Chapter 2
A Toy Language for Concurrency

Since the aim of this book is to introduce models and verification techniques for programming languages, our first task is to introduce the programming language through which we demonstrate the main ideas of this book. There are many possible choices for such a language, from theoretical ones (e.g., CCS, the π-calculus [124]) which abstract away implementation details, to real-world standards and languages (e.g., POSIX, Java) with large sets of tools to handle concurrency. We choose to invent an intermediate language which is relatively concise while being somewhat realistic. We begin by introducing the language (Sect. 2.1). We then describe its operational semantics, which formalizes the way programs are to be executed (Sect. 2.2). Finally, we describe the correctness properties that we will be interested in (Sect. 2.3). In this chapter, the execution model of programs running in parallel is formalized in the simplest possible way, as an interleaving of the actions of the programs, and will be refined in the next chapter by truly concurrent models.

2.1 A Toy Language

Throughout the book, we consider a concurrent, shared-memory, imperative, toy language for illustrative purposes. A program in this language consists of a sequence of instructions, as in the following example:

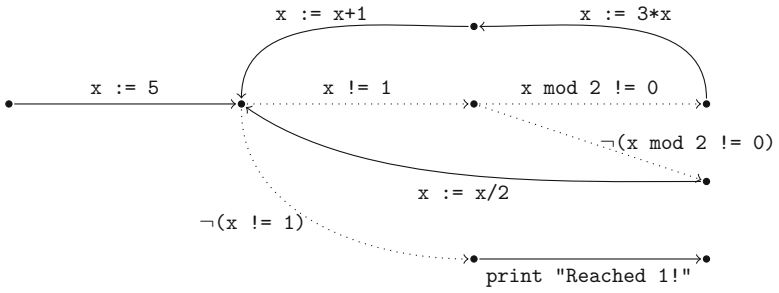
$$\mathtt{x} := 3; \mathtt{x} := \mathtt{x} + 1; \mathtt{y} := 2 * \mathtt{x}$$

The above sequence first assigns the value 3 to $\mathtt{x}$, then increments $\mathtt{x}$, and finally assigns twice the value of $\mathtt{x}$ to $\mathtt{y}$; here $\mathtt{x}$ and $\mathtt{y}$ are variables representing memory cells which are supposed to contain integers. Such sequences of instructions can be combined, using control flow constructs (e.g., conditional branching, conditional loops), as in the following example:

L. Fajstrup et al., *Directed Algebraic Topology and Concurrency*,
DOI 10.1007/978-3-319-15398-8_2

```
x := 5;
while x != 1 do (
    if x mod 2 != 0 then
        (x := 3*x; x := x+1)
    else
        x := x/2
);
print "Reached 1!"
```
(2.1)

This program computes the elements of the *Syracuse sequence*—the smallest sequence starting with $x_0 = 5$, ending with 1, and inductively defined for $n > 0$ by $x_{n+1} = x_n/2$ for x_n even and $x_{n+1} = 3x_n + 1$ for x_n odd—before printing a message. Here, `x != 1` denotes the condition $x \neq 1$. Such a program can be represented graphically by its *control flow graph*:



(2.2)

The vertices correspond to the positions in the program (a position is roughly a line number in the code), the solid arrows correspond to instructions, and the dotted arrows correspond to branches which might be taken depending on a condition. An *execution* of the program can be described as a path in this graph starting from the leftmost vertex.

The language we will use is a variant of the IMP language, often used as a setting for studying semantics [166], extended with a parallel composition operator. We choose to illustrate our methods on an imperative language because those are the most widespread, but they could be adapted to other flavors of programming languages (functional, object-oriented, etc.).

Definition 2.1 We suppose a fixed countable set Var of variables, and below x denotes a variable and n an integer. The language **PIMP** (*parallel* IMP) comprises three kinds of syntactic expressions, defined by their grammar:

- the set $\mathscr{A}$ of *arithmetic expressions*:

$$a \quad ::= \quad x \;\mid\; n \;\mid\; a + a \;\mid\; a * a$$

- the set $\mathscr{B}$ of *Boolean expressions*, or *conditions*:

$$b \ ::= \ \texttt{true} \mid \texttt{false} \mid a < a \mid b \ \texttt{and} \ b \mid \neg b$$

- the set $\mathscr{C}$ of *commands*, or *programs*:

$$c \ ::= \ x := a \mid \texttt{skip} \mid c; \ c \mid \texttt{if} \ b \ \texttt{then} \ c \ \texttt{else} \ c \mid \texttt{while} \ b \ \texttt{do} \ c \mid c \parallel c$$

A command of the form "$x := a$" is called an *action* and we write $\mathscr{C}_{\text{act}}$ for the set of actions. A program not containing the instruction "||" is called *sequential*.

A program is meant to be executed in an environment (the state of the memory) consisting of the values for the variables, which are integers only (for simplicity, we do not consider variables containing Booleans or other data types). Arithmetic and boolean expressions evaluate to integers and Booleans, respectively, in the usual way (e.g. , $*$ refers to integer multiplication, $\neg$ refers to logical negation). The commands have an effect on memory or on the control flow of the program. Their respective meanings are given in table below.

`x := a`	assign the result of the evaluation of the arithmetic expression a to the variable x
`skip`	do nothing
$\texttt{c}_1$`;` $\texttt{c}_2$	sequentially execute c_1 and then c_2
`if b then` $\texttt{c}_1$ `else` $\texttt{c}_2$	branch conditionally, i.e., evaluate the Boolean expression b and execute c_1 (resp. c_2) if the result is true (resp. false)
`while b do c`	execute the command c as long as the Boolean expression b evaluates to true
$\texttt{c}_1 \parallel \texttt{c}_2$	execute c_1 in parallel with c_2

We refer the reader to standard textbooks [166] for details about the first five operations, standard constructs in sequential programs. We will later discuss extensively the last, and novel, operation "$\parallel$" of *parallel composition*. The language is deliberately small in order to ease the definitions, but other operators can easily be added, and from time to time we incorporate such operators in our examples without comment, e.g. , the arithmetic operator `mod` and the Boolean operator `!=` in Example (2.1).

Convention 2.2 The sequence operator ";" binds more tightly than the parallel operator "||": the program $A; B||C$ is implicitly parenthesized as $(A; B)||C$. Moreover, multiple sequences or multiple parallels are parenthesized on the right: $A; B; C$ means $A; (B; C)$, $A||B||C$ means $A||(B|C)$, etc. This last convention is not really important though, because these operators are essentially associative; see Proposition 2.25.

We will focus on the study of *concurrent programs*, in which many subprograms (also called *threads* or *processes*) run in parallel. Programs of such form are often used in order to efficiently exploit the computing resources at our disposal and or

be more reactive to external events. A prototypical example is image processing on a multi-core machine, where each thread processes a part of the image for speed. Another prototypical example is the control of a power plant, where one thread controls physical hardware, another thread handles interactions between hardware and operators, and a failure of the latter does not imperil the former. Most common operating systems provide facilities for dynamic thread creation and associated operations, such as those formalized in the POSIX standard [79]. Once again, we abstract away implementations details. Instead we include the operation || in the very definition of a program; $p_1 || p_2$ means that the programs p_1 and p_2 are run in parallel, typically as two different threads. For instance, a simple image processing program might look like

$$p_{\mathrm{i}}; (p_{\mathrm{l}} || p_{\mathrm{r}}); p_{\mathrm{d}}$$

where p_{i} takes care of the initialization of the program, p_{l} and p_{r} process, respectively, the left and right parts of the image, and p_{d} displays the resulting image.

As a first approximation, the effect of $p||q$ is the same as some interleaving of actions of p and q: "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program", as phrased by Lamport [110]. We elaborate on the validity of this assumption, called *sequential consistency*, in Remark 2.17.

Remark 2.3 We limit ourselves in this book to the semantics of finitely many threads, whereas in most languages there are ways to define threads recursively, potentially creating an unbounded number of threads. In many ways, going to that level of generality would obscure the main purpose of the book without covering many more applications in practice: many programs are essentially structured as the image processing example above, creating all the threads after an initialization phase. To give a simple idea of the difference made by adding recursive thread creation to a parallel language, let us just mention that the state reachability problem for (our) multithreaded programs is PSPACE complete when there are finitely many finite-state threads [101] and undecidable with recursive threads [144].

2.2 Semantics of Programs

In this section, we formally introduce the notion of a transition graph (or a control flow graph) associated to a program [1]. This classical construction allows one to abstract away from the syntax of the programming language, to easily define a notion of execution trace for a program, and to provide an operational semantics.

2.2.1 Graphs

We begin by recalling some well-known constructions on graphs.

Definition 2.4 A *graph* $G = (V, \partial^-, \partial^+, E)$ consists of a set V of *vertices* (or *states*), a set E of *edges* (or *transitions*), and two functions $\partial^-, \partial^+ : E \to V$, respectively, associating to an edge its *source* and its *target*.

When graphs are indexed (such as G_1), we often index the associated sets of vertices (V_1), edges (E_1), and source and target functions (∂_1^- and ∂_1^+) correspondingly.

Definition 2.5 Given a set $\mathscr{L}$ of *labels*, a *labeled graph* (G, ℓ) consists of a graph G, as in Definition 2.4, together with a function $\ell : E \to \mathscr{L}$. Given an edge $e \in E$, the element $\ell(e)$ is called the *label* of the edge e.

We sometimes write $x \xrightarrow{A} y$ for an edge e such that $\partial^-(e) = x$, $\partial^+(e) = y$, and $\ell(e) = A$. This notation is not ambiguous because we usually consider graphs in which the edges between a given pair of vertices have different labels. A *path* $t = e_1 \,.\, e_2 \ldots e_n$ is a finite nonempty sequence of edges $e_i \in E$ such that $\partial^+(e_i) = \partial^-(e_{i+1})$ for $1 \leq i < n$, the integer n being the *length* of the path, or a vertex x denoting the empty path on this vertex, often written as ε_x. We write $\partial^-(t) = \partial^-(e_1)$ (resp. $\partial^+(t) = \partial^+(e_n)$) for the source (resp. target) of the path. Two paths with the same source (resp. target) are called *coinitial* (resp. *cofinal*). Given two paths t and u such that $\partial^+(t) = \partial^-(u)$, we write $t \,.\, u$ for their *concatenation*. Given two vertices $x, y \in E$, we say that y is *reachable* from x when there exists a path t with $\partial^-(t) = x$ and $\partial^+(t) = y$, denoted $t : x \twoheadrightarrow y$. When the graph is labeled in $\mathscr{L}$, the sequence of labels of edges in a path t forms a word in $\mathscr{L}^*$ that we denote as $\ell(t)$.

A *morphism* $f : G_1 \to G_2$ between two graphs consists of a pair of functions $f^V : V_1 \to V_2$ and $f^E : E_1 \to E_2$ such that the function on edges is compatible with source and target:

$$f^V \circ \partial_1^- = \partial_2^- \circ f^E \qquad \text{and} \qquad f^V \circ \partial_1^+ = \partial_2^+ \circ f^E$$

When the two graphs are labeled with the same set of labels, the morphism is moreover required to preserve the labeling of edges: $\ell_1 = \ell_2 \circ f^E$. The two functions f^V and f^E are often abusively denoted by the same symbol f, the context making clear which one we are referring to. Two (labeled) graphs G_1 and G_2 are isomorphic when there exists morphisms $f : G_1 \to G_2$ and $g : G_2 \to G_1$ such that $g \circ f = \mathrm{id}$ and $f \circ g = \mathrm{id}$.

In the following, we will make frequent use of the following operations in order to combine graphs:

Definition 2.6 Suppose given two labeled graphs $G_1 = (V_1, \partial_1^-, \partial_1^+, E_1, \ell_1)$ and $G_2 = (V_2, \partial_2^-, \partial_2^+, E_2, \ell_2)$. We define the following constructions on G_1 and G_2.

- Their **disjoint union** $G_1 \sqcup G_2$ is the graph

$$G_1 \sqcup G_2 \quad = \quad (V_1 \sqcup V_2, \partial^-, \partial^+, E_1 \sqcup E_2, \ell)$$

  where $\partial^-(e) = \partial_1^-(e)$ if $e \in E_1$ and $\partial^-(e) = \partial_2^-(e)$ if $e \in E_2$, and similarly for ∂^+ and ℓ.
- Their **tensor product** $G_1 \otimes G_2$ is the graph

$$G_1 \otimes G_2 \quad = \quad (V_1 \times V_2, \partial^-, \partial^+, (E_1 \times V_2) \sqcup (V_1 \times E_2), \ell)$$

  with $\partial^-(e, x) = (\partial_1^-(e), x)$ and $\ell(e, x) = \ell(e)$ for an edge $(e, x) \in E_1 \times V_2$, $\partial^-(x, e) = (x, \partial_2^-(e))$ and $\ell(x, e) = \ell(e)$ for an edge $(x, e) \in V_1 \times E_2$, and similarly for ∂^+.
- Given two vertices $x, y \in V_1$ the **quotient** graph $G_1[x = y]$ is the graph obtained by identifying the vertices x and y in G_1, i.e., formally

$$G_1[x = y] \quad = \quad (V_1/\approx, \partial^-, \partial^+, E_1, \ell_1)$$

  where $V_1/\approx$ is the quotient of the set V_1 by the equivalence relation $\approx$ on V_1 such that $x' \approx x''$ whenever $x' = x''$, or $x' = x$ and $x'' = y$, or $x' = y$ and $x'' = x$; and ∂^- and ∂^+ are the maps induced from ∂_1^- and ∂_1^+ by the quotient.
- Given a subset $V \subseteq V_1$ of vertices of G_1, the **restriction** of G_1 to V is the graph $G_1|_V = (V, E)$ where $E = \{e \in E_1 \mid \partial^-(e) \in V \text{ and } \partial^+(e) \in V\}$.

Example 2.7 From the two graphs

$$G = \; x \bullet \xrightarrow{A} \bullet y \qquad \text{and} \qquad H = \; z_0 \bullet \xrightarrow{B_1} \bullet_{z_1} \begin{array}{l} \xrightarrow{B_2} \bullet z_2 \\ \xrightarrow[B_3]{} \bullet z_3 \end{array}$$

we can compute the following graphs:

$$G \sqcup H \quad = \quad x \bullet \xrightarrow{A} \bullet y \qquad z_0 \bullet \xrightarrow{B_1} \bullet_{z_1} \begin{array}{l} \xrightarrow{B_2} \bullet z_2 \\ \xrightarrow[B_3]{} \bullet z_3 \end{array}$$

$$G \otimes H \quad = \quad \begin{array}{ccccc} (y,z_0) & \xrightarrow{} & (y,B_1) & \xrightarrow{} & (y,B_2) \\ & & & & \searrow (y,z_2) \\ & & & & \searrow (y,z_3) \\ (A,z_0) & & & & \downarrow (A,z_3) \\ (x,z_0) & \xrightarrow{} & (x,B_1) & & (x,B_3) \\ & & & & \searrow (x,z_3) \end{array}$$

$$H[z_0 = z_3] \quad = \quad z_0 \bullet \xrightarrow{B_1} \bullet_{z_1} \begin{array}{l} \xrightarrow{B_2} \bullet z_2 \\ \xleftarrow[B_3]{} \end{array} \qquad H|_{\{z_1, z_2\}} \quad = \quad \bullet_{z_1} \xrightarrow{B_2} \bullet z_2$$

Notice that the tensor product can be thought of as multiple copies of edges coming from either of the two graphs.

From these operations one can derive most usual operations on graphs. For instance, given a graph G, the graph obtained from G by adding an edge between two vertices x and y is the graph $(G \sqcup I)[x = x'][y = y']$ where I is the graph with two vertices x' and y' and one arrow $x' \to y'$.

Remark 2.8 The tensor product is sometimes referred to as the "cartesian product" of the two graphs. However, this terminology is incorrect since it is *not* a cartesian product in the usual category of graphs (and to add to the confusion, the proper cartesian product is usually called the tensor product). In fact, most of the classical interleaving semantics of such concurrent systems was originally done using yet another variant of product, called *synchronized product* [135]. The definition of the tensor product given above should become more natural when seen as a particular (one-dimensional) case of the tensor product of precubical sets, as defined in Sect. 3.4.

2.2.2 The Transition Graph

The operations introduced in the previous section easily allow us to formalize the notion of transition graph (or control flow graph) associated to a program as follows:

Definition 2.9 The **transition graph** $G_p = (G_p, \ell_p, s_p, t_p)$ associated to a program p is a graph G_p labeled in the set $\mathscr{L} = \mathscr{C}_{\text{act}} \sqcup \mathscr{B}$ together with two distinguished vertices $s_p, t_p \in E$ called the *beginning* and *end*. This graph is defined inductively as follows:

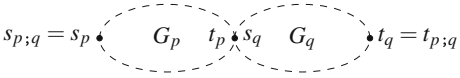
- G_A with $A \in \mathscr{C}_{\text{act}}$ is the graph with two vertices and one edge labeled by A:

$$s_A \bullet \xrightarrow{A} \bullet t_A$$

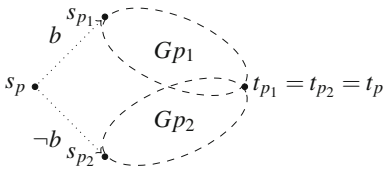
- $G_{\texttt{skip}}$ is the graph with one vertex (being both the beginning and the end) and no edge:

$$s_{\texttt{skip}} \bullet t_{\texttt{skip}}$$

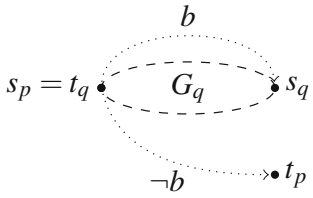
- $G_{p;q}$ is the graph obtained from the disjoint union of G_p and G_q by identifying t_p with s_q, such that $s_{p;q} = s_p$ and $t_{p;q} = t_q$:



- G_p, with $p = \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2$, is the graph obtained from the disjoint union of G_{p_1} and G_{p_2} by identifying t_{p_1} and t_{p_2}, the resulting vertex being t_p, adding a new vertex s_p and two transitions $s_p \xrightarrow{b} s_{p_1}$ and $s_p \xrightarrow{\neg b} s_{p_2}$:



- G_p, with p=`while` b `do` q, is obtained from G_q by adding a vertex t_p, adding an edge $t_q \xrightarrow{\neg b} t_p$, and adding an edge $t_q \xrightarrow{b} s_q$, with $s_p = t_q$:



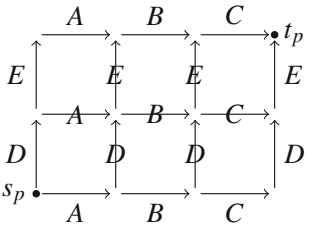
  (notice that source of the graph G_q is on the right and the target on the left in the above figure)
- $G_{p||q}$ is the graph $G_p \otimes G_q$ with $s_{p||q} = (s_p, s_q)$ and $t_{p||q} = (t_p, t_q)$.

A vertex of G_p is called a **position** of the program p.

Notice again that all the graphs above can be obtained using the constructions of Definition 2.6, for instance $G_{p;q} = (G_p \sqcup G_q)[t_p = s_q]$. As shown above and in the introductory example (2.1), the edges labeled by conditions in transition graphs are drawn with dotted arrows to distinguish them from those labeled by actions. This is only a drawing convention; there is no difference between the two types of edges except the sets in which they are labeled.

Example 2.10 The Syracuse program (2.1) gives rise to the transition graph depicted in (2.2).

Example 2.11 The transition graph of the program $p = (A; B; C) \,||\, (D; E)$, where A, B, C, D, E are arbitrary actions, is



By construction of the transition graph, we have

Lemma 2.12 *For every program p and vertex x of its transition graph there are, by construction, both a path from s_p to x and a path from x to t_p.*

Paths starting at the beginning vertex will be of particular importance since they encompass all the sequences of actions that the program can give rise to.

Definition 2.13 A **potential execution trace** of a program p is a path starting from s_p in G_p. We write $\mathrm{T}_{\mathrm{pot}}(p)$ for the set of such paths.

Of course, not every such path corresponds to an actual execution of the program, which is why they are called "potential." Determining the ones which can actually occur during an execution depends on the chosen semantics of the programming language, as formalized in Definition 2.19. Notice that the set $\mathrm{T}_{\mathrm{pot}}(p)$ is closed under prefix, its maximal elements are thus enough to describe it when traces are of bounded length.

Example 2.14 Consider a program p of the form A; (`if` b `then` B `else` C). Its maximal potential execution traces are labeled by $A \,.\, b \,.\, B$ and $A \,.\, \neg b \,.\, C$. The conditions occurring in those traces should be thought of as assumptions on the memory state under which those traces make sense, which is why we call them *potential*. For instance, the trace $A \,.\, b \,.\, B$ should be read as: do A, now suppose that the condition b is true, do B.

Example 2.15 Given an action A, the potential execution traces of the program `while` b `do` A are labeled by words which are prefixes of words in $(b \,.\, A)^* \,.\, \neg b$. In particular, when the condition b is `true`, all those labeled in $(b \,.\, A)^*$ are valid: a program can thus admit an infinite number of traces.

Example 2.16 Consider again the program $p = (A; B; C)||(D; E)$ introduced in Example 2.11. Its maximal potential execution traces are labeled by elements of the set $\{A \,.\, B \,.\, C\} \sqcup\!\sqcup \{D \,.\, E\}$ of shuffles of the words $A \,.\, B \,.\, C$ and $D \,.\, E$, i.e.,

$$\begin{array}{l}\{\ A \,.\, B \,.\, C \,.\, D \,.\, E,\ A \,.\, B \,.\, D \,.\, C \,.\, E,\ A \,.\, D \,.\, B \,.\, C \,.\, E,\ D \,.\, A \,.\, B \,.\, C \,.\, E,\ A \,.\, B \,.\, D \,.\, E \,.\, C,\\ \quad A \,.\, D \,.\, B \,.\, E \,.\, C,\ D \,.\, A \,.\, B \,.\, E \,.\, C,\ A \,.\, D \,.\, E \,.\, B \,.\, C,\ D \,.\, A \,.\, E \,.\, B \,.\, C,\ D \,.\, E \,.\, A \,.\, B \,.\, C\ \}\end{array}$$

Given two languages $P, Q \subseteq \mathscr{L}^*$, we recall that their *shuffle* $P \sqcup\!\sqcup Q$ is defined inductively by

$$P \sqcup\!\sqcup Q \quad = \quad \bigcup_{a \in \mathscr{L}} \{a\} \,.\, ((P/a \sqcup\!\sqcup Q) \cup (P \sqcup\!\sqcup Q/a))$$

with $P/a = \{u \in \mathscr{L}^* \mid a \,.\, u \in P\}$ and similarly for Q/a. Said otherwise, the words $u = a_1 \ldots a_n$ in $P \sqcup\!\sqcup Q$ are those for which there exists a subset $I \subseteq \{1, \ldots, n\}$ such that the subword of u consisting of letters with indices in I (resp. in $\{1, \ldots, n\} \setminus I$) is in P (resp. in Q). This example illustrates why we chose to interpret the parallel composition by the tensor product of graphs in Definition 2.9: in order to execute $p||q$, one should either execute the first action of p and then the rest of p in parallel with q, or the first action of q and then p in parallel with the rest of q. It is also interesting to notice the large number of execution traces, considering the small size of the program generating them: a word in the above shuffle is of length $3 + 2 = 5$,

and is characterized by the positions of the subword of length 2 in this word, so the cardinality of the above set is $\binom{5}{2} = 5!/(2!\,3!) = 10$.

Remark 2.17 The assumption that the only behaviors that can occur in a concurrent program are those which can be obtained as a sequential interleaving of the instructions of the processes executed in parallel is called *sequential consistency* [110]. Real multiprocessors, however, use sophisticated techniques to achieve high performance: the storage of buffers, hierarchies of local cache, speculative execution [154], etc. These implementation details are not observable by single-threaded programs, but in multithreaded programs different threads may see subtly different views of memory. Such machines exhibit *relaxed* (or *weak*) *memory models*. For instance, consider a standard x86 processor. Given two memory locations `x` and `y` (initially holding the value `0`), we look at the following program with two threads writing `1` to both `x` and `y` and then reading from `y` and `x`:

(`mov x, 1 ; mov eax, y`) || (`move y, 1 ; mov ebx, x`)

The instruction "`mov x, y`" is essentially the assembly notation for $x := y$, and `eax` and `ebx` are special memory locations called registers. Intuitively, the possible outcomes for (`eax`, `ebx`) are (`1`, `1`), (`1`, `0`), and (`0`, `1`). However, on standard processors, some executions can also lead to (`0`, `0`) [131]. Throughout the book we still make the assumption that the semantics is sequentially consistent for simplicity (our approach could be refined to encompass a semantics with a relaxed memory model) and because modern compilers help to ensure sequentially consistent semantics at a higher level.

2.2.3 Operational Semantics

We now need to describe, formally, the effect of executing a program, by describing a semantics for the language that we have been specifying. Notice that there can be many different semantics for a given language: most of the following constructions depend on our choice, and would be (slightly) different if we chose a different semantics. A program is meant to be executed in a *state* (also sometimes called an *environment*), which comprises the contents of the memory (as well as other resources to which the program would have access). For instance, the effect of an action like `x:=x+1` is to modify the memory cell corresponding to the variable `x` by incrementing `x`. Similarly, given a memory state, a condition such as `x<1` can be evaluated to either true or false depending on whether the cell `x` contains a value below 1 or not. In the following, we write $\mathbb{B} = \{\bot, \top\}$ for the set of *Booleans*,where $\bot$ (resp. $\top$) denotes false (resp. true).

Definition 2.18 We write $\Sigma = \mathbb{Z}^{Var}$ for the set of *states*, consisting of functions assigning an integer to each variable. The *initial state* $\sigma_0 \in \Sigma$ is the constant function

equal to 0. The **operational semantics** of our programming language consists of three functions:

- $[\![-]\!]_{\mathscr{A}} : \mathscr{A} \to (\Sigma \to \mathbb{Z})$ describing the evaluation of arithmetic expressions,
- $[\![-]\!]_{\mathscr{B}} : \mathscr{B} \to (\Sigma \to \mathbb{B})$ describing the evaluation of Boolean expressions,
- $[\![-]\!]_{\mathscr{C}^*_{\mathrm{act}}} : \mathscr{C}^*_{\mathrm{act}} \to (\Sigma \to \Sigma)$ describing the effect of sequences of actions on the state.

Thus, for instance, $[\![-]\!]_{\mathscr{A}}$ sends an element of $\mathscr{A}$ (an arithmetic expression) to a function from the set Σ of states to the set of integers. Given an arithmetic expression a, we write $[\![a]\!]_{\mathscr{A}} : \Sigma \to \mathbb{Z}$ for its semantic interpretation, and similarly for the other functions. Given a state $\sigma \in \Sigma$, the evaluation of arithmetic expressions is defined by

$$[\![x]\!]_{\mathscr{A}}(\sigma) = \sigma(x) \qquad [\![n]\!]_{\mathscr{A}}(\sigma) = n \qquad [\![a_1 + a_2]\!]_{\mathscr{A}}(\sigma) = [\![a_1]\!]_{\mathscr{A}}(\sigma) + [\![a_2]\!]_{\mathscr{A}}(\sigma) \qquad \dots$$

the evaluation of Boolean expressions by

$$[\![\mathtt{true}]\!]_{\mathscr{B}}(\sigma) = \top \qquad [\![\mathtt{false}]\!]_{\mathscr{B}}(\sigma) = \bot$$
$$[\![a_1 < a_2]\!]_{\mathscr{B}}(\sigma) = \begin{cases} \top & \text{if } [\![a_1]\!]_{\mathscr{A}}(\sigma) < [\![a_2]\!]_{\mathscr{A}}(\sigma) \\ \bot & \text{otherwise} \end{cases} \qquad \dots$$

and the evaluation of sequences of actions in $\mathscr{C}^*_{\mathrm{act}}$ by

$$[\![u \,.\, v]\!]_{\mathscr{C}^*_{\mathrm{act}}}(\sigma) = [\![v]\!]_{\mathscr{C}^*_{\mathrm{act}}} \circ [\![u]\!]_{\mathscr{C}^*_{\mathrm{act}}}(\sigma) \qquad [\![\varepsilon]\!]_{\mathscr{C}^*_{\mathrm{act}}}(\sigma) = \sigma$$
$$[\![x := a]\!]_{\mathscr{C}^*_{\mathrm{act}}}(\sigma) = \sigma\left[x \mapsto [\![a]\!]_{\mathscr{A}}(\sigma)\right] \qquad \dots \tag{2.3}$$

where ε denotes the empty word and, given $n \in \mathbb{Z}$, $\sigma\,[x \mapsto n]$ denotes the function in Σ which associates n to x and $\sigma(y)$ to each $y \neq x$.

In the following, we generally drop the subscripts when the function to which we are referring is clear from context. The operational semantics allows us to characterize those potential execution traces which are valid, in the sense that the assumptions corresponding to the conditions occurring in those traces are satisfied. Given a word u in $(\mathscr{C}_{\mathrm{act}} \sqcup \mathscr{B})^*$, we write $[\![u]\!] : \sigma \to \sigma$ defined as in (2.3), extended with $[\![b]\!](\sigma) = \sigma$, for a condition $b \in \mathscr{B}$, in other words $[\![u]\!] = [\![v]\!]_{\mathscr{C}^*_{\mathrm{act}}}$ where v is the projection of u onto $\mathscr{C}^*_{\mathrm{act}}$. Finally, given a path t in a transition graph G_p (see Definition 2.9), its labeling word $\ell(t)$ is an element of $(\mathscr{C}_{\mathrm{act}} \sqcup \mathscr{B})^*$, and we sometimes write $[\![t]\!]$ instead of $[\![\ell(t)]\!]$. In particular, notice that $[\![t]\!]$ is defined for each potential execution trace $t \in \mathrm{T}_{\mathrm{pot}}(p)$, even if it is not valid.

Definition 2.19 A word $u \in (\mathscr{C}_{\mathrm{act}} \sqcup \mathscr{B})^*$ is *valid* when it is either

- the empty word ε,
- of the form $u \,.\, A$ with u valid and $A \in \mathscr{C}_{\mathrm{act}}$,
- or of the form $u \,.\, b$ with u valid, $b \in \mathscr{B}$ and $[\![b]\!] \circ [\![u]\!](\sigma_0) = \top$.

An **execution trace** is a valid potential execution trace in $\mathrm{T}_{\mathrm{pot}}(p)$, i.e., a path $t \in \mathrm{T}_{\mathrm{pot}}(p)$ such that the word $\ell(t)$ is valid. We write $\mathrm{T}(p)$ for the set of execution traces of a program p.

Example 2.20 Consider the following program:

$$(x := 0 \;\;||\;\; x := 1);\;\; \texttt{if}\;\; x == 0 \;\; \texttt{then}\;\; c_0 \;\; \texttt{else}\;\; c_1$$

where the == operator compares two integer values for equality. Its four maximal potential execution traces are

$$\texttt{x:=0.x:=1.x==0.}c_0 \qquad \texttt{x:=0.x:=1.}\neg\texttt{x==0.}c_1$$
$$\texttt{x:=1.x:=0.x==0.}c_0 \qquad \texttt{x:=1.x:=0.}\neg\texttt{x==0.}c_1$$

and only traces up to the right and below to the left are valid. For instance, the one up to the left is not valid because when the condition x==0 is evaluated, the variable x contains 1, and the condition is not true. Notice that this example shows that our programs are nondeterministic because of parallelism: here, either c_0 or c_1 can be executed. It can be shown, however, that programs without parallelism are deterministic.

The operational semantics, as we formulated them, can be related to a more standard *small-step operational semantics* for the programming language. We describe them here briefly and refer to [166] for details about such definitions.

Definition 2.21 We define a **reduction** relation $\to$ on pairs $\langle \sigma, c\rangle$ consisting of a state $\sigma \in \Sigma$ and a command c, which formally describes how a command evaluates in a given environment. The rules defining this relation are as follows, where each "fraction" below should be interpreted so that the denominator also holds whenever the numerator holds.

$$\frac{\langle \sigma, c_1\rangle \to \langle \sigma', c_1'\rangle}{\langle \sigma, c_1\,;\,c_2\rangle \to \langle \sigma', c_1'\,;\,c_2\rangle} \qquad \frac{}{\langle \sigma, \texttt{skip}\,;\,c\rangle \to \langle \sigma, c\rangle} \qquad \frac{}{\langle \sigma, x := a\rangle \to \langle \sigma\left[x \mapsto [\![a]\!](\sigma)\right], \texttt{skip}\rangle}$$

$$\frac{[\![b]\!](\sigma) = \top}{\langle \sigma, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle \to \langle \sigma, c_1\rangle} \qquad \frac{[\![b]\!](\sigma) = \bot}{\langle \sigma, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle \to \langle \sigma, c_2\rangle}$$

$$\frac{[\![b]\!](\sigma) = \top}{\langle \sigma, \texttt{while}\ b\ \texttt{do}\ c\rangle \to \langle \sigma, c\,;\texttt{while}\ b\ \texttt{do}\ c\rangle} \qquad \frac{[\![b]\!](\sigma) = \bot}{\langle \sigma, \texttt{while}\ b\ \texttt{do}\ c\rangle \to \langle \sigma, \texttt{skip}\rangle}$$

$$\frac{\langle \sigma, c_1\rangle \to \langle \sigma', c_1'\rangle}{\langle \sigma, c_1 \parallel c_2\rangle \to \langle \sigma', c_1' \parallel c_2\rangle} \qquad \frac{\langle \sigma, c_2\rangle \to \langle \sigma', c_2'\rangle}{\langle \sigma, c_1 \parallel c_2\rangle \to \langle \sigma', c_1 \parallel c_2'\rangle}$$

$$\frac{}{\langle \sigma, \texttt{skip} \parallel c\rangle \to \langle \sigma, c\rangle} \qquad \frac{}{\langle \sigma, c \parallel \texttt{skip}\rangle \to \langle \sigma, c\rangle}$$

The relation between the semantics given in Definition 2.18 and the one of Definition 2.21 can be formalized as follows. We write $\to^*$ for the reflexive and transitive closure of $\to$.

Proposition 2.22 *Given states $\sigma, \sigma' \in \Sigma$ and a command c, there is an execution trace $t \in \mathrm{T}(c)$ such that $[\![t]\!](\sigma) = \sigma'$ if and only if there exists a command c' such that $\langle \sigma, c\rangle \to \langle \sigma', c'\rangle$.*

Remark 2.23 In order to show the previous proposition, it is important to remark that in Definition 2.21 the evaluation of arithmetic and Boolean conditions is "atomic", i.e., performed at once. Because of this, the two arithmetic expressions `x+x` and `2*x` are equivalent, in the sense that replacing one with another in a program does not change the results of the program. This would not be true anymore if we had chosen a small-step semantics for the evaluation of expressions as well, i.e., if we had added the rules

$$\frac{\langle \sigma, a_1 \rangle \to \langle \sigma', a_1' \rangle}{\langle \sigma, a_1 + a_2 \rangle \to \langle \sigma', a_1' + a_2 \rangle} \qquad \frac{\langle \sigma, a_2 \rangle \to \langle \sigma', a_2' \rangle}{\langle \sigma, a_1 + a_2 \rangle \to \langle \sigma', a_1 + a_2' \rangle} \qquad \frac{}{\langle \sigma, n_1 + n_2 \rangle \to \langle \sigma, n \rangle}$$

with $n_1, n_2 \in \mathbb{Z}$ and $n = n_1 + n_2$, replaced the rule for assignation by the rules

$$\frac{\langle \sigma, a \rangle \to \langle \sigma', a' \rangle}{\langle \sigma, x := a \rangle \to \langle \sigma', x := a' \rangle} \qquad \frac{}{\langle \sigma, x := n \rangle \to \langle \sigma\,[x \mapsto n]\,, \mathtt{skip} \rangle}$$

with $n \in \mathbb{Z}$, and similarly had replaced the other rules. Under these modified rules, consider the program c defined by `y:=(x+x)+1 || x:=x+1`. We have $\langle \sigma_0, c \rangle \to^* \langle \sigma, \mathtt{skip} \rangle$ where σ is a state such that $\sigma(\mathtt{y}) = 2$, because the incrementation of `x` can be interleaved between the two evaluations of the variable `x` occurring in the expression defining `y`. This is not possible anymore if we replace `x+x` with `2*x`. However, this kind of behavior could be simulated by introducing variables for intermediate results during the evaluation of expressions.

Finally, we would like to briefly mention the concept of contextual equivalence of programs, applied to concurrent programs.

Definition 2.24 Two commands c_1 and c_2 are *contextually equivalent*, written $c_1 \approx c_2$, when for every state $\sigma \in \Sigma$ we have $[\![c_1]\!](\sigma) = [\![c_2]\!](\sigma)$.

It is well known that, up to contextual equivalence, sequential composition is associative and admits `skip` as its neutral element. Similar such identities hold for parallel composition:

Proposition 2.25 *For all commands c, c_1, c_2, and c_3, the following equivalences hold:*

$$(c_1 || c_2) || c_3 \approx c_1 || (c_2 || c_3) \qquad \mathtt{skip} || c \approx c \approx c || \mathtt{skip} \qquad c_1 || c_2 \approx c_2 || c_1$$

Remark 2.26 In our programming language, the classical equivalence

$$\mathtt{while}\ b\ \mathtt{do}\ c \quad \approx \quad \mathtt{if}\ b\ \mathtt{then}\ (c; \mathtt{while}\ b\ \mathtt{do}\ c)\ \mathtt{else}\ \mathtt{skip}$$

can be shown, formalizing the intuition that a `while` loop can be seen as an infinite sequence of nested conditional branchings. If one is interested in verifying loops of a

program only up to a certain depth (i.e., imposing a bound on the number of times a loop can be taken), one can *unroll* the program by replacing every `while` construct by a finite series of conditional branchings as explained above. With this unrolling, we can only verify weaker properties on programs (since we only ensure that the properties are verified up to the depth of the loops). Still, it is sometimes useful since loops are quite difficult to handle in verification.

2.3 Verifying Programs

2.3.1 Correctness Properties

In order to check that a program is "correct," we have to specify what "correctness" means, i.e., what properties we are interested in. We can identify three major families of commonly encountered verification properties:

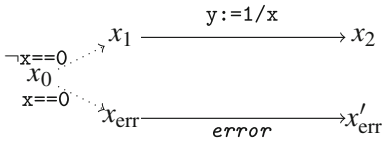
1. *Functional properties*. These describe how the result of the program complies with a mathematical specification. For instance, an implementation of the factorial function actually computes the factorial, i.e., given an integer $n \in \mathbb{N}$ as input, it returns the integer $n!$. These usually describe invariants or safety properties that will hold true for all executions of the program, expressed in proof-theoretic form [109] or using temporal logic, and generally verified using proof assistants, model checking [27], or abstract interpretation [127].
2. *Reachability properties*. These properties consist in ensuring that some position of a program, typically corresponding to an error, will not be reached. They can also be used to ensure that operations are used within their domain of definition. For instance, division is only defined if the denominator is non-null. In a program, an expression such as `y = 1/x` could be treated as

$$\texttt{if x} == 0 \ \texttt{then} \ \textit{error} \ \texttt{else y} := 1/x \tag{2.4}$$

   and then a reachability analysis could be used in order to ensure that the error is unreachable (i.e., never executed), which means that `x` is always non-null at this point of the program. One is also sometimes interested in knowing which positions of a program can be reached, such as in *code coverage* analysis, to ensure that every piece of code can be executed in some situation. Reachability is a particular case of a liveness property [2], and of more general temporal properties as below.
3. *Temporal properties*. They specify the shape of expected execution traces. For instance, in a computer graphics software, whenever the function to display the image is called, the function to compute the image should have been called before.

For simplicity, we are going to focus on reachability properties in the rest of the book: we want to ensure that a given set $\mathscr{X}$ of *forbidden positions* is never reached during the execution of a program. For instance, suppose that there is a special instruction

`error`, as in program p shown in (2.4), and we want to ensure that this instruction is never executed, i.e., the position in $\mathscr{X} = \{x_{\text{err}}\}$ in the transition graph corresponding to p is not reachable:



Definition 2.27 A program p is **correct** w.r.t. a set $\mathscr{X}$ of positions if there is no execution trace with a state in $\mathscr{X}$ as target.

In order to verify that a program is correct starting from X_0, we could thus verify all its traces, by exploring all the paths in G_p starting from the initial vertex s_p, following some exploration strategy (e.g. , depth-first, breath-first, etc.). During the exploration, the state $\sigma \in \Sigma$ reached by the path can be iteratively computed following Definition 2.18, and it can be thus checked whether the path is valid (see Definition 2.19), i.e., if it is an execution trace: if the path is valid, the algorithm should check that it does not reach a forbidden position, otherwise the exploration of paths extending the current path can be skipped (since such extended paths will not be valid either).

It can be noticed that the algorithm we have just described, essentially consists of executing the program, with all its possible schedulings, and observing whether an error occurs at some point. This state space exploration strategy is quite naive. For instance, because of `while` loops, the number of traces is not necessarily finite, see for instance, Example 2.15: this problem is not specific to concurrent programs, and traditional methods can be used in order to overcome this shortcoming (one can be interested in correctness properties on execution traces of bounded length and unroll loops, or use widening operators associated to abstract interpretation domains [29], etc.).

2.3.2 *Reachability in Concurrent Programs*

In the rest of the book, we will put aside the problems encountered in the verification of generic properties, such as the ones mentioned above, and will focus on those specific to concurrent programs. In this context, one of the main difficulties to overcome is the following one. In order to check that a program is correct, one has to check that all the execution traces coming from possible schedulings of the threads are correct: even without loops, a concurrent program can generate a number of traces which is exponential in the size of the program—namely, by generalizing observations made in Example 2.16! it is easy to see that a program p of the form $p = A||A||\ldots||A$ with n copies of some action A generates $n!$ maximal execution traces. This major problem is often referred to as the *state space explosion problem* [26].

The notion of correctness of programs depends on the set $\mathscr{X}$ of positions which we do not want to be reached by any execution. Apart from ensuring that the domain of definition of used operators is respected, such as in the division example (2.4), or more generally that invariants are preserved during computation, sets of positions satisfying the following properties are interesting for concurrent programs: such positions are witnessing potential errors in the code, and the properties are generic in the sense that they are not tied to some particular instructions.

Definition 2.28 A vertex x in the transition graph G_p of a program p is called

- **unreachable** when there is no execution trace with x as target,
- a **deadlock** when x is different from the terminal position t_p and there is an execution trace with x as target which is not a proper prefix of another execution trace,
- **unsafe** when there is an execution trace with x as target which is the prefix of an execution trace with a deadlock as target,
- **doomed** when there is an execution trace with x as target which is not a proper prefix of an execution trace reaching the terminal position t_p.

An unreachable position is a position that can never be reached during the execution of a program: these positions are witnessing the presence of *dead code*, code that will never be executed. In a critical system, every single piece of code is usually written for some purpose, and the fact that some part of the code is formally useless is generally a good indicator of some misconception on the part of the programmer regarding the possible executions of the program. Deadlocks are much more problematic per se: they indicate positions in which the program is blocked and cannot do anything, i.e., the program is "frozen." This situation typically occurs in concurrent programs when two threads (or more) are mutually waiting for each other to free a resource, whence comes the term *deadlock* or *deadly embrace* [31]. Finally, an unsafe position is one from which the program can reach a deadlock position, and a doomed position is one from which the program will eventually reach a deadlock or loop forever. A program p is **safe** when its beginning position s_p is not unsafe: in such a program, no execution will lead to a deadlock.

Remark 2.29 The deadlock and unsafe situations are really specific to concurrent programs (such a program exhibiting a deadlock is provided in Example 3.32): it can be shown that a sequential program does not have deadlock positions.

Remark 2.30 We could have introduced a variant of the notion of deadlock (and similarly for unsafe and doomed positions) by requiring that all the execution traces reaching x cannot be extended. This would have not made any difference in what follows, because we will restrict to programs (called "coherent programs") whose structure is such that a property of a position does not depend on the path reaching it: for those programs either all the paths reaching x cannot be extended, or none.

Remark 2.31 Following the tradition in classical automata theory, we will only consider finite executions. However, even in this case, there are some subtle differences related to the presence of infinite executions (that will be silently ignored in the rest of this book): for instance, a doomed position could have been defined as a reachable position x such that every maximal path with x as source has a deadlock as target. Notice that this definition is not equivalent to the one above for programs with loops.

The search for such positions in concurrent programs will be extensively discussed and illustrated in the next section (in particular, Examples 3.22 and 3.23 provide programs illustrating positions with these properties).

http://www.springer.com/978-3-319-15397-1

Directed Algebraic Topology and Concurrency

Fajstrup, L.; Goubault, E.; Haucourt, E.; Mimram, S.; Raussen, M.

2016, XI, 167 p. 1 illus., Hardcover
ISBN: 978-3-319-15397-1