Implementation and verification of ring termination
detection protocols using structured rv-programs

Cezara Drăgoi and Gheorghe Ştefănescu

Faculty of Mathematics and Computer Science, University of Bucharest
Str. Academiei 14, Bucharest, Romania 010014
Email: {cdragoi,gheorghe}@funinf.cs.unibuc.ro

Abstract

A model (consisting of rv-systems), a core programming language (for developing rv-programs), several specification and analysis techniques appropriate for modeling, programming and reasoning about interactive computing systems have been introduced by Stefanescu in 2004 using register machines and space-time duality, see [8]. In a couple of papers [3, 4] the authors have introduced structured programming techniques for rv-systems and have developed a Hoare-like verification logic.

The dual-pass ring termination detection protocol is used to detect the termination of a pool of processes, logically organized into a ring. In this paper we develop an implementation for the core activity of the processes and for their interactions using structured rv-programs. Then, we present a formal proof of the correctness of the protocol using our Hoare-like logic for structured rv-programs. A brief study of variations of the protocol is included, as well.

Keywords: ring termination detection, structured rv-systems, formal verification, Hoare logic, interactive systems, registers and voices

1 Introduction

A model (consisting of rv-systems), a core programming language (for developing rv-programs), several specification and analysis techniques appropriate for modeling, programming and reasoning about interactive computing systems have been introduced by Stefanescu in 2004 using register machines and space-time duality, see [8]. In a couple of papers [3, 4] the
authors have introduced structured programming techniques for rv-systems and have developed a Hoare-like verification logic.

Termination detection is quite a popular research topic in distributed systems. The aim is to find when a pool of distributed processes have terminated. There are many termination detection protocols, including those presented in [2, 6, 9, 7] (search the web for more information). The problem is particularly compound as one has to combine a local termination condition (each process has finished its current jobs) with a global condition (no messages are in transit, as such messages may reactivate already terminated processes).

In this paper we develop an implementation for a particular dual-pass ring termination detection protocol using structured rv-programs. Then, we present a formal proof of the correctness of the protocol using our Hoare-like logic for structured rv-programs [4].

2 The problem and its solution

In this section we briefly present the termination detection protocol to be implemented and verified, the implementation language, and the verification logic.

Ring termination detection protocols The dual-pass ring termination detection protocol is used to detect the termination of a pool of processes, logically organized into a ring. It can handle the case when processes may be reactivated after their local termination. To this end, it uses colored (i.e., black or white) tokens. Processes are also colored: a black color means global termination may have not occurred. Then, the algorithm works as follows:

- The root process $P_0$ becomes white when it has terminated and it generates a white token that is passed to $P_1$.
- The token is passed through the ring from one process $P_i$ to the next when $P_i$ has terminated. However, the color of the token may changed. If a process $P_i$ passes a task to a process $P_j$ with $j < i$, then it becomes a black process; otherwise it is a white process. A black process will pass on a black token, while a white process will pass on the token in its original color. After $P_i$ has passed on a token, it becomes a white process.
• When $P_0$ receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

An optimized version of the algorithm may be introduced, where a process which sends a job back becomes black only in the case the destination process has already terminated. In this case, additional termination information has to be passed through the network.

Structured rv-programs The language used to implement these termination detection protocols is provided by structured interactive rv-programs introduced in [3]. These programs are written in a structured, high level programming language based on the interactive rv-programs [8]. An example of a structured rv-program is the program $P$ presented in the next section. Here, we briefly touch on the key features of the language, with explicit reference to program $P$.

Structured rv-programs inherit the modules of the rv-programs as their basic blocks. Such a module has both a spatial and a temporal interface. The spatial interface is specified using registers, while the temporal interface is specified using voices, usually implemented on streams. A module has explicit read/write and listen/speak statements for its registers and voices. An example of module is $R$ in program $P$.

The structuring programming operations for rv-programs extend the classical structuring programming operations. Both, composition and iterated composition (while) have extensions to rv-programs which exploit the multiple possibilities to compose the blocks: spatial composition via registers; temporal composition via voices; it is also possible to compose the blocks on the diagonal (a “spatio-temporal composition”) and in this case, both, the output registers and voices of a block become the input registers and voices of the next block. Spatial, temporal, and diagonal compositions are denoted by ;, #, and $\$\$, respectively. The iterated versions are introduced using the spatial, temporal, and diagonal while statements, denoted by while_s, while_t, and while_st, respectively. Occurrences of most of these statements may be found in the program $P$, in the next section.

Hoare logic for structured rv-programs Hoare logic was developed to verify classical structured programs. It is based on Hoare triples $\{\phi\} P \{\psi\}$, where $\phi$ is an assertion on the input states of the program $P$ and $\psi$ is an assertion on the output states of $P$. There is a basic rule for the assignment statement: it is based on substitution to describe $\psi$ in terms of the input variables of $P$ and then to check if the resulted assertion follows from $\phi$. 
This situation is then extended to larger programs using inference rules for composition, if, and while, till one get a Hoare triple for the full program. See, e.g., [5] for a presentation of Hoare logic.

The extension of Hoare logic to structured rv-programs is difficult. It is rather straightforward to lift Hoare logic to programs which use either the spatial or the temporal interfaces, but not both. The extension to the spatio-temporal while or composition is also easy. The difficult part is when the program heavily use both interfaces. For instance, if for a usual temporal while in a statement\( \textbf{while} \ t(B) \{ \textbf{S} \} \) the body \( \textbf{S} \) is an open module with a nontrivial input/output temporal interface, then the rule has to capture the situation on these temporal interfaces, too. In such a case, if the control part of the while is complicate, then it may be rather difficult to come up with appropriate assertions. To alleviate this situation, we are using a simple and restricted \textbf{for} statement, where the number of iterations is a priori known. This is all we need to model ring termination detection protocols.

Later in the paper, we develop an implementation for the core activity of the processes and for their interactions as specified by the dual-pass ring termination detection protocol using structured rv-programs. Then, we present a formal proof of the correctness\(^2\) of the protocol using our Hoare-like logic for structured rv-programs described in [4].

### 3 Implementation

We describe a structured rv-program \( P \) that implements a dual pass ring termination detection algorithm for a network of distributed processes.

**The structured rv-program:** Suppose there are \( n \) processes, denoted \( 0, \ldots, n-1 \). Besides the input \( n \), the program uses the spatial variables \( \text{id} : \text{sInt}, \text{c} : \{ \text{white, black} \}, \text{active} : \text{sBool} \) and the temporal variables \( \text{tn}, \text{tid} : \text{tInt}, \text{msg} : \text{tIntSet}[\].

The program \( P \) is the diagonal composition of an initialization program \( I \) and a core program \( Q \),

\[
P = I \$ Q
\]

\(^1\)The correctness is usually split into the partial correctness part and the termination part.

\(^2\)Notice that only the “partial correctness” part may be proved: If the root receives the white token, then the pool of processes have terminated. It is however possible that the pool of processes never terminates.
where

\[
I = I_1# \text{ for } s(\text{tid}=0; \text{tid}<\text{tn}; \text{tid}++)\{I_2\}#
\]

\[
I_1 = \{ \text{read } n;
\text{tn}=n; \text{token.col}=\text{black}; \text{token.pos}=0;
\text{speak } \text{tn}, \text{tid}, \text{msg[ ]}, \text{token(col,pos)}; \}
\]

\[
I_2 = \{ \text{listen } \text{tn}, \text{tid}, \text{msg[ ]}, \text{token(col,pos)};
\text{id}=\text{tid}; \text{c}=\text{white}; \text{active}=\text{true}; \text{msg[id]}=\text{null};
\text{write } \text{id}, \text{c}, \text{active}; \text{speak } \text{tn}, \text{tid}, \text{msg[ ]}, \text{token(col,pos)}; \}
\]

\[
Q = \text{while } s(\text{! (token.col==white & token.pos==0))} \{
\text{for } s(\text{tid}=0; \text{tid}<\text{tn}; \text{tid}++)\{R\}\}
\]

\[
R = \{ \text{read } \text{id}, \text{c}, \text{active}; \text{listen } \text{tn}, \text{tid}, \text{msg[ ]}, \text{token(col,pos)};
\text{if(msg[id]}=\text{emptyset}) \{ \text{//take my jobs}
\text{msg[id]}=\text{emptyset};
\text{active}=\text{true}; \}
\text{if( active) } \{ \text{//execute code, send jobs, update color}
\text{delay( random_time)};
\text{r}=\text{random( tn-1)};
\text{for( i=0; i<r; i++)} \{ \text{k}=\text{random( tn-1)};
\text{if( k!=id)} \{ \text{msg[k]}=\text{msg[k]}∪\{id\}; \}
\text{if( k<id)} \{ \text{c}=\text{black}; \}
\text{active= random( true, false)}; \} \}
\text{if( !active & token.pos==id)} \{ \text{//termination}
\text{if( id==0) token.col=white;}
\text{if( id!=0 & c=black)} \{ \text{token.col=black; c=white};
\text{token.pos=token.pos+1[mod tn]}; \}
\text{write } \text{id}, \text{c}, \text{active}; \text{speak } \text{tn}, \text{tid}, \text{msg[ ]}, \text{token(col,pos)}; \}
\]

**Comments:** The spatial variables \text{id}, \text{c}, \text{active} represent the process identity, its color, and its active/passive status. The temporal variables used in this program are: (i) \text{tn, tid} - temporal versions of \text{n, id}; (ii) \text{msg[ ]} - an array of sets, where \text{msg[k]} contains the \text{id} of the destination processes for the pending messages sent by process \text{k}; (iii) \text{token.col} - an element of \{\text{white}, \text{black}\} representing the color of the token; and (iv) \text{token.pos} - the number of the process that has the token.

The program starts with the initialization of the network (program I) by activating all the processes (and setting the fields \text{id, c, active}). Initially,
msg[i] = ∅, for all 0 ≤ i < n, because no jobs were sent and the default color/position of the token is black/0.

After the initialization part and until the first process receives a white token back, each process executes its code. If one process has the token and terminates, it passes the token to the next process (only the first process has the right to change the color of the token into white once it terminates).

When a process executes the code R, whether active or passive, it checks if new jobs were assigned to it; if the answer is positive, it collects its jobs from the jobs lists and stays/becomes active. When it is active, it executes some code, sends new jobs to other processes, and randomly goes to an active or passive state. If it has the token, it keeps it until it reaches termination and afterward it passes it. A white process will pass the token with the same color as it was received and a black process will pass a black token (after passing the token, the process becomes white).

Variations  By passing the active/passive status of the processes together with the msg list, one can design a more efficient termination detection protocol by turning to black a process i which send a job back to a process j with j < i only if process j already terminated. The code may easily be modified to implement this version.

4 Verification

The program P is the diagonal composition of the initialization block and the repeated diagonal compositions given by the while statement. In each case, the temporal/spatial output of a block becomes the temporal/spatial input of the next block.

For I, the input is a spatial variable n. The output satisfies the condition:

$$\forall k \in [0, n) : (id, c, active)[k] = (k, white, true)$$

$$\land tn = n \land token = (black, 0) \land \forall k \in [0, n) : msg[k] = \emptyset.$$

Notice that the spatial interface is expanded on n processes 0, 1, ..., n-1. We use the notation (id,c,active)[k] to refer the the values of variables (id,c,active) in process k.

The invariant Inv:  For Q we need to find appropriate invariant properties. We define the following properties and prove they are satisfied by for s(tid=0;tid<tn;tid++) {R}:
P1: \( \text{token} = (\text{white}, i) \) \[\forall r \in [0, i-1]: \text{active}[r] = \text{false} \land \text{msg}[r] = \emptyset \]
\[\lor (\exists k > i-1: c[k] = \text{black})\]
where the value \( i - 1 \) is interpreted as \( tn - 1 \) for \( i = 0 \).

In words, if the token is white and reached process \( i \), then all processes with smaller id terminate and have no pending messages sent\(^3\) or a process with a larger id is black.

P2 \( \text{token}.\text{col} = \text{white} \) \[\forall k \in [0, n): \text{msg}[k] \neq \emptyset \rightarrow c[k] = \text{black}\]

In words, if a process has a job inserted in the pending message list, then its color is black.

We want to prove \( \text{Inv} = \text{P1} \land \text{P2} \) is really an invariant, i.e., the same assertion \( \text{Inv} \), translated to the output values of the variables, holds at the end of the \textbf{for} statement. Formally,

\[
\{\text{Inv}\}\textbf{for}(\text{tid}=0;\text{tid}<\text{tn};\text{tid}++)\{\text{R}\}\{\text{Inv}\}.
\]

Notice that due to the fact that the token is black, \( \text{Inv} \) holds at the beginning of the spatio-temporal while.

**Proof of the invariance of \( \text{Inv} \):** Suppose \( \text{Inv} \) holds at the start of the \textbf{for} statement. We want to prove that the property \( \text{Inv}' = \text{P1}' \land \text{P2}' \), where \( \text{Inv}' \) is just \( \text{Inv} \) translated to the output values of the variables\(^4\), holds at the end of the \textbf{for} statement.

First, we prove \( \text{P1}' \), where

\[ \text{P1'}: \text{token}' = (\text{white}, i') \rightarrow \]
\[\forall r \in [0, i'-1]: \text{active}'[r] = \text{false} \land \text{msg}'[r] = \emptyset \]
\[\lor (\exists k > i'-1: c'[k] = \text{black})\]

Suppose \( \text{token}'.\text{col} = \text{white} \); then \( \text{token}.\text{col} = \text{white} \), too. Notice that \( \forall r \in [i, i'-1]: \text{active}'[r] = \text{false} \land \text{msg}'[r] = \emptyset \) holds because: (i) the token could not reach the process \( i' \) unless processes \( i, \ldots, i'-1 \) hadn’t terminated and (2) \( \text{token}'.\text{col} \) hadn’t been white unless \( \text{msg}'[i], \ldots, \text{msg}'[i'-1] \) are all empty.

As \( \text{P1} \) holds and \( \text{token} = (\text{white}, i) \), either (i) or (ii) apply, where:

---

\(^3\)The pending message lists are the lists of messages that have been inserted in and not removed from the message lists during a complete passing through the ring. Formally, they are \( \text{msg}[r]'s \) at the start of the \textbf{for} statement.

\(^4\)We use the standard “prim” notation, i.e., if \( x \) is a variable, then \( x' \) refers to the value of the variable \( x \) at the end of the program.
\( \forall r \in [0, i-1] : \text{active}[r] = false \land \text{msg}[r] = \emptyset \): In this case:

(a) If all processes \(0, \ldots, i-1\) stay passive, then by the above observation this situation is extended to \( \forall r \in [0, i' - 1] : \text{active}'[r] = false \land \text{msg}'[r] = \emptyset \) and we are done.

(b) If one process \(0, \ldots, i-1\) becomes active, it may be reactivated only by a message from a process \( k \) with \( k > i - 1 \) (indeed, \( \text{msg}[0], \ldots, \text{msg}[i-1] \) are all empty). Then, by \( P2, c[k] = \text{black} \). Moreover \( k > i' - 1 \) (otherwise \( \text{token'.col} \) hadn’t been white), hence \( c'[k] = \text{black} \) and the second part is true.

\( \exists k > i - 1 : c[k] = \text{black} \): In this case, \( k > i' - 1 \) (otherwise \( \text{token'.col} \) hadn’t been white) and \( c'[k] = \text{black} \), hence the implication holds.

Next, we prove \( P2' \), where

\( P2' : \text{token'.col} = \text{white} \rightarrow (\forall k \in [0, n)) : \text{msg}'[k] \neq \emptyset \rightarrow c'[k] = \text{black} \)

Notice that after the execution of \( R \) by the process \( k \), \( \text{msg}'[k] \) consists in the processes that were contacted by \( k \). The execution of \( R \) for \( tid = k \) is followed by the execution of \( R \) for \( k < tid < tn \). All these executions of \( R \) that follows, will erase all the processes greater than \( k \), from \( \text{msg}'[k] \) and consequently, by the end of the \( \text{for.s} \), \( \text{msg}'[k] \subseteq [0, k] \).

Hence, if \( \text{msg}'[k] \neq \emptyset \), then the process \( k \) had sent a message to a process \( p \) with \( p < k \) and the color of the process became black. Moreover, if \( \text{token'.col} = \text{white} \), then the color of the process stayed black until the end of the \( \text{for.s} \) instruction, which is equivalent to \( c'[k] = \text{black} \).

The final step: Further applying the rule for the spatio-temporal while

\( \{\text{Inv}\}\text{while st(! (token=(white, 0))) \{Q'\}}\{\text{Inv} \land (token = (white, 0))\} \)

where \( Q' = \text{for.}(\text{tid}=0; \text{tid}<\text{tn}; \text{tid}++) \{R\} \), it follows that

\( \forall i \in [0, tn - 1] : \text{active}[i] = false \land \text{msg}[i] = \emptyset \)

hence all process have terminated and there are no pending jobs/messages in the communication lists.

This concludes the correctness proof for our implementation of the dual-pass ring termination detection protocol.
A more formal proof: In order to come up with a more formal verification proof, we need a more formal proof for

\[ \{ \|Inv\| \} \text{for}_s(\text{tid}=0;\text{tid}<\text{tn};\text{tid}++) \{ R \} \{ \|Inv\| \} \]

For this, one may introduce a few more detailed assertions, which essentially depend on \( \text{tid} \) and describe the effect of the computation within \( R \):

**Q1**: (case \( \text{token.col} = \text{white} \land \text{tid} < \text{token.pos} \))

\[
\begin{align*}
\text{token}' &= \text{token} \land \\
\text{active}[\text{tid}] &= \text{false} \land \exists k, \text{tid} \in \text{msg}[k].\text{list} \rightarrow \\
\text{active}'[\text{tid}] &= \text{false} \land \text{msg}' = \text{msg} \\
\text{active}[\text{tid}] &= \text{true} \lor \exists k, \text{tid} \in \text{msg}[k].\text{list} \rightarrow \\
\forall k \neq \text{tid} : \text{msg}'[k].\text{list} &= \text{msg}[k].\text{list} - \{\text{tid}\} \\
\land \text{msg}'[\text{tid}].\text{list} \cap [0, \text{tid}) \neq \emptyset &\rightarrow c'[\text{tid}] = \text{black}
\end{align*}
\]

**Q2**: (case \( \text{token.col} = \text{white} \land \text{tid} = \text{token.pos} \))

\[
\begin{align*}
\forall k \neq \text{tid} : \text{msg}'[k].\text{list} &= \text{msg}[k].\text{list} - \{\text{tid}\} \land \\
\text{active}'[\text{tid}] &= \text{true} \land \text{token}' = \text{token} \\
\land \text{msg}'[\text{tid}].\text{list} \cap [0, \text{tid}) \neq \emptyset &\rightarrow c'[\text{tid}] = \text{black} \\
\lor \text{active}'[\text{tid}] &= \text{false} \land \text{token}'.\text{pos} = \text{token.pos} + 1 \\
\land \text{token}'.\text{col} &= \text{white} \rightarrow \text{msg}'[\text{tid}].\text{list} \cap [0, \text{tid}) = \emptyset
\end{align*}
\]

**Q3**: (case \( \text{token.col} = \text{white} \land \text{tid} > \text{token.pos} \)) - the same as Q1.

To prove \( \text{Inv} \) we use a less strong version \( \text{Inv2} \) that may be proved by induction: if \( \text{Inv2} \) holds “up-to” to an \( \text{tid} \) and module \( R \) is applied, then \( \text{Inv2} \) holds up to \( \text{tid}+1 \). \( \text{Inv} \) follows from the fact that \( \text{Inv2} \) holds for the last value of \( \text{tid} \). The details are omitted.\(^5\)

5 Final comments

In this paper we presented an implementation and a verification of a popular ring termination detection protocol using structured rv-programs. The programming language turns out to be able to describe not only the code for each process, but also the complicated interactions between processes. While not easy\(^6\), the proof proves to be quite natural. The conclusion of the study is positive, rising our hope that structured rv-programs may be a good choice for those interested in the study of more complex parallel, concurrent, or distributed protocols.

\(^5\)A fully formal proof requires a detailed presentation of the Hoare logic for structured rv-programs and it is outside of the scope of this paper.

\(^6\)As usual, one has to come up with appropriate invariants, sometimes difficult to find.
References


