1. Static Execution Model

Let \( F = \langle V, A, s, E \rangle \) be a CFG. A serial execution of a program is modeled as a path in the CFG from the initial vertex \( s \) to a vertex in the set \( E \) of end vertices. We assume that program execution progresses from a start node through a series of steps which may be numbered starting from 0, and that execution continues until the program reaches an end state or enters an infinite loop.

We consider that a step involves execution of a single node in the CFG, and we speak of the execution being at a particular state when it executes a particular state. We define:

**Definition 1.1. Serial state:** The \( j \)th state \( \sigma_j \) of a serial execution is a tuple:

\[
\sigma_j = (x_j, D_j)
\]

where \( x_j \) denotes execution of the basic block \( x \) at step \( j \), and \( D_j \) is the set of data values stored in memory by the process when execution reaches the entry point of the basic block after performing \( j \) transitions from the start of execution.

\( D_j \) depends on the entire execution history. Every state implies execution of one basic block, and every transition is a change from one basic block to another along an arc of the CFG. The basic blocks appear in an order determined by the CFG, however the state number \( j \) is a simple count of steps or transitions; it always increases in a serial execution.

**Remark 1.2.** We have here defined states in terms of basic blocks. It should be evident that an equivalent definition could be made for smaller units of code, e.g. individual statements, or sets of consecutive statements inside a basic block.

We do not extend a state beyond a single basic block because we could not, in such a multi-block state, statically determine what actions or transformations on memory are performed in the state. Specifically, since we do not know that execution will always follow one block to a particular other block, we can not know whether statements in both blocks will or will not be executed consecutively until runtime.

We define a transition from one state to a next state:

**Definition 1.3.** Given a CFG, \( F = \langle V, A, s, E \rangle \), the transition \( \sigma_j \rightarrow \sigma_{j+1} \) is a relation between serial states such that \( x_j, y_{j+1} \in V \) are nodes in the CFG and there is an arc \( x \rightarrow y \in A \).

We now define a serial execution as follows:

**Definition 1.4.** Given \( F = \langle V, A, s, E \rangle \), let \( i \) be a step number, and let \( x_i \in V \) denote the particular node of the CFG executed at step \( i \). Let \( s \) be the start node and \( e \in E \) an end node. Then a **serial execution** is a series of states that correspond to a path from \( s \) to either an end node \( e \) or divergence designated by \( \perp \). We denote a path \( P \) of an execution with steps from 0 to \( n \) by:

\[
P = s \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_{n-1} \rightarrow e \land \perp
\]

and an execution \( X \) as:

\[
X = \sigma_s \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_e \land \perp
\]

The **serial steps**, denote successive execution of basic blocks in the CFG starting from the initial node \( s \) and ending in one of the final nodes \( e \in E \); the step number at a particular node indicates the number of blocks that have been executed from the start to reach that node in \( X \).
Note that the order of execution of the nodes \( x_i \) is only restricted by the CFG. There is no requirement that the nodes appear in any order. In particular, a given node in a loop may appear multiple times in \( X \); however the step number is different each time a particular node appears in \( X \).

We call such an execution time-like, in that the sequence of increasing step numbers gives us a notion of time and a concept of progress for the serial execution. Since we always transition to a higher-numbered step, we can say that such a step is always after, or later than, any lower-numbered step. Progress includes the assumption that the process does not fail to advance to a next step if one exists (the process is not at an end node).

Another way of putting this is to observe that, in order to reach a particular step number, we must transition through all the lower-numbered steps, and we must not have reached any of the higher-numbered steps in the sequence. This gives us a serial execution.

It can be shown that this form of execution is deterministic. A deterministic execution may not terminate, for example due to an infinite cycle. Execution could still be deterministic if it always enters such a cycle given the same initial conditions. We handle such a case (as do Apt and Olderog - “Verification of Sequential and Concurrent Programs” - Springer) by appending a divergent state \( \bot \) to the set of states \( E \).

We define:

**Definition 1.5.** A deterministic serial transition is a state change \( \sigma_j \rightarrow \sigma_{j+1} \) such that \( \sigma_{j+1} \) is the unique successor of \( \sigma_j \). A serial execution of transitions of this type is a sequence of states each of which has an unique successor and is therefore a deterministic execution. The execution of a basic block is deterministic if there are no explicitly random instructions. In a deterministic execution, we have that the basic block \( x_j \) reached at step \( j \) is a function of \( j \), since each state has a unique successor.

The serial transition gives us a relation between serial states which defines one state as later than, or occurring after another.

**Definition 1.6.** Later: Given two serial states such that \( \sigma_j \rightarrow^* \sigma_i \) we will say that \( \sigma_j \leq \sigma_i \), since by the definition of serial execution 1.4 we must have \( j \leq i \) (recall that \( j \) and \( i \) are step numbers).

Another useful way of considering a serial execution is by considering its view of memory. Although \( D_j \), \( t \) defines all data that the process could possibly have access to at step \( j \), the process usually does not in fact read or write all this data. When considering only a single serial process, we may also wish to consider the changes that a process makes to \( D_j \) in a transition \( \delta_j \rightarrow \delta_{j+1} \). Note that in considering parallel processes, there is also the possibility that some of the data at one process will change due to the action of some other process.

We define:

**Definition 1.7.** At state \( \sigma_j \) of a process, the data view \( \nu_j \subseteq D_j \) is the data that is read or modified by the process at that step. The data view trace of a process is the list of the data view at each step. Given an execution that reaches an end state:

\[
X = \sigma_s \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_e
\]
the data view trace is:
\[ V = (\nu_0, \nu_1, \nu_2, \ldots, \nu_c) \]

The data view at a given basic block can be determined by standard data flow analysis as described in Aho-Ullman.

**Remark 1.8.** It is evident that a deterministic program executing on the same data will access the same memory in the same steps, and perform the same computation at each step on repeated execution. Therefore it will generate the same data view trace. We can turn this statement around: if the same deterministic serial program executes the same path in the CFG and sees the same data trace in two different executions, then it computes the same results. This is because the data view trace includes all the values that the program sees in its computation, and the steps at which it sees them. Since the program reads no data outside the data view trace, and its computation is deterministic, the results are the same.

### 2. FUNCTIONAL EXECUTION MODEL

A dynamic view of execution: Let \( x_j \) in state \( \sigma_j = (x_j, D_j) \) denote a single executable statement (rather than a basic block), so we get a finer-grained view of execution. Let \( M = < m_0, \ldots, m_n > \) be the set of consecutive statements in \texttt{main} (the function that is first executed in a program).

Assume for the present that all logic and control flow is managed through function calls - so that for example, an if statement appears as a statement \( m_i \) that calls a function \( A = < a_0, \ldots, a_m > \).

An execution might then appear as follows:
\[
E = m_0 \rightarrow m_1 \rightarrow \ldots \rightarrow m_i \rightarrow m_i+1 \rightarrow \ldots \rightarrow m_n
\]

if \( m_i \) evaluates to FALSE, otherwise if \( m_i \) evaluates to TRUE:
\[
E = m_0 \rightarrow m_1 \rightarrow \ldots \rightarrow m_i \rightarrow a_0 \rightarrow \ldots \rightarrow a_m \rightarrow m_{i+1} \rightarrow \ldots \rightarrow m_n.
\]

We do not know which of these occurs until we reach state \( \sigma_k = ((m_i)_k, D_k) \).

(We here use the somewhat clumsy notation \((m_i)_k\) to denote the execution of statement \( m_i \) at step \((\text{in state number}) \ k, \) to bring out the fact that \( k \) and \( i \) do not have to be equal.)

Suppose \( m_i \) is actually an if-then-else construct that does \( A \) if true and \( B \) if false. Then we have the alternative execution:
\[
E = m_0 \rightarrow m_1 \rightarrow \ldots \rightarrow m_i \rightarrow b_0 \rightarrow \ldots \rightarrow b_m \rightarrow m_{i+1} \rightarrow \ldots \rightarrow m_n.
\]

Loops appear similarly: let \( m_i \) be a \texttt{while} statement that repeats \( A \):
\[
E = m_0 \rightarrow * m_i \rightarrow a_0 \rightarrow * a_m \rightarrow m_i \rightarrow a_0 \rightarrow * a_m \rightarrow \ldots \rightarrow m_i \rightarrow m_{i+1} \rightarrow * m_n
\]

Here the sequence of statements in \( A \) appears repeatedly, always coming back to the while statement \( m_i \) which eventually ends the loop and continues to \( m_{i+1} \). (For compactness, we replace the sequence \( \rightarrow \ldots \rightarrow \rightarrow * \) to denote the execution of 0 or more transitions.)

**Remark 2.1.** Bohm and Jacopini (1962) showed that in fact if-then-else and while loops are sufficient to express any control structure (think about how to demonstrate such a thing?) - this implies that any (sequential) program execution can fit into this model.

Key property: 'nestedness' - we are using the same notion as in matching parenthesis and recursion. The idea that an action (the thing inside the parenthesis, the loop body, the function code) is something that completes and returns to whatever called it. We can have functions that call themselves or other functions, loops
within loops, whatever; but a called action always ends before whatever called it
(or: innermost parentheses always match each other!). The execution model given
above has this property; note that a function A returns to whatever calls it after it
executes all its code; note that, if a stack is used for A’s local variables, they can
be initialized on the stack on top of M’s memory, and cleaned up when A exits,
leaving M’s variables on top of the stack.