Towards a Floyd Logic for Interactive RV-Systems

Gheorghe Ştefănescu
University of Bucharest, Faculty of Mathematics and Computer Science
Str. Academiei 14, Bucharest, Romania 010014
gheorghe@funinf.cs.unibuc.ro

Abstract

A model, a core programming language, specification and analysis techniques appropriate for modeling, programming and reasoning about interactive computing systems have been introduced by Ştefănescu in 2004 using register machines and space-time duality, see [17].

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.

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 finite interactive systems (FIS’s) and their grid languages.

The specification of rv-systems uses relations between input registers and voices and their output counterparts.

In this paper we show how Floyd’s method for the correctness of classical sequential programs may be adapted to prove the correctness of 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 [2, 4, 6, 9, 10, 18, 19], 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 Ştefănescu 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 [17] for more information.

In this paper we show how Floyd’s method [13] for the correctness of classical sequential programs may be adapted to prove the correctness of interactive rv-programs. After a brief recall of Floyd’s method for flowchart programs, the paper describes the main concepts needed to understand rv-programs and their semantics: grids and scenarios, finite interactive systems, spatio-temporal specifications using registers and voices. The semantics of rv-programs uses scenarios, a two-dimensional version of paths. The lifting of Floyd’s method to rv-program is essentially a two-dimensional extension of the method: cut-points with state assertions become contours (borders of certain scenarios) with state and class assertions. We illustrate the method starting with a flowchart program for perfect numbers, describing an interactive specification when the state space is decomposed into components (one for each variable), writing an interactive rv-program for this specification, and finally proving the correctness of this rv-program.
2 Verification of flowchart programs using Floyd’s method

Floyd’s method is used to verify classical flowchart programs. We briefly describe the method using the program in Fig. 5 - this program checks if a natural number is perfect. (A perfect number is a number equal to the sum of its proper divisors. 6 = 3 + 2 + 1 is the smallest perfect number.)

![Flowchart program for perfect numbers](image)

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., \( \phi_A : \mu \geq 2 \) and \( \phi_C = \phi_D : \nu = 1 \) if \( n \) is a perfect number, otherwise 0”. The other assertions should be found (guessed); in our case, \( \phi_B : 0 \leq x \land y = n \geq 2 \land z = n - \sum_{d|d,n,x<d<n} d \).

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 [13], as well as algebraic presentations based on iteration theories [3], relation algebras [12], or network algebras [14].

3. Grids and scenarios

3.1. Grids

**Grids and their representation.** A grid is a rectangular two-dimensional area filled in with letters of a given alphabet \( V \). 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. This typing is naturally extended to grids. 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. In classical two-dimensional language studies [7] the letters are left uninterpreted, hence grids tend to have a more rigid, geometrical structure.

![Grids](image)

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) - the emphasized border line helps to understand its representation presented below. 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 continues its execution. We show below that the convention to send messages left-to-right only is not restrictive.
Contour-and-contents representation of grids. A general grid may be represented using a contour-and-contents representation consisting of a pair of strings: one over \( \{ e, s, w, n \} \) for its contour and one for its letters collected in a top-to-bottom and left-to-right parsing. For the contour, we start from one point of the contour and follow the contour in a clockwise direction. The letters \( e, s, w, n \) are used to indicate the east, south, west, north directions. E.g.,

\[
e^4s^2w^3\ldots
\]

means

- go 4 steps towards east; then go 2 steps towards south; then go 3 steps towards west; etc.

For instance, the grid in Fig. 2(b) may be represented by

Contour: \( e^4s^2e^2n^1e^1s^3w^1n^1w^3 \times s^1w^1n^2w^1e^2n^1w^2n^1 \)

Contents: \( a^2b^0c^3d^0ab^0caca \).

Action vs. inter-action. The convention of having only a left-to-right causality in grids may raise an important question:

How inter-action may appear? It seems that a process may send a message to another process on the right, but can not receive an answer back!

Or, put in other words, it looks that grids properly describe actions (like sending messages from a master to slaves), but not real inter-actions.

Let us recall that our grids are logical, not geometrical objects. A two-ways communication situation is isomorphic to the situation described in Fig. 3(a). If the alphabet of grid letters is rich enough to contain empty cell, identities, corners, and crossings, then, as in Fig. 3(b), the picture may be converted into a grid which faithfully captures the situation.

Expressions for grids. One may define 2-dimensional regular expressions by the following BNF syntax:

\[
E ::= a | 0 | E + E | E \cap E | E : E | E^* | \varepsilon | E \triangleright E | E^0 | -
\]

Such expressions have a natural interpretation on grids. For two grids \( v, w \), the horizontal composite \( v \triangleright w \) is the grid obtained putting \( v \) on left of \( w \); the vertical composite \( v \cdot w \) is the grid obtained putting \( v \) on top of \( w \); families of vertical and horizontal identities \( \epsilon_k, A_k \) are naturally introduced.

\[
\begin{array}{c|c|c|c|}
\text{v} & \text{w} & \triangleright & \varepsilon \\
\hline
\end{array}
\]

The interpretation

\[
| \varepsilon : 2 \text{RegExp}(V) \rightarrow \mathcal{P}(V^*)
\]

from expressions to grid languages is defined by: \(|a| = \{a\}; \|0\| = \emptyset; |E + F| = |E| \cup |F|; |E \cap F| = |E| \cap |F|; |E : F| = \{v \cdot w: v \in |E| \land w \in |F|\}; |E^*| = \{v_1 \cdot \ldots \cdot v_k: k \in \mathbb{N} \land v_1, \ldots, v_k \in |E|\}; |\varepsilon| = \{\varepsilon_0, \ldots, \varepsilon_k, \ldots\}; |E \triangleright F| = \{v \triangleright w: v \in |E| \land w \in |F|\}; |E^0| = \{v \triangleright \ldots \triangleright v_k: k \in \mathbb{N} \land v_1, \ldots, v_k \in |E|\}; |\varepsilon| = \{\varepsilon_0, \ldots, \varepsilon_k, \ldots\}.
\]

3.2. Scenarios

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 in the picture below (i.e., a name \( A, B, 1, 2 \))

\[
\begin{array}{c|c|c|c|}
A & 0 & B & 1 \\
\hline
1 & 1 & 1 & A_1B_1B_1B_1 \\
2 & 1 & 1 & A_2B_1B_1B_1 \\
2 & 2 & 2 & A_2A_2B_1B_1 \\
\end{array}
\]

or in a more detailed form as in Fig. 6.

We do not describe the details of a scenario like the one in Fig. 6 now. At this stage just notice that 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.

Contour-and-contents representation of scenarios. To represent these scenarios we use the above contour-and-contents representation of grids enriched with state/class information for the border. A contour is written as

\[
e_1s_2B\ldots
\]

and means:

- go one step towards east leaving state 1 on the left hand; then go one step towards south leaving class B on the left hand; etc.

For instance, the grid of \( a \)'s, described in figure (b) above, has the following contour-and-contents representation:

Contour: \( e_1s_2e_1s_2e_3s_2w_2n_Aw_2n_Aw_2n_A \)

or, shortly, \( (e_1s_2)^3(w_2n_A)^3 \);

Contents: \( aaa \).
4. Finite interactive systems

4.1. Finite interactive systems

A finite interactive system (shortly FIS) is a finite hypergraph with two types of vertices and one type of (hyper) edges:

- A first type of vertices is for states; we label them using numbers or lower case letters;
- The second type of vertices is for classes; we use capital letters as labels;
- The edges (also called transitions) are labeled by letters denoting atoms of the grids; they obey the following constraints: (1) each transition has two incoming arrows: one from a class vertex, the other from a state vertex, and (2) each transition has two outgoing arrows: one to a class vertex, the other to a state vertex.

Some classes/states may be initial (in the graphical representation this is specified by small incoming arrows) or final (for this the double circle representation is used).

Finite interactive systems have been introduced in Lesson 11 of [15]; see also [16, 17]. An example is presented in Fig. 4.

\[
F_1 = \begin{array}{c}
\text{A} \\
1 \\
\text{a} \\
\text{B} \\
2 \\
\text{C} \\
\text{c}
\end{array}
\]

**Figure 4. A finite interactive system**

One may use a semi-textual representation for FIS’s. E.g., \(F_1\) in Fig. 4 may be defined by: \(A, 1\) initial; \(B, 2\) final; and transitions \(\text{A} \rightarrow \text{B} \rightarrow \text{A} \rightarrow \text{B} \rightarrow \ldots\). A fully textual representation of transitions as \(a : (A, 1) \rightarrow (B, 2)\) may also be used.

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 selecting unprocessed atoms having a state/class inserted at its north/west border. For each such atom \(a\), if \(s/C\) is its north state / west class, then choose a transition \(a \rightarrow a\) from \(F\) (if any), insert \(s'/C'\) at its south/east border and consider this atom to be already processed. Repeat the above as long as possible. The grid \(w\) is recognized if there is a parsing such that all of its atoms are processed and the south/east border contains final states/classes, only. The language \(L(F)\) is the set of grids recognized by \(F\).

To have an example, let us consider the grid \(\begin{array}{c}
\text{ab} \\
\text{c}
\end{array}\) and \(\begin{array}{c}
\text{can} \\
\text{bb}
\end{array}\)

\(F_1\) in Fig. 4. A parsing is

\[
\begin{array}{c}
\text{A} \rightarrow \text{B} \rightarrow \text{A} \rightarrow \text{B} \rightarrow \ldots
\end{array}
\]

showing this grid is recognized by \(F_1\). The first two rows show that \(\begin{array}{c}
\text{ab} \\
\text{c}
\end{array}\) is not recognized (1 is not final).

5. Spatio-temporal specifications

5.1. Data with temporal representation

Turing tape was a first spatial memory model used in machine-oriented models of computation; complex data structures may be implemented on top of this simple model. We propose to use a similar approach to create complex data structures in time. The resulting specifications inherit some features from [1, 4].

**Voices.** Registers holding numbers may be implemented on a Turing tape. At their higher level, the tedious aspects of identifying the positions of the numbers on the tape, the need to shift data when more space is needed, the computation of arithmetical or logical operations at the bit level, etc. are all hidden and one gets a more readable specification of the problem and of its solution.

Similarly, we start with a simple linear temporal data model: the stream structure. A stream, as used in the data-flow setting, is a finite or infinite sequence of data ordered in time. A stream is denoted as \(a_0 \_ a_1 \_ a_2 \_ \ldots\), where \(a_0, a_1, a_2, \ldots\) are its data (tokens) at time \(0, 1, 2, \ldots\), respectively.

The contents of our streams is always finite, but unbounded in time, in the same way the contents of a Turing tape is always finite, but unbounded in space. One may think of a stream as the result of observing the data transmitted along a channel: it exhibits a datum (corresponding to the channel type) at each clock cycle.

Now we can define a voice as the time-dual of a register (a register holds a number and it displays different values at different times):

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, voices may be specified giving their starting address and their length. E.g., a voice v holding 2006 is defined by its temporal address t0 (its starting time on the stream) and its length 4 (if decimal representation is used). This means, from the point in time t0, the cell corresponding to the stream is showing the digits 2,0,0,6 during 4 consecutive clock cycles.

At this new and higher level of abstraction we are interested in voices and their contents only. We are not interested in the implementation details as: representation, position on the stream, low-level manipulation, etc.

More data with temporal representation. The above setting is good for theoretical purposes (a voice may represent an arbitrarily long number), but in practice more 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: tBool (booleans), tInt (integers), tArray (arrays), tLinkedList (linked lists), etc.

5.2. Specifications

Relational spatio-temporal specifications. To distinguish between the products of data with spatial or temporal representation, we use the notation \(\sharedsymbol\) for the spatial product and “t” for the temporal product (mathematically, they are just the Cartesian product). Moreover, \(\mathbb{N}^{\times k}\) denotes \(\mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N}\) (k terms) and \(\mathbb{N}^{-k}\) denotes \(\mathbb{N}^{-} \times \ldots \times \mathbb{N}^{-}\) (k terms).

A spatio-temporal specification is a relation

\[ S \subseteq (\mathbb{N}^{-m} \times \mathbb{N}^{\times p}) \times (\mathbb{N}^{-n} \times \mathbb{N}^{\times q}) \]

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 \((v \mid r) \rightarrow (v' \mid r')\), where \(v, v'\) (resp. \(r, r'\)) are tuples of voices (resp. registers).

Examples. The constants used in Fig. 3
\[ c0 = \emptyset, c1 = \{\emptyset\}, c2 = \{\emptyset, \{\emptyset\}\}, c3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, c4 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset, \{\emptyset\}\}\}\}, c5 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset, \{\emptyset\}\}\}, \{\emptyset, \{\emptyset, \{\emptyset, \{\emptyset\}\}\}\}\} \]

have a natural relational interpretation:
\[ c0 = \emptyset; c1 = \{\emptyset\}; c2 = \{\emptyset, \{\emptyset\}\}; c3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}; c4 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset, \{\emptyset\}\}\}\}; c5 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset, \{\emptyset\}\}\}, \{\emptyset, \{\emptyset, \{\emptyset, \{\emptyset\}\}\}\}\}. \]

Composing specifications. Specifications may be composed horizontally and vertically, as long as their types agree. 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 \(S_1 \sqcup S_2\) is defined only if \(n_1 = m_2\); the type of \(S_1 \sqcup S_2\) is \((m_1, p_1 + p_2) \rightarrow (n_2, q_1 + q_2)\);

— the composite is defined as expected:
\[ (v \mid r_1, r_2) \mapsto (v' \mid r'_1, r'_2)\] in \(S_1 \sqcup S_2\) iff \(\exists v'. (v \mid r_1) \mapsto (v' \mid r'_1)\) in \(S_1\) and \(\exists v'. (v' \mid r'_2) \mapsto (v'' \mid r''_2)\) in \(S_2\)

— the vertical composition \(S_1 \cdot S_2\) is defined in a similar way: it is defined only if \(q_1 = p_2\); its resulting type is \((m_1 + m_2, p_1) \rightarrow (n_1 + n_2, q_2)\).

A specification for perfect numbers. The flowchart program for perfect numbers in Fig. 5 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 \(n \rightarrow n/2 \rightarrow \ldots \rightarrow 2 \rightarrow 1\) on its east side;

- \(C_y\): read \(n \rightarrow n/2 \rightarrow \ldots \rightarrow 1\) from its west side and write \(n \rightarrow \phi(n/2) \rightarrow \ldots \rightarrow \phi(2) \rightarrow 1\) on its east side, where \(\phi(k) = \text{“if } k \text{ divides } n \text{ then } k \text{ else } 0\”\);

- \(C_z\): read \(n \rightarrow \phi(n/2) \rightarrow \ldots \rightarrow \phi(2) \rightarrow 1\) from its west side, subtract from the first number (i.e., \(n\)) all the other received numbers (i.e., \(\phi([n/2]), \ldots, \phi(1)\)), 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 number (i.e., \(x\)) is \(n\), then the rightmost south number (i.e., \(z\)) is either 0 or 1, being 1 if \(n\) is perfect.

6. Interactive programs with registers and voices

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

We study programmable rv-systems specified using rv-programs. An example of rv-program is presented in Fig. 5. A computation is described by a scenario like in a FIS, but with concrete data around each action, see Fig. 6. Other examples of rv-programs and associated running scenarios may be found in [17].
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 : tInt
    y = tx;
    goto [C,2];

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

U::
(A,3) x : sInt
    tx : tInt;
    tx = x;
    x = x\(1);\)
    if (x > 0) goto [B,3] else goto [B,2];

V::
(B,2) y : sInt
    tx : tInt
    if(y%tx != 0) tx = 0;
    goto [C,2];

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

Figure 5. The rv-program “Perfect” (for perfect numbers)

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. 5. It has a name X and 4 areas.

(1) 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. Notice that similar pairs of labels are used inside the code (described in the bottom-right area), but the format \([B,3]\) and the meaning are different. We will return to this later.

(2) The top-right part declares the spatial input variables. These variables specify the memory state before the application of the module. To distinguish them from the variables used for interaction interfaces, we put an “s” in front of their types. For module X there is one spatial input variable of type integer, denoted \(s\text{Int}\).

(3) The bottom-left part is similar to the top-right one, but it declares the temporal input variables. These variables are used for the data appearing to the interaction interfaces between modules. We put a “t” in front of normal types to indicate these data have temporal representations. Module X has no temporal input variables. (Other modules have such variables, for instance Y has a temporal integer variable.)

(4) The body of a module is its bottom-right part. It may include type declarations for new variables and the code to be executed by the module, similar to C code. The computation within this part makes no distinction between a spatial and a temporal variable. The exit from the module is realized by a \(\text{goto}\) statement. The computation within an interactive systems has two dimensions: one vertical, the other horizontal. A statement like \(\text{goto } [B,3]\) indicates that:

- Vertically, the data of the spatial variables in the current module will be used in a next module with control state 3. (What will be the new interaction data in state 3 is not known from this module alone, as it depends on the global properties of the interactive system.)

- Horizontally, 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. (Again, it is not known from this module alone what will be the control state of the new process when it receives these data at its temporal interface B.)

The code in Fig. 5 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. Below, we describe a rule to determine the variables that may be used when a particular module is executed and their current values. 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. 6 (for the rv-program Perfect):

(1) Each cell of the associated grid has a label in \(\{X, Y, Z, U, V, W\}\) specifying the module of the program used in that particular cell. A cell has associated states on its top and bottom neighboring areas and associated classes on its left and right neighboring areas. The grid is built up starting with its top row and left column (containing the inputs) and progressively inserting new cells when their left and top neighboring areas are already computed/updated.

(2) An area may have an additional information as \(x=2\). This means, in that area \(x\) is updated to be 2.
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 \( t_x = 2 \).

(5) The 1st column has an input class (here \( A \)) and particular tuples of values for its temporal variables. The 1st row has an input state (here \( 1 \)) and particular tuples of values for its spatial variables.

(6) Once the above points have been clarified, the local computation of a cell is easy to describe:

(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 temporal variables of \( \alpha \) with their current values.

(iii) If the local execution of \( \beta \) is finished with a \texttt{goto} \([\Gamma, \gamma]\) statement, then the label of the right neighboring area of \( \alpha \) is set to \( \Gamma \) 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 \( \alpha \).

(7) 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. 6 is a complete scenario for the rv-program \textit{Perfect}.

7. Verification of rv-programs using a Floyd-like method

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.

There are variants of the verification method where cutpoints and assertions are omnipresent, surrounding each statement, for instance in the verification of while-programs using Hoare logic (see [11]). With such an extreme point of view, the extension is obvious: we have to provide assertions around each transition/module of an rv-program. But this approach is tedious and makes the program verification a nightmare.

On the other hand, 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.

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 cutpoints into a sequence \((p_1, p_2, \ldots, p_k)\) of small paths and (2) the set \( P \) of all these 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 checking of the invariance conditions for \( P \). Notice that condition (2) is fulfilled if each loop has at least one cut-point. For instance, if \( B \) is removed in Fig. 5, then the set of all simple paths from \( A \) to \( C \) becomes infinite.

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:

- find an appropriate set of contours and assertions (it should be a finite and complete set);
- fill in the contours with all possible scenarios; and
- prove these scenarios respect the border assertions.

Notice that, except for the guess of assertions, the proof is finite and can be done fully automatic.

The running example. We use the rv-program for perfect numbers in Fig. 5 to illustrate the method. It corresponds to the decomposed specification for perfect numbers presented in Section 5.
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 illustrated in Fig. 7. It is defined by: \( A, 1 \) initial, \( D, 2 \) final, and transitions:

![Figure 7. The FIS of the program “Perfect”](image)

Notice that \( U1/U2 \) denote the runs corresponding to the module \( U \) and to the 1st/2nd output (i.e., \( \{ B, 3 \} / \{ B, 2 \} \)), in the same way \( p_{yes} / p_{no} \) is used for the yes/no branches 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 [17].

The language recognized by \( F \) consists of grids

\[
\begin{align*}
X & \quad Y & \quad Z \\
U1 & \quad V & \quad W \\
\cdots & \quad \cdots & \quad \cdots \\
U1 & \quad V & \quad W \\
U2 & \quad V & \quad W
\end{align*}
\]

Assertions. To define these assertions one has to specify contours and particular relationships on states and classes along these contours. The role of such an assertion is the following:

- one has to complete the area within the contour with proper program actions (slices of program modules, like \( U1/U2 \) above) to get a program scenario \( f \) and to collect in a “path condition” \( C \) all conditions making this run \( f \) possible;

- the scenario must obey the border assertions, provided the path condition \( C \) is true;

- the above test must be true for all possible completions of the area within the contour, provided one gets valid scenarios.

An assertion is represented using the contour representation and inserting particular assertions on state and class variables in each point of the contour. For instance,

\[
e_1 \{ x = x_0 \} s_D e_2 \{ z = x_0 - 1 \} \ldots
\]

means:

go towards east having on left the state 1 satisfying condition \( x = x_0 \); then go towards south having on left the class \( D \) with no condition (i.e., True); then go towards east having on left the state 2 with condition \( z = x_0 - 1 \); etc.

Partial correctness, using a row partition. For this proof we use three cut-contours \( e_1 e_1 s_D w_2 w_3 w_A n_A, e_2 e_2 s_D w_2 w_A n_A, \) and \( e_3 e_3 s_D w_2 w_A n_A \), covering all scenarios of the program Perfect. The associated assertions and a formal proof are presented below. Before this, we describe how invariance conditions are tested.

The proof technique is described in details using the 2nd contour \( C = e_3 e_3 s_D w_2 w_A n_A \). For simplicity, we use numbers form 1 to 8 to indicate the positions along the contour. The associated assertion is:

\[
\exists k. 0 < k \leq \lfloor x_0 / 2 \rfloor \text{ such that } e_3 \{ x = k \} e_2 \{ y = x_0 \} \quad \text{where } e_2 \{ z = x_0 - \sum_{d|x_0, k<d<x_0} d \} s_D
\]

\[
\cdot e_2 \{ z = x_0 - \sum_{d|x_0, k-1<d<x_0} d \} w_2 \{ z = x_0 - \sum_{d|x_0, k-1<d<x_0} d \} w_2 \{ y = x_0 \} w_3 \{ x = k - 1 \} n_A
\]

There is only one way to fill in the interior of \( C \), respecting the state/class information on the borders, namely using \( U1 \triangleright V \triangleright W \). Moreover, the condition \( \psi_E \) to follow this run is \( x - 1 > 0 \) in module \( U \), hence \( k - 1 > 0 \). Finally, using a backwards substitution \( \sigma \) we have to translate the output conditions for the east/south borders \( \psi_E \) using the values for the west/north borders: \( z \) on position 5 (south) is \( z \) on position 3 (north) updated via \( U1 \triangleright V \), i.e., \( z = \phi(k); y \) is unchanged; and \( x \) on position 7 (south) is \( x - 1 \) from position 1 (north). (Recall that \( \phi(k) = k \) if \( k \mid x_0 \) then \( k \) else 0.)

The implication

\[
\psi_C \land \psi_W \land \psi_N \Rightarrow \sigma(\psi_E) \land \sigma(\psi_S)
\]

is trivially satisfied:

\[
z = x_0 - \sum_{d|x_0, k<d<x_0} d \\
\Rightarrow z = \phi(k) = x_0 - \sum_{d|x_0, k-1<d<x_0} d \\
x = k \Rightarrow x - 1 = k - 1
\]

Formal verification:

1. For the top row of the scenarios we use the cut-contour \( e_1 e_1 s_D w_2 w_3 n_A \) with the assertion
\[ e_1(x = x_0) e_1 e_1 s_D \]
\[ \cdot w_2(z = x_0) w_2(y = x_0) w_3(x = \lfloor x_0/2 \rfloor) n_A. \]

It can only be filled in with transitions \( X \triangleright Y \triangleright Z \) and the invariant condition holds true.

2. For the middle rows of the scenarios the cut-contour is
\[ e_3 e_2 s_D w_2 w_2 w_3 n_A \]

\[ \exists k. 0 < k \leq \lfloor x_0/2 \rfloor \] such that:
\[ e_3(x = k) e_2(y = x_0) \]
\[ \cdot e_2(z = x_0 - \sum_{d | x_0, k < d < x_0} d) s_D \]
\[ \cdot w_2(z = x_0 - \sum_{d | x_0, k-1 < d < x_0} d) \]
\[ \cdot w_2(y = x_0) w_3(x = k - 1) n_A. \]

It can only be filled in with transitions \( U_1 \triangleright V \triangleright W \) and, as we have seen above, the invariant condition holds true.

3. For the bottom row of the scenarios, the cut-contour is
\[ e_3 e_2 s_D w_2 w_2 w_3 n_A \]

\[ \exists k. (0 < k \leq \lfloor x_0/2 \rfloor) \] such that:
\[ e_3(x = k) e_2(y = x_0) \]
\[ \cdot e_2(z = x_0 - \sum_{d | x_0, k < d < x_0} d) s_D \]
\[ \cdot w_2(z = 1) w_3(x_0 - \sum_{d | x_0, 0 < d < x_0} d = 0, \text{ otherwise } 0) w_2 w_3 n_A. \]

It can only be filled in with transitions \( U_2 \triangleright V \triangleright W \), provided the condition \((- k - 1 > 0)\) is true. The invariant condition can be proved as in the previous case using the extra fact that \((0 < k) \land (k - 1 > 0) \Rightarrow (k = 1).\)

4. By combining matching contours of these types we get the partial correctness of this rv-program, namely for each scenario \((X \triangleright Y \triangleright Z) \cdot (U_1 \triangleright V \triangleright W) \cdot (U_2 \triangleright V \triangleright W)\)

\[ e_1(x = x_0) e_1 e_1 s_D \]
\[ \cdot w_2(z = 1) w_3(x_0 \text{ is a perfect number, otherwise } 0) \cdot w_2 w_2 n_A. \]

Partial correctness, using a column partition. The scenarios of the program Perfect may also be obtained as an horizontal composite of its columns, so it is enough to provide good correctness assertions for these columns.

1. For the first column of the scenarios, the contour is
\[ e_1(s_B)^{r+1} w_2(n_A)^{r+1}. \]

It can only be filled in with \( X \cdot (U_1)^r \cdot U_2 \).

The associated assertion is
\[ e_1(x = x_0) s_B \{ \{ x = x_0 \} \}
\[ \cdot \prod_{i=\lfloor x_0/2 \rfloor} s_B \{ \{ x = i \} \}
\[ \cdot w_2(x = \{ x = 0 \}) (n_A)^{r+1} \]

hence \( r = \lfloor x_0/2 \rfloor \). This assertion is obtained by composing simple assertions for \( X, U_1, U_2: \)

\[ e_1(x = x_0) s_B \{ \{ x = x_0 \} \}
\[ \cdot \prod_{i=\lfloor x_0/2 \rfloor} s_B \{ \{ x = i \} \}
\[ \cdot w_2(x = \{ x = 0 \}) \]

\[ \cdot w_2 w_2(n_A)^{r+1}. \]

2. For the second column of the scenarios, the contour is
\[ e_1(s_C)^{r+1} w_2(n_B)^{r+1}. \]

It can only be filled in with \( Y \cdot V^{r+1} \). The associated assertion is
\[ e_1 s_C \{ \{ x = x_0 \} \}
\[ \cdot \prod_{i=\lfloor x_0/2 \rfloor} s_C \{ \{ x = \phi(i) \} \}
\[ \cdot w_2(z = x_0) \cdot \prod_{i=\lfloor x_0/2 \rfloor} n_B \{ \{ x = i \} \}
\[ \cdot n_B \{ \{ x = x_0 \} \}

where \( \phi(i) = \text{if } i|x_0 \text{ then } i \text{ else } 0 \).

It follows by composing simple assertions for \( Y, V: \)

\[ e_1 s_C \{ \{ x = x_0 \} \} w_2(y = x_0) n_B \{ \{ x = x_0 \} \}
\[ \cdot e_2(y = x_0) s_C \{ \{ x = \phi(a) \} \}
\[ \cdot w_2(y = x_0) n_B \{ \{ x = a \} \}

3. Finally, for the third column of the scenarios, the contour is
\[ e_1(s_D)^{r+1} w_2(n_C)^{r+1}. \]

It can only be filled in with \( Z \cdot W^{r+1} \). The associated assertion is
\[ e_1(s_D)^{r+1} \]
\[ \cdot w_2(z = 1) \cdot \prod_{i=\lfloor x_0/2 \rfloor} n_C \{ \{ x = \phi(i) \} \}
\[ \cdot n_C \{ \{ x = x_0 \} \}

It follows by composing simple assertions for \( Z, W: \)

\[ e_1 s_D w_2(z = x_0) n_C \{ \{ x = x_0 \} \}
\[ \cdot \prod_{i=\lfloor x_0/2 \rfloor} s_D \{ \{ x = i \} \}
\[ \cdot w_2(z = x_0) \cdot \prod_{i=\lfloor x_0/2 \rfloor} n_A \{ \{ x = \phi(k) \} \}

or \[ k = 1 \]

\[ e_2(z = x_0) \cdot \prod_{i=\lfloor x_0/2 \rfloor} s_D \{ \{ x = i \} \}
\[ \cdot w_2(z = 1) \cdot \prod_{i=\lfloor x_0/2 \rfloor} n_A \{ \{ x = \phi(k) \} \} \]

\[ \cdot n_A \{ \{ x = \phi(k) \} \} \]
This way the proof is much more complicated, for sure. If we look at the assertion for $W$, as we do not know the context of its occurrence, we have to consider in an complicated “OR” condition all possible applications of the module.

To conclude, while more complicated, this second proof rests on the same principles: find good assertions for a finite numbers of contours (here 7 contours surrounding simple $1 \times 1$ blocks), check their invariance when we fill the contours with sub-scenarios of the program, and show that their set is complete for building up all scenarios of the program and to prove the global assertion.

**Termination, total correctness.** The termination of the rv-program Perfect is easy to establish: $x$ is decreasing one-by-one form $[x_0/2]$ down to 1, so in each running the first process (column) terminates within a finite number of steps. This induces a termination of the whole rv-program, as the second and third columns are simple reactive systems whose termination is controlled by the first column. To conclude, the rv-program Perfect is totally correct.

**Circular interaction.** The program for perfect numbers has a simple interaction structure (all scenarios have three columns), which is not always the case. For instance, the program MAFact for an interactive factorial function in [17] has a circular interaction. Such rv-programs may have scenarios extending both ways, vertically and horizontally. However, the check of the correctness is completely similar.

8. 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. Then, the verification logic for rv-programs is a spatio-temporal logic, for sure; a comparison to other similar logics (like existent monadic second-order logic EMSO in [8], the logic for mobile ambients [5], or the tile logic [6]) deserves to be investigated. Finally, Floyd’s method fits well with unstructured programs; consequently, it may be useful to introduce a Hoare-like logic for more structured rv-programs (structured programming “in space” requires a more restricted and structured object invocation compared to the current practice in OO programming).

Acknowledgments. Special thanks go to my students in Bucharest who patiently followed a course on the foundations of interactive systems and provide good feedback, particularly to Oana Bulibasa, Mihai Codescu, Paul Neata, Vlad Radu, Alexandru Sofronia, and Alexandru Trofin.

References