Formal Verification of the Ricart-Agrawala Algorithm*

Ekaterina Sedletsky[1], Amir Pnueli[1], and Mordechai Ben-Ari[2]

[1] Department of Computer Science and Applied Mathematics,
The Weizmann Institute of Science, Rehovot 76100, Israel.
{kate|amir}@wisdom.weizmann.ac.il
[2] Department of Science Teaching,
The Weizmann Institute of Science, Rehovot 76100, Israel.
moti.ben-ari@weizmann.ac.il

Abstract. This paper presents the first formal verification of the Ricart-Agrawala algorithm [RA81] for distributed mutual exclusion of an arbitrary number of nodes. It uses the Temporal Methodology of [MP95a]. We establish both the safety property of *mutual exclusion* and the liveness property of *accessibility*. To establish these properties for an arbitrary number of nodes, parameterized proof rules are used as presented in [MP95a] (for safety) and [MP94] (for liveness). A new and efficient notation is introduced to facilitate the presentation of liveness proofs by verification diagrams.
The proofs were carried out using the Stanford Temporal Prover (STeP) [BBC+95], a software package that supports formal verification of temporal specifications of concurrent and reactive systems.

1 Introduction

The Ricart-Agrawala algorithm (RA) [RA81] for achieving mutual exclusion in a network is one of the venerable and well-known algorithms in distributed computing. Nevertheless, the correctness of the algorithm has not been formally verified.

The only previous attempt to formally prove the RA algorithm is the unpublished work [Kam95], but it is restricted to the safety property of mutual exclusion and uses a simplified model. On the other hand, already [Lamp82] presented a non-mechanized proof of a similar algorithm.

The main motivation for this work was to attempt a fully mechanized formal deductive proof of the RA algorithm, establishing both its safety and liveness properties, and using the deductive methods of [MP91].

These methods have been mechanized in a software package called the Stanford Temporal Prover (STeP) [BBC+95]. A further motivation of this work was to push STeP to its limits, and see whether it could be used to prove an algorithm whose correctness proofs are quite complex.

* This research was supported in part by the Minerva Center for Verification of Reactive Systems, and a grant from the U.S.-Israel bi-national science foundation.

S. Kapoor and S. Prasad (Eds.): FST TCS 2000, LNCS 1974, pp. 325–335, 2000.

We successfully generated formal proofs of both mutual exclusion and accessibility using STeP. This research points the way for further improvements both in proof techniques and in software support for deductive verification methods.

2 Implementation of the Ricart-Agrawala Algorithm

To verify the RA algorithm, we have written it in a formal programming notation, the language SPL which is used in [MP91] as the programming language (Figure 1).

$$
\begin{array}{lll}
\textbf{in} & N & : \textbf{integer where } N \geq 2 \\
\textbf{local} & chq, chp & : \textbf{array } [1..N, 1..N] \textbf{ of channel } [1..] \textbf{ of integer} \\
 & & \qquad\qquad \textbf{where } chq = \Lambda, chp = \Lambda \\
 & y, z & : [1..N] \textbf{ where y=1} \\
\textbf{type} & Nar & = \textbf{array } [1..N] \textbf{ of integer} \\
 & Bar & = \textbf{array } [1..N] \textbf{ of boolean} \\
\textbf{value} & mini & : Nar \times Bar \rightarrow [1..N]
\end{array}
$$

$$
\begin{array}{l}
\left[\begin{array}{l}
\textbf{local } osn, hsn, p, c : \textbf{integer where } osn = 0, hsn = 0, c = 0, p = 1 \\
\qquad\quad rcs \quad\quad : \textbf{boolean where } rcs = F \\
\qquad\quad rd \quad\quad : \textbf{array } [1..N] \textbf{ of boolean where } rd = F \\
M :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
m_1 : \textbf{noncritical} \\
m_2 : \langle rcs := T; osn := hsn + 1; c := N\text{-}1; p := 1; \\
\qquad\qquad\qquad\qquad\qquad\qquad y := mini(osn, rcs)\rangle \\
m_{31} : \textbf{while } p \leq N \textbf{ do} \\
m_{32} : \quad \langle \textbf{if } p \neq s \textbf{ then } chq[s,p] \Leftarrow osn; p := p + 1\rangle \\
m_4 : \textbf{await } c\textbf{=0} \\
m_5 : \textbf{critical} \\
m_6 : \langle rcs := F; p := 1; y := mini(osn, rcs)\rangle \\
m_{71} : \textbf{while } p \leq N \textbf{ do} \\
m_{72} : \quad \langle \textbf{if } rd[p] \textbf{ then } [rd[p] := F; chp[s,p] \Leftarrow 1]; p := p + 1\rangle
\end{array}\right]
\end{array}\right] \\
\Big\| \\
Q :: \Big\|_{t=1}^{N} \left[\begin{array}{l}
\qquad \textbf{local } rq : \textbf{integer} \\
\textbf{loop forever do} \\
q_1 : \left\langle \left[\begin{array}{l}
chq[t,s] \Rightarrow rq \\
\textbf{if } hsn < rq \textbf{ then } hsn := rq \\
\textbf{if } (rq,t) \prec (osn,s) \vee \neg rcs \\
\quad \textbf{then } chp[s,t] \Leftarrow 1 \textbf{ else } rd[t] := T
\end{array}\right] \right\rangle
\end{array}\right] \\
\Big\| \\
P :: \Big\|_{u=1}^{N} \left[\begin{array}{l}
\qquad \textbf{local } rp : \textbf{integer} \\
\textbf{loop forever do} \\
r_1 : \langle chp[u,s] \Rightarrow rp; c := c - 1\rangle
\end{array}\right]
\end{array}\right]
\end{array}
$$

$$\Big\|_{s=1}^{N} Node[s] ::$$

Fig. 1. Implementation of the RA algorithm.

The structure of the program is as follows: we assume that there are N nodes, where N is a parameter of the program which stays fixed during execution. Each node is a concurrent process: in the notation $Node[s] :: [\ldots]$, the ellipsis indicates the program text for the s'th node and s is the index of the node which may be referenced within the program text. The entire program is $\underset{s=1}{\overset{N}{||}} Node[s]$, implying a concurrent execution of all the nodes.

The nodes are connected to each other in a complete graph: there is a pair of uni-directional asynchronous channels connecting each node to every other node, where *chq* is the outgoing channel for the REQUEST messages and *chp* is the incoming channel for the REPLY messages. The notation for output is $chq[a, b] \Leftarrow e$, meaning that node a sends the value of expression e to node b along channel *chq*, and similarly, $chq[a, b] \Rightarrow x$, means that node b removes the value coming from a and assigns it to x.

The additional global declarations are discussed below.
The program for process *Node* is composed of three concurrent processes:

- M is the main process containing the critical section and the protocols to be executed upon entry and exit.
- P is the process which receives and counts replies.
- Q is the process which receives requests and decides if to reply or to defer the reply.

Note that P and Q are themselves composed of concurrent processes, one for each channel. Within $Node[s]$, process $Q[t]$ (which can also be identified as $Q[t, s]$) is responsible for reading messages from channel $chq[t, s]$. Similarly, process $P[u]$ ($P[u, s]$) is responsible for reading messages from channel $chp[u, s]$. The synchronization among the processes within the same node is based on shared variables, and we use the notation $< \ldots >$, to imply that the statements are to be executed atomically. This can be easily implemented using semaphores. We include the assignments c := N-1 and p := 1 within the atomic statement of line m_2, and p := 1 within line m_6, to reduce the number of verification conditions in the proof of accessibility.

Within each node there are global variables which are shared among the processes of that node:

- *osn* - the sequence number chosen by the node.
- *hsn* - the highest sequence number seen in any request message received by the node.
- *rcs* - a flag that is true if the node is requesting to enter the critical section.
- *c* - a counter of the number of outstanding reply messages.
- *rd* - an array that lists deferred requests. $rd[j]$ is true when the node is deferring a reply to the request from node j.
- *p*, *rp*, *rq* are auxiliary variables and could have been declared as local to the processes of the node.

The following variables are not needed by the algorithm; they were added to facilitate the proof.

- z is the index of a generic node, which is used to specify and verify accessibility.
- y is the index of the node with the minimal value of the rank $(osn[i], i)$, where the minimum is taken over all nodes i such that $rcs[i]$ is true. If $rcs[i]$ are all false, $y = 1$.
- $mini(osn, rcs)$ is a function that computes y, the index of the node with the minimal rank.

3 Proof of the Mutual Exclusion Property

Invariance properties of the form $\Box\, p$, where p is an *assertion* (a *state formula*) can be verified by the *invariance rule* B-INV, given by

Rule B-INV	I1. $\Theta \rightarrow \varphi$ I2. $\varphi \wedge \rho_\tau \rightarrow \varphi'$ $\Box\, p$	For every transition $\tau \in \mathcal{T}$

where Θ is the initial condition and $\mathcal{T}$ is the set of transitions of the verified system. An assertion satisfying premises I1 and I2 of rule B-INV is called *inductive*.

In our case, the main invariance property is that of *Mutual exclusion*, which can be specified as

PROPERTY **excl**: $\Box\, \forall i, j : [1..N] : m_5[i] \wedge m_5[j] \rightarrow i = j$

Here and below, we use $m_5[i]$ to denote that process $M[i]$ is currently executing at location m_5.

3.1 Bottom Up Assertions

At first we use a *bottom-up* approach to deduce some simple properties of the program.

Locations at which $rcs = 1$ A first observed property is

PROPERTY **rcs_range**: $\Box(m_{31,32,4,5,6}[i] \leftrightarrow rcs[i])$.

Note, that whenever there is a free index, such as i in the above property, there is an implicit universal quantification, implying that the property holds for every $i \in [1..N]$.

Range of $p[i]$ The variable p serves as a loop counter for the loops at statements m_{31} and m_{71}. The upper limit of these two loops is N.

PROPERTY **p_range**: $\Box(1 \leq p[i] \leq N + 1 - m_{32,72}[i])$

This inductive assertion claims that $p[i] \leq N + 1$ at all locations, except for locations m_{32} and m_{72}, where the stronger inequality $p[i] \leq N$ holds.

The Message Chain Linkage

PROPERTY **message_chain**:
$\Box((m_{31,32}[i] \wedge p[i] > j) + m_4[i] \;\geq\; |chq[i,j]| + rd[j,i] + |chp[j,i]|)$

Here, $|chq[i,j]|$ and $|chp[j,i]|$ denote the sizes of the buffers of these asynchronous channels. This property states that the sum $|chq[i,j]| + rd[j,i] + |chp[j,i]|$ never exceeds 1, and can be positive only if process $M[i]$ is at location m_4 or at locations $m_{31,32}$ with $p[i] > j$.

The Reply Counter The role of the counter $c[i]$ is to count the number of positive replies $Node[i]$ received since it last sent out requests for entering the critical section. We would expect that, at any point, the value of $c[i]$ will equal the number of pending replies. This is stated by

PROPERTY **c_range**: $\Box(c[i] \;=$
$$\sum_{k=1}^{N} |chq[i,k]| + rd[k,i] + |chp[k,i]|) + m_{31,32}[i] \cdot (N - p[i] + (p[i] > i))$$

Neither of properties **message_chain** or **c_range** is inductive by itself. However, their conjunction, to which we refer as **msg_range_coun- ter**, together with the previously established property **p_range** form an inductive assertion.

The Value of a Request Message As the last bottom-up invariant, we formulate the following property:

PROPERTY **request_in_channel**:
$$\Box(|chq[i,j]| > 0 \quad \rightarrow \quad head(chq[i,j]) = osn[i])$$

This property states that if channel $chq[i,j]$ is not empty, then the value it contains is the *current* value of $osn[i]$.

3.2 Top Down Assertions

We now move to a set of assertions which are derived based on the goal we wish to prove, namely the property of mutual exclusion.

We start by introducing some definitions:

$$\begin{array}{ll} requested(i,j): & i \neq j \;\wedge\; (m_{4,5}[i] \;\vee\; (m_{31,32}[i] \wedge p[i] > j)) \\ request\_received(i,j): & requested(i,j) \;\wedge\; |chq[i,j]| = 0 \\ granted(i,j): & request\_received(i,j) \;\wedge\; \neg rd[j,i] \end{array}$$

Variable *hsn* Retains the Highest Message Number Seen So Far The following property states that, after having read the recent message from $Node[i]$, the variable $hsn[j]$ ("highest seen") has a value which is not lower than $osn[i]$.

PROPERTY **hsn_highest**:
$\Box(request\_received(i,j) \rightarrow osn[i] \leq hsn[j])$

The Implication of *Node*[*j*] Granting Permission to *Node*[*i*] The following property describes the implications of a situation in which $Node[j]$ has granted an entry permission to $Node[i]$ before $Node[i]$ exited its critical section:

PROPERTY **permitted**:
$\Box(granted(i,j) \rightarrow \neg rcs[j] \ \vee \ (osn[i],i) \prec (osn[j],j))$

Finally, Mutual Exclusion Finally, we establish the property of mutual exclusion, specified by

PROPERTY **excl**: $\Box(m_5[i] \ \wedge \ m_5[j] \ \rightarrow \ i = j)$

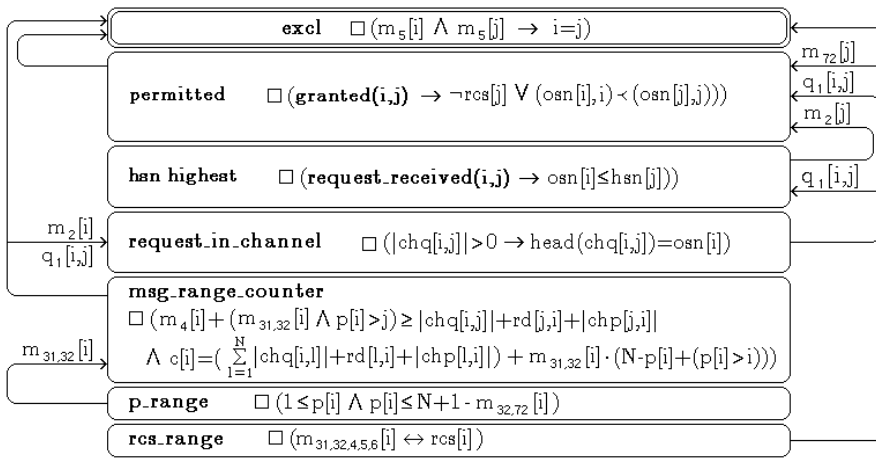

Fig. 2. Set of inductive properties leading to the proof of Mutual Exclusion. The labels on the dependence edges identify the transitions for which the verification of the higher placed property depends on the lower property.

4 A Proof Rule for Accessibility

Rule P-WELL For assertions p and $q = \varphi_0, \varphi_1[k], \ldots, \varphi_m[k]$,
transitions $\tau_1[k], ..., \tau_m[k] \in \mathcal{J}$
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_0, \delta_1[k], \ldots, \delta_m[k] : \Sigma \mapsto \mathcal{A}$

$W1.$ $p \rightarrow \bigvee_{j=0}^{m} \exists k : \varphi_j[k]$

For $i = 1..m$

$W2.$ $\rho_\tau[k] \wedge \varphi_i[k] \rightarrow \bigvee_{j=0}^{m} \exists u : (\varphi'_j[u] \wedge \delta_i[k] \succ \delta'_j[u]) \vee (\varphi'_i[k] \wedge \delta_i[k] = \delta'_i[k])$
for every $\tau \in \mathcal{T}$

$W3.$ $\rho_{\tau_i}[k] \wedge \varphi_i[k] \rightarrow \bigvee_{j=0}^{m} \exists u : (\varphi'_j[u] \wedge \delta_i[k] \succ \delta'_j[u])$

$W4.$ $\varphi_i[k] \rightarrow En(\tau_i[k])$

$p \Longrightarrow \diamondsuit q$

To verify liveness properties of parameterized programs, we can use a fixed number of intermediate formulas and helpful transitions but they may refer to an additional parameter k which is a process index. A parameterized rule for proving accessibility properties of parameterized systems has been presented in [MP95b]. However, to verify a complicated system such as the RA algorithm, it was necessary to introduce a new version of this rule, which we present here.

To improve readability of formulas, we write $\rho_{\tau_i[k]}$ as $\rho_{\tau_i}[k]$. Rule P-WELL uses parameterized intermediate assertion, parameterized helpful transitions, and parameterized ranking functions. For each $i = 1, ...m$, the parameter k in $\varphi_i[k]$, $\tau_i[k]$, and $\delta_i[k]$ ranges over some nonempty set, such as $[1..N]$.

The rule traces the progress of computations from an arbitrary p-state to an unavoidable q-state. With each non-terminal assertion φ_i, $i > 0$, the rule associates a *helpful transition* τ_i, such that the system is just (weakly fair) with respect to τ_i. Premise W2 requires that the application of any transition τ to a state satisfying a non-terminal assertion φ_i will never cause the rank of the state to increase. Premise W3 requires that if the applied transition is helpful for φ_i then the rank must decrease. Due to the well-foundedness of the ranking functions, we cannot have an infinite chain of helpful transitions, since this will cause the rank to decrease infinitely often. Premise W4 stipulates that the helpful transition τ_i is always enabled on every φ_i-state. Thus, we cannot have an infinite computation (which must be fair) which avoids reaching a $q = \varphi_0$ state.

4.1 Representation by Diagrams

The paper [MP94], introduced the graphical notation of *verification diagrams*. For our application here, verification diagrams can be viewed as a concise and optimized presentation of the components appearing in rule P-WELL. We refer the

reader to Figure 3 for explanation of some of the main elements which typically appear in such diagrams.

The diagram contains a node for each assertion φ_i that appears in the rule. The helpful transition tau_i associated with φ_i is identified by the label of one or more directed edges departing from the node (labeled by) φ_i. Thus, in the diagram of Figure 3, $m_2[z]$ is identified as the transition helpful for assertion φ_{13}, and the helpful transition for φ_{10} is $q_1[z, i]$ even though it labels two edges departing from φ_{13}.

The interconnection topology in the diagram provides a more specialized (and efficient) version of the P-WELL rule. For a node φ_i, let $succ(i)$ be the set of indices of the nodes which are the targets of edges departing for φ_i. Then the diagram suggests that, instead of proving premises W2 and W3 as they appear in the rule, it is sufficient to prove their following stronger versions:

$$U2.\ \ \rho_\tau[k] \wedge \varphi_i[k] \rightarrow \varphi'_i[k] \wedge \delta_i[k] = \delta'_i[k] \qquad \text{for every } \tau \in \mathcal{T}$$

$$U3.\ \rho_{\tau_i}[k] \wedge \varphi_i[k] \rightarrow \bigvee_{j \in succ(i)} \exists u : (\varphi'_j[u] \wedge \delta_i[k] \succ \delta'_j[u])$$

It is not difficult to see that U2 and U3 imply W2 and W3. For example, premise U3 for assertion φ_6 as implied by the diagram is

$$\varphi_6[i] \wedge \rho_{m_{72}}[i] \ \rightarrow\ (\varphi'_7[i] \wedge \delta_6[i] \succ \delta'_7[i]) \ \vee\ (\varphi'_4[i] \wedge \delta_6[i] \succ \delta'_4[i])$$

The more general notion of verification diagrams as presented in [MP94] admits two types of edges, one corresponding to the helpful transitions, which are the edges present in our diagram. The other type corresponds to unhelpful transitions. It is suggested there to use a double line for helpful edges. In our case, we only need to represent helpful transitions, so we draw them as single lines.

The rule also requires to associate with each non-terminal assertion φ_i a ranking function δ_i. By convention, whenever a ranking function is not explicitly defined within a node φ_i, the default value is the index of the node, i.e. $\delta_i = i$. For example, in the diagram, $\delta_{13} = 13$. However, as we will see below, this is not the end of the story.

4.2 Encapsulation Conventions

The diagram of Figure 3 contains, in addition to *basic nodes* such as those labeled by assertions, also *compound nodes* which are also called *blocks*. We may refer to compound nodes by the set of basic nodes they contain. For example, the successor of node φ_{13} is the compound node $\varphi_{31,32}$. Compound nodes may be annotated by λ-declarations, such as the compound node $\varphi_{4..7}$, by additional assertions, such as $m_4[y]$ for block $\varphi_{3..7}$, or ranking components, such as $(6, -p[i])$ for block $\varphi_{6,7}$. There are several encapsulation conventions associated with compound nodes.

– An edge stopping at the boundary of a block, is equivalent to individual edges which reach the basic nodes contained in the block. Thus, both φ_{11} and φ_{12} are immediate successors of node φ_{13}.

– For each basic node φ_i, the full assertion associated with this node is the conjunction of all the assertions labeling the blocks in which the basic node is contained. We denote this full assertion by $\widehat{\varphi}_i$. For example,

$$\widehat{\varphi}_7 \quad = \quad m_4[z] \ \wedge \ (\forall j : |chq[z,j]| = 0) \ \wedge \ m_4[y] \ \wedge \ rd[i,y] \ \wedge \ m_{71}[i]$$

– For each basic node φ_i, the full ranking function associated with this node is the left-to-right concatenation of all the ranking components labeling the blocks in which the basic node is contained. As the rightmost component, we add i. For example,

$$\delta_7 \quad = \quad (1, -osn[y], -y, 4, c[y], 6, -p[i], 7)$$

In Figure 3 we present the full ranking functions for each of the nodes alongside the diagram.

Note that whenever we have to compare ranking functions which are lexicographic tuples of different lengths, we add zeroes to the right of the shorter one. For example, to see that $\delta_{13} \succ \delta_{12}$, we confirm that $(13, 0, 0) \succ (11, -p[z], 12)$.

Note also that several components of the ranking functions are negative. When STeP is presented with any ranking function, one of the proof obligations which are generated require proving that all components are bounded from below. This has been done for all the components present in the diagram.

5 Proof of Accessibility Property

The property of accessibility may be written in the form

PROPERTY **m2m5**: $m_2[z] \Rightarrow \diamondsuit\, m_5[z]$

where $z \in [1..N]$.

5.1 Auxiliary Assertions Needed for the Proof

A crucial part in the proof is the computation of the index of the process y with minimal signature. We define

$$ismin(osn, rcs, y) : \left(\begin{array}{l} y = 1 \ \wedge \ \forall j : \neg rcs[j] \\ \vee \ rcs[y] \ \wedge \ \forall j : (rcs[j] \rightarrow (osn[y], y) \prec (osn[j], j)) \end{array} \right)$$

Thus, y has a minimal signature, if either there is no process j with $rcs[j]$ and then $y = 1$, or there exists some j with $rcs[j] = 1$ and y is such a j with the minimal signature. In fact, rather then defining the function *mini* explicitly, we inform STeP of the following axiom:

AXIOM **mini:** $ismin(osn, rcs, mini(osn, rcs))$.

There were several auxiliary assertions whose invariance was necessary in order to establish the proof obligations generated by STeP, when being presented by the verification diagram of Figure 3. We list them below:

rd_osn : $rd[i,j] \rightarrow (osn[i],i) \prec (osn[j],j)$
not_rd_range : $m_{1,2}[i] \vee (m_{71,72}[i] \wedge p[i] > j) \rightarrow \neg rd[i,j]$
y_eq_mini : $y = mini(osn, rcs)$
y_is_min : $ismin(osn, rcs, y)$
rd_to_y : $rd[i,y] \rightarrow m_{71,72}[i] \wedge p[i] \leq y$
y_not_change : $(\forall j : |chq[z,j]| = 0) \wedge m_4[z] \wedge m_2[s] \rightarrow (osn[y],y) \prec (hsn[s]+1, s)$

The last property **y_not_change** is very crucial in order to establish that y can stop being minimal only by retiring on exit from $m_6[y]$. In particular, no newcomer s can execute transition $m_2[s]$ and become minimal.

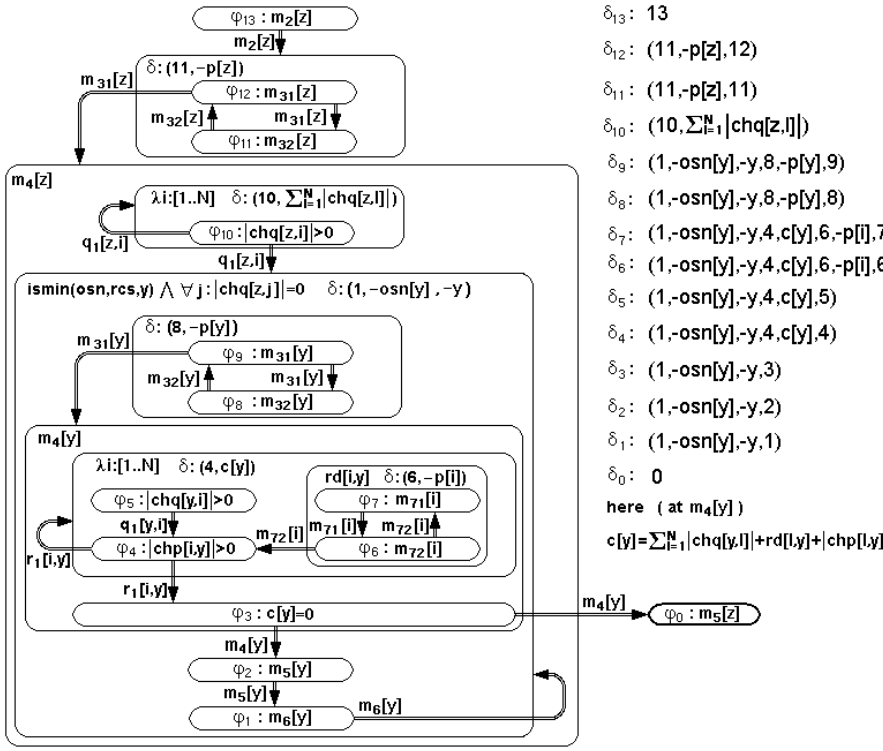

Fig. 3. Verification Diagram for the property $m_2[z] \Rightarrow \diamondsuit\, m_5[z]$.

5.2 Usage of STeP in the Proof

We used STeP version 1.4 from 2/XI/1999 in our proof. Some modifications of the source program were necessary in order for STeP to accept our SPL program. This version of STeP also fails to support *lambda*-blocks in the way there were used in Figure 3. To overcome this difficulty, we had to feed STeP with some processed fragments of this diagram and then modify manually some of the resulting verification conditions. We hope that future versions of STeP will provide direct support of *lambda*-blocks.

In spite of these minor inconveniences, we found STeP to be a very powerful and useful verification system, specially geared to the temporal verification of complex algorithms such as the Ricart Agrawala algorithm we considered here.

Acknowledgment

We gratefully acknowledge the help of Nikolaj Bjørner of the STeP research team for his advice and for his fast response to our requests for modifications in STeP.

References

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice-Hall International, Hemel Hempstead, 1990.

[BBC+95] N. Bjørner, I.A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[Kam95] J. Kamerer. Ricart and Agrawala's algorithm. Unpublished, http://rodin.stanford.edu/case-studies, 9 August 1995.

[Lamp82] L. Lamport *An Assertional Correctness Proof of Distributed Program.* Science of Computer Programming, 2, 3, December 1982, pages 175–206.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lect. Notes in Comp. Sci.*, pages 726–765. Springer-Verlag, 1994.

[MP95a] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MP95b] Z. Manna and A. Pnueli. Verification of parameterized programs. In E. Börger, editor, *Specification and Validation Methods*, pages 167–230. Oxford University Press, Oxford, 1995.

[RA81] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Comm. ACM*, 24(1):9–17, 1981. Corr. ibid. 1981, p.581.