Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Local Model Checking and Protocol Analysis*

Xiaoqun Du, Scott A. Smolka, Rance Cleaveland

Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400, USA

Abstract. This paper describes a local model-checking algorithm for the alternation-free fragment of the modal mu-calculus that has been implemented in the Concurrency Factory and discusses its application to the analysis of a real-time communications protocol. The protocol considered is *RETHER*, a software-based, real-time Ethernet protocol developed at SUNY at Stony Brook. Its purpose is to provide guaranteed bandwidth and deterministic, periodic network access to multimedia applications over commodity Ethernet hardware.

Our model-checking results show that (for a particular network configuration) *RETHER* makes good on its bandwidth guarantees to real-time nodes without exposing non-real-time nodes to the possibility of starvation. Our data also indicate that, in many cases, the state-exploration overhead of the local model checker is significantly smaller than the total amount that would result from a global analysis of the protocol.

In the course of specifying and verifying *RETHER*, we also identified an alternative design of the protocol that warranted further study due to its potentially smaller run-time overhead. Again, using local model checking, we showed that this alternative design also possesses the properties of interest. This observation points up one of the often-overlooked benefits of formal verification: by forcing designers to understand their designs rigorously and abstractly, these techniques often enable the designers to uncover interesting design alternatives.

Key words: model checking – modal mu-calculus – protocol verification – state explosion – real-time

* Research supported in part by NSF grants CCR-9505562 and CCR-9705998, and AFOSR grants F49620-95-1-0508 and F49620-96-1-0087.

1 Introduction

The term *model checking* [4, 17, 5] refers to a verification technique aimed at determining whether a system specification possesses a property expressed as a temporal logic formula. When the system in question contains a finite number of states, model checking typically becomes decidable, meaning that in principle, it can be done in a fully automatic fashion. Such fully automated model checkers have enjoyed increasing success in establishing the correctness of, and finding design errors in, real-life systems. An interesting account of a number of these success stories can be found in [6].

Model-checking algorithms rely on state-space exploration in order to determine if a system satisfies a temporal formula. Typically, temporal-logic-oriented formalisms view systems as transition systems in which all the possible states and transition steps of a system are represented, together with the initial state(s). The semantics of a temporal logic then defines when a state in a transition system satisfies a formula: the definition is usually recursive, meaning that determining whether or not a state satisfies a formula may require examining other states and other formulas. Thus, although a system satisfies a formula if its initial state(s) do, determining whether or not this is so will typically necessitate the analysis of other states in the system as well as other formulas.

Procedures for model checking may be classified as either *global* or *local* (sometimes also called *on-the-fly*), depending on how the structure of the formula is used to guide the construction of the state space. Global algorithms follow a "bottom-up" approach to analyzing formulas: starting with the smallest subformulas in a given formula, one calculates all the states satisfying a given formula and then uses the results of this analysis to compute the set of all states satisfying successively larger subformulas. This procedure returns the set of all the states satisfying the formula in question; one then checks whether the initial states lie within this set in order to determine if a system is correct. Because of this bottom-up strategy, global procedures require the *a pri-*

ori generation of all system states, and storing these states may consume large quantities of memory owing to the *state explosion problem*. In contrast, local model checkers [13, 20, 14] proceed via a "top-down" examination of the formula in question. In order to determine if a given state satisfies a given formula, such procedures generate subgoals involving states and subformulas that must hold in order for the initial goal to be true. As there is no need for the state space to be known in advance, it can be constructed incrementally, as the model-checking computation proceeds. An advantage of local model checking is that pruning is often possible: how much of the state space actually has to be explored depends on how much of it turns out to be relevant to establishing satisfaction of the formula to be verified. So although in the worst case local algorithms will exhibit the same storage requirements as those of global algorithms, in practice they generally use less.

This paper describes a local model-checking approach for a specific temporal logic, the (alternation-free) modal mu-calculus, and discusses a case study in which an implementation of the algorithm was used to analyze the recently developed *RETHER* (Real-time Ethernet) protocol [2], a protocol that provides real-time service guarantees while running on top of commodity Ethernet hardware. *RETHER* forms a key component of the implementation of the Stony Brook Video Server [3].

The rest of the presentation develops along the following lines. Part I contains sections describing the syntax and semantics of the alternation-free modal mu-calculus and our local model-checking algorithm. Part II then provides a detailed description of our use of the local model checker to analyze two different versions of *RETHER*. The analysis is conducted using the Concurrency Factory [7] verification platform, which contains an implementation of the model checker. After a brief overview of *RETHER*, we discuss its encoding in VPL, the design language supported by the Factory, and discuss the performance of the model checker on the protocol. Based on insights we obtained in formalizing the protocol, we then offer an alternative design and prove that it too meets the requirements set forth for *RETHER*. In all cases we report on the performance statistics of our algorithm. In our concluding remarks we comment on the dual role played by the case study: on the one hand, it gives us a good testbed for our algorithm, while on the other, it offers testimony to the utility of formal analysis as a means for uncovering design alternatives.

Part I: Local Model Checking in the Alternation-Free Modal Mu-Calculus

2 The Alternation-Free Modal Mu-Calculus

This section describes the syntax and semantics of the L_{μ_1}, the alternation-free modal mu-calculus. This logic enriches the traditional propositional calculus in two ways.

1. Single-step modalities allow the definition of properties that are intended to hold after a single execution step from the current system state.
2. Formulas may be defined recursively.

The modalities themselves appear quite weak when compared with the more expressive operators included in temporal logics such as CTL [4]. The facility for recursive definition, however, dramatically extends the expressive power of the logic; indeed, it can be shown that CTL is strictly less expressive that L_{μ_1}. The full modal mu-calculus, L_μ, which we do not consider in this paper, is in turn strictly more expressive than CTL* [9], the most expressive temporal logic in widespread use.

The literature contains several different presentations of L_{μ_1}; here we use an equational version presented in [8]. Before describing the logic, however, we first introduce *labeled transition systems*, which serve as models with respect to which formulas in the logic are interpreted.

Definition 1. A *labeled transition system* (LTS) is a 4-tuple $\langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$ where $\mathcal{S}$ is the set of *states*, Act is the set of *actions*, $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ is the *transition relation*, and $s_0 \in \mathcal{S}$ is the *start state*.

Intuitively, an LTS encodes the operational behavior of a system, with $\mathcal{S}$ containing the set of possible system states, Act the set of possible actions the system may engage in, and $\rightarrow$ the execution steps at every state.

2.1 L_{μ_1}: Basic Formulas

We now present the syntax of L_{μ_1} in two stages. The first stage introduces the modal operators, while the second deals with recursion.

The following grammar defines the syntax of first-stage, or *basic*, formulas. The grammar is parameterized by a set $(A \in) \mathcal{AP}$ of *atomic propositions*, a countably infinite set $(X, X_1, X_2) \in \mathcal{V}$ of *propositional variables*, and a set $(a \in) Act$ of actions.

$$\Phi ::= A \mid X \mid X_1 \vee X_2 \mid X_1 \wedge X_2 \mid \langle a \rangle X \mid [a]\, X$$

Basic formulas are interpreted with respect to an LTS $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$; a *valuation mapping* $V : \mathcal{AP} \rightarrow 2^{\mathcal{S}}$, relating every atomic proposition A to the set of states in which A holds; and an *environment* $e : Var \rightarrow 2^{\mathcal{S}}$, mapping each variable X to the set of states that satisfy X. For a fixed environment e, the meaning of basic formulas is given by the semantical function $[\![\cdot]\!]_{\mathcal{L},V} e : \Phi \rightarrow 2^{\mathcal{S}}$, defined in Figure 1. Intuitively, $s \in [\![\Phi]\!]_{\mathcal{L},V} e$ if state s satisfies Φ with respect to the given valuation. Atomic propositions, variables, and the propositional connectives $\vee$ and $\wedge$ have the obvious interpretations. The modal operators $\langle a \rangle$ and $[a]$ allow constraints to be placed on the outgoing a-labeled transitions of a state. A state s satisfies $\langle a \rangle X$ if it has an a-transition leading to a state satisfying X, while $[a]X$ holds of s if all its a-transitions lead to states satisfying X. Note that if s has no a-transitions then it vacuously satisfies $[a]X$.

$$
\begin{aligned}
[\![A]\!]_{\mathcal{L},V}\, e &= V(A)\\
[\![X]\!]_{\mathcal{L},V}\, e &= e(X)\\
[\![\Phi_1 \vee \Phi_2]\!]_{\mathcal{L},V}\, e &= [\![\Phi_1]\!]_{\mathcal{L},V}\, e \cup [\![\Phi_2]\!]_{\mathcal{L},V}\, e\\
[\![\Phi_1 \wedge \Phi_2]\!]_{\mathcal{L},V}\, e &= [\![\Phi_1]\!]_{\mathcal{L},V}\, e \cap [\![\Phi_2]\!]_{\mathcal{L},V}\, e\\
[\![\langle a\rangle \Phi]\!]_{\mathcal{L},V}\, e &= \{s \mid \exists s'.s \xrightarrow{a} s' \wedge s' \in [\![\Phi]\!]_{\mathcal{L},V}\, e\}\\
[\![[a]\Phi]\!]_{\mathcal{L},V}\, e &= \{s \mid \forall s'.s \xrightarrow{a} s' \Rightarrow s' \in [\![\Phi]\!]_{\mathcal{L},V}\, e\}
\end{aligned}
$$

Fig. 1. Semantics of basic formulas.

It should be noted that the logic of basic formulas is extremely weak, since the operators may only be applied to variables. The second stage of the definition will remedy this shortcoming.

2.2 L_{μ_1}: Equational Blocks

The second stage of the definition of the syntax of L_{μ_1} introduces recursive propositional definitions via the use of *equational blocks*. An equational block consists of a system (i.e. ordered set) E of mutually recursive equations together with a *tag* max or min. E has form

$$
\begin{aligned}
X_1 &= \Phi_1\\
&\vdots\\
X_n &= \Phi_n
\end{aligned}
$$

where each Φ_i is a basic formula and the X_i are pairwise distinct variables. The intention behind such a block is that the equations "define" n different formulas, one for each variable; since variables can appear in each other's right-hand sides, one may define properties recursively.

The tags attached to blocks are necessitated by the fact that mutually recursive equations by themselves do not define properties uniquely. To see why, consider the following equation system

$$
E = \left\{ \begin{array}{l} X_1 = \langle a\rangle X_2 \\ X_2 = \langle b\rangle X_1 \end{array} \right\}
$$

One consistent interpretation to attach to these equations is that X_1 and X_2 cannot be satisfied; that is, they are equivalent to "false". Another consistent interpretation is that X_1 is satisfied by states capable of an infinite sequence of alternating a's and b's, beginning with a, while X_2 is satisfied by states capable of a similar infinite alternating sequence beginning with b. The tags disambiguate equation systems by indicating which of the interpretations, or "solutions", is intended. Thus $\mathsf{min}\ E$ indicates that the most restrictive interpretation of E is intended, i.e. the one in which X_1 and X_2 are both "false", while $\mathsf{max}\ E$ signifies the most permissive one (the "alternating sequences of a's and b's" interpretation, in this case). That the notions of "most restrictive" and "most permissive" are well-defined is a consequence of the Tarski-Knaster Fixpoint Theorem, as we now show.

Semantically blocks may be understood as distinguished solutions to systems of equations whose variables and atomic propositions range over sets of system states, or, equivalently, as fixed points of functions mapping tuples of state sets to tuples of state sets To see this, fix LTS $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0\rangle$ and let E be a set of equations with left-hand sides $\overline{X} = \langle X_1, \ldots, X_n\rangle$. If $e : \mathcal{V} \rightarrow 2^{\mathcal{S}}$ is an environment and $\overline{S} = \langle S_1, \ldots, S_n\rangle$ is an n-tuple of sets of states, then we write $e[\overline{X} \mapsto \overline{S}]$ for the environment that maps X_i to S_i and is otherwise unchanged. For a fixed environment e we may now define a function f^e_E mapping n-tuples of state sets to n-tuples of state sets as follows.

$$
f^e_E(\overline{S}) = \langle [\![\Phi_1]\!]_{\mathcal{L},V} e[\overline{X} \mapsto \overline{S}], \ldots, [\![\Phi_n]\!]_{\mathcal{L},V} e[\overline{X} \mapsto \overline{S}]\rangle]
$$

In other words, $f^e_E(\overline{S})$ returns the results of evaluating the right-hand sides of the equations in environment e, where the values of each X_i are taken from the input tuple $\overline{S}$. We have the following facts.

- The set of n-tuples of state sets forms a complete lattice, with tuples ordered by pointwise set inclusion.
- For any equation system E and environment e, f^e_E is monotonic over this lattice.

These observations permit the Knaster-Tarski Fixpoint Theorem to be applied to f^e_E; therefore, there exist a unique least tuple $\mu f^e_E \in (2^{\mathcal{S}})^n$ and greatest tuple $\nu f^e_E \in (2^{\mathcal{S}})^n$ such that $f^e_E(\mu f^e_E) = \mu f^e_E$ and $f(\nu f^e_E) = \nu f^e_E$. The elements of these tuples are sets of states, and these sets are taken to be the meanings of the associated formulas in a block. That is, for a block of the form $B = \mathsf{min}\ E$,

$$
[\![B]\!]_{\mathcal{L},V} e = \mu f^e_E,
$$

while for a block of the form $B = \mathsf{max}\ E$,

$$
[\![B]\!]_{\mathcal{L},V} e = \nu f^e_E.
$$

So a block with n equations defines n different "formulas", one for each equation in the block.

2.3 Syntax and Semantics of L_{μ_1}

We can now give the syntax for the version of L_{μ_1} considered in this paper. Formulas in L_{μ_1} have form

$$
X\ \mathsf{where}\ \mathcal{B}
$$

where X is a variable and $\mathcal{B} = \{B_1, \ldots, B_m\}$ is a set of blocks obeying the following.

1. The left-hand sides in the B_i must be distinct from one another.
2. $\mathcal{B}$ must be *closed*: every variable used in a right-hand side in $\mathcal{B}$ must also be a left-hand side in $\mathcal{B}$.
3. X must be a left-hand side of some B_i.
4. The *dependency relation* (defined below) for $\mathcal{B}$ must be acyclic.

A block B_j *depends on* a block B_i if the right-hand side of some equation in B_j uses a variable appearing as the left-hand side of some equation in B_i. Requiring that this dependency relation be acyclic amounts to saying that all mutually recursive definitions of formulas occur within single blocks. This restriction ensures that there can be no alternating fixed points in the formulas defined by $\mathcal{B}$ [10].

To give the semantics of X where $\mathcal{B}$, we first show how to interpret $\mathcal{B}$. As with single blocks, $\mathcal{B}$ defines a collection of formulas, one for each equation in each block. To determine the meanings of these formulas, one does the following.

1. Topologically sort $\mathcal{B}$ using the dependency relation to obtain the sequence $B_1, \ldots, B_m$. This sequence has the property that blocks can only depend on blocks occurring earlier in the sequence.
2. Given an initial environment e_0, compute a sequence $e_1, \ldots, e_m$ as follows.

$$\begin{aligned} e_1 &= [\![B_1]\!]_{\mathcal{L},V} e_0 \\ &\vdots \\ e_m &= [\![B_m]\!]_{\mathcal{L},V} e_{m-1} \end{aligned}$$

3. Take $[\![\mathcal{B}]\!]_{\mathcal{L},V} e_0 = e_m$.

Since $\mathcal{B}$ is closed and the dependency relation is acyclic, it follows that for any e_0, e_0' and variable X that appears in the left-hand side of $\mathcal{B}$, $([\![\mathcal{B}]\!]_{\mathcal{L},V} e_0)(X) = ([\![\mathcal{B}]\!]_{\mathcal{L},V} e_0')(X)$. Thus we define

$$[\![X \text{ where } \mathcal{B}]\!]_{\mathcal{L},\mathcal{V}} = ([\![\mathcal{B}]\!] e)(X),$$

where e is arbitrary.

2.4 Notational Extensions

The version of L_{μ_1} used in this paper is tailored for the easy presentation of the model-checking algorithm given in the next section. However, the notation become unwieldy when writing down actual formulas; for the purposes of readability during the discussion of the model-checking algorithm, we therefore introduce several notational conveniences.

In the context of the *RETHER* protocol it is useful to label modalities in mu-calculus formulas with sets of actions rather than just single actions. The meaning of such formulas is derived in a straightforward manner from the semantics of basic formulas as follows, where $S \subseteq Act$ is a set of actions.

$$[\![\langle S \rangle \Phi]\!]_{\mathcal{L},V} e = \bigcup_{a \in S} [\![\langle a \rangle \Phi]\!]_{\mathcal{L},V} e$$

$$[\![[S] \Phi]\!]_{\mathcal{L},V} e = \bigcap_{a \in S} [\![[a] \Phi]\!]_{\mathcal{L},V} e$$

That is, a state satisfies $\langle S \rangle X$ if some transition labeled by some action in S leads to a state satisfying X, while it satisfies $[S]\Phi$ if every transition labeled by an action in S satisfies X.

When writing sets of actions in modalities, we usually omit set braces; we also use $-$ as set complementation. Thus the formula $[-a, b]\Phi$ holds of a state if every transition labeled by actions *other than* a or b leads to a state satisfying Φ. In this interpretation, $[-]\Phi$ is satisfied by a state if all its transitions (i.e. all transitions not labeled by an action in the empty set) lead to states satisfying Φ.

3 The Local Model-Checking Algorithm

The problem of *model checking* in L_{μ_1} can be defined as follows: given an LTS $\mathcal{L} = \langle \mathcal{S}, A, \rightarrow, s_0 \rangle$, a valuation $\mathcal{V}$, and a L_{μ_1} formula X where $\mathcal{B}$, determine whether or not $s_0 \in [\![X \text{ where} \mathcal{B}]\!]_{\mathcal{L},\mathcal{V}}$. Global algorithms proceed by calculating $[\![\mathcal{B}]\!]_{\mathcal{L},\mathcal{V}}$ and then examining the set associated with X to see if it contains s_0. Such a strategy can lead to unnecessary computation, as whether or not s_0 satisfies X might depend on only a few pieces of information contained in $[\![\mathcal{B}]\!]_{\mathcal{L},\mathcal{V}}$. Local algorithms aim to improve on global algorithms by working in a demand-driven manner, calculating only those parts of $[\![\mathcal{B}]\!]_{\mathcal{L},\mathcal{V}}$ needed to ascertain the status of s_0 *vis à vis* X. In addition to avoiding the calculation of unnecessary information in $[\![\mathcal{B}]\!]_{\mathcal{L},\mathcal{V}}$, local algorithms also allow $\mathcal{S}$, the set of states, to be computed on the fly. This section presents a local model checker for L_{μ_1}.

Given an LTS $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$, valuation $\mathcal{V}$, and an L_{μ_1} formula X where $\mathcal{B}$, our local model model-checking algorithm *LMC* determines if X where $\mathcal{B}$ holds in s_0 by using depth-first search to construct a *dependency graph* $G_{\mathcal{L},\mathcal{B}}$. $G_{\mathcal{L},\mathcal{B}}$ is an and-or graph whose vertices have form $\langle s, X_i \rangle$, where s is a state and X_i is a variable appearing as a left-hand side in $\mathcal{B}$. The vertices of $G_{N,\phi}$, and therefore the states in $\mathcal{S}$, are constructed in a lazy, demand-driven fashion, using the rules given in Figure 2 and starting with the initial vertex $\langle s_0, X \rangle$. For example, for a vertex $w = \langle s, Y \rangle$, if Y has right-hand side $[a]\, Z$ and $\{s_1, \ldots, s_k\} = \{s' | s \xrightarrow{a} s'\}$ contains the set of states reachable from s via a single a-labeled transition, then w will have k successors $\{\langle s_i, Z \rangle\}$. As the graph is explored in a depth-first fashion, however, only some of w's successor vertices may actually have been built at any point during the execution of the algorithm, .

With each vertex v of $G_{\mathcal{L},\mathcal{B}}$ we associate a *type*, denoted $\text{type}(v) \in \{\wedge, \vee, \lambda\}$. The type of a vertex is determined by the rule that may be applied to it to construct its successors; these types are shown to the left of each rule in Figure 2. For instance, for the vertex w considered above, its type is $\wedge$. For a vertex for which none of the rules in Table 2 apply, its type is λ, indicating a "leaf node." Note that the type of a vertex $\langle s, Y \rangle$is completely determined by the form of the right-hand side attached to Y in $\mathcal{B}$.

With each vertex v that has been constructed by our depth-first search procedure we also maintain a *value*, denoted

$$\wedge\colon \frac{\langle s, Y\rangle}{\langle s, W\rangle \;\; \langle s, Z\rangle} \;\; [Y = W \wedge Z]$$

$$\vee\colon \frac{\langle s, Y\rangle}{\langle s, W\rangle \;\; \langle s, Z\rangle} \;\; [Y = W \vee Z]$$

$$\wedge\colon \frac{\langle s, Y\rangle}{\langle s_1, Z\rangle \;\; \cdots \;\; \langle s_k, Z\rangle} \;\; \left[Y = [a]\, Z,\;\; s \xrightarrow{a} s_1, \cdots, s \xrightarrow{a} s_k\right]$$

$$\vee\colon \frac{\langle s, Y\rangle}{\langle s_1, Z\rangle \;\; \cdots \;\; \langle s_k, Z\rangle} \;\; \left[Y = \langle a\rangle Z,\;\; s \xrightarrow{a} s_1, \cdots, s \xrightarrow{a} s_k\right]$$

Fig. 2. Rules for constructing the dependency graph.

$\mathrm{value}(v) \in \{0, 1, U\}$. Intuitively, if $\mathrm{value}(v) = U$ then its truth-value is unknown; otherwise, it is known to be false (if $\mathrm{value}(v) = 0$) and true otherwise. In general, the value of a vertex will depend on its type and on the values of its successors as given in Figure 2. In particular, if $\mathrm{type}(v) = \wedge$ (respectively, $\vee$), then $\mathrm{value}(v)$ is the conjunction (respectively, disjunction)[1] of the values of v's successor, if any; note that if a vertex of type $\wedge$ (respectively, $\vee$) has no successors then its value is 1 (respectively, 0). Otherwise, i.e. if $\mathrm{type}(v) = \lambda$, then v is a leaf node, meaning that the right-hand side of its variable component is either an atomic proposition A or a negated atomic proposition $\neg A$. In this case, the value of v is defined as follows.

$$\mathrm{value}(\langle s, X\rangle) = \begin{cases} 1 & \text{if } X = A, s \in \mathcal{V}(A) \\ 0 & \text{if } X = A, s \notin \mathcal{V}(A) \\ 1 - \mathrm{value}(\langle s, A\rangle) & \text{if } X = \neg A \end{cases}$$

The pseudo-code for *LMC* appears in Figure 3. When a vertex that is not of type λ is first constructed, it is assigned value U, since its value depends upon those of its successors, which have not yet been determined. The algorithm maintains an up-to-date value for a vertex as new vertices are added to the graph. Whenever a new vertex v is constructed whose type is λ, the values of some or all of the U-vertices that are immediate predecessors of v may change; this change may, in turn, affect the values of the next-level predecessors. Such backwards propagation of updates is accomplished by lines 8 and 17 of the pseudo-code; a "one-step" back-propagation is performed at line 22. The back-propagation procedure itself, which is not shown in Table 3, merely traverses the graph *backwards* from v, in depth-first fashion, updating U-values of the vertices as required, and backtracking as soon as it encounters a vertex whose value has already been determined to be 0 or 1. A consequence of this strategy is that once a vertex is assigned a value in $\{0, 1\}$, it is never changed.

The presence of cycles in $G_{\mathcal{L},\mathcal{B}}$ complicates the algorithm somewhat. Cycles are possible since $\mathcal{B}$ may, in general, involve recursive variable definitions, and the transition relation of $\mathcal{L}$ may be cyclic. To determine the values of U-vertices lying on cycles, as we build $G_{\mathcal{L},\mathcal{B}}$ we run Tarjan's algorithm [21] "on the side" to identify strongly connected components (SCCs) of U-vertices (recall that an SCC is a maximal set of vertices that are reachable from one another). Among other data structures, the SCC algorithm uses a stack to record all vertices visited so far; when an SCC is detected, it is guaranteed that all vertices in the SCC will constitute a prefix of the stack content. Call such an SCC a U-SCC. Since the formula is alternation-free, all the fixed points associated with a U-SCC are of the same type. Therefore, as soon as a U-SCC $\mathcal{U}$ is found, the values of the vertices in it can be uniformly assigned to 1 if $\mathcal{U}$ is associated with maximal, hence most permissive, fixed points, and 0 otherwise.

The number of vertices in $G_{\mathcal{L},\mathcal{B}}$ is bounded by $|S| \cdot n$, where n is the number of equations in $\mathcal{B}$, and is thus finite. It is then easy to see that *LMC* will terminate. The following Theorem ensures that *LMC* produces the correct result upon termination.

Theorem 1. *In LMC, if the final value given to a vertex* $\langle s, X\rangle$ *is* 1*, then* $s \in [\![X \text{ where } \mathcal{B}]\!]_{\mathcal{L},\mathcal{B}}$*, and if its final value is* 0*, then* $s \notin [\![X \text{ where } \mathcal{B}]\!]_{\mathcal{L},\mathcal{B}}$.

Let the *SCC-graph* of a dependency graph $G_{\mathcal{L},\mathcal{B}}$ be the directed graph induced by collapsing the SCCs of $G_{\mathcal{L},\mathcal{B}}$ into single nodes. Clearly, an SCC-graph is acyclic. *LMC* processes the SCCs of $G_{\mathcal{L},\mathcal{B}}$ in post-order of $G_{\mathcal{L},\mathcal{B}}$'s SCC-graph. Thus, the values of vertices in a given SCC can depend upon the values of vertices in the same SCC and, at most, on SCCs earlier in the post-order. The lemma can thus be proved by induction on the "post-order number" of the SCC in which the vertex $v = \langle s, X\rangle$ occurs. It is clear that the algorithm assigns correct values to vertices of type λ (which are singleton SCCs). For the case of non-λ vertices, the correctness follows from the induction hypothesis, the soundness of the rules for assigning values to $\wedge$ and $\vee$ vertices, and the fact that assigning all vertices in a U-SCC the value 1 (0) gives maximum (minimum) fixed points.

Time and Space Complexity. Let $|\mathcal{S}_{s_0}|$ and $|\longrightarrow_{s_0}|$ denote, respectively, the size of $\mathcal{L}$'s state space and transition relation

[1] Let $Val = \{0, 1, U\}$. The conjunction of a sequence in Val^* is 0 if it is in $Val^* 0\, Val^*$, 1 if it is in 1^*, and U otherwise. Dually, the disjunction of a sequence in Val^* is 1 if it is in $Val^* 1\, Val^*$, 0 if it is in 0^*, and U otherwise.

```
0  procedure LMC (N: network with [[N]] = ⟨S, ⟶, Λ, s0⟩, X0 in B: Lμ formula)
1     Initialize the DFS stack D := ⟨⟨s0, X0⟩⟩;
2     Initialize the visited vertex table VT := {⟨s0, X0⟩};
3     Initialize the Tarjan stack to TS := ⟨ ⟩;
4     while D is not empty do
5        let v = ⟨s, X⟩ be the top vertex in D;
6        if value(v) ∈ {0, 1} then {
               /* type(v) = λ or value known through back-propagation */
7           pop D;
8           transitively propagate v's value backwards via its U-predecessors;
9        }
10       else {  /* value(v) = U */
11          obtain the next successor v' of v, using the rules in Table 2;
                  /* v' is set to null if v has no more successors */
12          if v' = null then {
13             pop D;
14             push v on TS;
15             if v is the root of an SCC C then {  /* ask Tarjan */
16                pop the vertices in TS that belong to C,
                        and set their values to True or False depending
                        on the type of the associated fixed points;
                        /* the values of the vertices in C are now determined */
17                transitively propagate v's value backwards via its U-predecessors;
18             }
19          }
20          else if value(v') ∈ {0, 1} then {  /* type(v') = λ or v' visited before */
21             add v' to VT;
22             update value(v);
23          }
24          else if v' ∉ VT then {  /* value(v') = U and first visit of v' */
25             add v' to VT;
26             push v' on D;
27          }
28       }
29    endwhile
30    return value(⟨s0, X0⟩);
31 end  /* LMC */
```

Fig. 3. Basic algorithm for local model checking in L_{μ_1}. At line 11, the successors of v are examined one after the other in left-to-right order, so that each call returns the leftmost successor that has not yet been examined; $null$ is returned when all successors have been examined.

reachable from s_0. Let n be the number of variables in $\mathcal{B}$. The number of vertices in $G_{\mathcal{L},\mathcal{B}}$, then, is bounded by $|\mathcal{S}_{s_0}| \cdot n$ and the number of edges is bounded by $(|\longrightarrow_{s_0}| + 2|\mathcal{S}_{s_0}|)\, n$. Assume that the visited vertex table VT is implemented as a balanced binary tree, allowing insertion and membership operations in time logarithmic in its size. *LMC* then requires time at most $O\left((|\longrightarrow_{s_0}| + |\mathcal{S}_{s_0}|)\, n \log(|\mathcal{S}_{s_0}| \cdot n)\right)$. [2]

The overall space complexity of the algorithm is easily seen to be bounded by $O\left(|\longrightarrow_{s_0}| + |\mathcal{S}_{s_0}| \cdot n\right)$.

As mentioned previously, the algorithm of Table 3 is for the alternation-free fragment of the modal mu-calculus. The algorithm actually implemented in the Concurrency Factory is for the full modal mu-calculus and supports an additional mechanism for state-space pruning known as partial-order reduction. See [19].

Implementation Issues. The vertex table VT may also be used to record the values of vertices. Whenever a vertex is visited for the first time, its value (0, 1, U) can be stored with it in VT and then updated as needed. Also, while a balanced binary tree implementation of VT yields the best asymptotic bound on complexity, in practice a hash table generally proves superior.

4 Example of the Algorithm in Action

To enhance the reader's understanding of *LMC* , we illustrate the algorithm's behavior on a simple example. The example consists of the two-state LTS $\mathcal{L}$ depicted in Figure 4 and the

[2] The logarithmic factor above drops out if, like in [22], we assume the use of a direct-access structure to store VT. We do not, however, make this assumption since the initialization of such a structure goes against the local spirit of the algorithm.

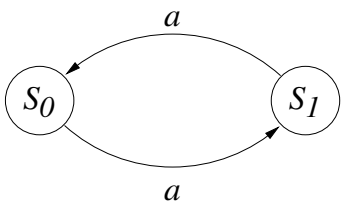

Fig. 4. Example labeled transition system.

L_{μ_1} formula X_0 where $\mathcal{B}$, where $\mathcal{B} = \{B_1, B_2\}$. Blocks B_1 and B_2 are defined as follows:

$$B_1 = \max \left\{ \begin{array}{l} X_0 = X_1 \wedge X_2 \\ X_1 = [a]X_0 \end{array} \right\} B_2 = \min \left\{ \begin{array}{l} X_2 = X_3 \vee X_4 \\ X_3 = p \\ X_4 = [a]X_2 \end{array} \right\}$$

Intuitively, X_0 where $\mathcal{B}$ expresses the property "alway eventually p", where p is an arbitrary atomic proposition, and can be expressed in the logic CTL as AG EF p. In our example LTS, p is assumed to be false in each of the two states, so X_0 where $\mathcal{B}$ will also be false.

LMC constructs the portion of the dependency graph $G_{\mathcal{L},\mathcal{B}}$ drawn in Figure 5 in solid lines. The dashed nodes and edges are not visit due to the top-down, goal-directed manner in which the dependency graph is constructed by the algorithm. The cycle containing the vertices $\langle s_0, X_2\rangle$, $\langle s_0, X_4\rangle$, $\langle s_1, X_2\rangle$, and $\langle s_1, X_4\rangle$ is a U-SCC and, as expected, all the vertices along the cycle are associated with the same type of fixed point, namely minimal. Hence all of these vertices are uniformly assigned the value 0 and this value is propagated back to the initial vertex, $\langle s_0, X_0\rangle$. *LMC* will then return the final value of 0 meaning that X_0 where $\mathcal{B}$ has been found to be false in the initial state of $\mathcal{L}$.

Part II: Local Model Checking and the *RETHER* Protocol

The remainder of this paper describes a case study in which we use LMC to analyze *RETHER* [2], a Real-time Ethernet protocol intended to provide real-time communication guarantees using commodity ethernet infrastructure. To achieve this, we encoded *RETHER* in the Concurrency Factory [7], a verification tool for finite-state concurrent systems in which a version of LMC is implemented. The Factory provides users with graphical and textual languages for describing hierarchical networks of communicating finite-state processes; we use the textual language, VPL, to encode *RETHER* and LMC to check whether *RETHER* has the properties it should.

This part of the paper is structured as follows. We first provide an overview of *RETHER* as described in [2], discuss its encoding in VPL, and formulate its requirements in the modal mu-calculus. We then present performance statistics exhibited by LMC in verifying that these properties hold. Finally, we repeat our analysis of *RETHER* on an alternative design of the protocol having potentially lower token-passing overhead.

5 The *RETHER* Protocol

RETHER is a contention-free token bus protocol for the datalink layer of the ISO protocol stack. It is designed to run on top of a CSMA/CD physical layer. A network running the protocol normally operates in CSMA/CD mode, transparently switching to *RETHER* mode when one or more nodes generate requests for real-time (RT) connections. An initialization protocol is used to coordinate the switch to *RETHER* mode. Once in *RETHER* mode, a token is used to control access to the medium: a node can transmit data on the network only when it has the token.

The token circulates the network in cycles, each of which is called a *token rotation cycle*. The period of a cycle, the *token cycle time*, is a configurable parameter determined by the frequency of real-time data transmissions. For instance, TCT (as this parameter is called) can be set to 33.3 *ms* for video applications, which typically transmit data at the rate of 30 frames per second. In each token cycle, all nodes that have successfully made a bandwidth reservation (the so-called "admitted nodes") can send out RT data in accordance with their reservations. In the time remaining in the cycle, nodes are permitted to send out non-real-time (NRT) data. To be fair to NRT transmission requests, part of each cycle, T_{nrt}, is set aside for NRT traffic; the value of this parameter is also configurable.

An RT application is admitted into the system by the protocol only if there is sufficient bandwidth available. To illustrate, suppose this application requests a reservation for an RT transmission of duration $THT_{RT_{new}}$ (where THT stands for "token-holding time"). Then this request is granted only if the following inequality holds:

$$\left(\sum_{i \in RT_set} THT_{RT_i} \right) + THT_{RT_{new}} + T_{nrt} \leq TCT$$

where RT_set is the set of admitted RT nodes, and THT_{RT_i} is the token holding time of the ith RT node. It is precisely this policy that enables the protocol to guarantee cycle bandwidth to RT applications.

Admitted RT nodes are serviced first in a cycle; we refer to this policy as RTF. A newly admitted RT application adds its node id to RT_set, which is stored in the token as a bit vector. A node releasing RT bandwidth removes its id from RT_set before forwarding the token to the next node. The network reverts back to CSMA/CD mode when the last RT node releases its bandwidth. The token also carries a *residual*

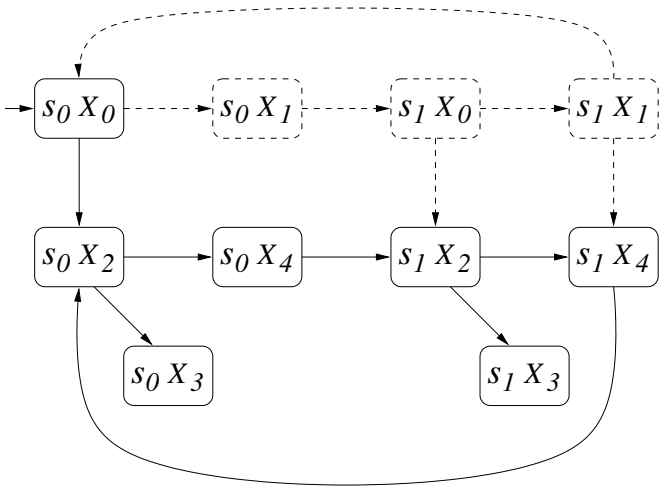

Fig. 5. Example dependency graph.

time counter that records the amount of time remaining in a cycle. At the start of a cycle this field is initialized to TCT. Following a data transmission by the token-holding node, the residual time counter is decremented by that node's token-holding time before the token is passed to the next node.

The admission policy ensures that at least T_{nrt} time remains in each cycle for NRT traffic. NRT transmissions are scheduled in a round-robin fashion to prevent the possibility of starvation. For this purpose, the token carries a field containing the id of the last node that failed to transmit its NRT data in a given cycle because of insufficient residual time.

To illustrate the behavior of the protocol, and the RTF node-servicing policy, consider the example network depicted in Figure 6. RT_set contains 1, 3, and 5, so these nodes are serviced first during each token rotation cycle. Assuming that each cycle can accommodate three NRT transmissions (in addition to the three guaranteed RT transmissions), and that each node has NRT data to transmit, it takes two successive cycles for the NRT transmissions to be serviced in round-robin fashion. In particular, the following sequences of node transmissions are possible in successive cycles:

1 – 3 – 5 – 1 – 2 – 3
1 – 3 – 5 – 4 – 5 – 1

6 VPL Specification of *RETHER*

In this section, we first introduce VPL, the textual input language of the Concurrency Factory. We then describe how the *RETHER* protocol is specified in VPL.

6.1 VPL

VPL is designed for specifying concurrent systems with non-trivial control structure and data structures that involve value passing (as opposed to pure synchronization). VPL-supported data structures include integers of limited range as well as arrays and records. Structural equivalence of data types is employed, i.e. two integer types are considered the same if their sizes are equal and two records are equivalent when they have fields of the same type appearing in the same order.

A system specification in VPL is a tree-like hierarchy of subsystems. A subsystem is either a *network* or a *process*. A network consists of a collection of subsystems running in parallel and communicating with each other through typed channels. Processes are at the leaves of the hierarchy. Each subsystem, whether process or network, consists of a header, local declarations and body. The header specifies a unique name for the subsystem and a list of "formal channels" (by analogy with formal parameters of procedures in programming languages). The names of the formal channels of a subsystem can be used in the body of the subsystem and represent events visible to an external observer.

Declarations local to a network include specifications of the subsystems of the network and channels for communication between subsystems that are to be hidden from the outside world. The body of a network is a parallel composition of its subsystems. A subsystem declared within a network can be used arbitrarily many times; each time a new copy of the subsystem is instantiated with actual channels substituted for the formal ones. Actual channels must match the formal ones introduced in the header and must be declared either as local channels or formal channels of the network immediately containing the subsystem.

Declarations local to a process consist of variable and procedure declarations. Procedure bodies, like process bodies, are sequences of statements. Simple statements of VPL are assignments of arithmetic or boolean expressions to variables, and input/output operations on channels. Complex statements include sequential composition, `if-then-else`, `while-do`, and nondeterministic choice in the form of the `select` statement.

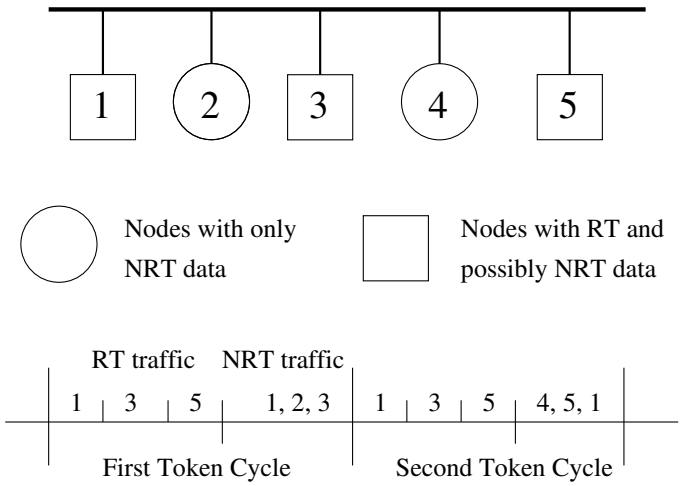

Fig. 6. A Sample Network Configuration.

6.2 The RETHER Protocol in VPL

In this subsection, we describe our specification of the *RETHER* protocol in VPL under the RTF node-servicing policy. The interested reader can find the VPL source code listing in Appendix A and on the worldwide web at URL `www.cs.sunysb.edu/~sas/rether.html`.

The key to the specifications, and subsequent model checking, is our abstraction of time into "time slots." That is, each token cycle is divided into a fixed number of time slots, and each RT and NRT transmission consumes one time slot. Since the properties we wish to verify (admitted RT nodes receive their guaranteed cycle bandwidth and NRT transmissions are not starved) do not depend on the exact value of the token holding times, our time slot abstraction is sufficient for this purpose.

The protocol is specified in VPL as a network of `N`+2 processes: `Node_0`, . . ., `Node_N-1`, `Token`, and `Bandwidth`. The `Node_i` processes capture the behavior of the various nodes in the network, `Token` represents the *RETHER* token, and `Bandwidth` is the process that nodes interact with to reserve or release cycle bandwidth. A separate process is set aside for the token in order to localize the token-passing logic deployed by the protocol. Modeling the token in this fashion decreases the size of the specification's state space.

Figure 7 depicts the network topology of the specification. In particular, it indicates the names of the channels that processes use to communicate with one another and with the outside world. To simplify the figure, only `Node_0` is depicted; the other nodes are connected to `Token`, `Bandwidth`, and the outside world in a similar fashion.

The connections to the outside world are for model-checking purposes. That is, communications over these channels will appear in the various modal mu-calculus formulas to be model checked. Each such action represents a pure synchronization; the intended meaning of these actions is given in Table 1. Since the contents of the transmitted data is irrelevant to the correct operation of the protocol, it suffices to use the (dataless) signals `rt_i` and `nrt_i`.

Process `Bandwidth` represents the simplest of the processes in the specification. It essentially behaves as a counting semaphore whose value should not exceed `RTSlots`, where `RTSlots` defines the total number of time slots available in a token cycle for RT data transmissions. `RTSlots` is less then the total number of time slots (`Slots`) in a cycle since some slots are reserved for NRT traffic. An RT node requests a bandwidth reservation via the signal `reserve`. The request is `granted` if a time slot is available; i.e., if `RT_count` $<$ `RTSlots`, where `RT_count` is a local variable that keeps track of the number of RT nodes in the system.

When an RT node is ready to release its reserved bandwidth, it signals `Bandwidth` with `release`. It is `ok` for the node to release its bandwidth and become an NRT node as long as the system contains at least one RT node at all times. This is to ensure that the protocol remains in *RETHER* mode: the protocol reverts back to CSMA/CD mode when all RT connections terminate, and we do not model this mode of the protocol. `Bandwidth` also writes `RT_count` to channel `RTC_chan` when `Token` wants to read the number of RT nodes in the system.

The `Token` process determines the order in which the token visits the nodes in the network. It is also responsible for starting and ending the token rotation cycle and updating the information in the token at the beginning of each cycle. To perform these duties, it maintains a local data structure `tok_value` with the following fields: a bit-vector `node[]` used to encode RT_set; an integer `NRT_count` indicating the number of time slots available for NRT data transmissions in the current cycle (`NRT_count` is decremented each time an NRT data transmission occurs); an integer `next`, which points to the next node with NRT data to be serviced in round-robin fashion; and a boolean variable `serving_rt`, that indicates whether RT traffic or NRT traffic is currently being serviced.

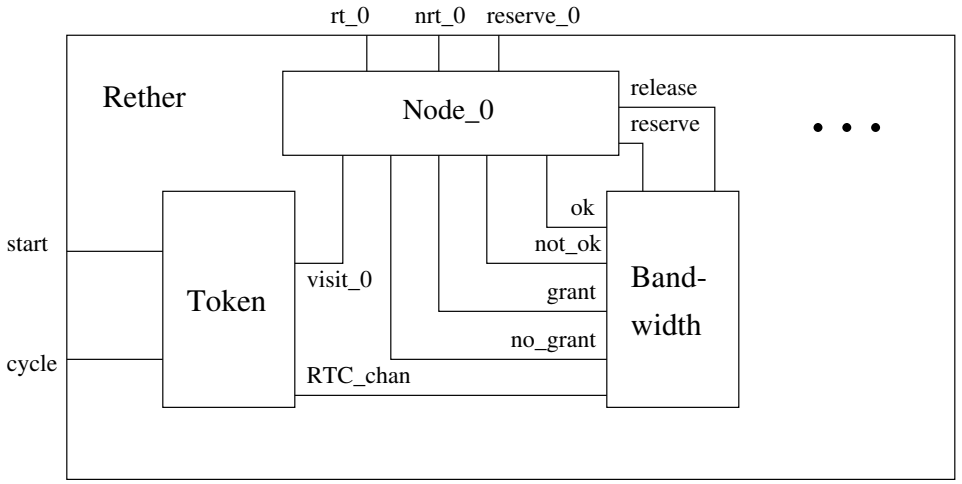

Fig. 7. Network topology of the specification.

Table 1. Actions and their interpretations.

`start`:	indicates the beginning of a token cycle
`cycle`:	indicates the end of a token cycle
`reserve_i`:	`Node_i` successfully makes a bandwidth reservation
`rt_i`:	`Node_i` performs an RT transmission
`nrt_i`:	`Node_i` performs an NRT transmission

`Token` starts a cycle by outputting the signal `start` to the outside world. It then passes the token to each of the nodes in RT_set in order of their node ids. `Token` passes the token to `Node_i` by writing the token data structure onto channel `visit[i]`; when `Node_i` is finished with the token it sends it back to `Token` over the same channel. After finishing with the RT nodes, `Token` circulates the token among the nodes in the system until all remaining time slots are consumed. At this point, `Token` terminates the cycle by outputting the signal `cycle` and setting `NRT_count` to `Slots` − `RT_count` (recall that `RT_count` is maintained by the bandwidth process). It then begins a new cycle.

Process `Node_i` first receives the token from process `Token` into the local variable `local_tok`. It then checks the `serving_rt` field in `local_tok` (written as `local_tok.serving_rt` in VPL) to determine whether it should call procedure `RT_action` or `NRT_action`. Procedure `RT_action` simulates an RT data transmission by emitting signal `rt_i`, and then nondeterministically chooses to release its bandwidth. If its request to release bandwidth is successful, the node is deleted from RT_set and `local_tok.node[i]` is updated accordingly.

`Node_i` executes procedure `NRT_action` when it receives the token and `local_tok.serving_rt` is false. The following conditions are guaranteed to hold at this point: `Node_i` is being pointed to by `local_tok.next`, and the value of `local_tok.NRT_count` is greater than zero. Therefore, it simulates an NRT transmission by emitting signal `nrt_i`, decrementing `local_tok.NRT_count`, and incrementing `local_tok.next.NRT_action` then checks `local_tok.node[i]` to see if it is an RT node; if not, it nondeterministically chooses to issue a request for bandwidth reservation. If granted, `local_tok.node[i]` is set to true, making `Node_i` an RT node, and signal `reserve_i` is emitted to the outside world, indicating that `Node_i` has successfully made a bandwidth reservation. Upon completion of `NRT_action`, `Node_i` outputs `local.tok` to `Token`.

7 Mu-Calculus Properties for *RETHER*

We now describe the mu-calculus properties that capture the intended behavior of *RETHER*. At the outset, we should note that to ensure that we are model checking a finite-state system, we fix the following parameters of the specification at compilation time: the number of nodes in the system is chosen to be 4, i.e. the system contains `Node_0` up to `Node_3`; the total number of time slots in each token rotation cycle (`Slots`) is 3; and one slot per cycle is reserved for NRT data transmission. Consequently, during each cycle there are either two RT transmissions plus one NRT transmission, or two NRT transmissions plus one RT transmission. These values may be seen in some of the formulas given below. Initially `Node_0` is designated as an RT node and is pointed to by `next`. The remaining nodes are initially NRT nodes.

In addition, the version of the mu-calculus we use contains only two atomic propositions: tt for "true", and ff for "false". Regardless of the labeled transition under consideration, the valuation assigns the set of all states to tt and the empty set of states to ff. In other words, every state is assumed to satisfy tt, and no state is assumed to satisfy ff.

The formulas and their intended meanings are as follows.

– **DLF**: Deadlock freedom.

$$X \text{ where } \left\{ \mathsf{max} \left\{ \begin{array}{ll} X = X_1 \wedge X_2 & X_2 = \langle - \rangle X_3 \\ X_1 = [-]\, X & X_3 = \mathsf{tt} \end{array} \right\} \right\}$$

– **CC**: Cycle completion.

$$X \text{ where } \left\{ \begin{array}{l} \mathsf{max} \left\{ \begin{array}{l} X = X_1 \wedge X_2 \\ X_1 = [-]\, X \\ X_2 = [\mathtt{start}]\, Y \end{array} \right\} \\ \\ \mathsf{min} \left\{ \begin{array}{ll} Y = Y_1 \wedge Y_2 & Y_4 = \langle - \rangle Y_6 \\ Y_1 = [-\mathtt{cycle}]\, Y & Y_5 = \mathsf{ff} \\ Y_2 = Y_3 \wedge Y_4 & Y_6 = \mathsf{tt} \\ Y_3 = [\mathtt{start}]\, Y_5 & \end{array} \right\} \end{array} \right\}$$

This formula states that each token rotation cycle must be completed. It requires that whenever the beginning of a cycle is seen ([start]), then eventually the end of the cycle must also be seen by observing the signal cycle, and that there cannot be another start before the cycle is observed. The requirement forbidding the occurrence of start is enforced by the subformula [start]ff. To see why, note that the semantics of the mu-calculus require that for a state to satisfy this subformula, all the state's transitions must lead to states satisfying ff. Since no such states can exist, the only way [start]ff can hold of a state is if the state has no transitions labeled by start.

– **RT0 - RT3**: Bandwidth guarantee for RT nodes.

$$X \text{ where } \left\{ \begin{array}{l} \mathsf{max} \left\{ \begin{array}{l} X = X_1 \wedge X_2 \\ X_1 = [-]\, X \\ X_2 = [\mathtt{reserve_i}]\, Y \end{array} \right\} \\ \\ \mathsf{min} \left\{ \begin{array}{ll} Y = Y_1 \wedge Y_2 & Z = Z_1 \wedge Z_2 \\ Y_1 = [-\mathtt{cycle}]\, Y & Z_1 = [-\mathtt{rt_i}]\, Z \\ Y_2 = Y_3 \wedge Y_4 & Z_2 = Z_3 \wedge Z_4 \\ Y_3 = [\mathtt{rt_i}]\, Y_5 & Z_3 = \langle - \rangle Z_5 \\ Y_4 = [\mathtt{cycle}]\, Z & Z_4 = [\mathtt{cycle}]\, Y_5 \\ Y_5 = \mathsf{ff} & Z_5 = \mathsf{tt} \end{array} \right\} \end{array} \right\}$$

These formulas state that whenever Node_i successfully makes a bandwidth reservation during a token rotation cycle ([reserve_i] ... [cycle]), then during the immediately succeeding cycle (i.e. before a second cycle signal is seen), Node_i will transmit RT data ([rt_i]). Also no RT data transmission for this node is allowed during the cycle in which it made the reservation (the node is not yet an RT node in that cycle).

– **NS0 - NS3**: No Starvation for NRT traffic.

$$X \text{ where } \left\{ \begin{array}{l} \mathsf{max} \left\{ \begin{array}{l} X = X_1 \wedge X_2 \\ X_1 = [-]\, X \\ X_2 = [\mathtt{start}]\, X_3 \\ X_3 = X_4 \wedge X_5 \\ X_4 = [-\mathtt{nrt_i}, \mathtt{cycle}]\, X_3 \\ X_5 = [\mathtt{cycle}]\, Y \end{array} \right\} \\ \\ \mathsf{min} \left\{ \begin{array}{l} Y = Y_1 \wedge Y_2 \\ Y_1 = [-\mathtt{nrt_i}]\, Y \\ Y_2 = \langle - \rangle Y_3 \\ Y_3 = \mathsf{tt} \end{array} \right\} \end{array} \right\}$$

If during a particular cycle, a Node_i does not transmit any NRT data (i.e. no nrt_i between start and cycle), then eventually an nrt_i signal will be observed, meaning that Node_i eventually transmits some NRT data.

– **RTT**: At least one RT data transmission in each cycle.

$$X \text{ where } \left\{ \begin{array}{l} \mathsf{max} \left\{ \begin{array}{l} X = X_1 \wedge X_2 \\ X_1 = [-]\, X \\ X_2 = [\mathtt{start}]\, Y \end{array} \right\} \\ \\ \mathsf{min} \left\{ \begin{array}{ll} Y = Y_1 \wedge Y_2 & Y_4 = \langle - \rangle Y_6 \\ Y_1 = [-S_{RT}]\, Y & Y_5 = \mathsf{ff} \\ Y_2 = Y_3 \wedge Y_4 & Y_6 = \mathsf{tt} \\ Y_3 = [\mathtt{cycle}]\, Y_5 & \end{array} \right\} \end{array} \right\}$$

Once a start is seen, at least one signal from the set $S_{RT} = \{\mathtt{rt_}i | 0 \leq i \leq 3\}$ must be observed before the cycle signal.

– **NRT**: At least one NRT data transmission in each cycle.

$$X \text{ where } \left\{ \begin{array}{l} \mathsf{max} \left\{ \begin{array}{l} X = X_1 \wedge X_2 \\ X_1 = [-]\, X \\ X_2 = [\mathtt{start}]\, Y \end{array} \right\} \\ \\ \mathsf{min} \left\{ \begin{array}{ll} Y = Y_1 \wedge Y_2 & Y_4 = \langle - \rangle Y_6 \\ Y_1 = [-S_{NRT}]\, Y & Y_5 = \mathsf{ff} \\ Y_2 = Y_3 \wedge Y_4 & Y_6 = \mathsf{tt} \\ Y_3 = [\mathtt{cycle}]\, Y_5 & \end{array} \right\} \end{array} \right\}$$

Once a start is seen, at least one signal from the set $S_{NRT} = \{\mathtt{nrt_}i | 0 \leq i \leq 3\}$ must be observed before the cycle signal.

– **T3T**: A total of three data transmissions in each cycle.

$$X \text{ where } \left\{ \begin{array}{l} \mathsf{max} \left\{ \begin{array}{l} X = X_1 \wedge X_2 \\ X_1 = [-]\, X \\ X_2 = [\mathtt{start}]\, Y \end{array} \right\} \\ \\ \mathsf{min} \left\{ \begin{array}{ll} Y = Y_1 \wedge Y_2 & \\ Y_1 = [-S]\, Y & Y_{10} = Y_{11} \wedge Y_{12} \\ Y_2 = Y_3 \wedge Y_4 & Y_{11} = [-S]\, Y_{10} \\ Y_3 = [\mathtt{cycle}]\, Y_5 & Y_{12} = Y_3 \wedge Y_{13} \\ Y_4 = [S]\, Y_6 & Y_{13} = [S]\, Z \\ Y_5 = \mathsf{ff} & \\ Y_6 = Y_7 \wedge Y_8 & \\ Y_7 = [-S]\, Y_6 & Z = Z_1 \wedge Z_2 \\ Y_8 = Y_3 \wedge Y_9 & Z_1 = [-\mathtt{cycle}]\, Z \\ Y_9 = [S]\, Y_{10} & Z_2 = [S]\, Y_5 \end{array} \right\} \end{array} \right\}$$

Once a `start` is seen, there must be exactly three signals from the set $\{\mathtt{rt_}i, \mathtt{nrt_}i | 0 \leq i \leq 3\}$ before the `cycle` signal.

– **RTF**: RT-first property.

$$X \text{ where } \left\{ \begin{array}{l} \mathsf{max} \left\{ \begin{array}{l} X = X_1 \wedge X_2 \\ X_1 = [-]\, X \\ X_2 = [S_{NRT}]\, Y \end{array} \right\} \\ \\ \mathsf{min} \left\{ \begin{array}{ll} Y = Y_1 \wedge Y_2 & Y_4 = \langle - \rangle Y_6 \\ Y_1 = [-\mathtt{cycle}]\, Y & Y_5 = \mathsf{ff} \\ Y_2 = Y_3 \wedge Y_4 & Y_6 = \mathsf{tt} \\ Y_3 = [S_{RT}]\, Y_5 & \end{array} \right\} \end{array} \right\}$$

This formula states that in each cycle, if an NRT data transmission is observed, no more RT data transmissions can be seen before the end of the cycle. This implies that all RT data transmissions must precede any NRT data transmissions in each cycle.

Note that all of the formulas given above are of the form:

$$X \text{ where } \{\mathsf{max}\, \{X = [-]\, X \wedge \Phi\}\}$$

where the variable X does not appear in Φ. Intuitively an L_{μ_1} formula of this kind holds of a state if Φ holds in every state reachable from the given state, and corresponds to the AG of CTL. AG formulas specify system *invariants*, i.e., properties intended to hold throughout the execution of the system.

8 Model Checking the *RETHER* Protocol

In this section, we summarize the results we obtained applying the Concurrency Factory's local model checker to the VPL specification of *RETHER* given above, which implements the RTF strategy. Tables 2 presents the answers computed and the system resources consumed (memory and time). All data is obtained from a Sun4m Sparc machine with 65MB RAM and 520MB swap space. As can be seen, the protocol satisfies all 14 properties devised for it.

Table 3 illustrates the benefit of using a local, as opposed to global, model checker. To quantify this benefit, we first calculated the size of the reachable state space of *RETHER* by using a special "DFS" option of the local model checker, which forces the local model checker to perform a depth-first search of the LTS underlying the VPL specification and return the number of states in the LTS. The result of the depth-first search for the configuration of *RETHER* described above is given below.

model	size of LTS
RTF	12.8 K

With this information, and given the number of variables in the blocks of a L_{μ_1} formula, we may arrive at an upper bound on the number of vertices in the dependency graph $G_{\mathcal{L},\mathcal{B}}$ constructed by LMC as follows.

$$\text{no. vertices in } G_{\mathcal{L},\mathcal{B}} = (\text{no. states in } \mathcal{L}) \times (\text{no. variables in } \mathcal{B})$$

Global algorithms would require storage for each such vertex [8], since they in essence construct the graph before exploring it. Because LMC builds the graph in a demand-driven manner, however, not every vertex need be constructed. Table 3 demonstrates this fact concretely. For each formula, the table records the number of variables in the formula, the number of vertices in the dependency graph constructed by LMC, the maximum number of vertices in the dependency graph, and the percentage of the total LMC therefore explores.

The data indicates that the local model checker indeed achieves significant pruning of the dependency graph. On average, 52% of the total number of vertices is explored by LMC. To understand why LMC can avoid traversing the whole dependency graph, one should first observe that in general, the value associated with the nodes of the dependency graph depends on the observation of certain signals. Once the value of a node is decided due to the observation of a signal, the values of its successor nodes in the dependency graph may not be needed. This explains the observed savings. For example, the formulas RTn need exploration of only about 20% of the state space, as in each cycle once a certain real-time data transmission is observed, LMC will proceed to the evaluation of the next cycle. On the other hand, the formula DLF does not allow much savings as no such pruning is possible. Even in this case, however, some savings devolves from the processing of "or" nodes $\langle - \rangle \mathtt{tt}$, which allows LMC to avoid exploring certain nodes of the dependency graph.

One may see small differences in the vertex count for the formulas RT0 through RT3, even though the formulas are symmetric variants of one another. This is due to the fact that our system is not strictly symmetric, since `Node_0` starts out as an RT node, while the others start out as NRT nodes. Also, the token visits the nodes in a specific order in each cycle. These reasons also explain the minor variation among the data obtained for formulas NS0 through NS3.

Table 2. Model checking results for *RETHER*.

Formula	Result	Memory (MB)	Time (min)
DLF	True	11.0	80.6
CC	True	16.2	125.3
RT0	True	7.9	42.6
RT1	True	8.2	45.5
RT2	True	8.4	47.4
RT3	True	8.2	45.7
NS0	True	17.2	129.8
NS1	True	17.3	129.2
NS2	True	17.3	128.9
NS3	True	17.5	131.4
RTT	True	8.4	47.8
NRT	True	14.0	100.2
T3T	True	13.0	82.7
RTF	True	9.7	59.0

Table 3. Percentage of dependency graph explored by LMC for *RETHER*.

Formula	no. of vars	vertices(K) visited	vertices(K) total	percentage explored
DLF	4	49.2	51.2	96%
CC	9	93.9	115.2	82%
RT0	15	40.5	192.0	21%
RT1	15	42.7	192.0	22%
RT2	15	43.9	192.0	23%
RT3	15	43.4	192.0	23%
NS0	10	100.2	128.0	78%
NS1	10	100.7	128.0	79%
NS2	10	100.8	128.0	79%
NS3	10	101.7	128.0	79%
RTT	9	44.7	115.2	39%
NRT	9	79.9	115.2	69%
T3T	18	82.1	230.4	36%
RTF	9	52.3	115.2	45%

9 Modifying *RETHER*: The SQO Node Servicing Policy

As mentioned above, in the process of specifying and verifying *RETHER*, we identified an alternative design of the protocol that warranted investigation because of potential efficiency gains. In this version, nodes are serviced in sequential order by their node ids, rather than in order of their RT/NRT status. We refer to this new policy as SQO. In order to ensure periodic access to the token by RT nodes, the node with the smallest node id number is the first to receive the token in each cycle. Upon receiving the token, a node will transmit its RT data if it has previously been admitted by the protocol as an RT node. It will also, during the same token visit, transmit a fixed amount of NRT data if: (1) it has NRT data to transmit, and (2) the "next" pointer, which is stored in the token and used to implement round-robin scheduling, indicates that sufficient NRT bandwidth remains in the current cycle to accommodate the node's transmission. The next-pointer is incremented when its value is i and node i has just completed an NRT transmission. Similarly, an NRT node (i.e. a node with only NRT data to transmit) transmits its data only if there is sufficient bandwidth. Otherwise, it simply passes the token to its neighbor. Note that, unlike RTF, there is no need in the SQO policy to store RT_set in the token.

To illustrate the SQO strategy, consider once again the network of Figure 6. The following sequences of node transmissions are possible in successive cycles:

(1-1) – 2 – (3-3) – (4) – 5
(1-1) – (2) – 3 – 4 – (5-5)

where a sequence item of the form (i-i) means that node i transmits both RT and NRT data during the same token visit. Also, an item of the form (i) means that i is an NRT node that transmits no data but rather just passes the token along to its neighbor. At the beginning of the first sequence, the next-pointer is set to 1, which means that the NRT data transmissions of nodes 1, 2, and 3 can be accommodated during this cycle. The value of the next-pointer will be 4 at the beginning of the second sequence, allowing nodes 4, 5, and 1 to transmit NRT data. Comparing these sequences to the corresponding ones for RTF given above, we see that the token

visits a total of five nodes per cycle in SQO, and six nodes per cycle in RTF.

In general, SQO will incur less token-passing overhead than RTF when the total number of per-cycle data transmissions (call this m) exceeds the number of nodes on the network (call this n). Moreover, the reduction in overhead increase linearly in $m - n$. If $m < n$, SQO could potentially require more token passing than RTF, depending on the relative ordering of RT and NRT nodes. In practice, though, the conditions favorable to SQO can be expected to occur frequently, and correspond to the situation where the network is not saturated by too many applications with RT demands.

9.1 VPL Specification of the Modified RETHER

The structure of the VPL specification for our modification of *RETHER* mirrors that of the VPL for the original *RETHER*. For a system containing N protocol entities the specification contains $N + 2$ processes: one for each entity, plus one, `Bandwidth`, for handling bandwidth reservations and one, `Token`, for implementing the distribution of the token. The `Bandwidth` process in the SQO model is exactly the same as the one for the RTF model, and we therefore do not discuss it further.

Process `Token` is somewhat simpler in the SQO-based specification of the protocol than in the RTF-based one. It has simpler token-passing logic—the token is passed from one node to another in sequential order by their node ids—and the token carries less information in it since it no longer stores RT_set. In particular, `tok_value` has the following four fields: `RT_count`, `NRT_count`, `next`, and `NRT_enabled`. `RT_count` and `NRT_count` are the residual number of time slots for RT and NRT data transmissions, respectively, and `next` and `NRT_enabled` are used to implement round-robin scheduling of NRT transmissions. `NRT_enabled` is set to true by `Token` if the NRT data transmission of the next node to receive the token can be accommodated during the current cycle. This is the case if the following (somewhat complicated looking) condition is met:

```
NRT_count > 0 &
( next = index
| ( index <next
    & total_NRT > N - next + index
  ))
```

Here `index` is the id of the next node to receive the token, `total_NRT` is the total number of time slots available in the current cycle for NRT transmissions, and `N` is the total number of nodes in the network. The reader is referred to the discussion of the SQO policy above for the intuition underlying this logic.

Process `Node_i` in the SQO case has a local variable `mode` indicating whether or not the node is an RT node. Similar to the RTF case, once `Node_i` receives the token, it calls `RT_action` or `NRT_action` depending on the value of `mode`. In `RT_action`, it emits signal `rt_i` and decrements `local_tok.RT_count` to simulate an RT data transmission. If `local_tok.NRT_enabled` is true, it also emits signal `nrt_i` and decrements `local_tok.NRT_count`. Moreover, it increments `local_tok.next` if `local_tok.next = i`. As in the RTF case, it can then nondeterministically choose to release its bandwidth reservation; if successful, it switches to `NRT_mode`.

The SQO version of `NRT_action` also checks `local_tok.NRT_enabled` to decide if it can perform an NRT data transmission. If it can, it behaves as does `RT_action` in this case, after which it may nondeterministically choose to request a bandwidth reservation. If this request is `granted`, `Node_i` switches to `RT_mode`.

9.2 Mu-Calculus Properties for Modified RETHER

We deem the modified *RETHER* to be correct if it satisfies all but the last of the formulas presented in Section 7. The last formula, RTF, captures the node-servicing policy of the original *RETHER* protocol; as such it describes a feature of the design of *RETHER* rather than an essential ingredient of its behavior.

10 Model Checking the Modified *RETHER* Protocol

We now summarize the results we obtained in using the Concurrency Factory's implementation of LMC to analyze *RETHER* with the SQO node-servicing policy. Table 4 presents the answers returned and the system resources consumed; as expected, the first 13 formulas indeed hold, while formula RTF is violated.

Table 5 contains the data regarding the amount of the dependency graph LMC explored in our analysis of the modified *RETHER* protocol. To understand these numbers, we first note that owing to its simpler token-passing logic, the SQO model has a smaller state space than the RTF model: there is roughly a factor of four difference between the number of the states explored by the Concurrency Factory's local model checker, as the following table makes clear.

model	size of LTS
SQO	3.2 K

Consequently, the number of transitions, the memory usage, and the execution time are all down by roughly a factor of four in the SQO model. However, there is a startling difference in the amount of the graph explored in the original and modified versions of *RETHER* when the formula RTF is considered. In particular, in the modified version only 6% of the graph was explored. The reason for this is that once a non-real-time data transmission is observed to happen before real-time data transmissions within a certain cycle, the formula becomes false, thus eliminating the need for examining any more of the graph. This phenomenon points to one of the most useful aspects of local model checking in system design; if an `AG` formula fails to hold of a system, then potentially huge parts of the dependency graph can be ignored.

Table 4. Model checking results for the SQO model.

Formula	Result	states (K)	Transitions (K)	Memory (MB)	Time (min)
DLF	True	12.0	13.7	3.0	19.9
CC	True	22.6	25.1	4.2	25.4
RT0	True	10.9	12.0	2.3	10.7
RT1	True	11.3	12.7	2.4	12.4
RT2	True	11.5	12.9	2.4	12.1
RT3	True	10.4	11.5	2.2	10.3
NS0	True	19.4	21.4	3.6	22.8
NS1	True	21.8	24.1	4.0	26.2
NS2	True	21.9	24.3	4.0	25.7
NS3	True	25.0	27.8	4.5	31.0
RTT	True	13.7	15.2	2.8	15.1
NRT	True	14.7	16.5	3.0	17.3
T3T	True	19.8	21.7	3.4	18.8
RTF	False	1.8	1.8	0.9	2.6

Table 5. Percentage of state space explored by LMC for the Modified *RETHER* protocol.

Formula	no. of vars	vertices(K) visited	vertices(K) total	percentage explored
DLF	4	12.0	12.8	94%
CC	9	22.6	28.8	78%
RT0	15	10.9	48.0	23%
RT1	15	11.3	48.0	24%
RT2	15	11.5	48.0	24%
RT3	15	10.4	48.0	22%
NS0	10	19.4	32.0	61%
NS1	10	21.8	32.0	68%
NS2	10	21.9	32.0	68%
NS3	10	25.0	32.0	78%
RTT	9	13.7	28.8	48%
NRT	9	14.7	28.8	51%
T3T	18	19.8	57.6	34%
RTF	9	1.8	28.8	6%

This can be advantageous during system design; most system requirements take the form of AG properties, and it is usually the case that one must check a number of different versions of a design before finally eliminating all the bugs. Local model checkers would perform especially well in such a scenario, as the AG properties would in general not hold of the buggy versions and the model checker could terminate early.

11 Conclusions

We have presented LMC, a local model-checking algorithm for the alternation-free modal mu-calculus. LMC works by constructing dependency graphs describing the relationships between system states and "subformulas" of the formula in question; as it constructs this graph in a demand-driven manner, only the portion of the state space necessary to determine the outcome of the verification is explored.

LMC has been implemented in the Concurrency Factory, and, using this implementation, we verified a number of key properties of the *RETHER* real-time ethernet protocol. LMC is shown to achieve significant state-space pruning on a number of formulas.

In the course of specifying and verifying *RETHER*, we also identified an alternative design of the protocol that warranted further study because of potential efficiency gains. Model checking was used to show that this alternative design also possessed the properties of interest.

The discovery of the alternative design can be regarded as an example of the positive impact formal specification and verification can have on protocol design: formal verification requires a system designer to write an abstract and rigorous specification of the system under analysis, and this often enables the designer to distinguish what is essential in a design (e.g. the use of token-passing in *RETHER*) from what is peripheral (e.g. the order of node-servicing in *RETHER*). In our case, this resulted in an alternative design that is at once potentially more efficient than the original protocol and also possesses a smaller state space.

Another point to notice is that interesting properties of real-time protocols can be verified without the use of real-time formalisms, such as timed automata [1] or real-time

logic [16]. For our *RETHER* case study, we abstracted real time into "time slots" and, in this context, it sufficed to use an untimed value-passing language (VPL) along with a non-real-time temporal logic (the modal mu-calculus [12]).

As for future work, it would be interesting to implement the SQO node-servicing policy in the FreeBSD kernel, so that its performance could be compared to the original protocol. It would also be interesting to observe its relative impact on the quality of service (e.g. jitter) in video applications.

Another research direction is to prove *RETHER* correct for an arbitrary number of nodes and cycle time slots. This will require induction-based techniques, e.g. [15], symbolic techniques for parameterized systems, e.g. [11], or more general theorem-proving techniques, e.g. [18].

References

1. R. Alur and D. Dill. The theory of timed automata. *TCS*, 126(2), 1994.
2. T. Chiueh and C. Venkatramani. The design, implementation and evaluation of a software-based real-time ethernet protocol. In *Proceedings of ACM SIGCOMM '95*, pages 27–37, 1995.
3. T. Chiueh, C. Venkatramani, and M. Vernick. Design and implementation of the Stony Brook video server. *Software—Practice and Experience*, jan 1997.
4. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.
5. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.
6. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.
7. R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory — practical tools for specification, simulation, verification, and implementation of concurrent systems. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Proceedings of DIMACS Workshop on Specification of Parallel Algorithms*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 75–90, Princeton, NJ, May 1994. American Mathematical Society.
8. R. Cleaveland and B. U. Steffen. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.
9. E. A. Emerson and J. Y. Halpern. 'Sometime' and 'not never' revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.
10. E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278, 1986.
11. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proceedings of the 9th International Conference on Computer-Aided Verification*, Haifa, Israel, July 1997. Springer-Verlag.
12. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
13. K. G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In *Proceedings of 13th Colloquium on Trees in Algebra and Programming*, volume 299 of *Lecture Notes in Computer Science*, 1988.
14. K. G. Larsen. Efficient local correctness checking. In *Proceedings of the 4th Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, 1992.
15. K. L. McMillan and R. Kurshan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.
16. A. K. Mok. Toward mechanization of real-time system design. In A. van Tilborg and G. Koob, editors, *Foundations of Real-Time Computing: Formal Specifications and Methods*, pages 1–38. Kluwer Academic Publishers, 1991.
17. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.
18. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Computer Aided Verification (CAV '95)*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liége, Belgium, July 1995. Springer-Verlag.
19. Y. S. Ramakrishna and S. A. Smolka. Partial-order reduction in the weak modal mu-calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the Eighth International Conference on Concurrency Theory (CONCUR '97)*, volume 1243 of *Lecture Notes in Computer Science*, Warsaw, Poland, July 1997. Springer-Verlag.
20. C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1), 1991.
21. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.
22. B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating boolean equation systems. In *Proceedings of ICALP'94*, pages 304–315. LNCS 820, 1994.

In the appendices all addition and subtraction operations are modulo N, the number of nodes.

A VPL Listing of *RETHER*– RTF Version

```
{======================beginning of vpl code=======================}

value N : 4 {a network of 4 nodes}
value Slots : 3 {total number of time slots}
value RTSlots : 2 {max number of RT time slots}

type count_type : N {NRT_count, RT_count and next are all <= N}

type tok_var : record
  node: array [N] of boolean; {node[i] is true if it is a RT node}
  NRT_count : count_type; {number of NRT slots left in the cycle}
  next : count_type;
  serving_rt: boolean;
end

{=====================================================================}

network rether(start: synch, cycle: synch,
                reserve0: synch, reserve1: synch,
                reserve2: synch, reserve3: synch,
                rt0: synch, rt1: synch, rt2: synch, rt3: synch,
                nrt0: synch, nrt1: synch, nrt2: synch, nrt3: synch)

begin

  process bandwidth(reserve: synch, release:synch,
                        grant: synch, no_grant: synch,
                        ok: synch, not_ok: synch,
                        RTC_chan: count_type)
  begin

    var RT_count : count_type

    RT_count :=1;  {initially only node 0 is a RT node, so RT_count = 1}
    while true do
        select
           begin
             reserve?*;
             if (RT_count < RTSlots)  then
                begin  grant!*; RT_count := RT_count+1 end
             else   no_grant!*
           end
           %
           begin
             release?*;
             if (RT_count > 1)  then
                begin ok!*; RT_count := RT_count-1 end
             else  not_ok!*
           end
           %
           RTC_chan!RT_count
        end   {select}
```

```
44    end  {while}
45  end;  {process bandwidth}

46  {-------------------------------------------------------------------}
47  process token( visit[]: tok_var,
48                 RTC_chan: count_type,
49                 tok_start: synch, tok_cycle: synch)
50  begin
51    var tok_value: tok_var
52    var RT_count: count_type
53    var index: count_type

54    tok_value.NRT_count := Slots-1;
55    tok_value.next := 0;
56    tok_value.node[0] := true;
57    tok_value.node[1] := false;
58    tok_value.node[2] := false;
59    tok_value.node[3] := false;

60    while true do
61            tok_start!*;
62            tok_value.serving_rt := true;
63            index :=0;
64            while (index < N) do
65              if (tok_value.node[index] = true) then
66                 begin visit[index]!tok_value; visit[index]?tok_value end;
67              index := index + 1
68            end;

69            {start visiting NRT nodes}
70            tok_value.serving_rt := false;
71            while ( ~(tok_value.NRT_count = 0) ) do
72                  visit[tok_value.next]!tok_value;
73                  visit[tok_value.next]?tok_value
74            end;

75            tok_cycle!*;
76            RTC_chan?RT_count;
77            tok_value.NRT_count := Slots-RT_count
78    end  {outer while}
79  end;  {process token}

80  {-------------------------------------------------------------------}

81    process node(var i: count_type, visit_i: tok_var,
82                 reserve: synch, release: synch, grant: synch,
83                 no_grant: synch,  ok: synch, not_ok: synch,
84                 reserve_i: synch, rt_i: synch, nrt_i: synch)
85    begin
86      var local_tok: tok_var

87      procedure RT_action
88      begin
89          rt_i!*;
90          select
91                  skip
92                  %
```

```
 93                 begin
 94                   release!*;
 95                   select
 96                       begin
 97                         ok?*;
 98                         local_tok.node[i] := false
 99                       end
100                       %
101                       not_ok?*
102                    end  {inner select}
103                 end {2nd choice of outer select}
104          end {outer select}
105      end; {procedure RT_action}

106      procedure NRT_action
107      begin
108                nrt_i!*;
109                local_tok.NRT_count := local_tok.NRT_count -1;
110                local_tok.next := local_tok.next+1;
111                if (local_tok.node[i] = false) then
112                begin
113                  select
114                    skip
115                    %
116                    begin
117                      reserve!*;
118                      select
119                          begin
120                            grant?*;
121                            local_tok.node[i] := true;
122                            reserve_i!*
123                          end
124                          %
125                          no_grant?*
126                        end  {inner select}
127                    end {2nd choice of outer select}
128                  end {outer select}
129                end {of if local_tok.node = false}
130      end;  {procedure NRT_action}

131    {-----------process node body-------------------}
132      while true do
133          visit_i?local_tok;
134          if (local_tok.serving_rt = true)  then
135                  call RT_action
136          else    call NRT_action;
137          visit_i!local_tok
138      end {while}
139    end;  {process node}

140  { --------------- body of  network rether --------}

141    channel reserve: synch
142    channel release: synch
143    channel grant: synch
144    channel no_grant: synch
145    channel ok: synch
```

```
146    channel not_ok: synch
147    channel RTC_chan: count_type
148    channel visit: array [N] of tok_var

149    bandwidth(reserve, release, grant, no_grant, ok, not_ok, RTC_chan)
150    | token(visit[], RTC_chan, start, cycle)
151    | node(0, visit[0], reserve, release,
152           grant, no_grant, ok, not_ok, reserve0, rt0, nrt0)
153    | node(1, visit[1], reserve, release,
154           grant, no_grant, ok, not_ok, reserve1, rt1, nrt1)
155    | node(2, visit[2], reserve, release,
156           grant, no_grant, ok, not_ok, reserve2, rt2, nrt2)
157    | node(3, visit[3], reserve, release,
158           grant, no_grant, ok, not_ok, reserve3, rt3, nrt3)

159  end; {network rether}
```

B VPL Listing of *RETHER*– SQO Version

```
 1 {=========================beginning of vpl code=======================}

 2 value N : 4 {a network of 4 nodes}
 3 value Slots : 3 {total number of time slots in each token rotation cycle}
 4 value RTSlots : 2 {max number of RT time slots}

 5 value RT_mode : 1
 6 value NRT_mode : 0

 7 type count_type : N {NRT_count, RT_count and next pointer are all <= N}

 8 type tok_var : record
 9   NRT_count : count_type; {number of NRT slots remaining in the cycle}
10   RT_count : count_type; {number of RT slots remaining in the cycle}
11   next : count_type;
12   NRT_enabled: boolean;
13        {true if during a token visit this node will
14         be allowed to transmit NRT data.}
15 end

16 {=================================================================}

17 network rether(start: synch, cycle: synch,
18                 reserve0: synch, reserve1: synch,
19                 reserve2: synch, reserve3: synch,
20                 rt0: synch, rt1: synch, rt2: synch, rt3: synch,
21                 nrt0: synch, nrt1: synch, nrt2: synch, nrt3: synch)

22 begin

23   process bandwidth(reserve: synch, release:synch,
24                         grant: synch, no_grant: synch,
25                         ok: synch, not_ok: synch,
26                         RTC_chan: count_type )
27   begin

28     var RT_count : count_type
```

```
29     RT_count := 1;  {initially node 0 is a RT node, so RT_count = 1}
30     while true do
31         select
32            begin
33              reserve?*;
34              if (RT_count < RTSlots)  then
35                 begin  grant!*; RT_count := RT_count+1 end
36              else   no_grant!*
37            end
38            %
39            begin
40              release?*;
41              if (RT_count > 1)  then
42                 begin ok!*; RT_count := RT_count-1 end
43              else  not_ok!*
44            end
45            %
46            RTC_chan!RT_count
47         end   {select}
48   end  {while}
49 end;  {process bandwidth}

50 {-------------------------------------------------------------------------}

51 process token(visit[]: tok_var,
52                RTC_chan: count_type,
53                tok_start: synch, tok_cycle: synch)
54 begin
55   var tok_value: tok_var
56   var RT_count: count_type
57   var index: count_type
58   var total_NRT: count_type

59   tok_value.NRT_count := Slots-1;
60   tok_value.RT_count := 1;
61   tok_value.next := 0;
62   tok_value.NRT_enabled := false;

63   while true do
64           index := 0;
65           tok_start!*;
66           while ((tok_value.NRT_count >0) | (tok_value.RT_count >0)) do
67             tok_value.NRT_enabled := false;
68             if (tok_value.NRT_count >0) then
69               begin
70                 if  (tok_value.next = index)
71                    then tok_value.NRT_enabled := true;
72                 if ( (index < tok_value.next) &
73                        (total_NRT > N - tok_value.next + index) )
74                 then tok_value.NRT_enabled := true
75               end;
76             visit[index]!tok_value;
77             visit[index]?tok_value;
78             index := index +1
79           end;
```

```
 80           tok_cycle!*;
 81           RTC_chan?RT_count;
 82           tok_value.NRT_count := Slots-RT_count;
 83           tok_value.RT_count := RT_count;
 84           total_NRT := tok_value.NRT_count
 85   end  {outer while}
 86 end;  {process token}

 87 {-------------------------------------------------------------------}

 88   process node(var i: count_type, visit_i: tok_var,
 89                reserve: synch, release: synch, grant: synch,
 90                no_grant: synch,  ok: synch, not_ok: synch,
 91                reserve_i: synch, rt_i: synch, nrt_i: synch)
 92   begin
 93     var local_tok: tok_var
 94     var mode: boolean {1=RT, 0=NRT}
 95
 96     procedure RT_action
 97     begin
 98         if ~(local_tok.RT_count = 0) then
 99         begin
100            rt_i!*;
101            local_tok.RT_count := local_tok.RT_count-1
102         end;
103         if (local_tok.NRT_enabled = true)
104         then begin
105                 nrt_i!*;
106                 local_tok.NRT_count := local_tok.NRT_count-1;
107                 if (local_tok.next = i)  then
108                    local_tok.next := local_tok.next+1
109                 {if next != ID, then do not move the next pointer}
110         end;
111         select
112                 skip
113                 %
114                 begin
115                   release!*;
116                   select
117                       begin
118                         ok?*;
119                         mode := NRT_mode
120                       end
121                       %
122                       begin
123                         not_ok?*
124                       end
125                    end  {inner select}
126                end {2nd choice of outer select}
127         end {outer select}
128     end; {procedure RT_action}

129     procedure NRT_action
130     begin
131       if (local_tok.NRT_enabled = true)
132         then  begin
133               nrt_i!*;
```

```
134              local_tok.NRT_count := local_tok.NRT_count-1;
135              if (local_tok.next=i) then local_tok.next:=local_tok.next+1;
136              select
137                skip
138                %
139                begin
140                    reserve!*;
141                    select
142                        begin
143                          grant?*;
144                          mode := RT_mode;
145                          reserve_i!*
146                        end
147                        %
148                        begin
149                          no_grant?*
150                        end
151                    end  {inner select}
152                end {2nd choice of outer select}
153              end {outer select}
154         end {outer if}
155     end;  {procedure NRT_action}

156   {-----------process node body--------------------}
157     if (i = 0) then mode := RT_mode  {0th  node initially RT}
158     else mode := NRT_mode;
159     while true do
160         visit_i?local_tok;
161         if (mode = RT_mode)  then
162                 call RT_action
163         else    call NRT_action;
164         visit_i!local_tok
165     end {while}
166   end;  {process node}

167 {--------------- body of  network rether --------}

168   channel reserve: synch
169   channel release: synch
170   channel grant: synch
171   channel no_grant: synch
172   channel ok: synch
173   channel not_ok: synch
174   channel RTC_chan: count_type
175   channel visit: array [N] of tok_var

176   bandwidth(reserve, release, grant, no_grant, ok, not_ok, RTC_chan)
177   | token(visit[], RTC_chan, start, cycle)
178   | node(0, visit[0], reserve, release,
179           grant, no_grant, ok, not_ok, reserve0, rt0, nrt0)
180   | node(1, visit[1], reserve, release,
181           grant, no_grant, ok, not_ok, reserve1, rt1, nrt1)
182   | node(2, visit[2], reserve, release,
183           grant, no_grant, ok, not_ok, reserve2, rt2, nrt2)
184   | node(3, visit[3], reserve, release,
185           grant, no_grant, ok, not_ok, reserve3, rt3, nrt3)
186 end;
```