Formal Methods – An Appetizer

Chapter 1: Program Graphs

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

The Role of Models

A model is an abstraction of a system – it focuses on some parts of the system and ignores others.

A model may focus on the workflow of an organisation and ignore details about how the individual steps are carried out.

A model may dig into the details of how a cryptographic protocol or a sorting algorithm works.

A model may be extremely concrete and describe the detailed bit-level operation of computers.

What is common is that a model should describe not only the structure of the IT system but also its meaning – at an appropriate level of abstraction.

Program Graphs as Models

A Graphical Representation

A program graph focuses on how a system proceeds from one point to the next by performing one of the actions identified by the abstraction.

An Operational Approach

The meaning of the actions is formalised by defining their semantics and this gives rise to a precise description of the meaning of the program graphs themselves.

Different Levels of Abstraction

Depending on our focus the semantics may be at the level of bit strings, mathematical integers, or data structures such as arrays.

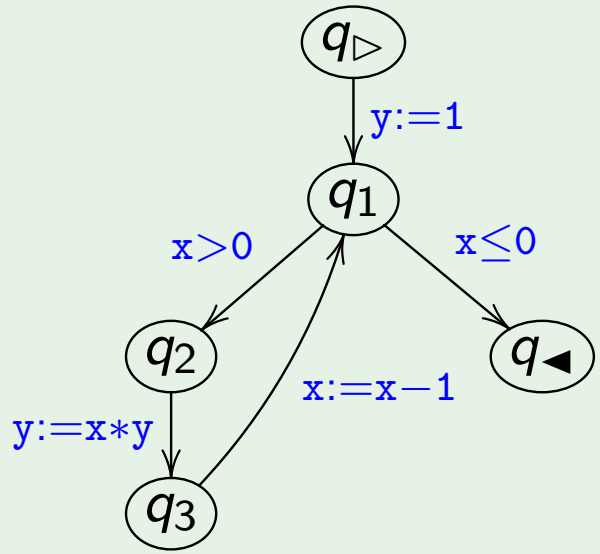
1.1 Program Graphs

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Program Graphs

Example (FM p 1)



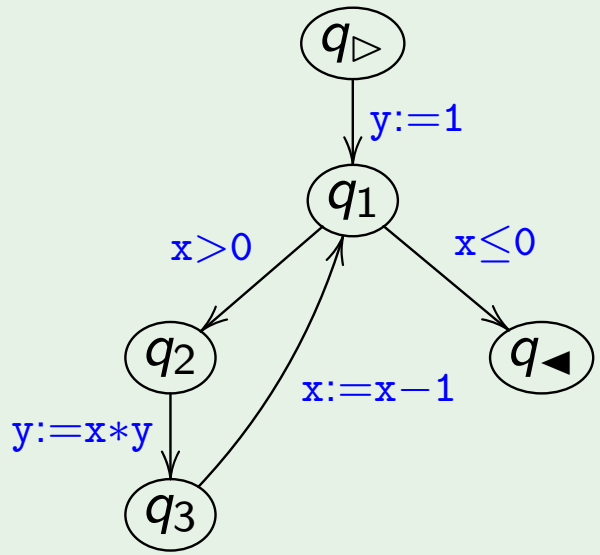
Idea: upon termination y should hold the value of the factorial of the initial value of x.

A program graph **PG** consists of (FM p 2)

- **Q**: a finite set of *nodes*
  $\{q_\triangleright, q_1, q_2, q_3, q_\blacktriangleleft\}$
- $q_\triangleright, q_\blacktriangleleft \in \mathbf{Q}$: two nodes where
  - $q_\triangleright$ is the *initial node* and
  - $q_\blacktriangleleft$ is the *final node*
- **Act**: a set of *actions*
  $\{\mathtt{y} := 1, \mathtt{x} > 0, \mathtt{x} \leq 0,$
  $\mathtt{y} := \mathtt{x} * \mathtt{y}, \mathtt{x} := \mathtt{x} - 1\}$
- $\mathbf{E} \subseteq \mathbf{Q} \times \mathbf{Act} \times \mathbf{Q}$: a finite set of *edges*
  $\{(q_\triangleright, \mathtt{y} := 1, q_1), (q_1, \mathtt{x} > 0, q_2),$
  $(q_1, \mathtt{x} \leq 0, q_\blacktriangleleft), (q_2, \mathtt{y} := \mathtt{x} * \mathtt{y}, q_3),$
  $(q_3, \mathtt{x} := \mathtt{x} - 1, q_1)\}$

Try It Out: Program Graphs

Example (FM p 1)



Idea: upon termination $\mathtt{y}$ should hold the value of the factorial of the initial value of $\mathtt{x}$.

Try It Out (FM p 2)

Specify a program graph for the power function computing the power 2^n of a number $n \geq 0$: upon termination the variable $\mathtt{y}$ should hold the value of the $\mathtt{x}$'th power of 2.

As actions you should use simple tests and assignments similar to the ones used for the factorial function.

1.2 Semantics

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Semantics of Program Graphs

A semantics consists of (FM p 4)

- a non-empty set **Mem** called the memory
- a semantic function $\mathcal{S}[\![\cdot]\!] : \mathbf{Act} \to (\mathbf{Mem} \hookrightarrow \mathbf{Mem})$ specifying the meaning of the actions

Idea

If $\mathcal{S}[\![\alpha]\!]\sigma = \sigma'$ and σ is the memory before executing α then σ' is the memory afterwards.

Example (FM p 4)

Memory: $(v_{\mathtt{x}}, v_{\mathtt{y}})$

Semantic function:

$$\mathcal{S}[\![\mathtt{y} := 1]\!](v_{\mathtt{x}}, v_{\mathtt{y}}) = (v_{\mathtt{x}}, 1)$$

$$\mathcal{S}[\![\mathtt{x} > 0]\!](v_{\mathtt{x}}, v_{\mathtt{y}}) = \begin{cases} (v_{\mathtt{x}}, v_{\mathtt{y}}) & \text{if } v_{\mathtt{x}} > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![\mathtt{x} \leq 0]\!](v_{\mathtt{x}}, v_{\mathtt{y}}) = \begin{cases} (v_{\mathtt{x}}, v_{\mathtt{y}}) & \text{if } v_{\mathtt{x}} \leq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![\mathtt{y} := \mathtt{x} * \mathtt{y}]\!](v_{\mathtt{x}}, v_{\mathtt{y}}) = (v_{\mathtt{x}}, v_{\mathtt{x}} * v_{\mathtt{y}})$$

$$\mathcal{S}[\![\mathtt{x} := \mathtt{x} - 1]\!](v_{\mathtt{x}}, v_{\mathtt{y}}) = (v_{\mathtt{x}} - 1, v_{\mathtt{y}})$$

Semantics of Program Graphs

Configurations (FM p 5)

A configuration is a pair $\langle q; \sigma \rangle$ where $q \in \mathbf{Q}$ and $\sigma \in \mathbf{Mem}$.

An initial configuration has $q = q_\triangleright$.

A final configuration has $q = q_\blacktriangleleft$.

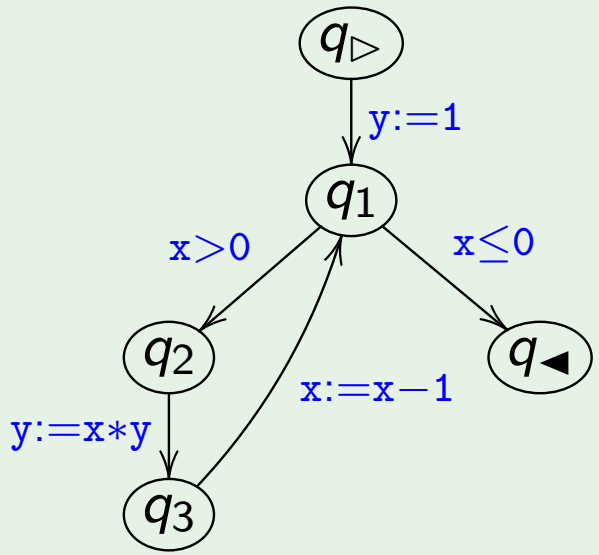
Execution Step (FM p 5)

Whenever $(q_\circ, \alpha, q_\bullet) \in \mathbf{E}$ we have an execution step

$$\langle q_\circ; \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle q_\bullet; \sigma' \rangle \text{ if } \mathcal{S}[\![\alpha]\!]\sigma = \sigma'$$

If $\mathcal{S}[\![\alpha]\!]\sigma$ is undefined there is no execution step.

Example (FM p 1)



Example configuration:

$$\langle q_2; (3,1) \rangle$$

Example execution step:

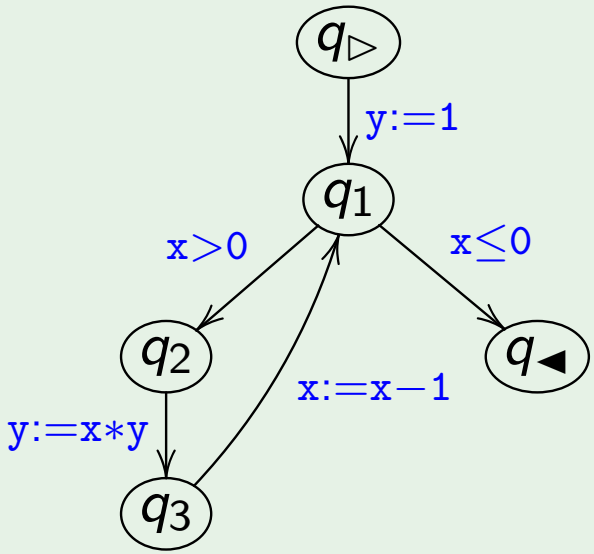
$$\langle q_\triangleright; (3,7) \rangle \stackrel{\text{y:=1}}{\Longrightarrow} \langle q_1; (3,1) \rangle$$

Semantics of Program Graphs

Complete Execution Sequence (FM p 6)

This is a sequence of execution steps starting in an initial configuration and ending in a final configuration.

Example (FM p 1)



$$
\begin{aligned}
&\langle q_\triangleright; (3,7)\rangle &&\overset{\texttt{y:=1}}{\Longrightarrow}\\
&\langle q_1; (3,1)\rangle &&\overset{\texttt{x>0}}{\Longrightarrow}\\
&\langle q_2; (3,1)\rangle &&\overset{\texttt{y:=x*y}}{\Longrightarrow}\\
&\langle q_3; (3,3)\rangle &&\overset{\texttt{x:=x-1}}{\Longrightarrow}\\
&\langle q_1; (2,3)\rangle &&\overset{\texttt{x>0}}{\Longrightarrow}\\
&\langle q_2; (2,3)\rangle &&\overset{\texttt{y:=x*y}}{\Longrightarrow}\\
&\langle q_3; (2,6)\rangle &&\overset{\texttt{x:=x-1}}{\Longrightarrow}\\
&\langle q_1; (1,6)\rangle &&\overset{\texttt{x>0}}{\Longrightarrow}\\
&\langle q_2; (1,6)\rangle &&\overset{\texttt{y:=x*y}}{\Longrightarrow}\\
&\langle q_3; (1,6)\rangle &&\overset{\texttt{x:=x-1}}{\Longrightarrow}\\
&\langle q_1; (0,6)\rangle &&\overset{\texttt{x≤0}}{\Longrightarrow}\\
&\langle q_\blacktriangleleft; (0,6)\rangle
\end{aligned}
$$

Try It Out: Complete Execution Sequence

Example (FM p 4)

Memory: (v_x, v_y)

Semantic function:

$$\mathcal{S}[\![y := 1]\!](v_x, v_y) = (v_x, 1)$$

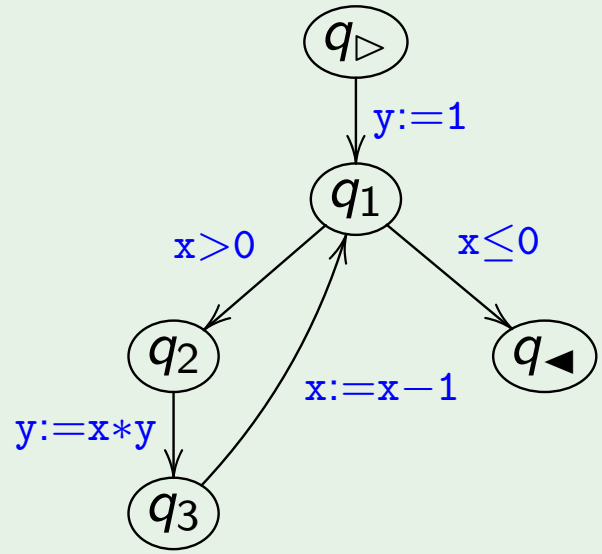
$$\mathcal{S}[\![x > 0]\!](v_x, v_y) = \begin{cases} (v_x, v_y) & \text{if } v_x > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![x \leq 0]\!](v_x, v_y) = \begin{cases} (v_x, v_y) & \text{if } v_x \leq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![y := x * y]\!](v_x, v_y) = (v_x, v_x * v_y)$$

$$\mathcal{S}[\![x := x - 1]\!](v_x, v_y) = (v_x - 1, v_y)$$

Example (FM p 1)



Try It Out (FM p 6)

Determine the complete execution sequence

$$\langle q_\triangleright; (4,4)\rangle \overset{\omega}{\Longrightarrow}^* \langle q_\blacktriangleleft; (v_x, v_y)\rangle$$

1.3 The Structure of the Memory

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Memories for Variables

Variables

For variables we model memories as mappings

$$\sigma : \textbf{Var} \to \textbf{Int}$$

We write $\sigma(x)$ for the value of x.

We write $\sigma[x \mapsto v]$ for the memory that is the same as σ except that the value of the variable x now has the value v.

Example (FM p 7)

A memory σ and the updated memory $\sigma[\mathtt{x} \mapsto 13]$:

x	12
y	5

σ

x	13
y	5

$\sigma[\mathtt{x} \mapsto 13]$

Memories

Variables and Arrays

We model memories as mappings

$$\sigma : \Big(\ \mathbf{Var} \cup \{A[i] \mid A \in \mathbf{Arr}, 0 \leq i < \mathrm{length}(A)\} \ \Big) \to \mathbf{Int}$$

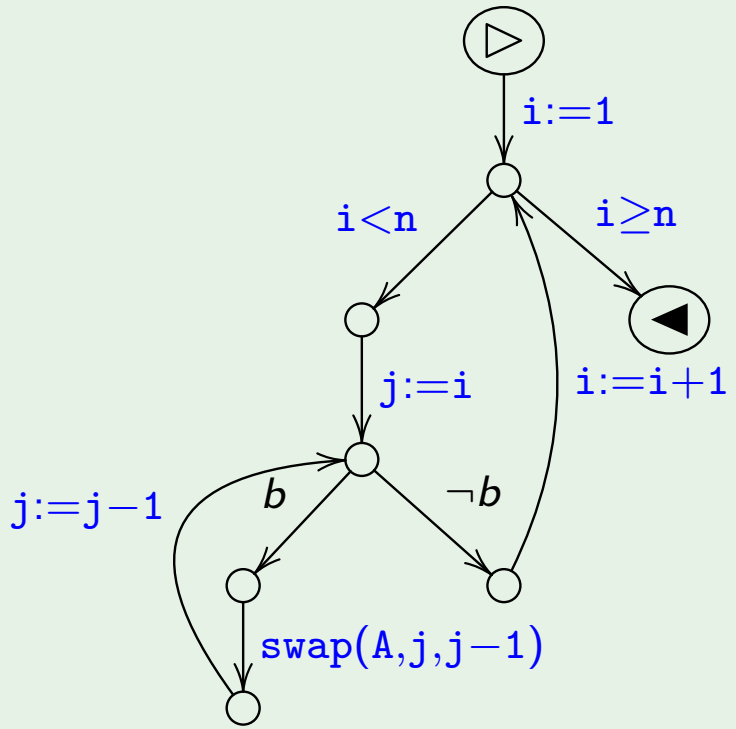
We write $\sigma(A[i])$ for the value of the i'th entry of the array A (provided $0 \leq i < \mathrm{length}(A)$).

We write $\sigma[A[i] \mapsto v]$ for the memory that is the same as σ except that the i'th entry of the array A now have the value v.

Example (FM p 7)

n	4
i	1
j	2
A[0]	4
A[1]	2
A[2]	17
A[3]	9

Example: Insertion sort (FM p 8)



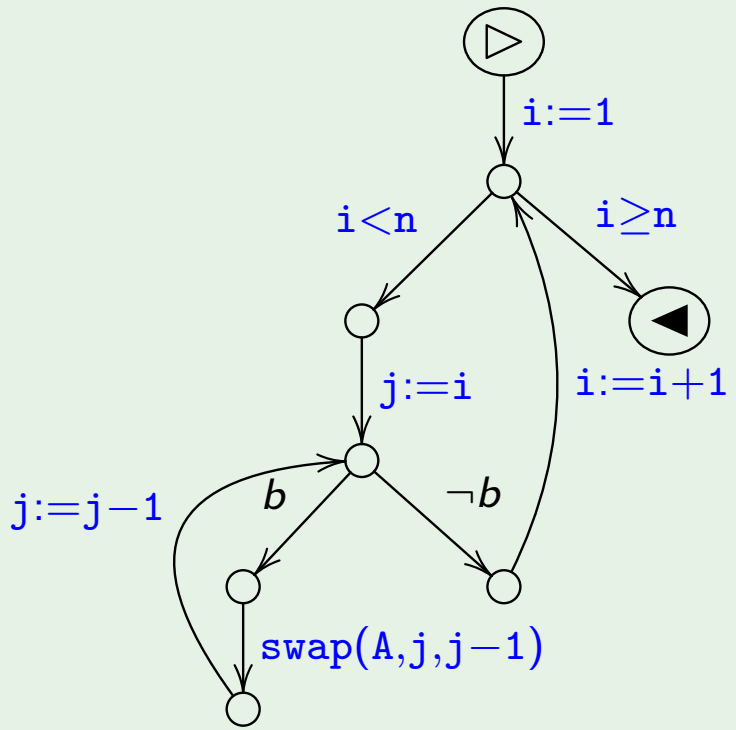
b abbreviates $\texttt{j} > 0 \;\&\&\; \texttt{A}[\texttt{j}-1] > \texttt{A}[\texttt{j}]$

n	4
i	1
j	2
A[0]	4
A[1]	2
A[2]	17
A[3]	9

$$\mathcal{S}[\![\texttt{swap}(\texttt{A},\texttt{j},\texttt{j}-1)]\!]\sigma = \begin{cases} \sigma[\texttt{A}[j] \mapsto v_{j-1}][\texttt{A}[j-1] \mapsto v_j] \\ \qquad \text{if } \sigma(\texttt{j}) = j \text{ and } \texttt{A}[j], \texttt{A}[j-1] \in \mathrm{dom}(\sigma) \\ \qquad \text{and } v_j = \sigma(\texttt{A}[j]) \text{ and } v_{j-1} = \sigma(\texttt{A}[j-1]) \\ \text{undefined} \qquad \text{otherwise} \end{cases}$$

Try It Out: Insertion sort

Program Graph (FM p 8)



b abbreviates $\texttt{j} > 0 \;\&\&\; \texttt{A[j} - 1\texttt{]} > \texttt{A[j]}$

Try It Out (FM p 8)

Modify the program graph for the insertion sort algorithm to use a temporary variable to do the swapping of array entries.

Define the corresponding semantics.

How is indexing outside the array bounds captured in your semantics?

1.4 Properties of Program Graphs

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Properties of Systems

Deterministic System

We would often expect the value computed to be the same each time we perform the computation and that the same computations are performed. This leads to the notion of deterministic system.

Evolving System

We would generally like that a value is always computed. This is clearly not always the case because programs may loop, and proving the absence of looping behaviour requires careful analysis of the program.

A weaker demand is that the program does not stop in a stuck configuration. This leads to the notion of evolving system.

A stuck configuration is a configuration that is not a final configuration and that has no next execution step.

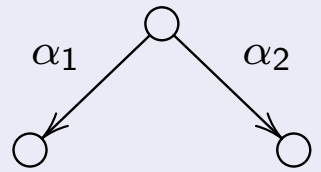
Deterministic System

Definition (FM p 9)

A program graph **PG** and its semantics $\mathcal{S}$ constitute a *deterministic system* whenever

$$\langle q_\triangleright ; \sigma \rangle \overset{\omega'}{\Longrightarrow}{}^{*} \langle q_\blacktriangleleft ; \sigma' \rangle \text{ and } \langle q_\triangleright ; \sigma \rangle \overset{\omega''}{\Longrightarrow}{}^{*} \langle q_\blacktriangleleft ; \sigma'' \rangle$$

imply that $\sigma' = \sigma''$ and that $\omega' = \omega''$.



$\mathcal{S}[\![\alpha_1]\!]\sigma$ or $\mathcal{S}[\![\alpha_2]\!]\sigma$ undefined.

Proposition (FM p 9)

A program graph **PG** and its semantics $\mathcal{S}$ constitute a *deterministic system* whenever distinct edges with the same source node have semantic functions that are defined on non-overlapping domains:

$$\forall (q, \alpha_1, q_1), (q, \alpha_2, q_2) \in \mathbf{E} : \left(\begin{array}{c} (\alpha_1, q_1) \neq (\alpha_2, q_2) \Rightarrow \\ (\mathrm{dom}(\mathcal{S}[\![\alpha_1]\!]) \cap \mathrm{dom}(\mathcal{S}[\![\alpha_2]\!]) = \{\}) \end{array} \right)$$

Evolving System

Definition (FM p 10)

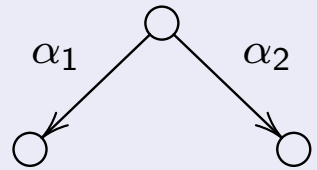
A program graph **PG** and its semantics $\mathcal{S}$ constitute an *evolving system* whenever

$$\langle q_\triangleright ; \sigma \rangle \overset{\omega'}{\Longrightarrow}{}^{*} \langle q' ; \sigma' \rangle \text{ with } q' \neq q_\blacktriangleleft$$

can always be extended to

$$\langle q_\triangleright ; \sigma \rangle \overset{\omega''}{\Longrightarrow}{}^{*} \langle q'' ; \sigma'' \rangle$$

for some ω'' of the form $\omega'' = \omega' \alpha'$.



$\mathcal{S}[\![\alpha_1]\!]\sigma$ or $\mathcal{S}[\![\alpha_2]\!]\sigma$ defined.

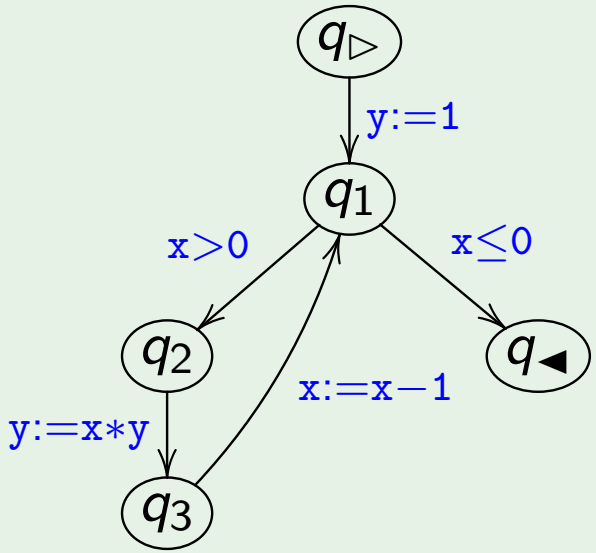
Proposition (FM p 11)

A program graph **PG** and its semantics $\mathcal{S}$ constitute an *evolving system* if for every non-final node and every memory there is an edge leaving it such that the memory is in the domain of the semantic function for that edge:

$$\forall q \in \mathbf{Q} \setminus \{q_\blacktriangleleft\} : \forall \sigma \in \mathbf{Mem} : \exists (q, \alpha, q') \in \mathbf{E} : \sigma \in \mathrm{dom}(\mathcal{S}[\![\alpha]\!])$$

Try It Out: Deterministic and Evolving Systems

Example (FM p 1)



Example (FM p 4)

$$\mathcal{S}[\![\mathtt{y} := 1]\!](v_\mathtt{x}, v_\mathtt{y}) = (v_\mathtt{x}, 1)$$

$$\mathcal{S}[\![\mathtt{x} > 0]\!](v_\mathtt{x}, v_\mathtt{y}) = \begin{cases} (v_\mathtt{x}, v_\mathtt{y}) & \text{if } v_\mathtt{x} > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![\mathtt{x} \leq 0]\!](v_\mathtt{x}, v_\mathtt{y}) = \begin{cases} (v_\mathtt{x}, v_\mathtt{y}) & \text{if } v_\mathtt{x} \leq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![\mathtt{y} := \mathtt{x} * \mathtt{y}]\!](v_\mathtt{x}, v_\mathtt{y}) = (v_\mathtt{x}, v_\mathtt{x} * v_\mathtt{y})$$

$$\mathcal{S}[\![\mathtt{x} := \mathtt{x} - 1]\!](v_\mathtt{x}, v_\mathtt{y}) = (v_\mathtt{x} - 1, v_\mathtt{y})$$

Try It Out (FM p 11)

Determine whether or not the factorial program constitutes a deterministic system.

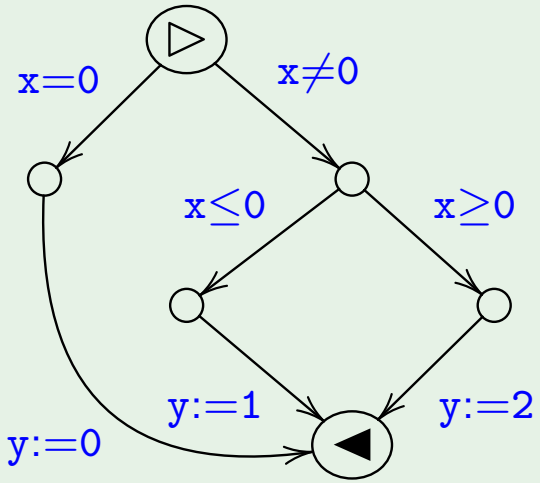
Try It Out (FM p 10)

Determine whether or not the factorial program constitutes an evolving system.

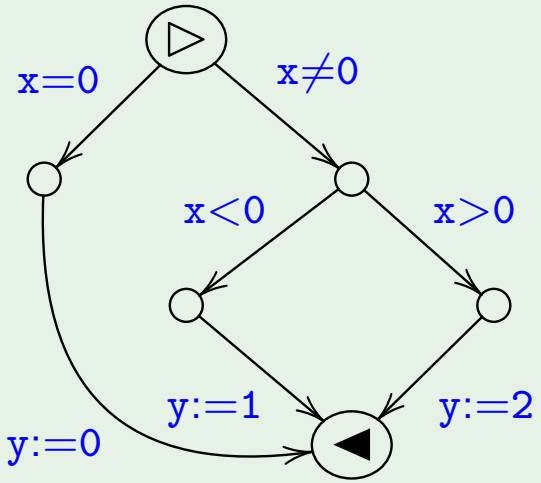
Deterministic and Evolving Systems

The conditions of the two propositions are *sufficient* but *not necessary*.

A Deterministic System (FM p 10)



An Evolving System (FM p 11)



$$\forall (q, \alpha_1, q_1), (q, \alpha_2, q_2) \in \mathbf{E}: (\alpha_1, q_1) \neq (\alpha_2, q_2) \Rightarrow (\mathrm{dom}(\mathcal{S}[\![\alpha_1]\!]) \cap \mathrm{dom}(\mathcal{S}[\![\alpha_2]\!]) = \{\})$$

$$\forall q \in \mathbf{Q} \setminus \{q_{\blacktriangleleft}\}: \forall \sigma \in \mathbf{Mem}: \exists (q, \alpha, q') \in \mathbf{E}: \sigma \in \mathrm{dom}(\mathcal{S}[\![\alpha]\!])$$

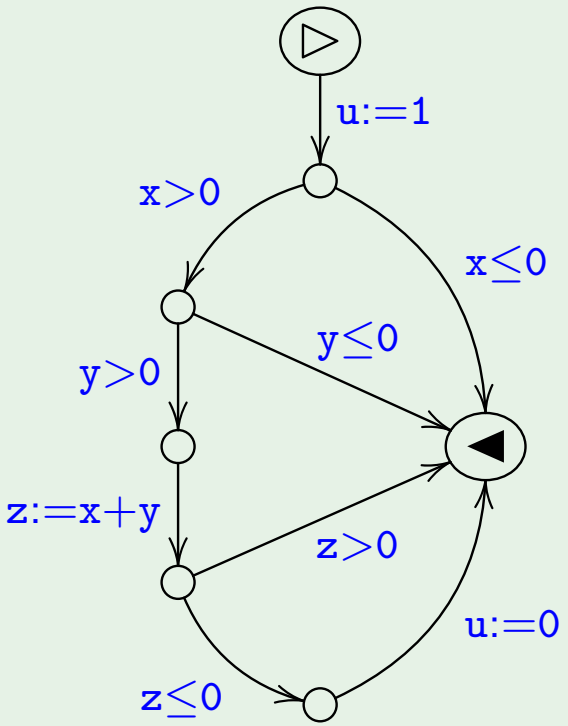
1.5 Bit-Level Semantics

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Bit-Level Semantics

Example program (FM p 11)



Questions

- Is it always possible to reach a final configuration?

$$\forall \sigma : \exists \sigma' : \langle q_{\triangleright}; \sigma \rangle \Longrightarrow^* \langle q_{\blacktriangleleft}; \sigma' \rangle$$

- Will the final value of u always be 1?

$$\forall \sigma, \sigma' : \; (\langle q_{\triangleright}; \sigma \rangle \Longrightarrow^* \langle q_{\blacktriangleleft}; \sigma' \rangle) \Rightarrow \sigma'_{\mathtt{u}} = 1$$

Answers depend on integer representation

- mathematical integers
- unsigned bytes
- signed bytes