Towards a Hoare logic for structured interactive 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), 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 [14]. In the setting of rv-programs one freely uses powerful go-to statements with both temporal and spatial labels. A Floyd-like logic for interactive rv-programs has been presented in [15].

In a companion paper [4] the authors have presented structured programming techniques to liberate rv-programs of go-to statements. A particular emphasis is put on developing a structural spatial programming discipline. The technique allows for more structured interactions between processes and it is of great help in simplifying the construction and the analysis of interactive programs.

In the present paper the authors show how Hoare’s method for the correctness of classical structured sequential programs may be adapted to prove the correctness of structured interactive rv-programs.

1 Introduction

Interactive systems are omnipresent - they range from describing low level interacting processes on the same machine, cluster, or distributed system to communicating agents in the Internet, human-computer, or human-human interaction. While there are many proposals for both the foundations and the practice of interactive computation, including [1, 3, 5–7, 16, 17], the subject is still open, far from having a common agreement, most of the approaches being tailored on specific aspects or application areas.

A model, a core programming language, 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: (1) The model consists of rv-systems (interactive systems with registers and voices); it includes register machines, is space-time invariant, is compositional, may describe computations extending in both time and space, and is applicable to open, interactive systems. To achieve modularity in space
the model uses voices (a voice is the time dual of a register) - they provide a high level organization of temporal data and are used to describe interaction interfaces of processes. (2) The programming language uses novel techniques for syntax and semantics to support computation in space paradigm. It uses rv-programs and bases their syntax and operational semantics on FIS's (finite interactive systems) and their grid languages. (3) The specification of rv-systems uses relations between input registers and voices and their output counterparts. (4) The analysis techniques developed for rv-programs use finite automata, FIS’s, and an intermediary class of decomposable FIS’s. See [14] for more information.

In a companion paper [4] the authors have shown how structured programming techniques for classical sequential programs may be lifted to the class of rv-programs. A particular emphasis is put on developing a structural spatial programming discipline. The technique allows for more structured interactions between processes and it is of great help in simplifying the construction and the analysis of interactive programs.

A Floyd-like method for the correctness of interactive rv-programs has been presented in [15]. The semantics of rv-programs uses scenarios, a two-dimensional version of paths. The lifting of Floyd’s method to rv-programs is essentially a two-dimensional extension of the method where cut-points with state assertions become contours (borders of certain scenarios) with state and class assertions.

In the present paper we show how Hoare’s method for the correctness of structured programs may be adapted to prove the correctness of structured interactive rv-programs. While similar to Floyd’s method, Hoare’s method takes advantage on the structural form of the programs to simplify the verification task. After a brief recall of Hoare’s method for structured programs, the paper describes the main concepts needed to understand structured rv-programs and their semantics: rv-programs and their scenarios, spatio-temporal specifications using registers and voices, structured programming for rv-programs. Then Hoare’s method for structured rv-programs is developed and illustrated on specific examples.

2 Floyd’s and Hoare’s methods for program verification

Floyd’s method is used to verify classical flowchart programs. We briefly describe the method using the program in Fig. 1 - this program checks if a natural number

\[
\text{(A perfect number is a number equal to the sum of its proper divisors.)
}
\]

For partial correctness, a set of “cut-points” are chosen such that each cycle has at least one cut-point and each point has an associated assertion. In our case, cut-points are A, B, C, D. The assertions in A, C, D are given and specify the input-output requirement for this program, e.g., \(\phi_A : \, "n \geq 2"\) and \(\phi_C = \phi_D : \, "z = 1 \text{ if } n \text{ is a perfect number, otherwise } 0"\). The other assertions should be found (guessed); in our case, \(\phi_B : \, "0 \leq x \land y = n \land n \geq 2 \land z = n - \sum_{d|n,x<d<n}d"\). The conditions to be proved are \(\phi_X \land C_p \Rightarrow \sigma(\phi_Y)\), where \(X, Y\) are cut-points, \(\phi_X, \phi_Y\) are the assertions in these points, \(p\) is a path from \(X\) to \(Y\), \(C_p\) is the condition to take this path, and \(\sigma\) a substitution. Notice that, both
(x,y,z) = (n/2,n,n);

while (! x == 0 ) {
  if ( y%x == 0 ) {
    z = z - x;
    x = x - 1;
  }
}

if ( z == 0 ) {
  z = 1;
} else {
  z = 0;
}

Fig. 1. A flowchart and a while-program for perfect numbers

the assertion in Y and the condition to take the path p are written in terms of the variables in X - for this reason σ is used. For the program in Fig. 1, the conditions are: \( \phi_A \land C_{(A,B)} \Rightarrow \sigma_1(\phi_B) ; \phi_B \land C_{(B,E1,B)} \Rightarrow \sigma_2(\phi_B) ; \phi_B \land C_{(B,E2,B)} \Rightarrow \sigma_3(\phi_B) ; \phi_B \land C_{(B,C)} \Rightarrow \sigma_4(\phi_C) ; \phi_B \land C_{(B,D)} \Rightarrow \sigma_5(\phi_D) \), for appropriate substitutions \( \sigma_1, \ldots, \sigma_5 \). If these conditions are fulfilled, then any terminating computation will meet the input-output requirement.

For total correctness, one also has to prove termination. This is generally a simpler task. In our case, x decreases by 1 along each loop, hence eventually becomes 0 and the computation exits the loop.

There is a rich literature on these topics, including the classical book by Z. Manna [10], as well as algebraic presentations based on iteration theories [2], relation algebras [9], or network algebras [11].

In Hoare’s method structured while-programs are used leading to cut points implicitly defined by the structured programming operations. The assertions and the verification is similar, but a bit more tedious as we have assertions at each point/statement of the program; see [8] for more information, examples, etc.

3 Interactive programs with registers and voices

3.1 Grids and scenarios

Grids. A grid is a rectangular two-dimensional area filled in with letters of a given alphabet. Their set is denoted by \( V^+ \). Each letter in V is a two-dimensional atom having its own type of data on its north, south, west, or east side. More general grids may be introduced removing the condition to have a rectangular area. It is important to notice that our grids are logical, not geometrical objects.

Examples of grids are presented in Fig. 2. The grid in (a) is a normal, rectangular grid - by default, a grid is considered of this type. A general grid is
presented in (b). In (c) the standard relationship between cells in grids is described: as one can see, each cell directly depends on its top and left neighbors.

In our standard interpretation the columns correspond to processes, the top-to-bottom order describing their progress in time. The left-to-right order corresponds to process interaction in a nonblocking message passing discipline: a process sends a message to the right, then resumes its execution. The convention of having only a left-to-right causality in grids is not restrictive and two-ways interactions may be naturally encoded, see [14].

Scenarios. As we already said, grids are used to describe computations and a letter in a grid represents a statement to be executed. A scenario is a grid enriched with information about data used at the borders of its letters. The additional information on data around each letter may be given in an abstract form as Fig. 3(a) and (b), or in a more detailed form as in Fig. 3(c) (the letters of the associated grid are those in the boxes $X, U, V, \ldots$, while the neighboring areas are used to put extra information).

3.2 Finite interactive systems

Definition. A finite interactive system (shortly FIS) is a finite hyper-graph with two types of vertices and one type of (hyper) edges: (1) the first type of vertices is for states, labeled by numbers or lower case letters; (2) the second type of
vertices is for classes, labeled with capital letters; (3) the edges/ transitions are labeled by letters denoting atoms of the grids and, moreover, each transition has two incoming arrows (one from a class, the other from a state), and two outgoing arrows (one to a class, the other to a state). Some classes/states may be initial (indicated by small incoming arrows) or final (indicated by double circles).

Finite interactive systems have been introduced in Lesson 11 of [12]; see also [13,14]. An example is presented below.

**Parsing procedure.** Given a FIS $F$ and a grid $w$, insert initial states/classes at the north/west border of $w$ and parse the grid completing the scenario according to the FIS transitions; if the grid is fully parsed and the south/east border contains final states/classes, the grid $w$ is recognized by $F$. The language $L(F)$ is the set of grids recognized by $F$. Below, a FIS $F_1$ and a parsing for $\text{abb}$ are presented showing this grid is recognized by $F_1$. The first two rows show that $\text{cab}$ is not recognized (1 is not final).

### 3.3 Spatio-temporal specifications

**Data with temporal representation.** Spatial data may be easily represented in the memory using registers. For the temporal data we use streams: a stream is a sequence of data ordered in time $a_0, a_1, \ldots$, where $a_0, a_1, \ldots$ are its data at time $0, 1, \ldots$. A stream may be seen as the result of observing the data transmitted along a channel: it exhibits a datum (corresponding to the channel type) at each clock cycle.

A voice is defined as the time-dual of a register: A *voice is a temporal structure that holds a natural number. It can be used ("heard") at various locations. At each location it displays a particular, possible different, value.*

Voices may be implemented on top of a stream in a similar way registers are implemented on top of a Turing tape, for instance specifying their starting address and their length. At this higher level of abstraction only voices and their contents matter, not the implementation details as position on the stream, low-level bit manipulation, etc.

In practice many concrete data structures are needed. Most of usual data structures have natural temporal representations. We add a “t” in front of normal types to denote these new temporal types. Examples: $\text{tBool}$ (booleans), $\text{tInt}$ (integers), $\text{tArray}$ (arrays), $\text{tLinkedList}$ (linked lists), etc.
Relational spatio-temporal specifications. We use the notation $\cdot \circ \cdot$ for the spatial product and $\cdot \lhd \cdot$ for the temporal product (mathematically, they are just the Cartesian product). Moreover, $\mathbb{N}^\circ k$ denotes $\mathbb{N} \cdot \circ \cdot \cdots \circ \cdot \mathbb{N}$ ($k$ terms) and $\mathbb{N}^\lhd k$ denotes $\mathbb{N} \cdot \lhd \cdot \cdots \lhd \cdot \mathbb{N}$ ($k$ terms).

A spatio-temporal specification is a relation $S \subseteq (\mathbb{N}^\circ m \times \mathbb{N}^\circ p) \times (\mathbb{N}^\lhd n \times \mathbb{N}^\lhd q)$ between input and output registers and voices. It is denoted as $S : (m, p) \rightarrow (n, q)$, where $m$ (resp. $p$) is the number of input voices (resp. registers) and $n$ (resp. $q$) is the number of output voices (resp. registers). On elements, it is defined as a relation between concrete tuples, written as $(v \mid r) \mapsto (v' \mid r')$, where $v, v'$ (resp. $r, r'$) are tuples of voices (resp. registers).

Specifications may be composed horizontally and vertically, as long as their types agree; e.g., for two specifications $S_1 : (m_1, p_1) \rightarrow (n_1, q_1)$ and $S_2 : (m_2, p_2) \rightarrow (n_2, q_2)$ the horizontal composition $S_1 \cdot S_2$ is defined only if $n_1 = m_2$ and the type of $S_1 \cdot S_2$ is $(m_1, p_1 + p_2) \rightarrow (n_2, q_1 + q_2)$.

A specification for perfect numbers. The flowchart program for perfect numbers in Fig. 1 may be decomposed into interacting components $C_x, C_y, C_z$ corresponding to its variables. The general specification is $C_x \cdot C_y \cdot C_z$. The relevant parts of the specification of the components are:

- $C_x$: read $n$ from its north side and write $n \circ ([n/2] \circ ([n/2] - 1) \circ \ldots \circ 2 \circ 1)$ on its east side;
- $C_y$: read $n \circ ([n/2] \circ ([n/2] - 1) \circ \ldots \circ 2 \circ 1$ from its west side and write $n \circ \phi_n([n/2]) \circ \ldots \circ \phi_n(2) \circ \phi_n(1)$ on its east side, where $\phi_n(k) = \{k \text{ divides } n \text{ then } k \text{ else } 0\}$;
- $C_z$: read $n \circ \phi_n([n/2]) \circ \ldots \circ \phi_n(2) \circ \phi_n(1)$ from its west side, subtract from the first number all the other received numbers (i.e., $\phi_n([n/2]), \ldots, \phi_n(1)$), and finally write on its south side “if the final difference is 0 then 1 else 0”.

The global input-output specification is: if the leftmost north number (i.e., $x$) is $n$, then the rightmost south number (i.e., $z$) is either 0 or 1, being 1 iff $n$ is perfect.

3.4 Interactive programs with registers and voices

RV-Programs. An rv-system (interactive system with registers and voices) is a FIS enriched with: (i) registers associated to its states and voices associated to its classes; and (ii) appropriate spatio-temporal transformations for actions.

We study programmable rv-systems specified using re-programs. An example of rv-program is presented in Fig. 4. A computation is described by a scenario like in a FIS, but with concrete data around each action, see Fig. 3(c). Other examples of rv-programs and associated running scenarios may be found in [14].

Syntax. The syntax is based on the syntax used in imperative programming languages. The basic block is a module. To explain the syntax, let us focus on the first module of the program Perfect in Fig. 4. It has a name $X$ and 4 areas: (1)
in: A,1; out: D,2

X::

(A,1) x : sInt
tx : tInt;
tx = x;
x = x/2;
goto [B,3];

Y::

(B,1) y : sInt
tx : y = tx;
tInt goto [C,2];

Z::

(C,1) z : sInt
tx : z = tx;
tInt goto [D,2];

U::

(A,3) x : sInt
tx : tInt;
tx = x;
x = x - 1;
if (x > 0) {
    goto [B,3]
} else {
    z = 1
    goto [B,2];
}

W::

(C,2) z : sInt
tx : z = z - tx;
tInt if(tx == 1){
    if(z == 0) {
        z = 1
    } else {
        z = 0;
    }
} goto [D,2];

Fig. 4. The rv-program Perfect (for perfect numbers)

In the top-left part we have a pair of labels (A,1) which specifies the interaction and control coordinates where this module has to be applied. (2) The top-right part declares the spatial input variables. (3) The bottom-left part declares the temporal input variables. (4) The body of a module is its bottom-right part, including type declarations and C-like code. The exit from the module is realized by a goto statement, a statement like goto [B,3] indicates that: (i) the data of the spatial variables in the current module will be used in a next module with control state 3; (ii) the data of the temporal variables in the current module will be used for the interaction interface of a new module with interaction label B.

The code in Fig. 4 is very simple. One may have more complex code in the body, for instance using while-statements. Notice that the new variables are not restricted to the module where they are declared - the output of a module contains both the inputs and the newly declared variables.

Operational semantics. The operational semantics of rv-programs is given in terms of scenarios. Scenarios are built up with the following procedure, described using the scenario in Fig. 3(c) (for the rv-program Perfect):

Steps for building scenarios:
(1) Each cell of the associated grid has as label a module name X, Y, Z, . . .
(2) An area around a cell may have an additional information as x=2 - that means, in that area x is updated to be 2.
(3) The full information on the current state of a process is obtained going vertically up and collecting the last updated values of the spatial variables. For instance, the bottom V in the second column has y=4.
(4) The full information on temporal variables in a current place is obtained collecting their last updated values going horizontally on left. For instance, W in the second row has tx=2.
The first column has an input class and particular tuples of values for its temporal variables; the first row has an input state and particular tuples of values for its spatial variables.

The computation in a cell $\alpha$ is done as follows: (i) Take a module $\beta$ of the program bearing the class label of the left neighboring area of $\alpha$ and the state label of the top neighboring area of $\alpha$. (ii) Follow the code in $\beta$ using the spatial and the temporal variables of $\alpha$ with their current values. (iii) If the local execution of $\beta$ is finished with a $\text{goto } [I, \gamma]$ statement, then the label of the right neighboring area of $\alpha$ is set to $I$ and the label of the bottom neighboring area of $\alpha$ is set to $\gamma$. (iv) Insert the values of the temporal variables updated by $\beta$ in the right neighboring area of $\alpha$ and the values of the spatial variables updated by $\beta$ in the bottom neighboring area of $\alpha$.

A partial scenario (for an rv-program) is a scenario built up using the above rules; it is a complete scenario if the bottom row has only final states and the rightmost column has only final classes.

The scenario in Fig. 3(c) is a complete scenario for the rv-program Perfect.

**Associating a FIS and a grid language.** One naturally associates a FIS to an rv-program. For the program Perfect, the associated FIS $F$ is defined by: $A, 1$ initial, $D, 2$ final, and transitions:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>X</td>
<td>B</td>
<td>1</td>
<td>C</td>
<td>Z</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>Y</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Notice that $U_1/U_2$ denotes the run corresponding to the module $U$ and to the 1st/2nd output (i.e., $[B, 3]/[B, 2]$), in the same way $p_{\text{yes}}/p_{\text{no}}$ is used for the yes/no branch of a test $p$ in a flowchart program. These actions are slices of the program modules and they represent the basic blocks for building up rv-program scenarios. Technically, the advantage of this convention is that the associated FIS is decomposable and its language can be easily described using the state and class projection finite automata, see [14]. The language recognized by $F$ consists of the grids presented in Fig. 5(a).

### 4 Structured rv-programs

**Syntax.** Classical structured programming avoids go-to statements and the related labels by using three basic operations to build up programs: composition, if, and while. The restriction to the flowcharts corresponding to structured programs simplifies to a great deal the analysis and verification tasks.

Extending structured programming to rv-systems is not so easy. The overall goal is to “liberate rv-programming of go-to statements”, hence to use no temporal or spatial labels, at all. A main source of complications is to match the typing of the modules at their composing interfaces. For usual structured
Fig. 5. Grids (a) and row (b)/column (c) decompositions for “Perfect” program scenarios.

sequential programs the state space is unique, hence there are no such typing problems.

The type of a program $P$ is a 4-uple $(w(P), n(P), e(P), s(P))$ indicating the types at its west, north, east, and south border. Alternatively, one may use the notation $P : (w(P), n(P)) \to (e(P), s(P))$.

The operations of structured programming, as proposed in [4], are the following extensions of the composition, if and while constructs:

**Composition:** We use three types of compositions: horizontal composition, vertical composition, and diagonal composition.

- For two programs $P_i : (w_i, n_i) \to (e_i, s_i)$, $i = 1, 2$, the horizontal composition $P_1 \# P_2$ is well defined only if $e_1 = w_2$; the type of the composite is $(w_1, n_1 \sqcap n_2) \to (e_1, s_1 \sqcap s_2)$.
- Similarly, the vertical composition $P_1 ; P_2$ is defined only if $s_1 = n_2$; the type of the composite is $(w_1 \wedge w_2, n_1) \to (e_1 \wedge e_2, s_2)$.
- The diagonal composition $P_1 \$ P_2$ is a derived operation - it connects the east border of $P_1$ to the west border of $P_2$ and the south border of $P_1$ to the north border of $P_2$; it is defined only if $e_1 = w_2$ and $s_1 = n_2$; the type of the composite is $(w_1, n_1) \to (e_2, s_2)$.

**If:** For the “if” operation, given two programs with the same type $P, Q : (w, n) \to (e, s)$, a new program if $(c)$ then $P$ else $Q : (w, n) \to (e, s)$ is constructed, for a condition $c$ involving both, the temporal variables in $w$ and the spatial variables in $n$.

**While:** There are various possible extensions of the while statement.

- For a program with a dummy interaction interface $P : (0, n) \to (0, n)$, a temporal while is defined as a natural extension of the usual “while” statement, namely $\textbf{while}_t (c) \{P\}$, where $c$ is a condition on the spatial variables in $n$; the type of the result is $(0, n) \to (0, n)$.
- Similarly, for a program with a dummy state interface $P : (w, 0) \to (w, 0)$, a spatial while is defined as $\textbf{while}_s (c) \{P\}$, where $c$ is a condition on the temporal variables in $w$; the type of the result is $(w, 0) \to (w, 0)$.
— A spatio-temporal while may be defined for a program $P : (w, n) \rightarrow (w, n)$, namely $\texttt{while}$ \_\_ $\texttt{st}$ \_\_ (c) \{} P \}, where $c$ is a condition on the temporal variables in $w$ and the spatial variables in $n$; the type of the result is $(w, n) \rightarrow (w, n)$.

— The temporal while statement may be extended to the case of programs with nonempty interaction interfaces. In such a case, as the while loop may be executed an indefinite number of times depending on its current data, the west interface has to provide as many interaction data as needed. This is generally not possible, as the programs have fixed types on their interfaces. However, one may introduce an appropriate construct $\texttt{forall}$ \_\_ $\texttt{x}$ \{} $P$ \}, for a $P : (w, n) \rightarrow (e, n)$, which repeat the application of $P$ to fit its temporal input interfaces. A similar extension may be applied to the spatial while. Finally, let us notice that no such problems appear in the case of the spatio-temporal while.

Structured rv-programs for perfect numbers. An rv-program for perfect numbers was presented in Fig. 4. It corresponds to the decomposed specification for perfect numbers presented in Section 3.3. Below, we present two structured versions. We slightly change to program to shorten the code removing the last 5 lines from Fig.1(b); now, the input number $x$ is perfect iff the output $z$ is 0.

A first structured version, corresponding to the row decomposition of scenarios in Fig.5(b), is

\begin{verbatim}
In: x; Out: z; x,y,z : sInt; t : tInt;>{{t=x; x=x/2;}# {y=t; }# {z=t;}}#; while $t(x>0)$\{
  {t=x; x=x-1;}#
  {if(!(y$t==0)}{t=0;}#
  {z=z-t;}#
};
\end{verbatim}

The second version, corresponding to the column decomposition in Fig.5(c), is described below

\begin{verbatim}
In: x; Out: z; x,y,z : sInt; t : tInt
{t=x; speak t; x=x/2; while $t(x>0)$\{t=x; speak t; x=x-1;}\}#
{listen t; y=t; speak t; forall t{listen t; if(!(y$t==0)}{t=0;}; speak t;}#
{listen t; z=t;
  forall t{listen t; z=z-t;}#
\end{verbatim}

Semantics. The operational-relational semantics of structured rv-programs is indirectly defined via a translation $Tr : strRV \rightarrow RV$ from structured rv-programs to rv-programs; once the translation is specified, the scenarios and the input-output relation are obtained from the translated program.
Each program will have an appropriate label for each of its borders; the labels are to be carefully handled, eventually renamed, to avoid name clashes. The operations are defined as follows (see [4] for details):

**Composition:** For $Tr(P_1 \# P_2)$ identify the output class label in $Tr(P_1)$ with the input class label in $Tr(P_2)$, identify the input state labels in $Tr(P_1)$ and $Tr(P_2)$, and identify the output state labels in $Tr(P_1)$ and $Tr(P_2)$.

Similar procedures may be used for $Tr(P_1; P_2)$ and $Tr(P_1 \& P_2)$.

**If:** In the case of $Tr(\text{if} (c) \text{ then } P \text{ else } Q)$, one has to use a preprocessing rv-program to collect the input values from all states on the north border and from all classes on the west border, then to test the condition $c$, and, finally, to use a go-to statement to execute either $Tr(P)$ or $Tr(Q)$ according to the value of $c$.

**While:** For $Tr(\text{while} t (c) \{P\})$, denoted $Q$, a similar preprocessing rv-program is used to collect the input values from all states on the north border and to test the condition $c$. If $c$ is true, then use a go-to statement to start $Tr(P)$; moreover, the output state label of $Tr(P)$ is identified to the input state label of $Q$ to repeat the loop. If $c$ is false, then use a go-to statement to the exit of $Q$.

Similar procedures may be used to define the translations $Tr(\text{while} \& (c) \{P\}$ and $Tr(\text{while} \& (c) \{P\})$.

5 Verification of structured rv-programs using a Hoare-like method

5.1 Assertions and inference rules

A framework for rv-program verification. The lifting of program verification techniques from flowchart programs (one-dimension) to rv-programs (two-dimensions) is not completely straightforward. There are a few key points where the design decision should be clearly motivated.

As explained in Section 2, the common practice for flowchart programs is to find assertions for a few key points of the program and to prove the invariance conditions. This technique demands to have at least one cut-point along each loop. We want to lift this more relaxed technique to rv-programs, first.

For flowchart programs, the role of the cut-points is to ensure that: (1) each possible run $p$ (i.e., each syntactically possible path from input to output) is decomposed by cut-points into a sequence $(p_1, p_2, \ldots, p_k)$ of small paths and (2) the set $SPath$ of all such paths $p_i$ (i.e., the set of all simple paths from one cut-point to another) form a finite set. Finally, the proof is reduced to the verification of the invariance conditions for $SPath$. Notice that condition (2) is fulfilled if each loop has at least one cut-point.

For rv-programs, cut-points becomes contours, surrounding finite scenarios. Their set must be finite. The condition to break all loops becomes the following: each syntactically possible scenario (i.e., the scenarios of the associated FIS) can be decomposed in pieces corresponding to these contours.
To conclude, the verification procedure for rv-programs consists of the following three steps: (i) find an appropriate set of contours and assertions (it should be a finite and complete set); (ii) fill in the contours with all possible scenarios; and (iii) prove these scenarios respect the border assertions. Notice that, except for the guess of assertions, the proof is finite and can be done fully automatically.

In the verification of while-programs using Hoare logic cut-points and assertions are omnipresent, surrounding each statement, see, e.g., [8]. With such a simplified point of view, the extension looks conceptually simpler, but the proofs becomes more tedious: one has to provide assertions around each transition/statement of a structured rv-program and to lift these assertions piece-by-piece to the full program using appropriate inference rules.

**Assertions.** To define assertions for rv-programs one has to specify contours and particular relationships on states and classes along these contours. The role of such an assertion is the following: (i) the area within the contour is completed with program actions to get a program scenario \( f \), while a “path condition” \( C \) collects all conditions making this run \( f \) possible; (ii) the scenario must obey the border assertions, provided the path condition \( C \) is true; (iii) the above test must be true for all possible completions of the area within the contour, provided one gets valid scenarios.

For structured rv-programs the contours have simple rectangular forms, hence their representation may be simplified and included in Hoare-like triples, a notation mixing assertions with program code. A *Hoare triple* is a combination of border assertions and program code of the following form

\[
\{ \phi_w | \phi_n \} P \{ \phi_c | \phi_s \}
\]

where each assertion is a sequence of usual assertions separated by “;”. We need to have such *sequences of assertions* on the north/south parts as our scenarios have multiple processes, each with its own variables. A similar situation is on the east/west borders.

**Inference rules.** In the proofs to follow we will use the following proof rules:

- **Basic rule:** For a module, the validity of an assertion \( \{ \phi_w | \phi_n \} M \{ \phi_c | \phi_s \} \) is reduced to the validity of the assertion \( \{ \phi_w \land \phi_n \} M \{ \phi_c \land \phi_s \} \) in the setting of usual while programs.

- **Rule for composition:** If \( \{ \phi_w | \phi_n \} P \{ \phi_c | \phi_s \}, \{ \phi'_w | \phi'_n \} P' \{ \phi'_c | \phi'_s \} \) and \( \phi_c \Rightarrow \phi'_c \), then \( \{ \phi_w | \phi_n; \phi'_w | \phi'_n \} P \# P' \{ \phi'_c | \phi'_s \} \). Similar rules hold for the vertical and the diagonal compositions.

- **Rule for “if”:** If \( C \) is a condition on states, \( \{ \phi_w | \phi_n \land C \} P_1 \{ \phi_c | \phi_s \} \) and \( \{ \phi_w | \phi_n \land \neg C \} P_2 \{ \phi_c | \phi_s \} \), then \( \{ \phi_w | \phi_n \} \text{if}(C)(P_1)\text{else}(P_2) \{ \phi_c | \phi_s \} \). A similar rule holds when \( C \) is a condition on classes.

- **Rule for “while”:** If \( P \) is a program of type \((0, n) \rightarrow (0, s)\), \( C \) is a condition on states, \( \{ \eta \land C \} P \{ \eta \} \), then \( \{ \eta \} \text{while}(C)(P) \{ \eta \land \neg C \} \). Similar rules holds for \texttt{while_st} and \texttt{while_set}. 
When the program $P$ has a nontrivial temporal interface, some useful “ad-hoc” rules may be obtained for $\texttt{while}_{t}(C)\{P\}$ by induction using the assertions on the temporal interfaces of $P$ - an example is presented in the proof of the correctness of the second program for perfect numbers. Extensions to conditions which combines both temporal and spatial data may also be handled, but no such programs are included in the present paper.

5.2 Correctness of programs for perfect numbers

Partial correctness of the 1st “Perfect” program. This case is similar to the verification of the original while program in Fig.1(b). To avoid confusion, we use a more detailed representation of the program, with explicit read/write listen/speak actions even when this is pretty obvious, e.g., $\{x=a; y=x;\}$ is rewritten $\{x=a; \texttt{write} \  x; \} \{\texttt{read} \  x; \ y=x;\}$. This program corresponds to a row decomposition of scenarios. The input/output and the type information is as above: IO $[\text{In: } x; \text{Out: } z]$; Types $[x,y,z: s\text{Int}; \ t: t\text{Int}]$.

The program is $P = Q; S$, where

$Q =$

\[
\{\texttt{read} \ x; \ t=x; \ x=x/2; \ \texttt{speak} \ t; \ \texttt{write} \ x;\}
\]

\[
\{\texttt{listen} \ t; \ y=t; \ \texttt{speak} \ t; \ \texttt{write} \ y;\}
\]

\[
\{\texttt{listen} \ t; \ z=t; \ \texttt{write} \ z;\}
\]

$S =$

\[\texttt{while}_{t}(x>0)\{R\}\]

where $R =$

\[
\{\texttt{read} \ x; \ t=x; \ x=x-1; \ \texttt{speak} \ t; \ \texttt{write} \ x;\}
\]

\[
\{\texttt{read} \ y; \ \texttt{listen} \ t; \ \text{if}(!(y\%t==0))\{t=0;\}; \ \texttt{speak} \ t; \ \texttt{write} \ y;\}
\]

\[
\{\texttt{read} \ z; \ \texttt{listen} \ t; \ z=z-t; \ \texttt{write} \ z;\}
\]

First, the Hoare triples for the parts of $Q$ are:

\[
\{n\} \ \texttt{read} \ x; \ t=x; \ x=x/2; \ \texttt{speak} \ t; \ \texttt{write} \ x; \ \{t=n; x=[n/2]\}
\]

\[
\{t=n\} \ \texttt{listen} \ t; \ y=t; \ \texttt{write} \ y; \ \texttt{speak} \ t; \ \{t=n; y=n\}
\]

\[
\{t=n\} \ \texttt{listen} \ t; \ z=t; \ \texttt{write} \ z; \ \{z=n\}
\]

By the horizontal composition rule

\[
\{n\} \ Q \ {\{x=[n/2]; y=n; z=n\}
\]

Let $k \in [1, [n/2]]$. We have

\[
\{x=k\} \ \texttt{read} \ x; \ t=x; \ x=x-1; \ \texttt{speak} \ t; \ \texttt{write} \ x; \ \{t=k; x=k-1\}
\]

\[
\{t=k; y=n\} \ \texttt{read} \ y; \ \texttt{listen} \ t; \ \text{if}(!(y\%t==0))\{t=0;\}; \ \texttt{speak} \ t; \ \texttt{write} \ y; \ \{t=\phi_n(k); y=n\}
\]

\[
\{t=\phi_n(k); z=n-\sum_{d|n,k<d<n} d\} \ \texttt{read} \ z; \ \texttt{listen} \ t; \ z=z-t; \ \texttt{write} \ z; \ \{z=n-\sum_{d|n,k-1<d<n} d\}
\]

By the horizontal composition rule we get for $k \in [1, [n/2]]$

\[
\{x=k; y=n; z=n-\sum_{d|n,k<d<n} d\} \ \{R\}
\]

\[
\{x=k-1; y=n; z=n-\sum_{d|n,k-1<d<n} d\}
\]
We may relax the condition for $k$ to $k \in [0, [n/2]]$ and use the condition “$x > 0$” to follow the loop. That way, we get a fully invariant form for north/south part, as in the bottom assertion $k = 1 \in [0, [n/2]]$, indeed.

By the rule for while $<$ we get for a $k \in [0, [n/2]]$:
\[
\begin{align*}
|x = [n/2]; y = n; z = n \} & S \\
\{ |n = k \land y = k; y = n; z = n - \sum_{d \mid n, k < d < n} d \} & \text{and the last part is reduced to } \{ x = 0; y = n; z = n - \sum_{d \mid n, 0 < d < n} d \}.
\end{align*}
\]

By the vertical composition rule we get
\[
\{ |n \} P \{ |z = n - \sum_{d \mid n, 0 < d < n} d \}
\]
where $z = 0$ if $n$ is a perfect number.

**Partial correctness of the 2nd “Perfect” program.** This program corresponds to a decomposition of scenarios by columns. As above, IO [In: x; Out: z]; Types $[x, y, z : sInt; t : tInt]$

\[
P = Q \# R \# S, \text{ where}
\]
\[
Q = Q_1 \{ t = n \mid x = [n/2] \}
\]

For $Q_1$, \{ $|n \} Q_1 \{ t = n \mid x = [n/2] \}$. For $Q_2$, for a $k \in [1, [n/2]]$, \{ $|x = k \} Q_2 \{ t = k \mid x = k - 1 \}$, hence by induction \{ $|x = [n/2] \} while_r(x > 0) \{ Q_2 \} \{ t = n \mid t = [n/2] - 1; \ldots ; t = 1 \mid x = 0 \}$

By vertical composition, \{ $|n \} Q \{ t = n ; t = [n/2] ; t = [n/2] - 1 ; \ldots ; t = 1 \mid x = 0 \}$

For $R_1$, \{ $t = n \} R_1 \{ t = n \mid y = n \}$. For $R_2$, for a $k$, we have \{ $t = k \mid y = n \} R_2 \{ t = \phi_n(k) \mid y = n \}$, hence,
\[
\{ t = [n/2]; t = [n/2] - 1; \ldots ; t = 1 \mid y = n \} forall_r(T) \{ t = \phi_n([n/2]); t = \phi_n([n/2] - 1); \ldots ; t = \phi_n(1) \mid y = n \}
\]

By vertical composition, \{ $t = n ; t = [n/2] ; t = [n/2] - 1 ; \ldots ; t = 1 \mid x = 0 \} R
\[
\{ t = n ; t = \phi_n([n/2]); t = \phi_n([n/2] - 1); \ldots ; t = \phi_n(1) \mid y = n \}
\]

For $S_1$, \{ $t = n \} S_1 \{ |z = n \}$. For $S_2$, for some $k, r$, we have \{ $t = k \mid z = r \} S_2 \{ |z = r - k \}$. Hence,
\[
\{ t = \phi_n([n/2]); t = \phi_n([n/2] - 1); \ldots ; t = \phi_n(1) \mid z = n \} forall_r(S_2) \{ |z = n - \phi_n([n/2]) - \phi_n([n/2] - 1) - \ldots - \phi_n(1) \}
By vertical composition, \( t = n; t = \phi_n([n/2]); t = \phi_n([n/2] - 1); \ldots; t = \phi_n(1) \}

Finally, by horizontal composition of the assertions for \( Q, R, S \) we get the partial correctness of the program:

\[
\text{\( t = n; t = \phi_n([n/2]); t = \phi_n([n/2] - 1); \ldots; t = \phi_n(1) \)}
\]

Termination, total correctness. The termination of the 1st “Perfect” program is easy to establish: \( x \) is decreasing one-by-one form \([n/2]\) down to 1, so the full program terminates. For the 2nd “Perfect” program, the first process terminates within a finite number of steps. This induces a termination of the whole structured rv-program, as the second and third columns are simple reactive systems whose termination is controlled by the first column. To conclude, both structured rv-programs “Perfect” are totally correct.

Correctness of space-time dual versions. By space-time duality, one gets two more structured rv-programs, whose proofs are obtained by applying space-time duality to the previous proofs. While simple consequences of the above results, we mention them as they exhibit an unbounded horizontal composition (in the first 2 versions, the horizontal composition was finite).

5.3 Correctness of a distributed termination detection protocol

To go to more interesting programs, we studied an implementation of a distributed termination detection algorithm for a pool of distributed processes, i.e., the “dual-pass ring termination algorithm”. The algorithm can handle the case when processes may be reactivated after their local termination. It uses colored (black/white) tokens. Processes are also colored: a black color means global termination may have not occurred.

The algorithm: (1) The root process \( P_0 \) becomes white when it has terminated and it generates a white token that is passed to \( P_1 \). (2) 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 be 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 color a token black and pass on, 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. (3) When \( P_0 \) receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

An implementation as a structured rv-program using \texttt{while} has been written and verified. (Notice that only the “partial correctness” part may be proved - if the root receives the white token, the pool of processes have terminated. It is however possible that the pool of processes never terminates.) A proof is presented in the full version of the paper.
6 Final comments

There are many possible developments of the results presented in this paper, a few being sketched below. A first objective may be to develop and verify rv-programs for certain interesting medium-scale applications. Next, the theory associated to the method should be developed. Finally, a clarification of the relationship between structured programming “in space” and class manipulation techniques used in OO programming may be useful to apply this method to properly modified OO-programs.

References