Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

Informal Languages and Programming

Seventh Colloquium, Noordwijkerhout, the Netherlands

Edited by J. van Leeuwen

Springer-Verlag Heidelberg New York 1980
1. Introduction

We present semantic analysis techniques for concurrent programs which are designed as networks of nondeterministic sequential processes, communicating with each other explicitly, by the sole means of synchronous, unbuffered message passing. The techniques are introduced using a version of Hoare[78]'s programming language CSP (Communicating Sequential Processes).

One goal is to propose an invariance proof method to be used in the development and verification of correct programs. The method is suitable to partial correctness, absence of deadlock, and non-termination proofs. The design of this proof method is formalized so as to prepare the way to possible alternatives.

A complementary goal is to propose an automatic technique for gathering information about CSP programs that can be useful to both optimizing compilers and program partial verification systems.

Aims

The aim of formally verifying programs is to formally define so as to capture all the essential features of CSP.

Programs map from \( P_1(t) \), \( P_2(t) \), etc., where \( t \) is the program consisting of a single parallel-composition of sequential executions.

Processes consist of \( P_1(t) \), \( P_2(t) \), etc., where \( t \) is the program consisting of a single parallel-composition of sequential executions.

Each process has a unique name \( P_i(t) \) and consists of a sequence of simple commands, which consist of local variables.

Process labels \( P_i(t) \), \( i \in \{1, 2\} \).

Declarations \( D_i(t) \), \( i \in \{1, 2\} \).

Variables \( X_i(t) \), \( i \in \{1, 2\} \).

Types \( T_i(t) \), \( i \in \{1, 2\} \).

Program locations \( M_1(t) \), \( M_2(t) \), etc.

Each command has been labeled to assist future references.

Simple commands \( S_i(t) \), \( i \in \{1, 2\} \).

Null commands \( \lambda_x(t) \), \( \lambda_y(t) \), where \( x, y \) are variables.

Université de Metz, Laboratoire des Sciences de l'Ile de Sauley, 57000 Metz, France.

This work was supported by INRIA (SESORI, 76208) and by CNRS (AP Intelligence Artificielle).
(The pattern-matching feature introduced in Hoare[78] is treated using dynamic typing. Multiple assignments or assignments to parts of structured variables are realized using global assignments to variables).

Test commands $S(j, i) \mid 1 \leq i \leq 1$.

Stop commands $S(j, i) \mid 1 \leq i \leq 1$.

Specifying termination of process $P(1)$

Communication commands $S(i, j) \mid 1 \leq i \leq 1$.

Input-Output actions $G(j, i) \mid 1 \leq i \leq 1$.

Input guards $G(j, k) \mid 1 \leq i \leq 1$.

Expressions $e(i, j) \mid 1 \leq i \leq 1$.

Boolean expressions $b(i, j) \mid 1 \leq i \leq 1$.

The following abbreviations will be used:

$P(i, j) \equiv (i, j) \in 1$.

$P(i, j) \equiv (i, j) \in 1$.

This syntax is best understood as a 3-tuple C.A.T; use \texttt{with} syntax for some examples.
2 Characterization of the States that a Process can Reach after a Communication

When process $P(i)$ is at location $c$ with values $x$ of its local variables, the transition $\tau(i) \in \mathcal{S}(i) \times \mathcal{S}(i) + \mathcal{B}$ is successfully executable only if $\text{Oge}(i, x, c)$ is true.

Oge(i, x, c) = $\mathcal{S}(i) \times \mathcal{B}$, $\text{ie}(i, c, x, c')$

Oge(i, x, c) = $\lambda(x, c) \cdot \text{ie}(i, c, x, c') \land x \in \text{dom}(\mathcal{B}(i, j, k), x)$

If $\mathcal{B}(i, j, k)$ is a relation then $\text{ie}(i, c, x, c')$ denotes its reflexive transitive closure.

The states $(x, c)$ that process $P(i)$ can reach after execution of output guard, $\text{ie}(i, j, k)$, in state $(x, c)$, and before meeting a communication or stop command are such that $\text{Read}(i, j, k)(x, c, c')$ is true.

Read(i, j, k) = $\mathcal{S}(i) \times \mathcal{S}(i) + \mathcal{B}$, $\text{ic}(i, k, x, c)$

Read(i, j, k) = $\lambda(x, c) \cdot \text{ic}(i, k, x, c) \land x \in \text{dom}(\text{b}(i, j, k), x)$

When process $P(i)$ is at location $c$ with values $x$ of its local variables the input guard $\text{Ige}(i, j, k)$ is successfully executable only if $\text{Ige}(i, j, k)(x)$ is true.

Ige(i, j, k) = $\mathcal{S}(i) + \mathcal{B}$, $\text{ic}(i, j, k, x)$

Ige(i, j, k) = $\lambda(x, c, c') \cdot \text{ic}(i, j, k, x) \land x \in \text{dom}(\text{b}(i, j, k), x)$

If $\text{Oge}(i, x, c)$ is a family of sets and $x \in \mathcal{O}(i; \text{ie}(i, c), j, k)$ then $\text{sub}(x)(i, j, k)$, which is a subset of $\{ \text{v}(i, j, k) | \text{v}(i, j, k) = \text{v}(i, j, k) \}$ for all $i, j, k$ such that $\text{ie}(i, c, x, c')$, is true.

sub(x)(i, j, k) = $\{ \text{v}(i, j, k) | \text{v}(i, j, k) \in \text{sub}(x)(i, j, k) \}$

2.2.5 Operational Semantics of Communicating Processes

We introduce the transition relations $\tau(i)$ and $\gamma(i)$ which describe the cooperation of concurrently operating processes. Concurrency in the execution of a program is modeled by global nondeterminism in the selection of successor states. The resolution of the global nondeterminism is left unspecified since CSP definition specifies no scheduling policy whether fair or unfair.
2.2.3.1 States

\[ S = \Sigma \times L \]

(When a process is willing to accept a rendez-vous, the states of all other processes may have to be checked in order to determine which processes are ready to communicate or have terminated and next which data are exchanged.)

2.2.3.2 Transition Relations

- \( C(i) = ([i] \cup H(i), ic[i, m]) \)
  (The only program locations relevant to cooperation between processes are those corresponding to communication or stop commands).

- \( 1 \in [[S \times S] \rightarrow B] \)

- \( \lambda((x, a), (x, b)) \quad \forall (i, 1) \in [i] \cup H(i) \quad \forall (a, b) \in \alpha[i] \) \quad \forall \chi \in \{ \lambda \}

\[ \chi \in \text{dom}(\lambda) \]

(if e, d, i, e are sets, \( (e \cup i) \) and \( d \in \chi \) then \( \chi \) is defined as \( \{ \chi \} \)

\[ \chi \in \text{dom}(\lambda) \]

(If \( C(i) \) is a set of communication channels isomorphic with the set of statically matching pairs of input-output guards).

- \( \mu \in [[S \times S] \rightarrow B] \)

- \( \lambda((x, a), (x, b)) \quad \forall (i, 1) \in [i] \cup H(i) \quad \forall (a, b) \in \alpha[i] \) \quad \forall \chi \in \{ \lambda \}

\[ \chi \in \text{dom}(\lambda) \]

(All \( C(i) \) for \( i = 1 \) to \( m \) are morphisms.]

3. A FRAMING CHARACTERIZATION OF CORRECTNESS PROPERTIES

3.1 Fundamental Theorem
By definition the set of states which may be reached during any execution of program $P$, starting with an initial value of the variables satisfying the entry specification $\phi E$ is characterized by $\text{Post}(\phi)(\text{Init}(\phi))$. Notice that when programs are non-deterministic $\text{Post}$ characterizes possible but not necessarily certain descendants of the entry states. The following fixpoint characterization of $\text{Post}(\phi)(\text{Init}(\phi))$ is the basis of our approach (Cousot [73]).

$$f \in [E \rightarrow [P \rightarrow P]]$$
$$f = A\phi \left[ A\phi \left[ \text{Init}(\phi) \vee \text{Post}(\phi)(\phi) \right] \right]$$

where $< \left[ E \rightarrow [P \rightarrow P] \right>$ is the least fixpoint operator for $\phi E$ over the complete lattice $P$ (Cousot & Cousot [73]).

**Theorem 3.1.1**

$$\forall \phi E, \text{Post}(\phi)(\text{Init}(\phi)) = \text{Lfp}(f(\phi))$$

The above fixpoint theorem leads to sound and complete invariant proof methods for verifying invariance properties of programs. However, in order to put these methods into practice one or several applications of the following step are required.

### 3.2 [Pre]homomorphic Image of the Predicate Algebra

Let $A = \langle \text{false}, \text{true}, \wedge, \neg \rangle$ be a uniquely complemented complete lattice of "assertions". The meaning of $\phi$ is defined by a false-strict $\wedge$-complete morphism from $\langle \text{false}, \text{true}, \wedge, \neg \rangle$ into $A = \langle \text{false}, \text{true}, \wedge, \neg \rangle$. $\langle B \rangle$ is the representation of a "predicate" $\phi E$ by an "assertion" belonging to $A$. Corresponding to $\phi$, let us introduce $\text{Fp}(\phi) \in [A \rightarrow A]$ defined as $\lambda \phi \langle \text{Init}(\phi) \vee \text{Post}(\phi) \rangle$, where $\text{Init}(\phi) \longrightarrow A$ and $\text{Post}(\phi) \rightarrow A$. It is easy to see that $\text{Fp}(\phi)$ is monotone and $\phi \preceq \phi'$ if and only if $\forall \phi E, \text{Fp}(\phi) \preceq \text{Fp}(\phi')$.

**Theorem 3.2.1**

The importance of the homomorphism $\phi E \rightarrow \text{Fp}(\phi)$ is that it shows that the ever increasing upper approximation of a program, $\text{Fp}(\phi)$, is monotone equivalent to the predicate morphism of the program, and it is a valid upper approximation of the program.
3.4 Analysis of the Behavior of Individual Processes

A global assertion about the states of process $P(i)$ can be replaced by a set of assertions about the values of the process variables preceding each communication.

3.4.1 Analysis of the Behavior of Individual Processes Independently of Communication

By definition, $Tr(i)[j,k] \in [t(x_1)t(x_2)] \Rightarrow [E(x_1) + E(x_2)]$. This is true because $P(i)$ starts from location $A(i,j)$ with the initial state $x_0$ and without encountering communication commands. The following characterization of $Tr(i)[j,k]$ as a fixpoint together with Conlat's theorem characterizes the iteration, where $\Lambda(i)$ is the symbolic execution with this difference that all paths are followed simultaneously and infinite paths handled by induction.

- $Post(i) = [E(x_1) + E(x_2)]$.
- $Post(i) = A(i) + B(x_i)$. 
- $FK(i) = A(i) + B(x_i)$.
- $FK(i) = \lambda x. (x_1)(x_2) + Post(i)[t(x_1)[i,i+1] + Post(i)[t(x_2)[i,i+1]]$
3.4.2 Analysis of the Behavior of Individual Processes taking Communications into Account

We define \( Desc(\varphi) \) characterizing the possible values that local variables \( x(i) \) can possess at run-time when location \( A[i,j] \) of process \( P(i) \) is reached during an execution of the program starting from an initial state of the local variables \( x(k) \). \( k \in [1, \pi] \), satisfying the entry specification \( \varphi \).

\[
\begin{align*}
\text{Desc} & \in [E \times A] \\
\text{Desc} & = \lambda x.[A \times \lambda x \times \lambda y . [\{\text{Post} A[i] y x \times \lambda k \times \lambda \varphi (y) x \times \lambda \} \times \lambda i \times \lambda j \times \lambda \varphi x \}]
\end{align*}
\]

The local descendants of the entry states are either direct descendants or the entry states of the states following either an output or an input transition. Since no new values can be generated when an initial state for the local variables is provided and the input from other processes (as determined at paragraph 3.3) is known, at a stage of the program before a communication, the state of the communicating program before this communication is obtained by the following representation:

\[
\begin{align*}
\text{Desc} & \in [E \times A] \\
\text{Desc} & = \lambda x.[A \times \lambda x \times \lambda y . [\{\text{Post} A[i] y x \times \lambda k \times \lambda \varphi (y) x \times \lambda \} \times \lambda i \times \lambda j \times \lambda \varphi x \}]
\end{align*}
\]

3.5 Example

The systems of equations \( X = F(X)(B(X)) \), where \( A, B \) are the following...
non-trivial programs are not mechanically computable. Even by hand, such calculations cannot be worked out since they are amazingly complex. The solution to this intricacy is the idea of approximation which is central to proof methods and automatic program analysis technique.

4. INVARIANCE PROOF METHODS

4.1 Outline of Our Approach

Let a program be invariant during execution of program P if starting with any state satisfying the empty specification of P if and only if satisfying P.

4.1.1 Fundamental Invariance Proof Method

The explicit characterization of Post(P)(n) given by theorem 3.1.1 leads to the general complete invariance-proof method:

\[ \text{Theorem 3.1.1:} \quad \text{Post}(P)(n) = \text{Post}(P)(0) \quad \text{if} \quad \text{Post}(P)(n) = \text{Post}(P)(n+1) \]

This invariance proof method fits for use in partial correctness, absence of loops, and non-termination proofs (the only differences is with respect to the closure of P).

4.1.2 Prehomomorphic Variants of the Fundamental Proof Method

Prehomomorphic images of the predicate algebra, i.e. homomorphic images, need to be introduced to space to call the fundamental completeness completeness of the fundamental proof method.

The soundness of soundness and completeness of these variants of the fundamental proof method follow from the following:

\[ \text{Theorem 4.1.2:} \quad \text{Let} \quad \text{and} \quad \text{P}, \text{ respectively, be some operators on the complete lattices} \]

\[ \text{and} \quad \text{and} \quad \text{P}, \text{ respectively, be some operators on the complete lattices} \]

\[ \text{P}(x) = \text{Post}(A)(x) \quad \text{if} \quad \text{Post}(A)(x) = \text{Post}(A)(x+1) \]

\[ \text{and} \quad \text{P}(x) = \text{Post}(B)(x) \quad \text{if} \quad \text{Post}(B)(x) = \text{Post}(B)(x+1) \]

\[ \text{Lemma 4.1.2:} \quad \text{Let} \quad \text{and} \quad \text{P}, \text{ respectively, be some operators on the complete lattices} \]

\[ \text{and} \quad \text{and} \quad \text{P}, \text{ respectively, be some operators on the complete lattices} \]

\[ \text{P}(x) = \text{Post}(A)(x) \quad \text{if} \quad \text{Post}(A)(x) = \text{Post}(A)(x+1) \]

\[ \text{and} \quad \text{P}(x) = \text{Post}(B)(x) \quad \text{if} \quad \text{Post}(B)(x) = \text{Post}(B)(x+1) \]

\[ \text{for example using the homomorphic image of the predicate algebra described above, one can get the following particular invariance approximation,} \]

\[ \text{Post}(A)(x) = \text{Post}(B)(x) \quad \text{if} \quad \text{Post}(B)(x) = \text{Post}(B)(x+1) \]

\[ \text{and} \quad \text{P}(x) = \text{Post}(A)(x) \quad \text{if} \quad \text{Post}(A)(x) = \text{Post}(A)(x+1) \]

\[ \text{and} \quad \text{P}(x) = \text{Post}(B)(x) \quad \text{if} \quad \text{Post}(B)(x) = \text{Post}(B)(x+1) \]
4.3 A Sound Variant Without Program Location Counters

For the sake of completeness in the invariance proof method 4.2.1 the assertions \( \text{IsAg} \) may have to take the values of the program location counters into account. Yet, on grounds of methodology, reasonings about program location counters are usually not desirable. In this section, we will introduce a variant that introduces a new function \( \text{truecount} \).

\[
\text{truecount} = \left[ \begin{array}{c}
\text{if } \phi \text{ is } \text{false}\text{-strict then choose } \text{false} \text{-strict } \\
\text{else } \text{truecount} \text{ is upper semi-continuous } \\
\end{array} \right]
\]

For example, one can choose:

\[
\begin{array}{l}
\text{truecount} = \lambda \chi. \lambda x. \left[ \begin{array}{c}
\text{if } x = \text{false then choose } \text{false} \\
\text{else } \text{truecount} \text{ is upper semi-continuous } \\
\end{array} \right]
\end{array}
\]

The soundness of the corresponding invariance proof method is shown in a general setting by the following:

\[
\text{truecount} \text{ is a } \text{false}\text{-strict }\text{v}\text{-complete morphism.}
\]

Corollary 4.3.2

The soundness of the corresponding invariance proof method is shown in a general setting by the following:

\[
\begin{array}{l}
\text{false}\text{-strict }\text{v}\text{-complete morphism.}
\end{array}
\]

Introducing Auxiliary Variables for Completeness

The reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.

Corollary 4.3.2

Introducing Auxiliary Variables for Completeness

In general, the reciprocal of theorem 4.3.2 is not true for programs involving two processes. A completeness result can nevertheless be obtained using auxiliary variables. The use of auxiliary variables is, in fact, essential in proving 4.2.1.
Involving total functions, therefore whenever \( ax \) or \( ay \) we have:

\[
\forall x, y \in \text{dom}(f), f(x, y) = f(y, x)
\]

Assume the program \( P_{ax} \) is transformed into a program \( P_{ay} \) by replacing all assignments to auxiliary variables by null statements and by deleting all declarations of auxiliary variables. The operational semantics of \( P_r \) defines a set of program locations \( L_{ax} \), a set of states \( S_{ax} \), and transition relations \( r_{ax} : L_{ax} \times S_{ax} \rightarrow L_{ax} \times S_{ax} \) such that:

- \[ r = \lambda(x, y, z) \rightarrow (x, y, z) \in S \]
- \[ r = \lambda(x, y, z) \rightarrow (x, y, z) \in S \]

A correspondence can be established between the entry specifications \( \phi \in E_{ax} \) for \( P_{ax} \) and \( \phi \in E_{ay} \) by elimination of the auxiliary variables:

\[
E_{ax} = \lambda \phi \cdot [\lambda x : [\lambda x : [\phi(\lambda x : x, y)]]]
\]

The same way, assertions \( \psi \in A_{ax} \) interleaved in \( P_{ax} \), can be connected to assertions \( \psi \in A_{ay} \) interleaved in \( P_{ay} \) by eliminating auxiliary variables:

\[
A_{ax} = \lambda \psi \cdot \lambda \psi(\lambda x : \lambda x : \lambda x : \lambda x : \psi(\lambda x : \lambda x : \lambda x : \lambda x : \lambda x : x, y, z))
\]

Proofs about \( P_{ax} \) using \( F_{ax} \) (hence referring to auxiliary variables) but program counters can be directly used for proving invariance properties of \( P_{ay} \).

**Theorem 4.4.1.1**

If \( f : ax \rightarrow ay \) is a false-strict v-complete morphism, \( f : ax \rightarrow ay \) is upper semi-continuous, and \( \forall a, b \in ax \; f(a, b) = f(b, a) \), then:

\[
\forall \phi \in E_{ax} \; \exists \psi \in E_{ay} \; \phi(a) = \psi(b)
\]

**Completeness**

Theorem 4.4.2.1

Given an arbitrary program \( P_{ax} \), let \( P_{ay} \) be the equivalent transformation obtained from \( P_{ax} \) such that every command is preserved by a null command. Assume \( f \) such that \( f : ax \rightarrow ay \) is a false-strict v-complete morphism and \( f : ax \rightarrow ay \) is upper semi-continuous. Then exist \( f : ax \rightarrow ay \) and \( P_{ay} \) validating \( \phi(a) = \psi(b) \), where:

\[
\forall \phi \in E_{ax} \; \exists \psi \in E_{ay} \; \phi(a) = \psi(b)
\]
Examples of Proofs

Given two disjoint sets of integers $S_0$ and $T_0$, $S_0 \cup T_0$ has to be partitioned into two sets $S$ and $T$ such that $S \cap T = \emptyset$, and every element of $S$ is smaller than any element of $T$.

Entity specifications:

- $\phi(x) = [a \in S \mid a + x \leq a]$ (for $x$ in $S$)
- $\psi(x) = [a \in S \mid a + x > a]$ (for $x$ in $S$)

To set it short, the program used is:

```
if $x$ in $S$
    if $x < a$
        add $x$ to $S$
    else
        add $x$ to $T$
```

The following version of a program given by Apt, Françoise, and Genon (1978) uses the

```
if $x$ in $S$
    if $x < a$
        add $x$ to $S$
    else
        add $x$ to $T$
```

which exchange the current maximum of $S$ with the current minimum of $T$ until $\max(S) = \max(T)$.

P1: % mx.x = mx(S) % S = S invocation

```
repeat
    mx = max(S)
    $S,mx \cup S,mx$ % S := S \cup {mx}
    $T,mx \leftarrow T,mx$ % T := T \setminus {mx}
    $S \leftarrow S \setminus \{mx\}$
    $T \leftarrow T \setminus \{mx\}$
    if $x \in S$
        if $x < mx$
            add $x$ to $S$
        else
            add $x$ to $T$
    until $mx = \max(S)$
```

---

We first obtain the following assertions $TH(x)(1,1)$ of the transformation of the
values of the variable $x$: when $x$ is executed starting at an entry point in after

```
if $x$ in $S$
    if $x < a$
        add $x$ to $S$
    else
        add $x$ to $T$
```

intermediate communications. Since no loop is involved, no detective assertion is

```
if $x$ in $S$
    if $x < a$
        add $x$ to $S$
    else
        add $x$ to $T$
```

necessary for the equations of the loop.

---

The assertions $ICH(1)$, $ICH(2)$, $ICH(3)$ respectively associated with the
communications

```
if $x$ in $S$
    if $x < a$
        add $x$ to $S$
    else
        add $x$ to $T$
```

are the following:

```
ICH1(x) = \{ \text{if} \emptyset \in \{x \mid x < a \} \text{ then } S \text{ else } T \}
ICH2(x) = \{ \text{if} \emptyset \in \{x \mid x > a \} \text{ then } S \text{ else } T \}
ICH3(x) = \{ \text{if} \emptyset \in \{x \mid x < a \} \text{ then } S \text{ else } T \}
```

---

The independent analysis of the program:

```
repeat
    mx = max(S)
    $S,mx \cup S,mx$ % S := S \cup {mx}
    $T,mx \leftarrow T,mx$ % T := T \setminus {mx}
    $S \leftarrow S \setminus \{mx\}$
    $T \leftarrow T \setminus \{mx\}$
    if $x \in S$
        if $x < mx$
            add $x$ to $S$
        else
            add $x$ to $T$
    until $mx = \max(S)$
```

end loop

---
5. AUTOMATIC PROGRAM ANALYSIS TECHNIQUES

Automatic program analysis techniques can be used to gather information about programs, which can help in program verification and reliability.

5.1 Outline of Our Approach

The design of a variety of automatic program analysis techniques (Cousot & Cousot[79a]) is tantamount to defining for each program a predomainic image \( A(I,L,F) \) by \( P[t\in\{\text{false},\text{true}\}] \) using an approximation operator \( P[t(F+A)] \) which is a strict join-complete monoid. In addition to the hypotheses of paragraph 4.2, the elements of \( A \) are chosen to be computer-representable and \( F \) must be such that \( \forall \overline{M} \in \text{LPP}(F) \) is either iteratively computable (using any chaotic iteration strategy, Cousot[77]) or approximable from data.

Once understood in this way, the numerous global flow analysis algorithms which have been developed in the literature for sequential programs can be generalized to scSP. The main difficulty of these generalizations has been solved, at paragraph 3.3 where \( \gamma \), which was a priori not necessary, was shown necessary for the program text. Program location counters can be dispensed with, too, for programs involving more than one dimension.
6. CONCLUSIONS

We have shown that program analysis techniques can be used to reason about the behavior of systems with...
7. REFERENCES


