Issues Arising in the Analysis of L.0

Linda Ness
Bellcore, Morristown, NJ
linda@bellcore.com

1. Introduction

L.0 is a programming language, which is currently being experimentally used at Bellcore as an executable specification language for communications software. It's design was driven by the special nature of communications systems. The most fundamental aspect of communications systems is that many fairly simple things are happening at the same time, for all time. It is precisely this aspect that is so difficult to specify in languages based on a sequential or asynchronous model. It is easy to specify this in L.0 because L.0 has an abstract concurrent-read-common-concurrent-write shared memory model, a notion of *next*, and because the basic composition operator is *conjunction*. Another fundamental aspect of communications systems is that many of the things happening simultaneously are very similar. Thus, another basic feature of L.0 is quantification, which permits parametrized specifications.

The semantics of the language given originally[1] was a combination of declarative and operational. Now that there is evidence that it is fairly easy to specify communications software in L.0, it is important to try to develop analysis tools. For this it is preferable to have declarative versions of the semantics given in several different standard paradigms so that analysis tools or algorithms previously developed might be applied to L.0.

One goal of this paper is to present the fundamental semantic constructs of L.0 in the framework of *linear predicate temporal logic*. Temporal logic was chosen because there is a fairly natural fit, and because there has been a great deal of work on its application to the specification and verification of reactive systems[2] [3]. Most of communications software can be viewed as a reactive system. The algorithm for translating L.0 syntax into the constructs given here is not given in this brief paper. It is however given in[4]. Neither is the focus of this paper is not on examples. They may be found in a number of papers[1] [5] [6] [7].

The main reason, that this presentation of the semantics does not immediately lead to the applicability of the various decision procedures and model checkers developed for temporal logic, is that L.0 is based on a subset of predicate (not propositional) temporal logic, and hence permits restricted forms of quantification. Thus, the second goal of this paper is to indicate the uses of quantification in L.0. The simple approach to developing some analysis tools for L.0, is to simply develop them for programs, in which the quantification is restricted to be finite, and the state spaces constructed during execution, are restricted to be contained in a finite universe. This approach is very unsatisfying because quantification is one of the main reasons that L.0 is so expressive. Furthermore, even if this restriction is made, the size of the finite universe, arising in applications of interest is likely to be huge, relative to the size that can be handled by existing tools. Kurshan[8] has developed a formal notion of homomorphism as one approach to handling this problem. An approach would be to develop tools to perform more restricted analysis which would work in presence of less stringents restrictions on quantification.

There has been quite a bit of work on the identification of executable subsets of temporal logic, and the development of executable temporal logic languages, i.e. languages in which a program is a temporal logic formula, and whose execution constructs a model for that formula. The subset of temporal logic exploited by L.0 is very similar to that identified by Dov Gabbay[9] and developed in MetateM. [10] The languages Tempura[11], Lustre[12] and Artic[13] can each be viewed as executable temporal logic languages. Tempura is based on interval temporal logic, rather than propositional, and uses quantification in a more limited way. Unlike L.0 it does not assume any solution to the frame problem. It has been used primarily to specify hardware. The semantics of Lustre can be given in terms of linear temporal logic. However, it appears that quantification is more restricted, since boolean Lustre may be compiled into a finite automaton. Arctic was designed for the specification and implementation of real-time computer systems, and has been applied to music. Its primary data-type is a real-valued function of time.

2. The fundamental constructs of L.0 in terms of temporal logic

Recall that a predicate temporal logic formula is constructed from predicates using the standard boolean operators, the temporal future operators *next* and *until* and universal and existential *quantification.* In addition, temporal past operators may be used, although it has been shown that the past operators do not add expressiveness[14]. The only temporal past operator that will be used here is *previous.*

Here the *until* used will be the weak *until.* In other words it is not required that the second argument of *until* is eventually true on a state. Precisely,

$S_i \models f$ *until* g if

$S_j \models f$ for all $j \geq i$
or (*opp*$E\, k \geq i$ such that $S_k \models g$ and $S_j \models f$ for all j such that $i \leq j < k$)

The temporal future operators *always* and *eventually* may be defined in terms of *until*, using the predicate *false* and negation.

2.1 Predicates on history and on extended history

These are the simplest components of L.0 programs. A *predicate on extended history* is inductively defined to be either:

1. *a predicate or*
2. *a boolean combination of predicates on extended history or*
3. *the past temporal operator previous applied to a predicate on extended history or*
4. *a universally quantified formula of the form* $r \Rightarrow f$, *where r is a restricting predicate and f is a predicate on extended history or*

5. *an existentially quantified formula of the form r andsign f, where r is a restricting predicate and f is a predicate on extended history.*

A *predicate on history* has the form *previous p*, where *p* is a predicate on extended history. The *restricting predicates* are a subset of the predicates on history. They may be used to ensure the the quantification is finite but unbounded.

The "truth" of a predicate on history on a state in a model, may be deduced by examining only the interpretations for predicates in the previous states, i.e. the *history*. For predicates on extended history, the interpretation for predicates in the current state, may need to be examined as well. In other words, no future interpretations need to be examined. This is the key to executability, since the future states of the model need not have been constructed yet.

2.2 Cause-effect formulas

The second fundamental building block of L.0 programs is the cause-effect formula. Causality is specified using formulas of the form:

cause => effect

where the *cause* is restricted to be a *predicate on history*, and the effect is any L.0 program. Such a formula is true on a state in a sequence, if either the cause was not true on the history, or the cause was true on the history and the effect is true on the current state. If general formulas were allowed for the cause, the intuition would not correspond to causality, for the truth of the cause might depend on the current and future states.

2.3 A restricted class of until formulas

A restricted class of until formulas is used to specify that several cause-effect formulas apply until the first occurrence of at least one of a set of events, and to also specify the effects of some of those events. Thus, the only until formulas allowed have the form:

- *the binary until operator, applied to a conjunction of cause-effect formulas and a deactivator formula*

where a *deactivator formula d* either has form:

cause(d) and effects(d)

where

$cause(d) = (c_0 \text{ or } c_1 \text{ or } ... \text{ or } c_n)$

and

$effects(d) = andsign_{i:\ i\ member\ S}\ (c_i => e_i)$

Here S *subset* $\{1, ...,n\}$, each of the conjuncts of *effects(d)* is a cause-effect formula, and *cause(d)* is a predicate on history. If none of the events in $cause(d)$ have specified effects, the deactivator specification has the simpler form.

$d = cause(d)$

2.4 Universally quantified cause-effect formulas

Universal first order quantification of cause-effect formulas is allowed, providing that the cause contains a conjunct which is a restricting predicate.

2.5 Names of formulas

Formulas may be referred to by name, so that formulas may be structured modularly. Definitions of formula names may be recursive, providing the recursive reference is "over time". This can be expressed in temporal logic using second order quantification.

3. An inductive definition of temporal formulas permitted in L.0

The temporal formulas permitted in L.0 programs are either:

- a predicate
- a conjunction of L.0 programs
- a cause-effect formula where the cause is a predicate on history and the effect is an L.0 program
- a binary until operator, applied to a conjunction of cause-effect formulas and a deactivator formula[1]

- the next operator applied to an L.0 program
- a universally quantified cause-effect formula, where the cause contains a conjunct which is a restricting predicate
- the name of an L.0 program

The original until operator in L.0 had a slightly different semantics in the case of nested untils. However, as in proven in[4], there is an algorithm for mapping these non-standard until formulas to the until formulas of temporal logic, which applies in all reasonable cases.

4. Only safety properties are expressible in L.0

It is not possible to express *eventually* in this subset of temporal logic, because negation of programs is not permitted. The omission of eventually, means that it is not possible to express liveness properties in L.0. However, this omission is not as alarming as it first may seem, for it is possible in L.0 to express pseudo-random sequences, using predicates. This is probably more practical than specifying sequences via the *eventually* operator. Also, the notion of fairness, for which *eventually* is crucial, is not essential to L.0, because L.0 does not base its semantics on the interleaving of atomic events. Instead, the semantics of L.0 is fundamentally the semantics of temporal logic, which is synchronous, and which permits simultaneous occurrence of "events".

4.1 An execution strategy for basic L.0 programs

The subset of temporal logic, consisting of basic L.0 programs is executable, in the sense that there is an algorithm for inductively determining from a basic L.0 program, the obligations it imposes on the current state, and the obligation it imposes on the subsequent sequence of states. This strategy extends to an execution algorithm for general L.0 programs. The decomposition of a basic L.0 program into current obligations and future obligations uses a recursive semantic tautology for basic L.0 until formulas. In fact, an operational semantics for L.0 can be given in terms of a dynamically changing rule set[15].

A fundamental characteristic of the execution strategy is that history, once constructed will never be altered. The key to this it that cause formulas must be predicates on history.

An L.0 program may impose several obligations on the current state, in the sense that a conjunction of predicates must be true on the current state. It is in this sense that L.0 exploits the synchronous semantics of temporal logic. If these obligations conflict, execution of the L.0 program halts. Since there is more than one possible state which satisfies the current obligations, the execution strategy is intrinsically non-deterministic.

5. The current restrictions on the data domain, expressions, and states

Currently the data domain is restricted to be the set of all trees with labeled edges, with the property that all of the child edges of a node have unique labels. The labels are restricted to be from the alphabet of strings of aschii characters. Such a tree is called a namespace. States in L.0 are restricted to be namespaces. Each non-root node of a namespace has a *name*, which is the sequence of labels along the path from the root to the node. Because of the restriction on namespaces, the name of a non-root node in a namespace uniquely identifies it. In general, a *names* is a sequence of labels. Note that namespaces may be equivalently characterized as prefix-closed sets of names. The *suffix-value* of a name on a namespace is the set of strings prefixed by that name in the namespace.

An expression in L.0 programs is either a *name*, a function symbol applied to expressions, a concatenation or union of expressions, or the *suffix value of a name*. The *suffix value* of a name on a namespace is just the set of strings prefixed by the name in that namespace. Geometrically, if the namespace is finite, the value is the tree with labeled edges rooted at the name. The suffix value expression permits simulation of standard variable-value programming, within the more general declarative paradigm. Note that indirection is possible because the suffix value of a namespace is a namespace(which might be a name).

The value of an expression in an L.0 program is a namespace. The algorithm used to interpret expressions is the expected one. The expressions in an L.0 program are only used as arguments of predicates or functions.

6. A restriction on the interpretations of predicate symbols

In an L.0 model, each predicate symbol must be assigned a *predicate definition*. The simplest type of definition is for a predicate symbol of arity 0. Then the definition is either *undefined* or consists of a single *set-theoretic predicate*, which consists of two sets: a *domain*, which is a set of names, and a set of *solutions*, each of which is a subset of the domain[15]. Such a predicate is interpreted as being true on a state, if the intersection of the domain, with the state, is one of the solutions. This is equivalent to a very strong disjunctive normal form, where each element of the domain or its negation occurs in each disjunct. Negation is interpreted as non-existence. There is no restriction that the domain must be finite, here. (With this set-theoretic interpretation, finding a solution of a conjunction of predicates, can be interpreted as finding a global section of a sheaf.)

The definition of an n-ary predicate symbol p, is a mapping from the n-fold cartesian product of *Namespaces(Labels)* to the set of such set-theoretic predicates *cup undefined*.

6.1 Restrictions on the set of predicate symbols permitted

Currently, the predicate definitions allowed are equality of expressions, existence and non-existence of a name, *true* and *false*, and membership in a set of consecutive integers, or in an explicitly given set of names. Furthermore, only limited kinds of equality may be used in effects, and even that in a restricted way: equality of the suffix values of two names, and equality of the suffix value of a name and an expression.

Restricting predicates must contain a conjunction which is either a set membership predicate, or an exists predicate.

Users may provide interpretations for function symbols in "C".

7. A frame assumption

A model for a temporal formula consists of a data domain, an interpretation for expressions, an interpretation for predicate symbols, and a sequence of states. Each of these have been restricted. However, there is one final restriction imposed on a sequence of states, that is to be a model for an L.0 program. This restriction is that successive states are related by one of the replacement rules determined by the predicate, which is the conjunction of predicates, which are to be true on the next state, i.e. the current program obligations. The point is that each predicate (when specific values are supplied for its arguments), has an associated domain and set of solutions. Each solution determines a replacement rule for namespaces, namely replace the intersection of the domain with the namespace by one of its solutions. For a restricted set of predicates (such as are currently in L.0), the result is again prefix-closed, and hence a namespace.

This is nothing but a generalization of the usual frame assumption in sequential programming: namely replace the value of the variable by the new value being assigned. However, this generalization permits the size of the state space to be dynamic. For if the intersection of the domain of a predicate with the previous state space is empty, the set of names making up a solution are added. Conversely, if the intersection of the domain of a predicate with the previous state space is "too large" to be a solution, some of the names are removed to obtain a solution. One of the conveniences in using L.0 as a specification language lies is this concise dynamic allocation and deallocation of "space". To apply analysis algorithms developed for logical formulas, it would be necessary to be able to explicitly specify the frame within an L.0 program, and then consider only the class of programs which explicitly specified it. Fortunately, it appears that this can be done easily, and does not restrict the class of programs, because of the restriction that each predicate must have a specified domain.

8. Uses of quantification in L.0

Universal quantification is extremely powerful. For, as is shown in[4] it permits specification of parameter passing by value, indirection, and the analogue of a set of simultaneous "calls" to the same "procedure". When all three of these uses are combined, one can program in a table driven manner. Thus it is easy to specify a generic non-deterministic finite state machine[1], request handlers that are able to handle a finite but unbounded set of requests each time[16] and specify reconstruction of predicates of particular restricted types[17] In fact using universal quantification, one can easily specify SIMD parallelism.

As seems to be well-known[11], encapsulation can be added by adding existential quantification. Finally, since L.0 permits equality of names, pass by reference can be specified using equality of names and existential quantification. Thus, L.0 augmented by existential quantification, provides permits programmers to program, in a well-structured manner, in temporal logic.

9. Remarks about conjunction and equality

Conjunction permits functional decomposition of specifications. The communication is via shared variables. It seems that well structured L.0 programs are based on a clearly articulated dynamic read-write protocol, among functional components. This permits writing of observer programs which may, for example, filter data, watch for bugs, or write to the screen, to animate the program. Conjunction also makes programs exponentially shorter.

It is interesting to note that when programs are written using encapsulation, it seems possible to restrict the use of equality of names to equality between names local to the parent and child modules. Thus, apart from parameter passing, the uses of equality that seem necessary are standard assignment, and one-way derivations, which deduce that the suffix-value of a name can be the value of a function applied to arguments, which may refer to other current suffix-values.

10. Acknowledgments

L.0 was developed primarily through the joint efforts of Jane Cameron, David Cohen, B. Gopinath, and the author. Prem Uppaluru and Diane Sonnenwald also made contributions to the language, as did a number of the users. B. Gopinath was also the head of the IC* project, during the period when the first version of L.0 was developed. The language implementation was done by David Cohen and Bill Keese (on a sequential machine). The most recent debugger was done by Tim Guinther.

A connection between L.0 and temporal logic has also been recognized by Bob Kurshan, Fred Schneider, Ambuj Singh, and Prem Uppaluru.

REFERENCES

1. E.J. Cameron, D.M. Cohen, L.A. Ness, H.N. Srinidhi, "L.0: A Language for Modeling and Prototyping Communications Software",(to appear in *Proceedings of the Third International Conference on Formal Description Techniques*, Madrid, November 5-8, 1990.)

2. A. Pnueli, "The Temporal Logic of Programs", *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*(1977) pp. 46-57.

3. A. Pnueli, "The Temporal Logic of Programs", *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*(1977) pp. 46-57.

4. L. Ness, "L.O: A Parallel Executable Temporal Logic Language", Bellcore Public Released TM-ARH-014974 September, 1989.

5. E. J. Cameron, N. H. Petschenik, L. Ruston, S. Shah, H. Srinidhi, "From Description to Simulation to Architecture: An Approach to Service-Driven System development", *Proceedings of the First International Conference on Systems Integration*, Morristown, N.J. April 23-26, 1990.

6. D. M. Cohen, T. M. Guinther, L. Ness, "Rapid Prototyping of a Communication Protocol Using a New Parallel Language", *Proceedings of the First International Conference on Systems Integration*, Morristown, N.J. April 23-26, 1990.

7. S. Aggarwal, F.S. Dworak, and P.Obenour, "An Environment for Studying Switching System Software Architecture", *Proceedings of IEEE Global Telecommunications Conference*, 1988.

8. Kurshan, R.P., "Reducibility in Analysis of Coordination", *Discrete Event Systems: Models and Applications*, LNCIS 103(1987), pp. 19-39.

9. D. Gabbay, "Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems", in A. Galton, editor, In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification*, Altrincham, 1987, pages 402-450. Springer-Verlag, LNCS Volume 398, 1989.

10. H. Barringer, M. Fisher, D. Gabbay, G. Gough, R. Owens, "MetateM: A Framework for Programming in Temporal Logic".

11. B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge. 1987.

12. D. Pilaud, N. Halbwachs, "From a synchronous declarative language to a temporal logic dealing with multiform time", *Proc. Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Warwick, Sept 88.

13. R. Dannenberg, "Arctic: Functional Programming for Real-Time Systems", *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, 1986.

14. O. Lichtenstein, A. Pnueli, L. Zuck, "The Glory of the Past", *Proc. Conf. on Logics of Programs*, Springer-Verlag LNCS #193, 1985, pp. 196-218.

15. E.J.Cameron, D.M.Cohen, B.Gopinath, L.Ness, W.M.Keese, P.Uppaluru, J.R.Vollaro, "The IC* Model of Parallel Computation and Programming Environment," *IEEE Transactions on Software Engineering*, Vol. 14, No 3, March 1988, pp. 317-327.

16. E.J. Cameron, D.M. Cohen, B. Gopinath, L. Ness, "IC*: An Environment for Designing Communications Software", *Proceedings of SETSS '90 7th Int'l Conference on Software Engineering for Telecommunication Switch Systems*, Bournemouth, England, July 3-6, 1989.

17. E.J. Cameron, L. Ness, A. Sheth, "A Universal Executor for Flexible Transactions Which Permits Maximal Parallelism".