Formal methods for software and hardware verification

LECTURES:

Vladimir Anatolyevich Zakharov,
Vladislav Vasilyevich Podymov

http://mk.cs.msu.ru/

Lecture 3.

Modeling programs and circuits

Model Checking
The main principles of modeling
Kripke structures
First-order representation
Granularity of models
Translation of programs into Kripke structures

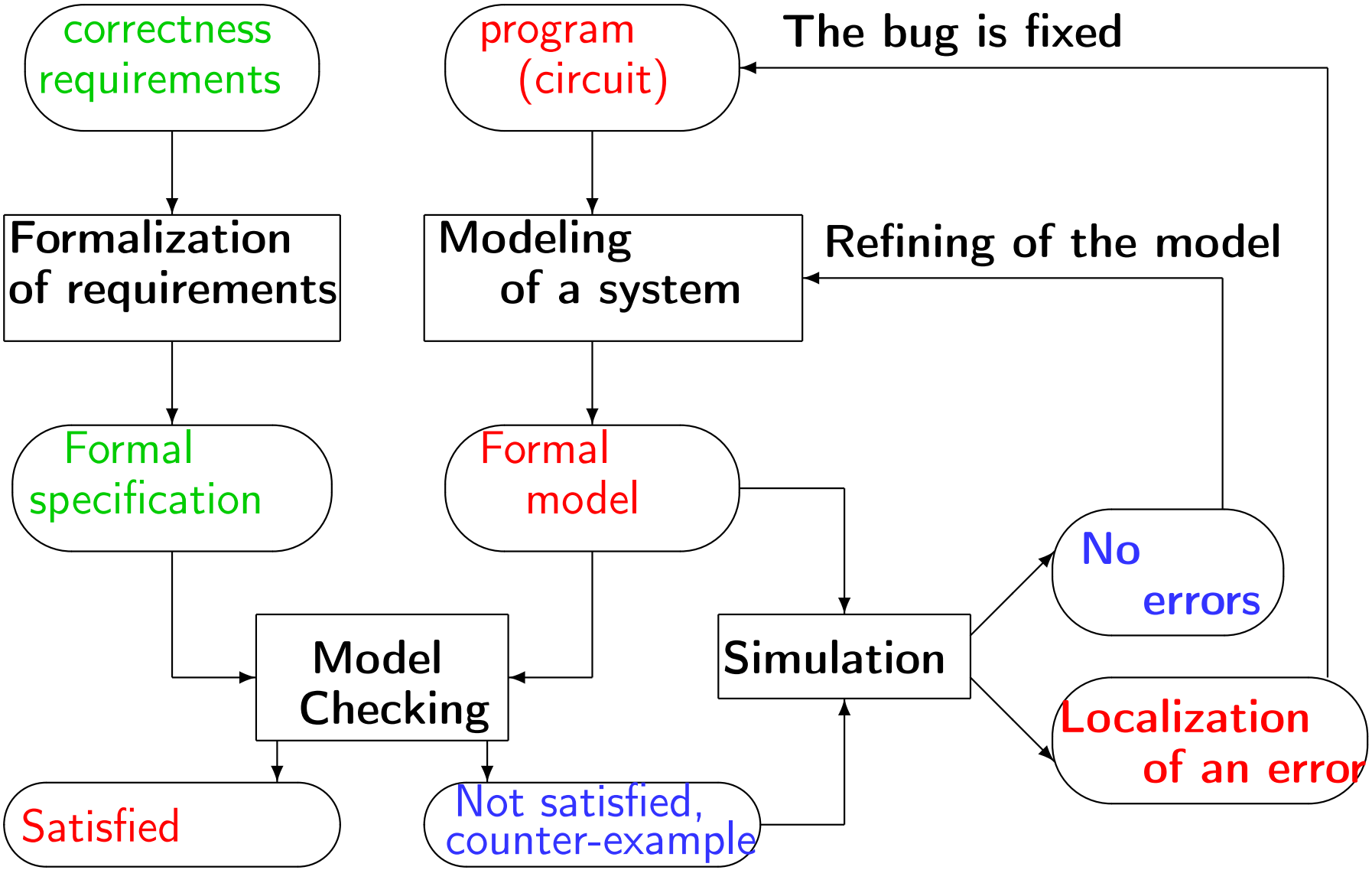
Model Checking

Model Checking consists in an exhaustive traversal of the state space of the system. If sufficient resources are available, this procedure always terminates and can be implemented by a rather efficient algorithm.

In some cases, systems with an infinite number of states can be checked by this method in combination with abstraction/refinement and induction techniques.

Since Model Checking can be applied purely automatically, it is preferable to proof-theoretic approach in those cases where it is applicable.

Model Checking



Model Checking

The efficiency
of model checking
depends on how much
concise and precise
is a program model.

Overly simplistic models yield
useless results.

Overly detailed models yield no
results at all.

The main principles of modeling

The first step in checking the correctness of the system is discussion and formal specification of properties to be checked.

As soon as it becomes known what properties are crucial, the next step is to build a formal model of the systems.

A model is suitable for verification if it displays all those properties that need to be checked to establish its correctness.

At the same time, a model should be free of insignificant details that do not affect properties to be analyzed, but only hinder verification.

The main principles of modeling

For example, for modeling digital circuits, it is advisable to reason in terms of logical cells and boolean values, not in terms of levels voltage.

And when checking communication protocols, it is reasonable to focus on the message exchange scenario, while ignoring the processing of their payload.

The main principles of modeling

When designing microelectronic hardware, one deals with reactive systems ; the behavior of such systems displays itself in the interaction of a systems with the environment.

The first characteristic feature of a reactive system is its state — a "snapshot" of the system, which captures the values of all variables at a given moment in time.

One also needs to know how the states of the system change as a result of performing the computing actions of the system. These changes can be described by specifying the state of the system before the action was taken and its state after performing the action. This pair of states defines a transition of the system.

The computations (run) of a reactive system is defined in terms of system transitions. A run is an infinite sequence of states, such that every next state in this sequence is reachable from the previous one by some transition.

The main principles of modeling

To formalize behaviors of reactive systems we will use a certain kind of labeled graphs which are called the Kripke structures (or, Kripke models , or Labeled Transtion Systems (LTSs)).

Typically, a Kripke structure consists of

- a set of states,
- a set of transitions, and
- a labeling function which marks every state with a set of basic properties that are true in this state.

Paths in the Kripke model correspond to runs of the system.

Kripke structures

Let AP be a set of atomic propositions (basic state properties).

A Kripke structure M over a set of atomic propositions AP is a quadruple $M = (S, S_0, R, L)$, where:

1) S is a finite set of states;
2) $S_0 \subseteq S$ is a subset of initial states;
3) $R \subseteq S \times S$ is a transition relation which is a total binary relation, i. e. for every state $s \in S$ there exists such a state $s' \in S$ that $R(s, s')$ holds;
4) $L \colon S \to 2^{AP}$ is a labeling function which assigns to every state $s \in S$ a set $L(s) \subseteq AP$ of atomic propositions which are regarded true at this state.

Kripke structures

A path in a model M from a state s is such an infinite sequence of states $\pi = s_0, s_1, s_2, \ldots$ that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for every $i \geqslant 0$.

A state s is called a reachable state of a model if some path from an initial state of the model goes through s .

A state s is called deadlock state of a model if every path from s goes only through the state s .

Sometimes transitions in Kripke model are labeled with the names of those program actions that fire these transitions. In this case a set of actions Act is introduced and the transitions of the model are defined as triples $R \subseteq S \times Act \times S$.

Kripke structures: Ferry.

Kripke structures: Ferry.

Kripke structures: Ferry.

Kripke structures: Ferry.

Kripke structures: Ferry.

Kripke structures: Ferry.

The boatsmsan can cross the river

alone

Kripke structures: Ferry.

The boatsmsan can cross the river

alone

Kripke structures: Ferry.

or with a passanger

Kripke structures: Ferry.

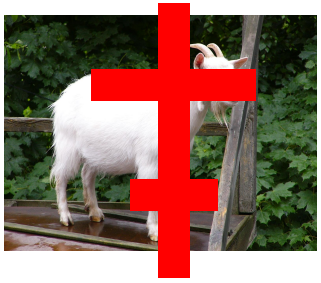
or with a passanger

Kripke structures: Ferry.

Some passengers may get hurt

without the boatman's supervision

Kripke structures: Ferry.



Some passengers may get hurt

without the boatman's supervision

Kripke structures: Ferry.

What is important to know?

It is important to know

whether the person is alive or not,

where is the person.

Kripke structures: Ferry.

The set of states in the Kripke structure Ferry

$$S = \{-1, 0, 1\}^4 \ .$$

For every state (x_1, x_2, x_3, x_4)

$x_i = -1$ means that the person i is on the left side,

$x_i = 1$ means that the person i is on the right side.

$x_i = 0$ means that the person i no longer lives.

Kripke structures: Ferry.

The set of states in the Kripke structure Ferry

$$S = \{-1, 0, 1\}^4 \ .$$

For every state (x_1, x_2, x_3, x_4)

$x_i = -1$ means that the person i is on the left side,

$x_i = 1$ means that the person i is on the right side.

$x_i = 0$ means that the person i no longer lives.

The set of imitial states

$$S_0 = \{(-1, -1, -1, -1)\}$$

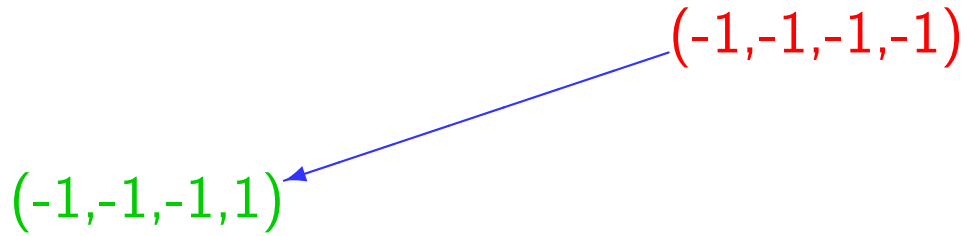
Kripke structures: Ferry.

Transition relation R :

(-1,-1,-1,-1)

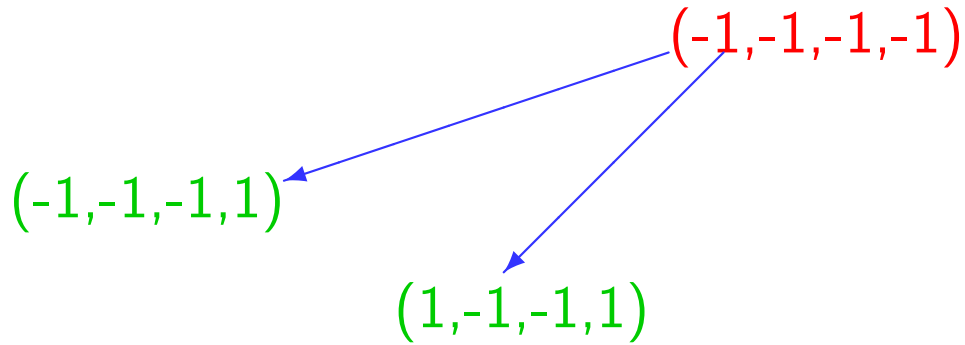
Kripke structures: Ferry.

Transition relation R :



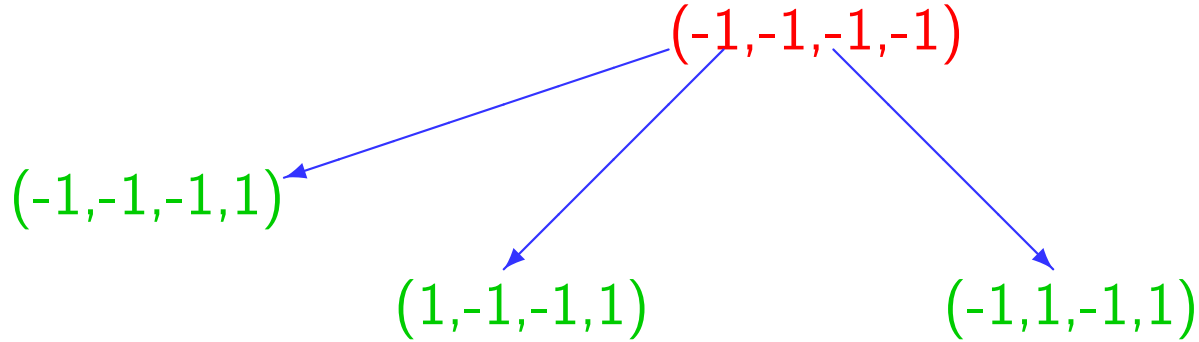
Kripke structures: Ferry.

Transition relation R :



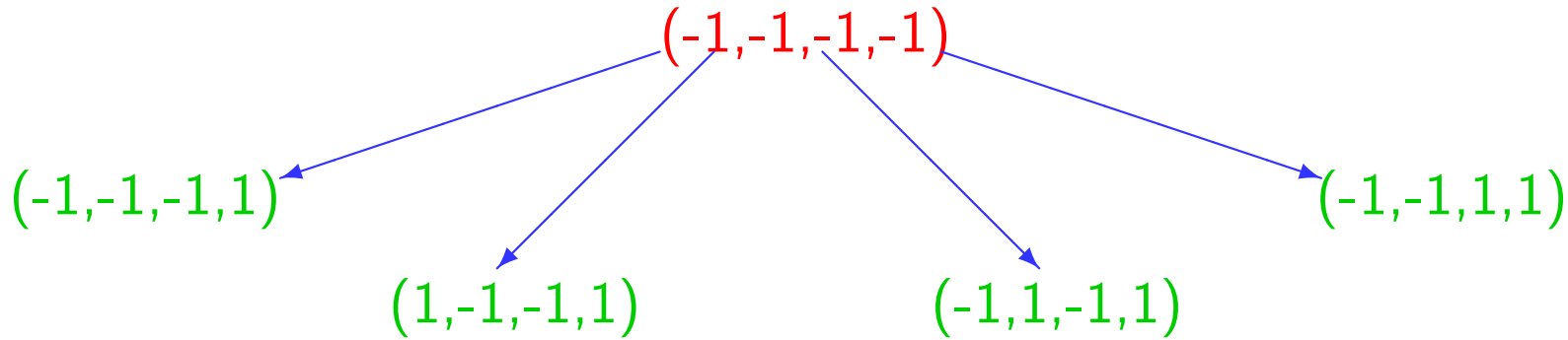
Kripke structures: Ferry.

Transition relation R :



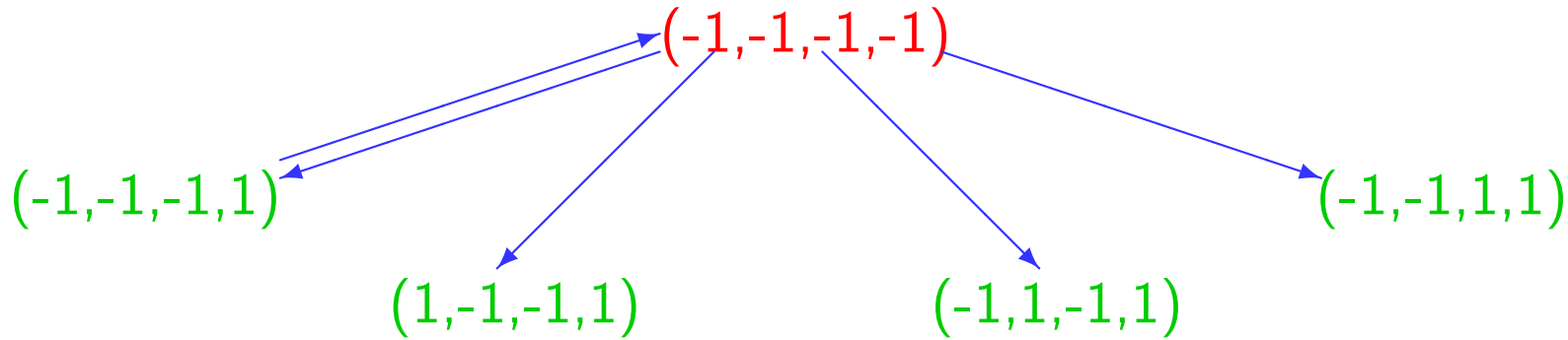
Kripke structures: Ferry.

Transition relation R :



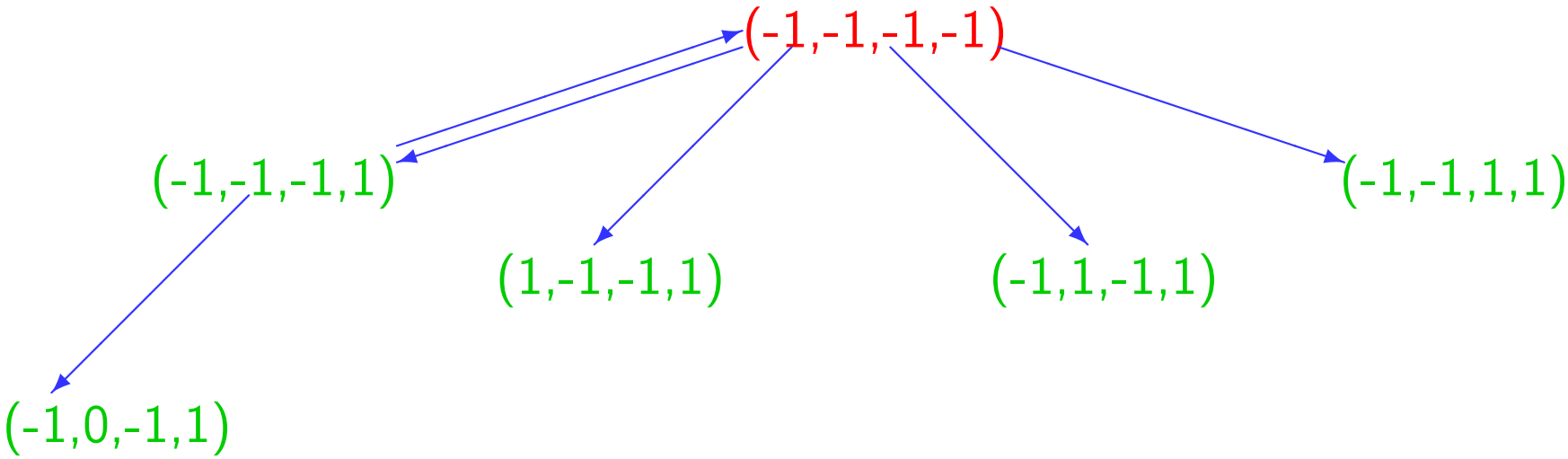
Kripke structures: Ferry.

Transition relation R :



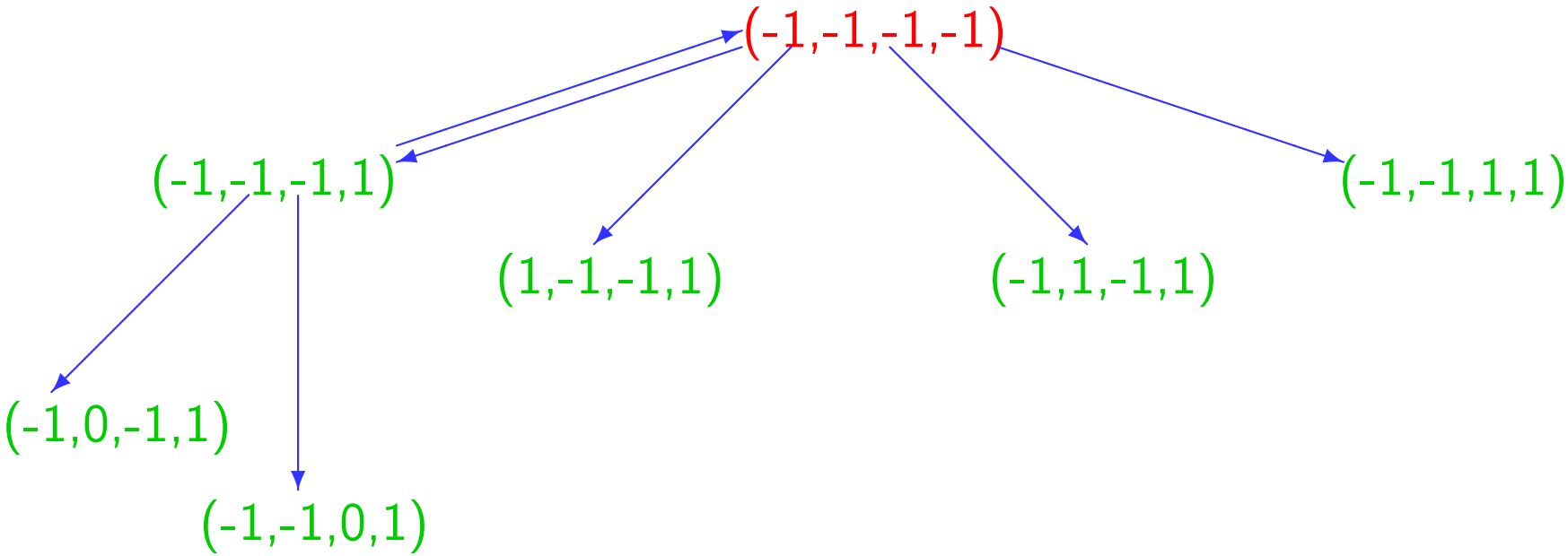
Kripke structures: Ferry.

Transition relation R :



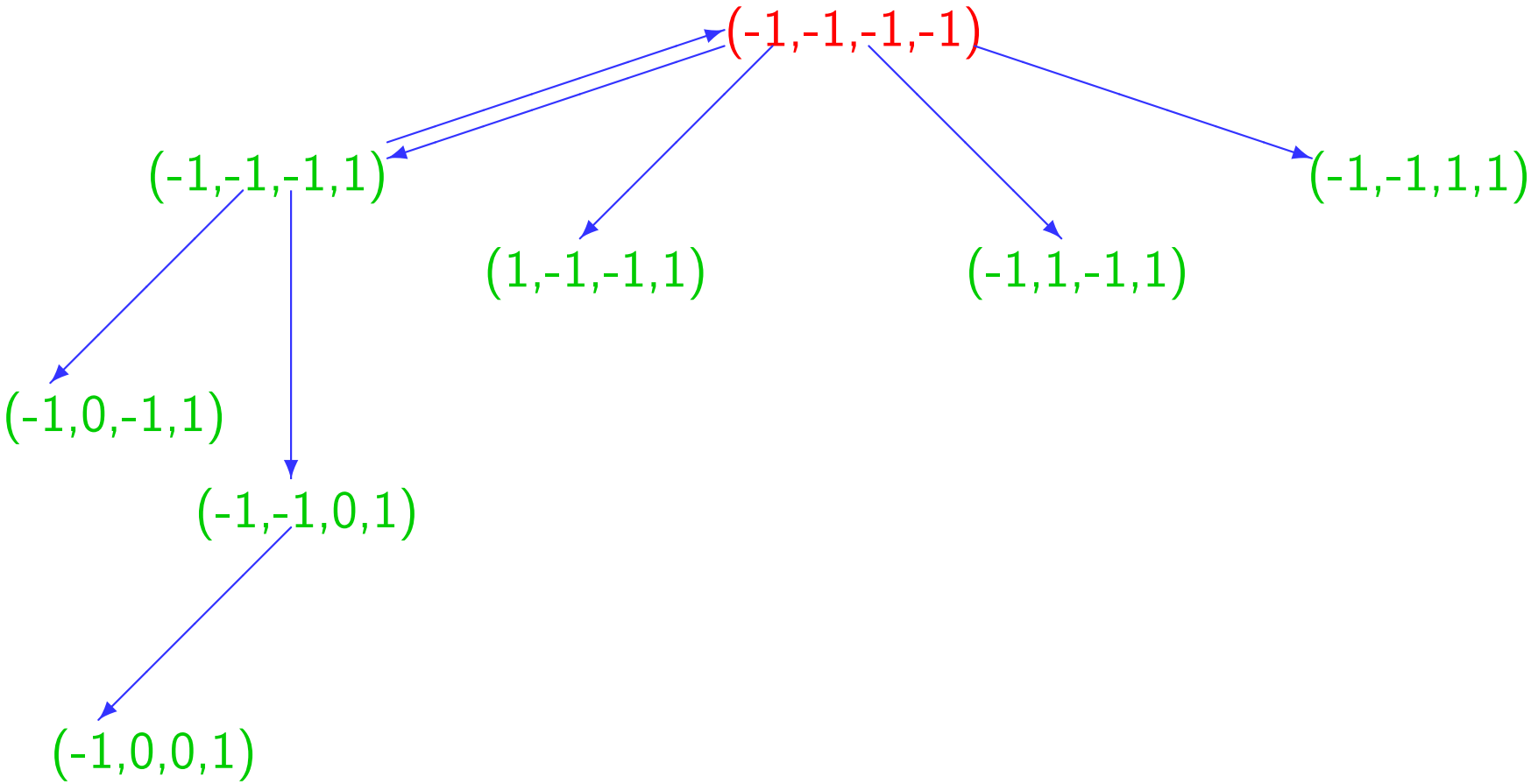
Kripke structures: Ferry.

Transition relation R :



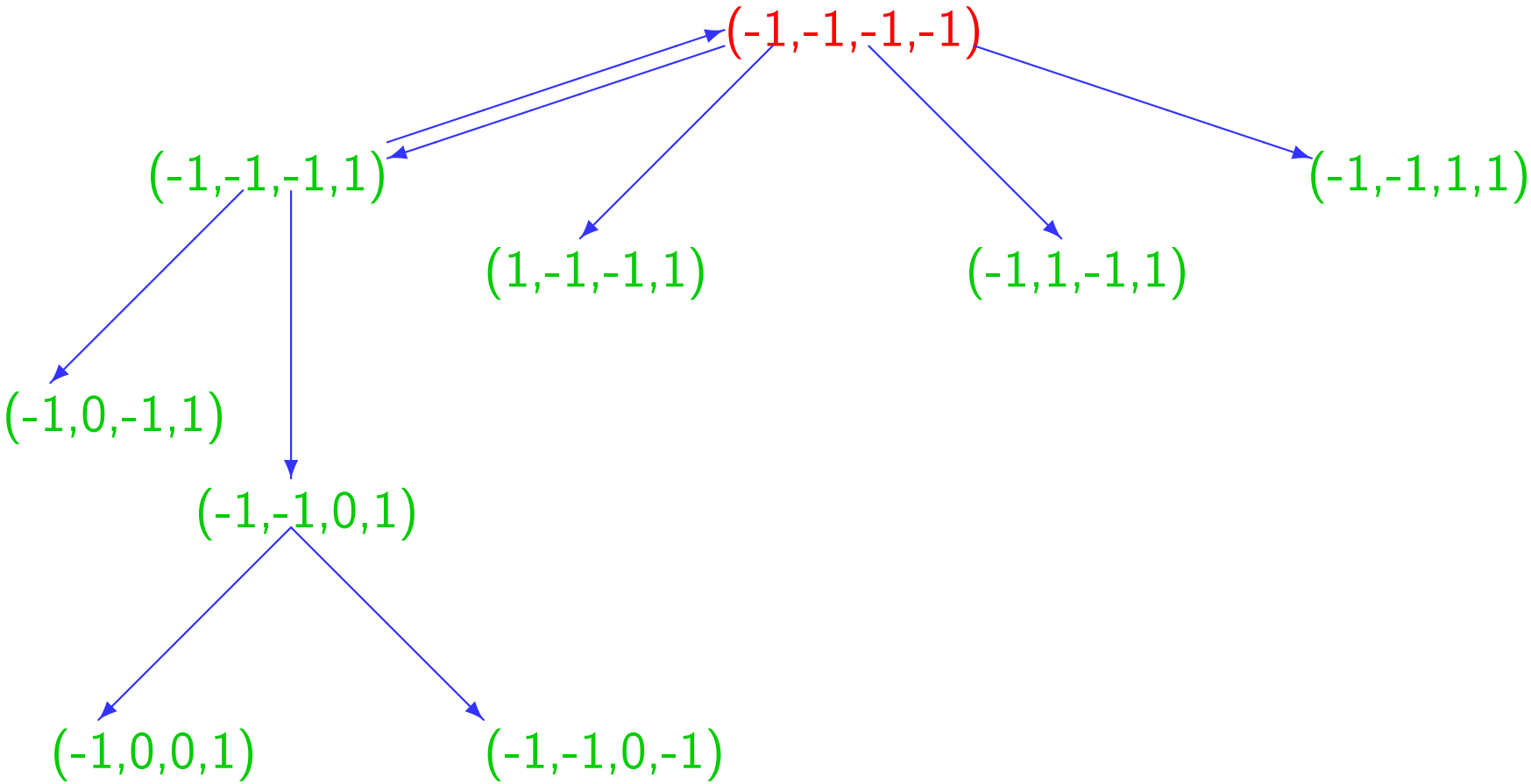
Kripke structures: Ferry.

Transition relation R :



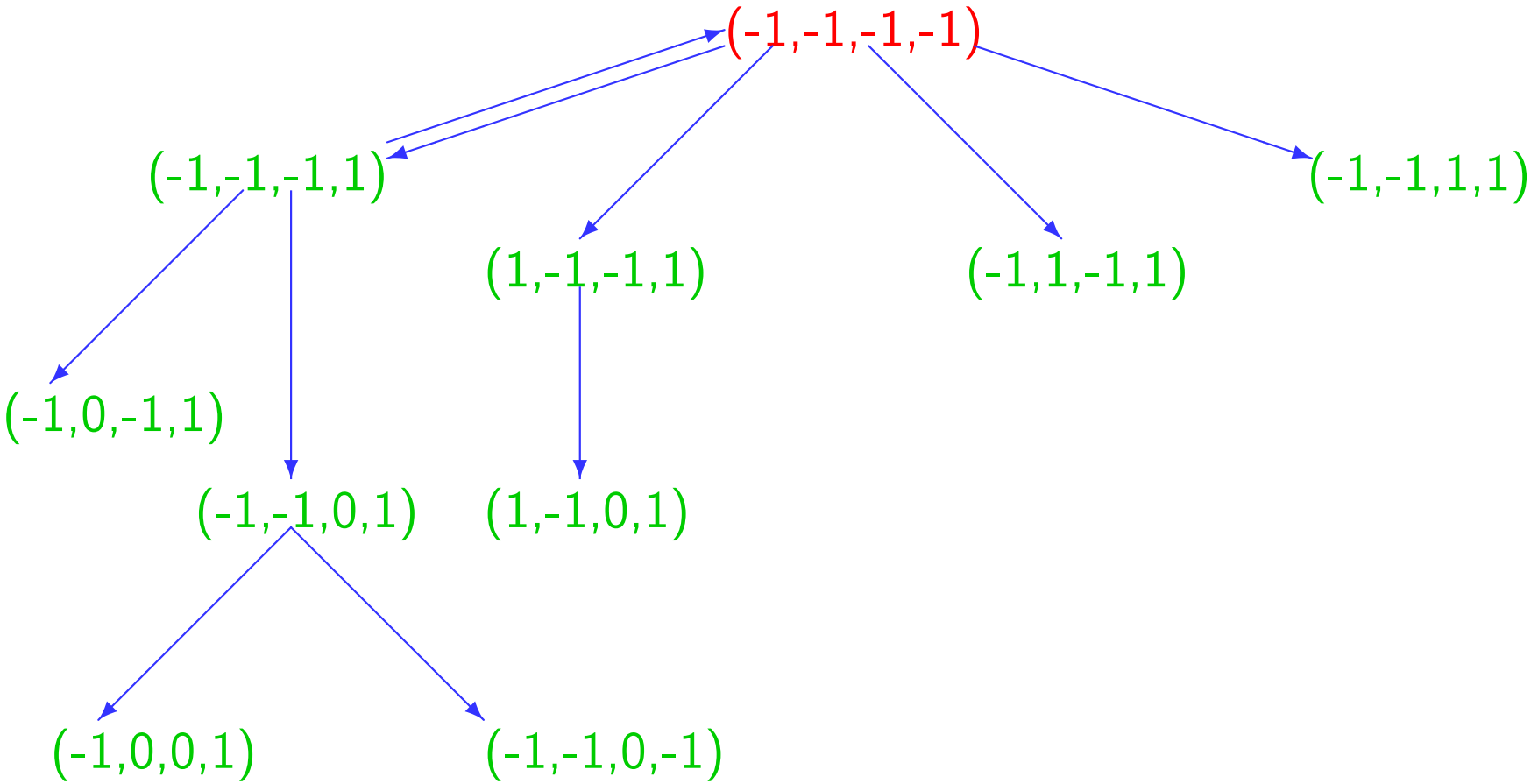
Kripke structures: Ferry.

Transition relation R :



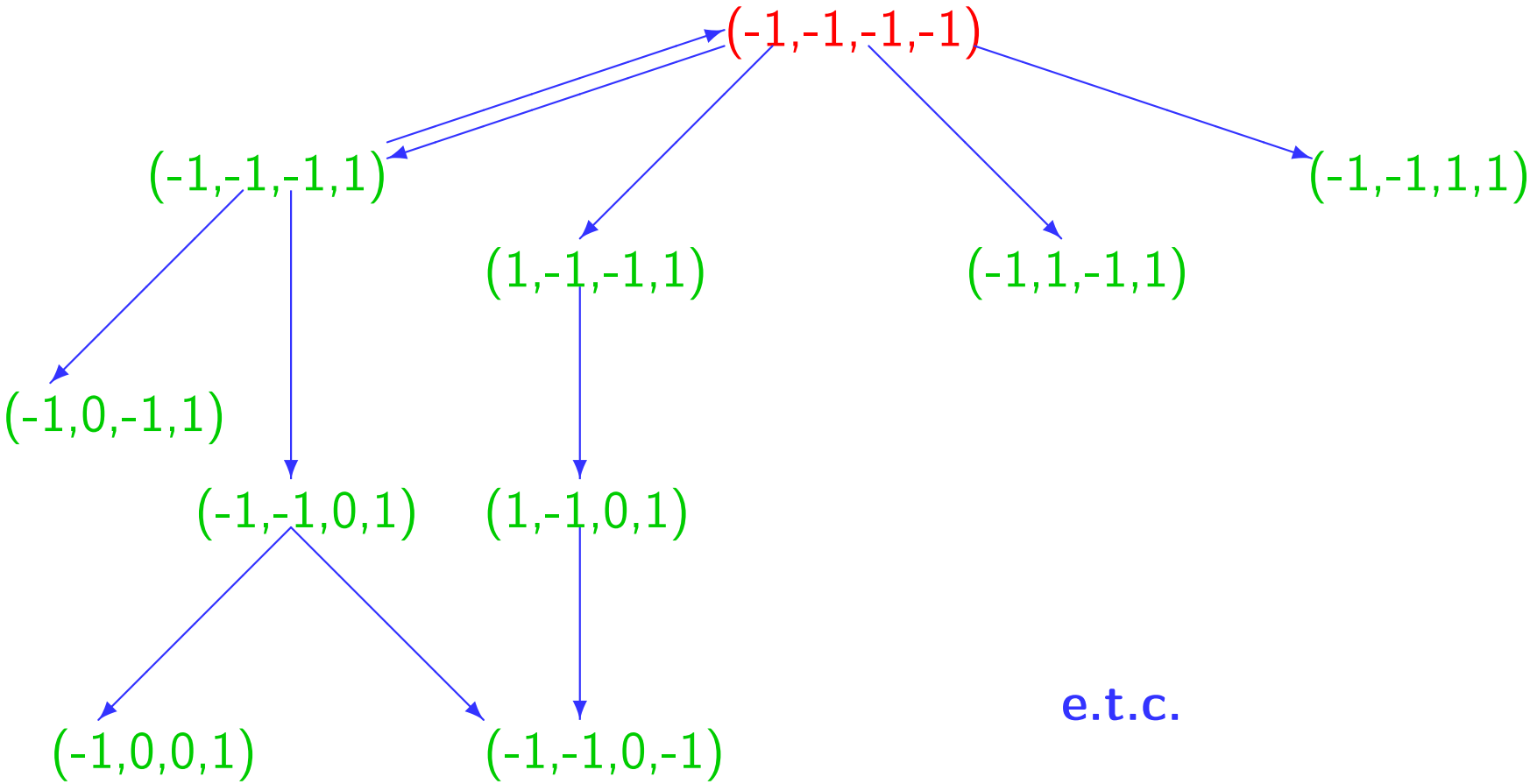
Kripke structures: Ferry.

Transition relation R :



Kripke structures: Ferry.

Transition relation R :



e.t.c.

Kripke structures: Ferry.

Consider a set of atomic propositions

$$AP = \{alive_i, left_i : i = 1, 2, 3, 4\}$$

Функция разметки L :

Kripke structures: Ferry.

Consider a set of atomic propositions

$$AP = \{alive_i, left_i : i = 1, 2, 3, 4\}$$

Функция разметки L :

$L((-1,-1,-1,-1)) = AP$,

$L((-1,0,1,1)) = \{alive_1,\ alive_3,\ alive_4,\ left_1\}$,

e.t.c.

The main principles of modeling

There are different types of parallel systems (synchronous and asynchronous circuits, programs with shared variables, programs interacting via message exchange, etc.). Due to this diversity, one needs a universal formalism, within which it would be possible to represent a parallel system of any type.

First-order logic formulas are very well suited for this purpose. Having such a formula that defines some parallel system, one can construct a corresponding Kripke structure that serves as an adequate model of the system.

First-order representations

Only those first-order formulas are suitable for describing formally parallel systems which are interpreted in some fixed first-order structure.

This means that predicate and function symbols occurred in these formulas have some predefined meaning.

Let $V = \{u_1, \ldots, u_n\}$ be a set of the variables of a system.

We assume that the variables from V take values from some finite set D, which is called a domain of the interpretation.

A valuation of V is any function which maps V to D.

First-order representations

A state of a parallel system is completely specified by the values of all variables in V. In other words, a state is a valuation $s\colon V \to D$ of the set of variables V.

For a given valuation one can write a formula, which is true exactly on this valuation. For example, for the set of variables $V = \{u_1, u_2, u_3\}$ a valuation $\langle u_1 \leftarrow 2, u_2 \leftarrow 3, u_3 \leftarrow 5\rangle$ is characterized by the formula $(u_1 = 2) \wedge (u_2 = 3) \wedge (u_3 = 5)$.

The same formula can be true on many valuations.
We may assume that any first-order formula Φ specifies the set of valuations (states)

$$S_\Phi = \{s : s \models \Phi\}$$

which make this formula true.
In particular, we denote by $\mathcal{S}_0$ any formula over a set of variables V which specifies the set of initial states S_0 of a system.

First-order representations

To specify transitions between states we use formulas to represent the set of ordered pairs of states.

Given a set of variables V, we create another set of variables V' which are the copies of the variables in V. Every variable u in V corresponds to some variable in V' which will be denoted u'. A valuation of variables from V will be regarded as a source state of a transition, whereas a valuation of variables from V' as a target state of the same transition.

Every valuation of variables from both sets V and V' may be viewed as a description of an ordered pair of states (s, s'), i.e. a transition from a state s to a state s'. Sets of transitions can be specified in the same way as sets of states — by means of first-order formulas.

Any set of ordered pairs of states will be called a transition relation. If R is a transition relation, then we write $\mathcal{R}(V, V')$ to denote a first-order formula which specifies R.

First-order representations

To write formal specifications of the system's properties one needs to choose a set of basic properties as atomic propositions AP.

The most simple basic properties are expressed usually by such formulas as $u = d$, where $u \in V$ and $d \in D$.

An atomic proposition $u = d$ is true at a state s if $s(u) = d$.

If u is a variable over the Boolean domain $\{0, 1\}$ (Boolean variable) then there is no need to write the equalities $u = 0$ and $u = 1$. Instead of writing $u = 0$ we will use a notation $\neg u$, and instead of $u = 1$ we will write u.

More generally, any relation over a domain D

$$P(u_{i_1}, u_{i_2}, \ldots, u_{i_k}) \subseteq D \times D \times \cdots \times D$$

can be an atomic statement.

First-order representations

Now let us see what is a Kripke model $M = (S, S_0, R, L)$ which is defined by first-order formulas $\mathcal{S}_0$ and $\mathcal{R}$.

- The set of states S is the set of all valuations of variables V
- The set of initial states S_0 is the set of all those valuations s_0 of V, which satisfy the formula $\mathcal{S}_0$.
- For every pair of states s and s', a relation $R(s, s')$ holds iff the formula $\mathcal{R}$ is evaluated to $True$ whenever each variable $u \in V$ takes the value $s(u)$ and each variable $u' \in V'$ takes the value $s'(u')$.
- A labeling function $L\colon S \to 2^{AP}$ is defined so that $L(s)$ is the set of all those basic propositions which are true at the state s. If a formula $\mathcal{P}(x_{i_1}, \ldots, x_{i_k})$ stands for a basic proposition P, then $P \in L(s)$ iff this formula evaluates to $True$ on the tuple $(s(u_{i_1}), \ldots, s(u_{i_k}))$. If u is a Boolean variable, then $u \in L(s)$ means that $s(u) = 1$, and $u \notin L(s)$ means that $s(u) = 0$.

First-order representations

Example.

Consider a simple system which has 2 variables x and y, and these variables take values from the set $D = \{0, 1\}$.

Hence, the valuations of x and y are all pairs $(d_1, d_2) \in D \times D$, where d_1 is a value of x, and d_2 is a value of y.

The system has the only transition which is defined by the action

$$x := (x + y) (\mathrm{mod}\ 2),$$

and the initial values $x = 1$ and $y = 1$.

Example.

The system is characterized by 2 first-order formulas.

$$\mathcal{S}_0(x, y) \equiv x \wedge y \ ,$$

$$\mathcal{R}(x, y, x', y') \equiv x' = (x + y)(\mathrm{mod}2) \wedge y' = y \ .$$

A Kripke structure $M = (S, S_0, R, L)$ is as follows:

- $S = D \times D$;
- $S_0 = \{(1, 1)\}$;
- $R = \{\langle(1, 1), (0, 1)\rangle, \langle(0, 1), (1, 1)\rangle,$
  $\langle(1, 0), (1, 0)\rangle, \langle(0, 0), (0, 0)\rangle\}$;
- $L((1, 1)) = \{x, y\}$, $L((0, 1)) = \{\neg x, y\}$,
  $L((1, 0)) = \{x, \neg y\}$, $L((0, 0)) = \{\neg x, \neg y\}$.

The only initial path in this Kripke structure is
$(1, 1), (0, 1), (1, 1), (0, 1), \ldots$

Granularity of a model description

The crucial aspect of modeling parallel systems is granularity of operations. It is important to achieve such atomicity of transitions that no state of the system can be observed as the result of performing only some part of a single transitions.

When a single transition models an execution of a whole sequence of actions, a Kripke model does not allow one to observe the results obtained after every step of such execution. Therefore, it may be so that such a Kripke model hides some errors of computation by making invisible those intermediate states where these errors occur.

The problems arise also when a description of a model is overly detailed, and an atomic action of a program is represented by a sequence of transitions. In this case parallel composition of such chains of micro-transition may bring a system into some states which never appear in real computations of the system.

Kripke model: Forge.

Kripke model: Forge.

Is it reasonable to represent

by separate transitions such actions as

river sailing?

Kripke model: Forge.

Is it reasonable to represent

by separate transitions such actions as

river sailing?

embarking?

Kripke model: Forge.

Is it reasonable to represent

by separate transitions such actions as

river sailing?

embarking?

disembarking?

Kripke model: Forge.

Is it reasonable to represent

by separate transitions such actions as

river sailing?

embarking?

disembarking?

What happens if we regard

as atomic such action as

boatman sailing in both directions?

Granularity of a model description

Consider a system with two variables x and y and two transitions α and β which can be executed in parallel:

$$\alpha: \quad x{:=}x+y, \qquad \| \qquad \beta: \quad y{:=}y+x$$

The set of initial states is specified by $x = 1 \ \wedge \ y = 2$.

Granularity of a model description

Consider a system with two variables x and y and two transitions α and β which can be executed in parallel:

$$\alpha\colon \quad x{:=}x+y, \qquad \| \qquad \beta\colon \quad y{:=}y+x$$

The set of initial states is specified by $x=1 \ \wedge \ y=2$.

Also consider a more detailed implementation of these transitions in assembly language:

	α	$\|$	β
α_0:	load R_1, x		β_0: load R_2, y
α_1:	add R_1, y		β_1: add R_2, x
α_2:	store R_1, x		β_2: store R_2, y

Granularity of a model description

A run α , β brings the program to a state $x = 3 \ \wedge \ y = 5$.
And a run β , α brings the program to a state $x = 4 \ \wedge \ y = 3$.

Granularity of a model description

A run α , β brings the program to a state $x = 3 \ \wedge \ y = 5$.
And a run β , α brings the program to a state $x = 4 \ \wedge \ y = 3$.

A more detailed implementation of the same program can have such run as

$$\alpha_0, \ \beta_0, \ \alpha_1, \ \beta_1, \ \alpha_2, \ \beta_2,$$

and it brings a program to a state $x = 3 \ \wedge \ y = 3$.

The correctness of the system depends on which model of parallel computing this program is implemented in.

Parallel syatems

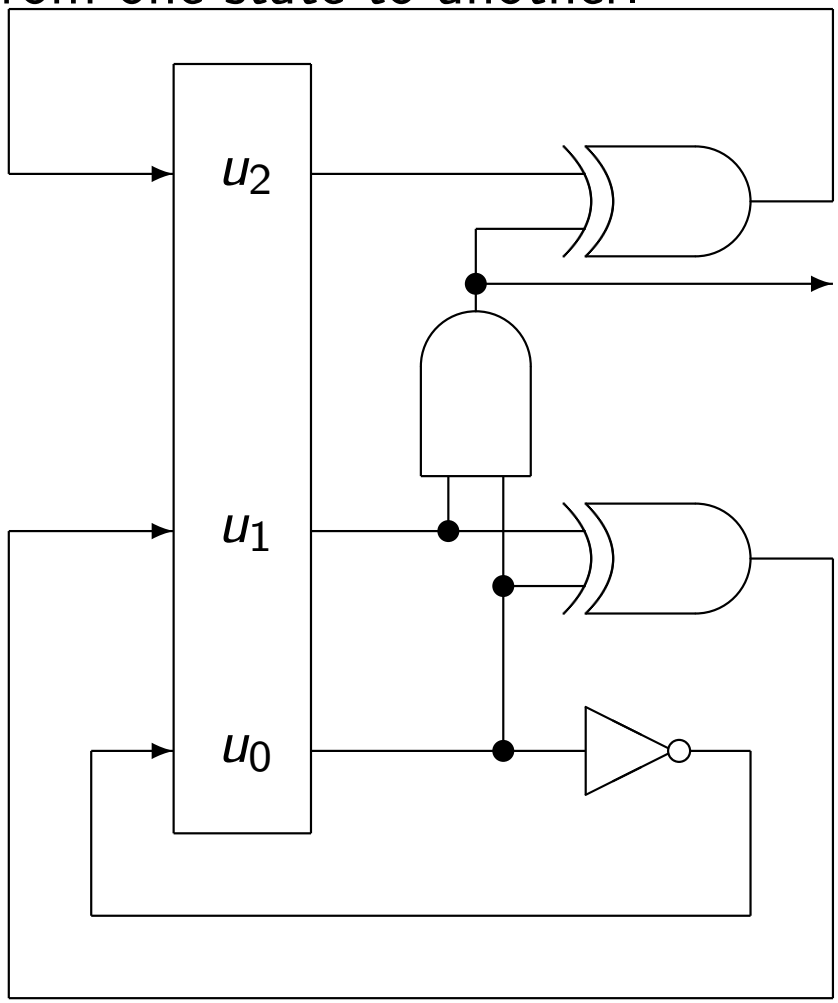
Parallel systems are composed of sequential programs which are executed simultaneously. Usually the components of parallel systems are supplied with certain means for interaction. The princples of parallel execution and the interaction machinary varies in different parallel systems.

There are parallel executions of two types: asynchronous , or interleaving execution , when every time only one component executes its computing action, and synchronous execution , when every time all components execute their computing actions simultaneously.

Two types of interaction are the most common: by reading and updating the values of shared variables , or by message passing .

Synchronous circuits

At every step of execution a synchronous electronic circuit receives signals at its input. After a synchronizing pulse passes through the circuit, these signals act on the circuit elements and transfer them from one state to another.



Synchronous circuits

Transitions of the counter are specified by a system of equations

$$u_0' = \neg u_0 \ ,$$
$$u_1' = u_0 \oplus u_1 \ ,$$
$$u_2' = (u_0 \wedge u_1) \oplus u_2 \ ,$$

These equation can be used to define the following relationships

$$\mathcal{R}_0(V, V') \equiv (u_0' \Leftrightarrow \neg u_0) \ ,$$
$$\mathcal{R}_1(V, V') \equiv (u_1' \Leftrightarrow u_0 \oplus v_1) \ ,$$
$$\mathcal{R}_2(V, V') \equiv (u_2' \Leftrightarrow (u_0 \wedge u_1) \oplus u_2) \ ,$$

which specify the constraints on the variables in any admissible transition. Since the values of all variables change simultaneously during the passage of synchronization pulse, to build a formula $\mathcal{R}$ which formally specifies the transition relation these constraints are joined by means of conjunctions:

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \wedge \mathcal{R}_1(V, V') \wedge \mathcal{R}_2(V, V') \ .$$

Asynchronous circuits

Transition relations for asynchronous systems are expressed with the help of disjunction. Suppose that every component of a system has a single input and does not have internal state variables. In this case the computing capability of every such component can be characterized by a function $f_i(u)$; at every state by the curent values of variables u this component outputs $f_i(u)$.

Since the components of asynchronous systems operate independently and with a high performance it is practically impossible for any two components to change their states simultaneously. Therefore, it is suitable to use the interleaving semantics which is based on the following principal assumption: at every step of computation only one component (process) of a parallel system changes its state. This can be expressed by means of the disjunction

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \vee \cdots \vee \mathcal{R}_{n-1}(V, V') \ ,$$

где $\mathcal{R}_i(V, V') \equiv \big(u_i' \Leftrightarrow f_i(V)\big) \wedge \bigwedge_{j \neq i} (u_j' \Leftrightarrow u_j)$.

Translation of programs to Kripke models

Let us define the main rules of a translation $\mathcal{C}$ of sequential and parallel programs P into first-order formula $\mathcal{R}$ which specifies the set of transitions of the program.

The syntax of our programs:

- $x := e$, **skip**, **wait**(b);
- $\pi = \pi_1; \pi_2;$
- $\pi =$ **if** b **then** π_1 **else** π_2 **fi**,
- $\pi =$ **while** b **do** π_1 **od**,
- **cobegin** $\pi_1 \parallel \pi_2 \parallel \cdots \parallel \pi_m$ **coend**.

Translation of programs to Kripke models

Without loss of generality we will assume that every program statement has the only entry and the only exit. All labels are pairwise different. Translation merges exit of one statement and entry of the next statement. As the result we obtain the unambiguous labeling of entries and exits of all statements.

Translation of programs to Kripke models

Without loss of generality we will assume that every program statement has the only entry and the only exit. All labels are pairwise different. Translation merges exit of one statement and entry of the next statement. As the result we obtain the unambiguous labeling of entries and exits of all statements.

x:=y; **if** x>0 **then** y:=z **else** z:=x **fi**; x:=z;

Translation of programs to Kripke models

Without loss of generality we will assume that every program statement has the only entry and the only exit . All labels are pairwise different. Translation merges exit of one statement and entry of the next statement. As the result we obtain the unambiguous labeling of entries and exits of all statements.

x:=y; **if** x>0 **then** y:=z **else** z:=x **fi**; x:=z;

0: x:=y;:1 2: **if** x>0 **then** 3: y:=z:4 **else** 5: z:=x:6 **fi**;:7 8: x:=z;:9

Translation of programs to Kripke models

Without loss of generality we will assume that every program statement has the only entry and the only exit . All labels are pairwise different. Translation merges exit of one statement and entry of the next statement. As the result we obtain the unambiguous labeling of entries and exits of all statements.

x:=y; **if** x>0 **then** y:=z **else** z:=x **fi**; x:=z;

0: x:=y;:1 2: **if** x>0 **then** 3: y:=z:4 **else** 5: z:=x:6 **fi**;:7 8: x:=z;:9

0: x:=y;:1 1: **if** x>0 **then** 3: y:=z:4 **else** 5: z:=x:4 **fi**;:4 4: x:=z;:9

Translation of programs to Kripke models

We introduce a variable pc of special type which is called command counter ; its domain is the set of all program labels and a special element $\perp$ (undefined value). The undefined value is used when we deal with parallel programs. In this case $pc = \perp$ means that the program is not active yet.

Let V be the set of all program variables. This set is accompanied with the set V' of primed variables u' which are in one-to-one correspondence with the variables $u \in V$, the set of primed varibles also includes pc' as a counterpart of command counter pc .

Since every transition usually updates only a small fraction of program variables, we will write $same(Y)$ to denote a formula

$$\bigwedge_{y \in Y} (y' = y) \ .$$

Translation of programs to Kripke models

First, we build a formula which specifies the set of initial states of a program P. Given a certain pre-condition $pre(V)$, which specifies the initial values of variables of the program P, this formula looks as follows

$$\mathcal{S}_0(V, pc) \equiv pre(V) \wedge pc = m\,.$$

Translation $\mathcal{C}$ depends on three parameters: an entry label ℓ, a labeled statement P and an exit label ℓ'. This is a recursive procedure which uses one rule per every type of program statements. A predicate $\mathcal{C}(\ell, P, \ell')$ describes the set of transitions of the program P as a disjunction of subformulas which specify transitions from this set.

Translation of programs to Kripke models

Assignment statement:

$$\mathcal{C}(\ell,\ u := e,\ \ell') \ \equiv$$
$$pc = \ell \ \wedge\ pc' = \ell' \ \wedge\ u' = e \ \wedge\ same(V \setminus \{u\}) \ .$$

Translation of programs to Kripke models

Assignment statement:

$$\mathcal{C}(\ell,\ u := e,\ \ell') \ \equiv \\ pc = \ell \ \wedge\ pc' = \ell' \ \wedge\ u' = e \ \wedge\ same(V \setminus \{u\}) \ .$$

Instruction **skip**:

$$\mathcal{C}(\ell, \textbf{skip},\ \ell') \ \equiv\ pc = \ell \wedge\ pc' = \ell \ \wedge\ same(V) \ .$$

Translation of programs to Kripke models

Assignment statement:

$$\mathcal{C}(\ell,\ u := e,\ \ell') \ \equiv \\ pc = \ell \ \wedge\ pc' = \ell' \ \wedge\ u' = e \ \wedge\ same(V \setminus \{u\}) \ .$$

Instruction **skip**:

$$\mathcal{C}(\ell, \textbf{skip},\ \ell') \ \equiv\ pc = \ell \wedge\ pc' = \ell \ \wedge\ same(V) \ .$$

Sequential composition of statements:

$$\mathcal{C}(\ell,\ P_1;\ \ell'' : P_2,\ \ell') \ \equiv\ \mathcal{C}(\ell, P_1, \ell'') \ \vee\ \mathcal{C}(\ell'', P_2, \ell') \ .$$

Translation of programs to Kripke models

Branching statement **if-then-else**:

$\mathcal{C}(\ell,$ **if** b **then** $\ell_1 : P_1$ **else** $\ell_2 : P_2$ **end if**, $\ell')$
is a disjunction of the following four formulas:

- $pc = \ell \ \wedge \ pc' = \ell_1 \ \wedge \ b \ \wedge \ same(V)$,
- $pc = \ell \ \wedge \ pc' = \ell_2 \ \wedge \ \neg b \ \wedge \ same(V)$,
- $\mathcal{C}(\ell_1, P_1, \ell')$,
- $\mathcal{C}(\ell_2, P_2, \ell')$.

The first subformula covers the case when the condition b is true. In this case the statement P_1 is executed. The second subformula corresponds to the case when the condition b is false. In this case the control passes to the statement P_2 . Both subformulas change the value of command counter only. The third and the forth subformulas specify the transitions of the statements P_1 and P_2 .

Translation of programs to Kripke models

Loop statement **while-do**:

$\mathcal{C}(\ell,$ **while** b **do** $\ell_1 : P_1$ **end while**, $\ell')$
is a dijunction of three subformulas:

- $pc = \ell \ \wedge \ pc' = \ell_1 \ \wedge \ b \ \wedge \ same(V)$,
- $pc = \ell \ \wedge \ pc' = \ell' \ \wedge \ \neg b \ \wedge \ same(V)$,
- $\mathcal{C}(\ell_1, P_1, \ell)$.

The first subformula specifies the case when the condition b is true. In this case at the next step the statement P_1 is executed. The second subformula corresponds to the case when b is false. Then the execution of the loop statements terminates. The third subformula specifies the transitions of the statement P_1. It should be noted that the exit from the statement P_1 is joined with the entry to the loop statement. Thus as soon as the execution of the statement P_1 ends the loop statement is started again.

Translation of programs to Kripke models

Parallel composition P :
$P = \ell : \textbf{cobegin } \ell_1 : P_1^{\mathcal{L}} \ell'_1 \parallel \ell_2 : P_2^{\mathcal{L}} \ell'_2 \parallel \ldots \parallel \ell_n : P_n^{\mathcal{L}} \ell'_n \textbf{ coend} : L'.$

Translation of programs to Kripke models

Parallel composition P :
$P = \ell : \textbf{cobegin}\ \ell_1 : P_1^{\mathcal{L}} \ell_1' \parallel \ell_2 : P_2^{\mathcal{L}} \ell_2' \parallel \ldots \parallel \ell_n : P_n^{\mathcal{L}} \ell_n'\ \textbf{coend} : L'.$

The formula
$\mathcal{C}(\ell,\ \textbf{cobegin}\ P_1 \parallel P_2 \parallel \ldots \parallel P_n\ \textbf{coend},\ \ell')$
is a disjunction of three subformulas:

- $pc = \ell\ \wedge\ pc_1' = \ell_1\ \wedge \ldots \wedge\ pc_n' = \ell_n\ \wedge\ pc' = \bot$,
- $pc = \bot\ \wedge\ pc_1 = \ell_1'\ \wedge \ldots \wedge\ pc_n = \ell_n'\ \wedge\ pc' = \ell'\ \wedge$
  $\wedge\ \bigwedge_{i=1}^{n}(pc_i' = \bot)$,
- $\bigvee_{i=1}^{n}\big(\mathcal{C}(\ell_i, P_i, \ell_i')\ \wedge\ same(V \setminus V_i)\ \wedge\ same(PC \setminus \{pc_i\})\big)$.

The first subformula specifies initialization of parallel processes. The second subformula specifies completion of the execution of parallel program. and the third subformula specifies the execution of parallel processes.

Translation of programs to Kripke models

Instruction **wait**.

The instruction **wait**(b) permanently checks the value of Boolean variable b until it finds that b is evaluated to true. As soon as b becomes true, the instruction passes the control to the next statement in the program.

Formula $\mathcal{C}(\ell,\ \mathbf{wait}(b),\ \ell')$ is a disjunction of two subformulas:

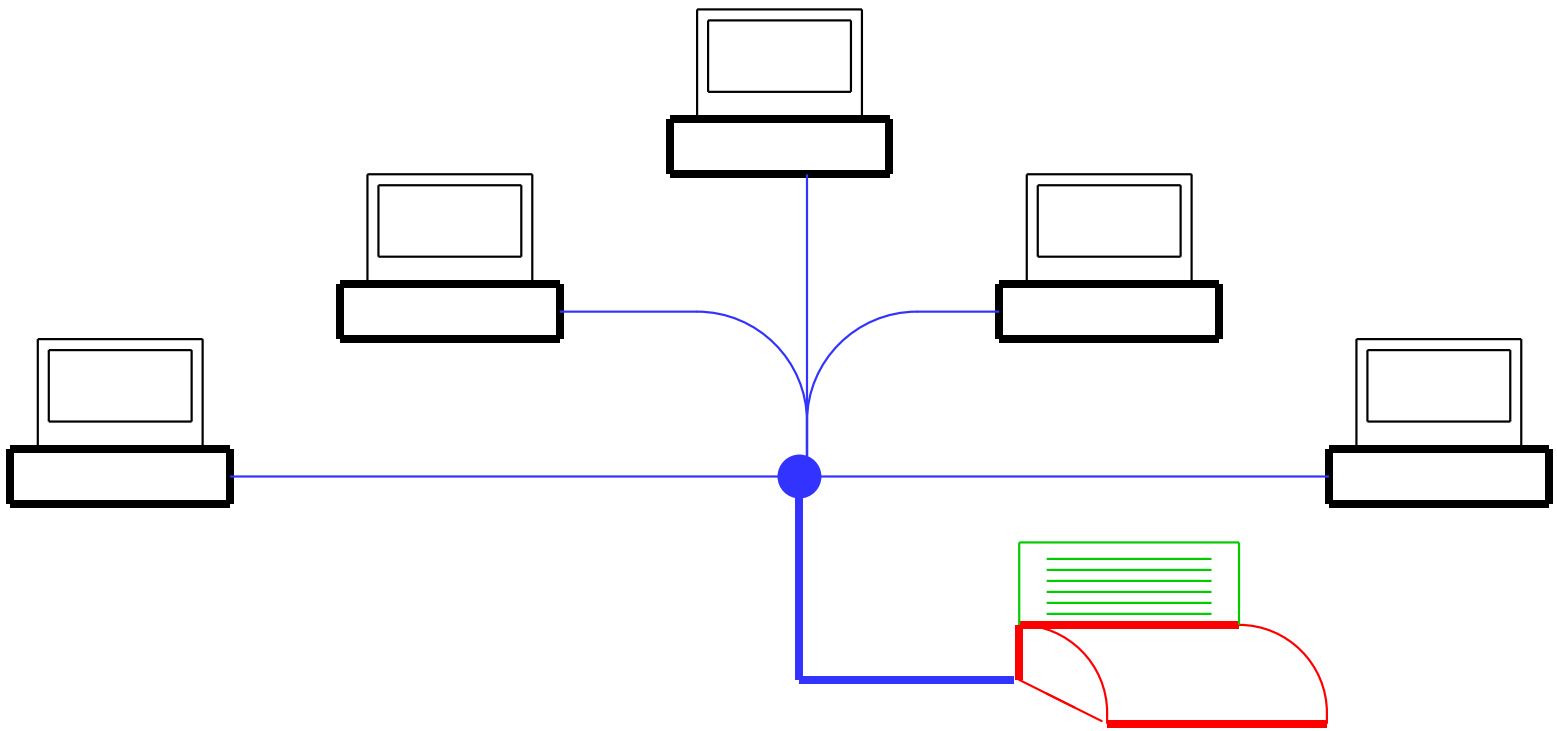
$$pc_i = \ell \ \wedge\ pc_i' = \ell \ \wedge\ \neg b \ \wedge\ same(V_i)\ ,$$

$$pc_i = \ell \ \wedge\ pc_i' = \ell' \ \wedge\ b \ \wedge\ same(V_i)\ .$$

Example of a program

Task

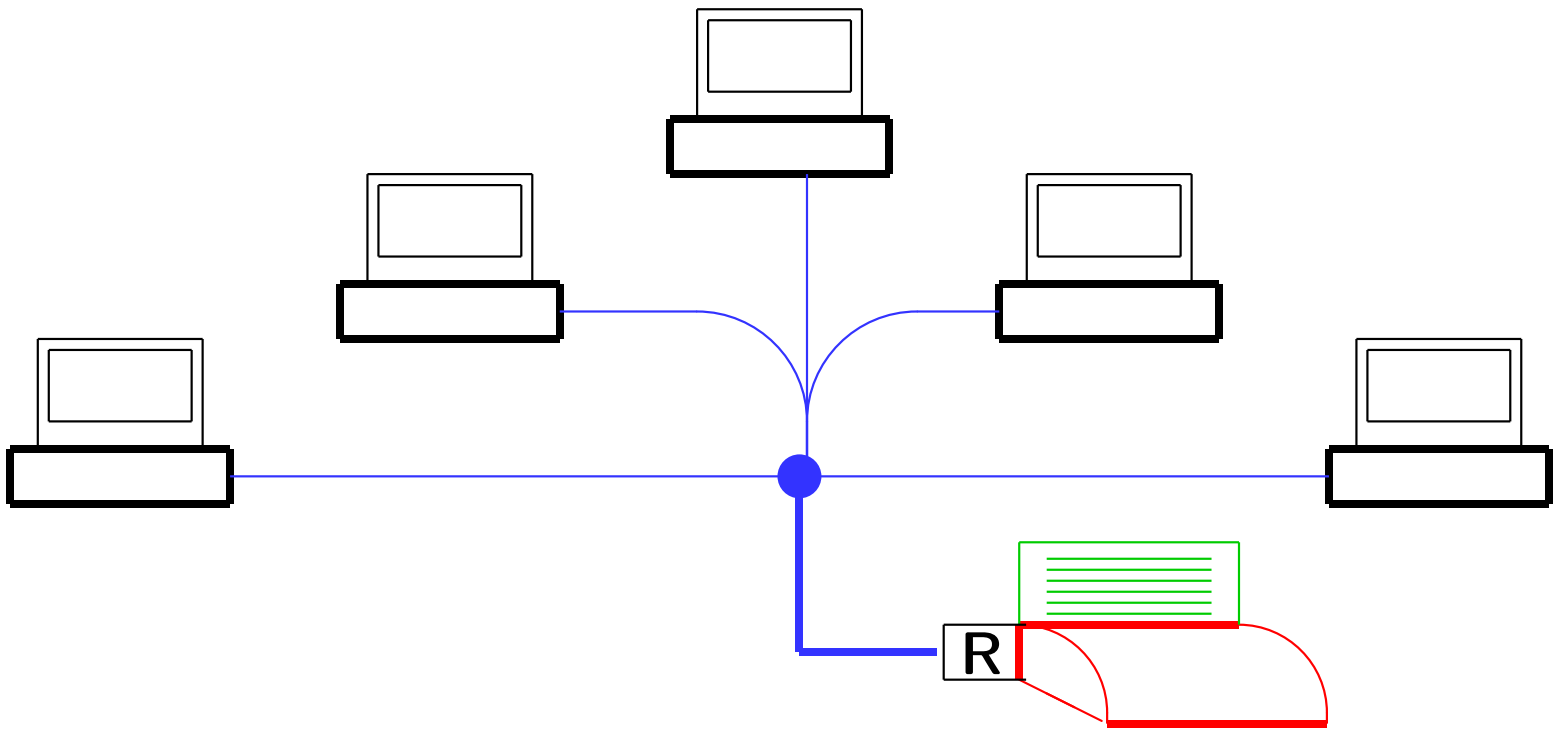
There are several computers and only one printer. No computer is aware of the existence of other computers. How to organize their interaction correctly so that they can all use this printer?



Example of a program

Task

It is assumed that the printer has a single 1-bit shared CRCW memory R (Concurrent Read — Concurrent Write). This memory can be either in the state *busy* (the printer is occupied), or *free* (the printer is free).



Example of a program

Before writing a program (driver) that ensures the interaction of each computer with a printer, one needs to formulate the requirements to this program.

Example of a program

Before writing a program (driver) that ensures the interaction of each computer with a printer, one needs to formulate the requirements to this program.

1. Whenever the printer is free and at least one computer is about to send data to print, the printer will eventually be busy;

Example of a program

Before writing a program (driver) that ensures the interaction of each computer with a printer, one needs to formulate the requirements to this program.

1. Whenever the printer is free and at least one computer is about to send data to print, the printer will eventually be busy;
2. Whenever the printer is busy, it must start printing sometime

Example of a program

Before writing a program (driver) that ensures the interaction of each computer with a printer, one needs to formulate the requirements to this program.

1. Whenever the printer is free and at least one computer is about to send data to print, the printer will eventually be busy;
2. Whenever the printer is busy, it must start printing sometime
3. The computer that has finished printing must free the printer sometime;

Example of a program

Before writing a program (driver) that ensures the interaction of each computer with a printer, one needs to formulate the requirements to this program.

1. Whenever the printer is free and at least one computer is about to send data to print, the printer will eventually be busy;
2. Whenever the printer is busy, it must start printing sometime
3. The computer that has finished printing must free the printer sometime;
4. Data to the printer is always sent by no more than one computer.

Example of a program

To communicate with the printer, the programmer suggested to supply each computer with the same program

π : **while** true **do**
wait (R=free);
R:=busy;
output(X,*printer*);
R:=free
od

Example of a program

To communicate with the printer, the programmer suggested to supply each computer with the same program

π : **while** true **do**
wait (R=free);
R:=busy;
output(X,*printer*);
R:=free
od

This program seems both simple and reasonable.
But will the system of computers equipped with this program behave in accordance with the specified requirements?

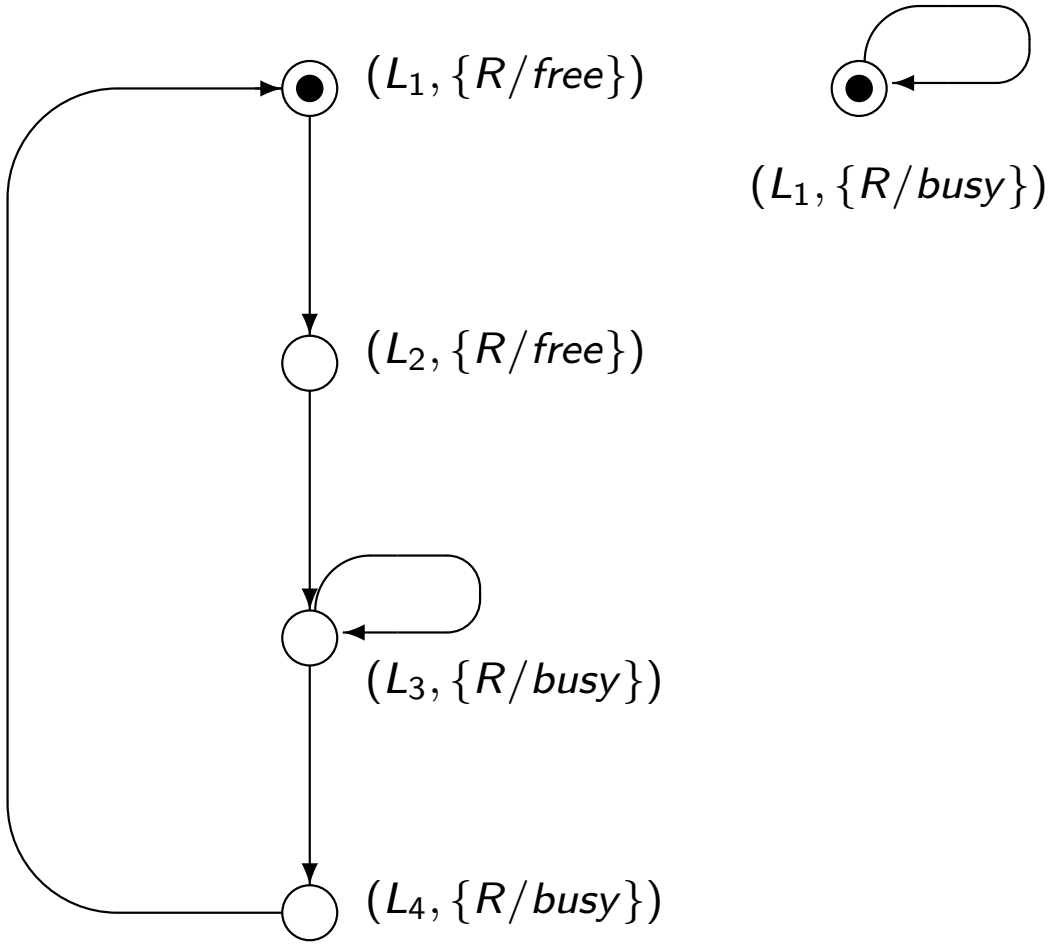
Пример программы

Consider a parallel composition of these programs

cobegin
  π' : **while** true
    do wait (R=free); R:=busy; **skip**; R:=free; **od**
  $\|$
  π'' : **while** true
    do wait (R=free); R:=busy; **skip**; R:=free; **od**
coend

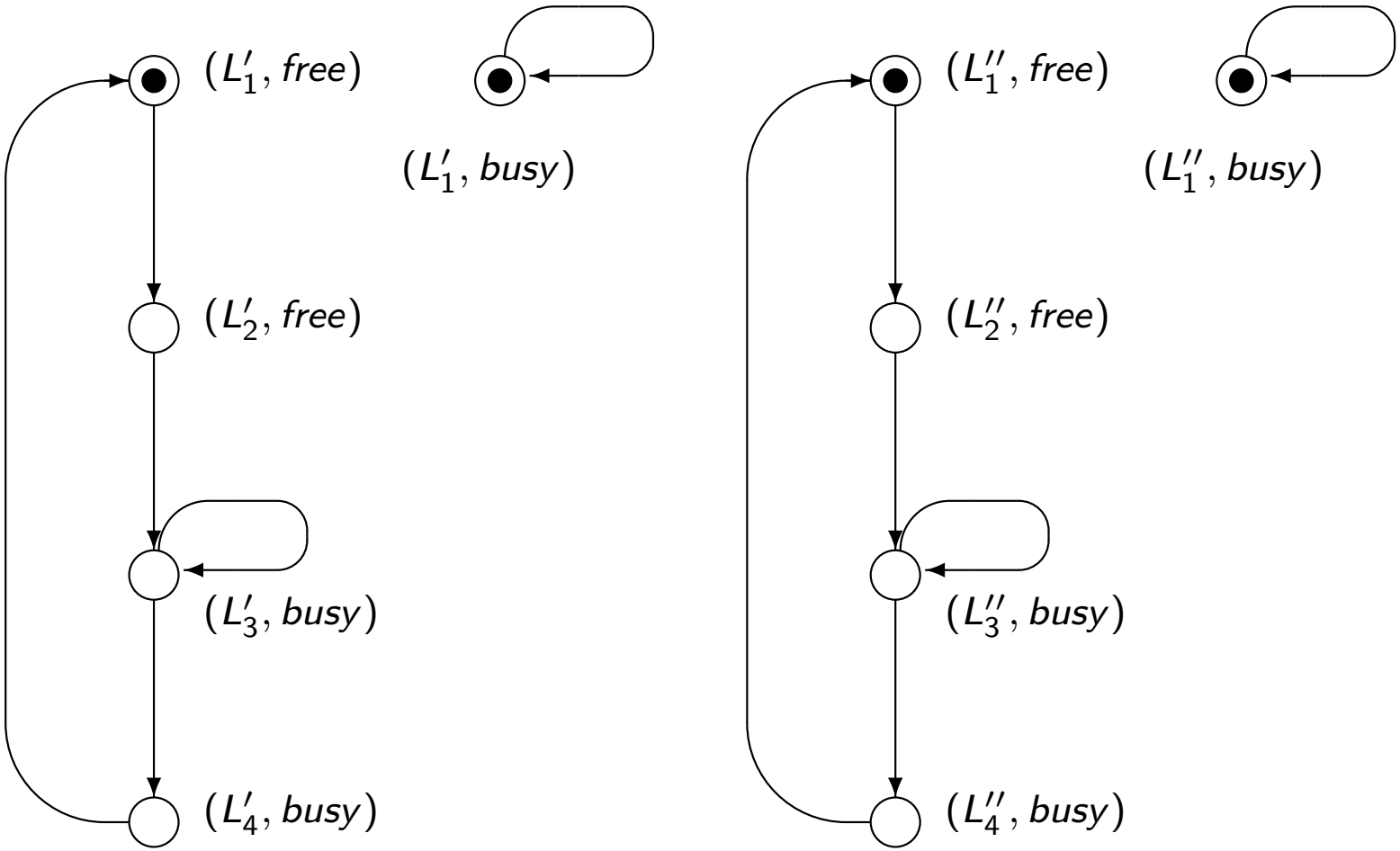
Example of a program

Kripke model for the program π'



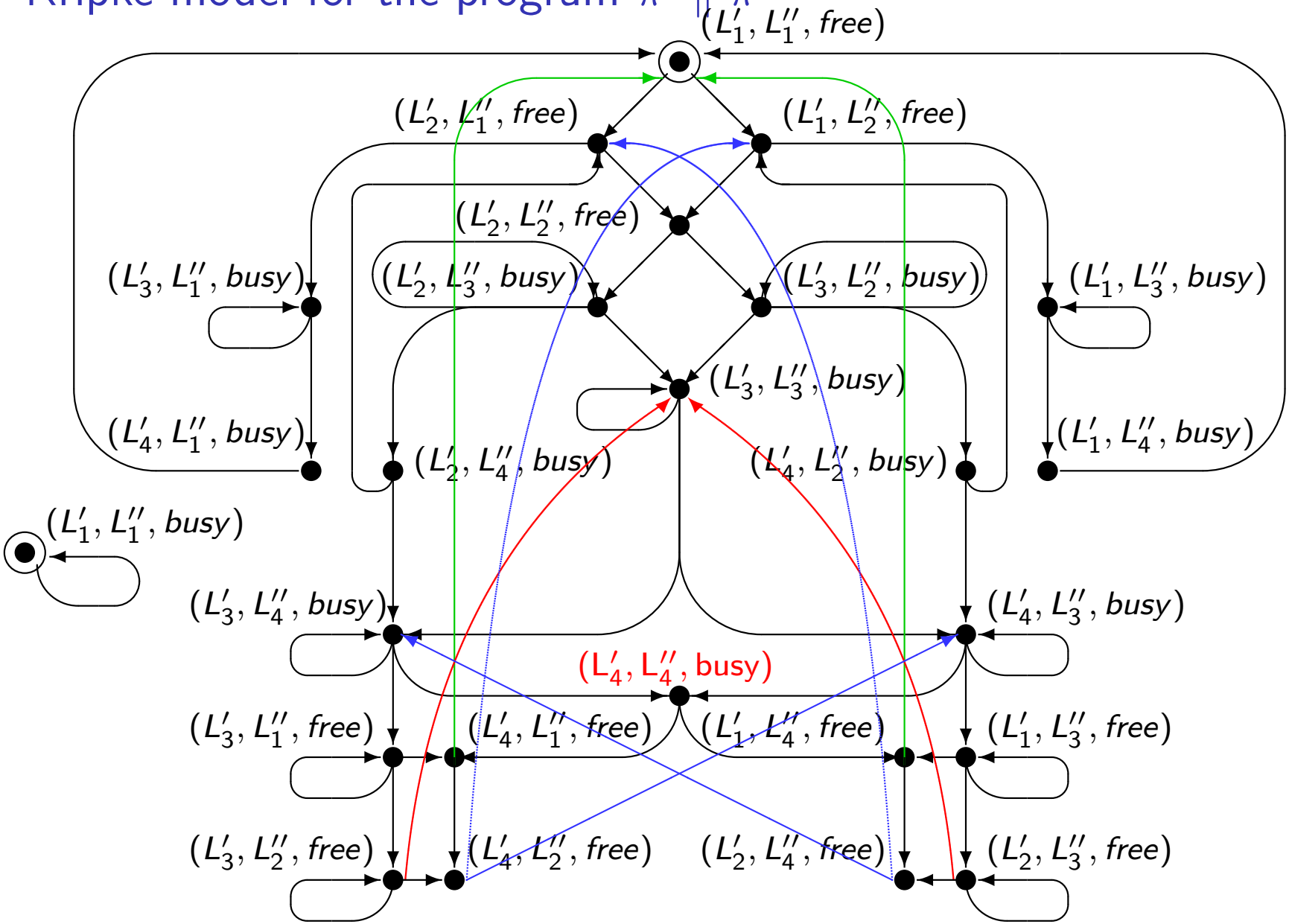
Example of a program

Kripke model for the programs π' and π''



Example of a program

Kripke model for the program $\pi' \parallel \pi''$



Peterson's Algorithm

Look at the pareallel composition of programs

cobegin
π_1 : **while** true
do $\langle$ b_1:=true; x:=2$\rangle$;
wait (x=1 $\vee$ $\neg b_2$); **skip**; b_1:=false;
od
$\|$
π_2 : **while** true
do $\langle$ b_2:=true; x:=1$\rangle$;
wait (x=2 $\vee$ $\neg b_1$); **skip**; b_2:=false;
od
coend

Peterson's Algorithm

Look at the pareallel composition of programs

cobegin
π_1 : **while** true
do $\langle$ b_1:=true; x:=2$\rangle$;
wait (x=1 $\vee$ $\neg b_2$); **skip**; b_1:=false;
od
$\|$
π_2 : **while** true
do $\langle$ b_2:=true; x:=1$\rangle$;
wait (x=2 $\vee$ $\neg b_1$); **skip**; b_2:=false;
od
coend

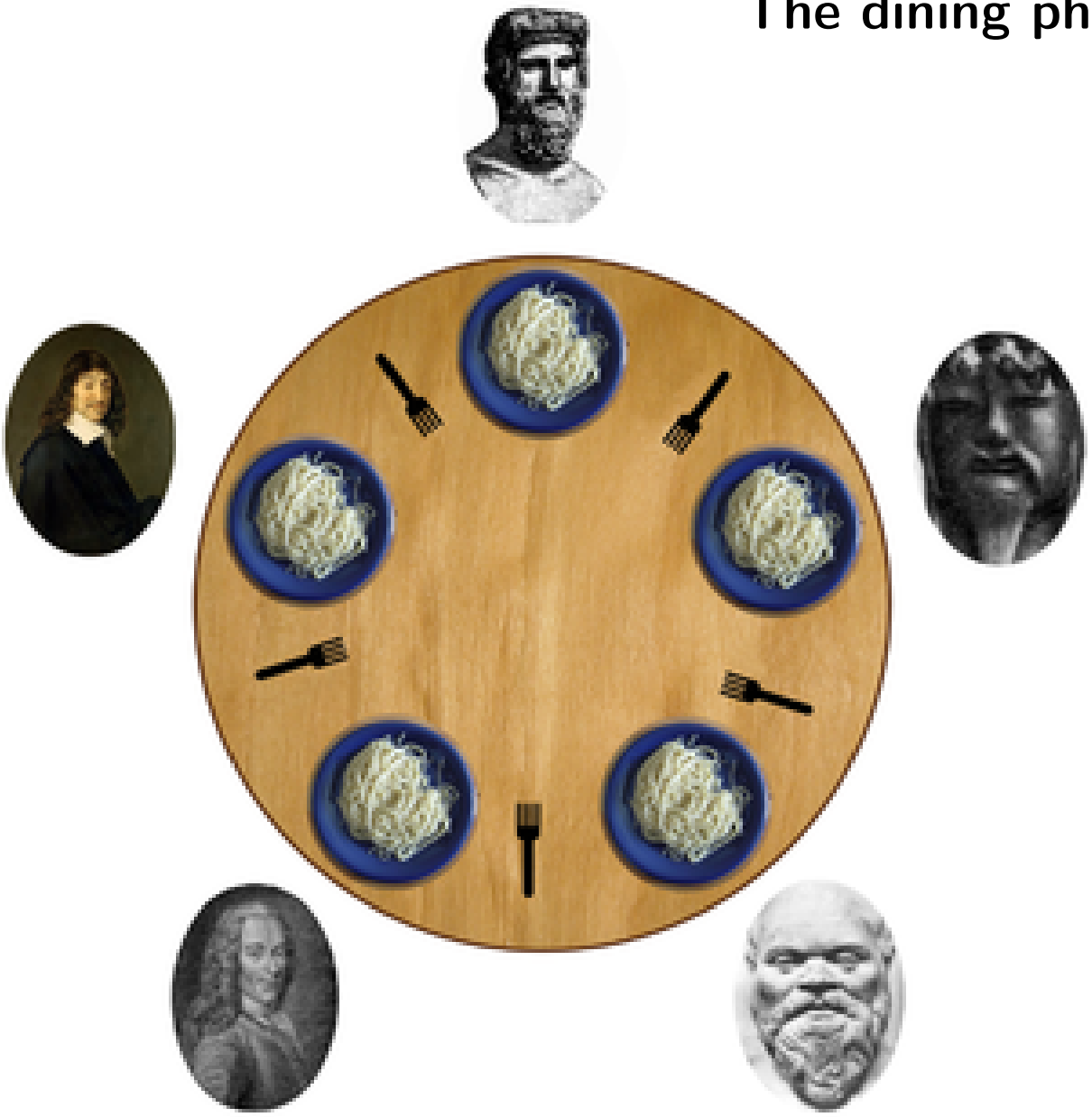
Is it possible that two processes enter simultaneously into the critical section?

Example of a program

How to formulate the correctness requirements?

Example of a program

The dining philosophers problem



The dining philosophers problem

Five silent philosophers are seated around a round table.
Each has a plate of spaghetti in front of them.
Forks lie between every pair of neighbors.
Every philosopher can either eat or think.
A philosopher can only eat when he holds two forks — taken from the right and from the left.
Each philosopher can take the nearest fork (if available), or put down - if he is already holding it.
Taking each fork and returning it to the table are separate actions that must be performed one after the other.

The dining philosophers problem

Five silent philosophers are seated around a round table.
Each has a plate of spaghetti in front of them.
Forks lie between every pair of neighbors.
Every philosopher can either eat or think.
A philosopher can only eat when he holds two forks — taken from the right and from the left.
Each philosopher can take the nearest fork (if available), or put down - if he is already holding it.
Taking each fork and returning it to the table are separate actions that must be performed one after the other.

The essence of the problem: to develop a model of behavior (parallel algorithm) in which none of the philosophers will starve, that is, they will forever alternate between eating and thinking.

END OF LECTURE 3.