#1.first = this; packet#1.last = anyother(this);
6                         order = { order, order+1 }; packet#2 = order;
7                         neighbours!rr(packet);
8           this?rr (packet) ->
9                         if this == packet#1.last then
10                                  packet#1.add(this);
11                                  packet#1.previous!unicast(packet);
12                        else
13                                  if not(packet#1.contains(this)) then
14                                            packet#1.add(this);
15                                            neighbours!rr(packet);
16                                  endif
17                        endif
18          this?unicast (packet) ->
19                        if this == packet#1.first then
20                                  packet#1.next!comm(packet);
21                        else
22                                  packet#1.previous!unicast(packet);
23                        endif
24          this?comm (packet) ->
25                        if this <> packet#1.last then
26                                  if neighbours.contains(packet#1.next) then
27                                            packet#1.next!comm(packet);
28                                  else
29                                            sender!packet_error(packet);
30                                            packet.remove(packet.range(this,packet#1.last));
31                                  endif
32                        endif
33          this?packet_error (packet) ->
34 endmodule
```

Fig. 4. Excerpt of DSR specification in LEP

Here we describe some syntactic details of LEP used in the excerpt of the DSR specification (figure 4) which were not discussed previously. The semantics of this specification is discussed in the next paragraph. Since parameter *dynamic* is used in the definition of the topology, the movements of hosts are implicitly and automatically generated. The symbol '#' used in the local declaration (line 3) is a type constructor of LEP, which aggregate other types to create a composed one. Beyond the basic types *int* and *bool*, a type can be a sequence of values (*seq*) or a set of values *set*. For each of those collections we have a set of ordinary functions to treat them. A complete list is presented at [1]. In lines 5-6 a mobile host chooses a partner for communicating non-deterministically. In the specification the symbol '#' works as a selector for a composed type. In line 6 the symbols ({ }) define a non-deterministic choice. Finally, in line 33 the treatment of the receiving of a *packet-error* message is left unspecified. The other commands work as previously described (section 2.2).

About the semantics, the reason for using the parameter *dynamic* in the description of the topology is that mobile hosts can move at any time and at any speed without notice. I.e., these movements are independent of the specification. So, we decide not to do it explicitly. The initial behaviour of a mobile host is non-deterministic since in the DSR protocol a host can send a message whenever he wants. The variable *order* is used to identify the request of routes in order to avoid the duplication of responses because of flooding. Since LEP still does not deal with real-time issues, we model the retransmissions of DSR, which are caused by the timeout of a delayed response to a request, through a non-deterministic choice of the value of the variable *order* (line 6). If we increment this variable, it means a new request. Otherwise, it means a retransmission. The sending commands at lines 7 and 15 mean that a request route message is sent through flooding to the entire network of reachable nodes. In the line 26 a mobile host, after receiving a packet to be transmitted, verifies whether it still is connected with the next mobile host in the packet. If it is not connected, an error message (*packet_error*) is sent to the sender of this packet. The other transitions work as expected.

About DSR we may have the following properties:

Property	*Result*
<>p!rr(q) and []([]q!unicast(none) and q!rr(none))	*true*
[](p!rr)→<> (p!comm)	*true*
[](none!packet_error)	*false*
[](none!unicast(p) U p!comm)	*false*

In the translation to Promela made by the architecture, the dynamic movement of nodes is simulated through non-deterministic changes to a matrix of connections among nodes. This is possible since movements reflect changes in reachability rather than physical position. Because of these changes, the

property *[](none!packet_error)* is false. The property (*[](none!unicast(p) U p!comm)*) is clearly a mistake in the order of formulae.

Properties about a model in Spin regard global variables in the specification. Then, many variables have to be generated at the translation phase in order to specify and verify the properties listed above. It highlights a disadvantage of manually using specification languages less abstract as Promela. About counter-examples, the counter-example of the property *[](none!unicast(p) U p!comm)* returned by Spin as a result of verifying the LEP specification translated to Promela is presented in the figure 5.

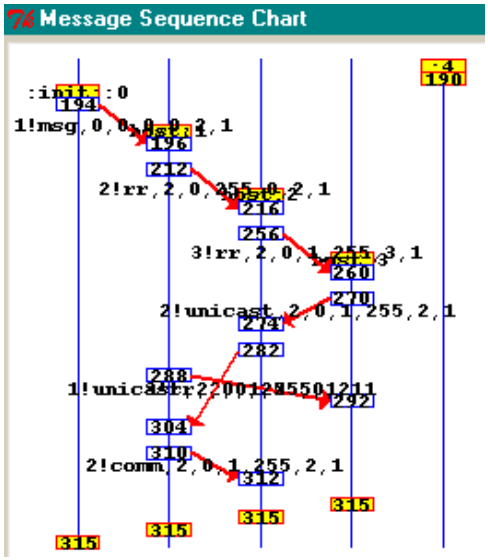

Fig. 5. Counter-example of the property *[](none!unicast(p) U p!comm)* in XSpin

The counter-example that we aim to obtain at the abstract level of LEP for the property *[](none!unicast(p) U p!comm)* is the following:

$$\boxed{\boldsymbol{some_1!unicast(p) \Rightarrow p!comm}}$$

A counter-example returned by the architecture is a sequence of actions $a_1 \Rightarrow a_2 \Rightarrow ... \Rightarrow a_k$, where each a_j is a send/receive command, whose sender/receiver (actors) must occurs in the property to be verified, a_1 is the starting action of the counter-example, meaning that no relevant action was taken before it, and a_k is the ending action, meaning that no action could be taken after it. Any identifier other than the reserved words of the properties (*none*, *everyone*, *any(k)*) is a variable in the counter example. Moreover, like in the properties, variables with the same name regard the same actor or message. For instance, in the counter-example $some_1$ ($some_1$ is a generated variable that regards an actor that not occurs in the property) sends a message *unicast* to actor p and, after some steps, it sends a message *comm*, which is a contradiction to the formula $[]none!unicast(p)Up!comm$.

Although the counter-example presented in figure 5 is not so hard to un-

derstand, since it only contains 3 actors and 7 messages exchanged, if we add 2 more actors for instance, the number of exchanged messages grows up to 37. Regardless of the different initial configurations, the counter-example returned by the architecture is the same in both cases. Then, comparing with counter-examples in MSC (Message Sequence Chart), the counter-example in the level of LEP can be seen as a simplification that contains just those elements that directly regard the property analysed.

In order to provide information for translating the chosen model checker's counter-example into a new one at the abstract level of LEP, we insert into the intermediate code a control variable called *codePositionLEP* that works as a marker. This variable is updated at key points of a LEP specification: at the beginning of each module; at the beginning of a set of transitions; and at the beginning of a transition's action, i.e. after a satisfied pre-condition. Then, when a counter-example is returned at Promela's level, we can check the value of the variable and detect which part of the LEP specification is probably causing the error.

The counter-example returned by the architecture is produced through de following steps: (1) Generate the negation $\bar{\phi}_{Promela}$ of the property $\phi_{Promela}$ to be verified; (2) Search in the counter-example for the state where $\bar{\phi}_{Promela}$ is satisfied. At this point we have the simulated command and the marker (related to LEP); (3) Given the marker and the command at Promela, we look in the original specification for the related LEP command; (4) If the execution of the related command depends upon a pre-condition in LEP, the agent of the pre-condition must also occurs in the counter-example; (5) We repeat the process until we do not find any new pre-condition. Then, the counter-example is the sequence built.

4 Related Work

IF toolbox [3] is a validation platform for timed asynchronous systems. It is built upon an intermediate representation language based on extended timed automata. The toolbox contains dedicated tools on the intermediate language, like compilers, static analysers and model-checkers, as well as front-ends to various specification languages and validation tools. Aside the timed features, our approach differs from the IF toolbox in the input language, since we just accepts LEP specifications for now and IF has various front-ends, the focus of the architecture and the attribute network grammar that adds the capability of validating the specification with many different topologies.

SAL [2] is a framework that combines different tools for abstraction, program analysis, theorem proving, and model checking toward calculating properties of concurrent systems. The core of SAL is its intermediate language for specifying concurrent systems in a compositional way. SAL specification language is closer to a general-purpose model checker's input language. Comparing the input languages, abstraction caused by the use of pronouns in LEP

is the main difference. It also suggests an implementation that replaces our intermediate language by SAL's intermediate language.

In [5] is presented an extension of the framework ASM-WB (Abstract State Machines Workbench) for SMV and MDG. ASM-SL is the specification language and ASM-IL the intermediate one that enables the use of different verification tools. This work is similar to our approach. In both cases there is an input language where models are specified. ASM-SL is based on domains and functions and is more suitable to transition systems, while in LEP process calculus take place. As an advantage, LEP provides pronouns for simplifying the specification. The authors argue that the model checker's result can be parsed in order to generate a higher-level result. However, it is still not implemented.

In [13], the author uses an abstract language called TAP and its computation model in order to talk about assumptions taken in the verification of network protocols. The execution model can be used in translating an abstract protocol specification from this language into C program as a way of turning the specification executable. In our case the generated model checker's models provide the executability of our architecture. In spite of being an abstract language, elements in the network are referred explicitly through their identifiers. In many scenarios in this work we could see how valuable would be the use of pronouns. The idea of translating between specifications is similar to ours. However, we do it in two steps in order to ease the use of different verification tools.

5 Conclusion

From this work we can conclude that domain specific languages can improve greatly the task of mobile protocol specification when comparing with languages designed to general-purpose. In addition, LEP's pronouns can really simplify the specification of intricating mobile protocols behaviours as broadcast, multicast or agreement algorithms. Implementation concerns as the replication of messages that arises when flooding in the translation of the pronoun *everyone* are treated implicitly through the translations to the intermediate representation.

About pronouns, *everyone* interpretation is a little bit tricky since one can imagine that it regards to the whole set of nodes that are reachable from a specific node. However, as this set changes accordingly to the dynamic behaviour of the network, probably a node would wait indefinitely for everyone's response. Furthermore, when using this pronoun in the model's properties for verification, it regards a state of the system where every process has received a specific message rather than a state of a process, which received messages from every process in the system. Another point is that some pronouns are not suitable to every situation. For example, it does not make sense to use a *sender* pronoun in a receive command, since its value is only assigned in the

action associated to a receive command. Despite these details, the advantage of using pronouns seems to be clear.

From the related work, we see that the integration of the proposed architecture with projects that aim the translation among different model checker's input language like SAL [2] is a promising step. We would join the power of translation of those projects with the expressiveness of LEP.

Considering the manual specifications of the mobile protocols in Spin, the whole specification of DSR is about five times greater than the specification in LEP. The specification in SMV is even bigger due to limitations in its input language. It affects readability, maintainability and mainly trustfulness in the specification and in the counter-examples obtained. Nevertheless, since our architecture still do not optimize the generated code, our approach does not attack the state explosion problem.

Another extension will be to allow the input of the specifications in the MSC (*Message Sequence Chart*) format. It is quite desirable since the majority of protocol designers are used to this format. Probably we will face some difficulties like to turn the specification of these charts more precise. In addition, we will have to propose a graphical representation for the LEP pronouns.

About DSR and other protocols analysed, due to smart state exploration of most model checking techniques, it was possible to verify properties without having to instantiate so many objects as found in real life. The specification raised the question about how we can find a minimal model (regarding the number of instances) that is enough to represent the system in the context of the properties to be verified. It would become useless the number of instances given in the topology declaration of the specification in LEP. This is beyond the scope of this paper and will be part of our future work.

References

[1] Bazilio, C. and E. H. Haeusler, *An architecture for the verification of protocolos and distributed algorithms* (2005), www.inf.puc-rio.br/˜bazilio/research.html, PUC-RJ.

[2] Bensalem, S. and et al, *An overview of sal*, in: C. M. Holloway, editor, *LFM 2000*, 2000.
URL citeseer.ist.psu.edu/article/bensalem00overview.html

[3] Bozga, M., S. Graf and L. Mounier, *IF-2.0: A validation environment for component-based real-time systems*, in: *CAV'02, Copenhagen*, number 2404 in LNCS (2002).

[4] Carnap, R., "An Introduction to the Philosophy of Science," Ed.Martin Gardner, Dover Publications, Inc, 1995.

[5] Castillo, G. D. and K. Winter, *Model checking support for the asm high-level language*, in: *Tools and Algorithms for Construction and Analysis of Systems*,

2000.
URL citeseer.ist.psu.edu/delcastillo00model.html

[6] Cimatti, A., E. M. Clarke, F. Giunchiglia and M. Roveri, *NUSMV: A new symbolic model checker*, Int. Journal on Software Tools for Technology Transfer **2** (2000).

[7] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," The MIT Press, 2000.

[8] Cordy, J., *Txl - a language for programming language tools and applications*, in: *Proc. LDTA 2004*, 2004.

[9] Cruz, S. O., C. J. P. Lucena and J. L. M. Rangel, *Identifying objects through pronouns* (2001), monographs in Computer Science; 39/01, PUC-RJ, 2001.

[10] Holzmann, G. J., *The model checker SPIN*, Software Engineering **23** (1997), pp. 279–295.

[11] Johnson, D. B., D. A. Maltz and Y.-C. Hu, *The dynamic source routing protocol for mobile ad hoc networks (dsr)* (2003), iETF MANET Working Group.

[12] Maibaum, T. S. E. and A. M. Haeberer, *Scientific rigour, an answer to a pragmatic question: A linguistic framework for software engineering*, in: *ICSE 2001*, 2001.

[13] McGuire, T. M., *Correct implementation of network protocols* (2004), ph.D. thesis, The University of Texas at Austin.

[14] Milner, R., "Communication and Concurrency," Int. Series in Computer Science. Prentice Hall, 1989.

[15] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, parts i and ii*, Information and Computation **100(1)** (1992), pp. 1–77.

[16] Plotkin, G. D., *A Structural Approach to Operational Semantics*, Technical Report DAIMI FN-19, University of Aarhus (1981).
URL citeseer.ist.psu.edu/plotkin81structural.html

[17] Popper, K., "Conjectures and Refutations," Routledge and Kegan. Paul Limited, 1963.