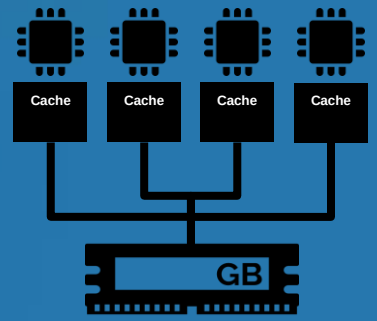
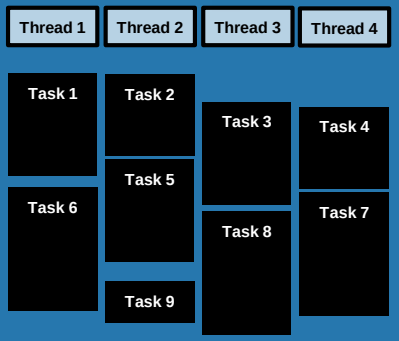
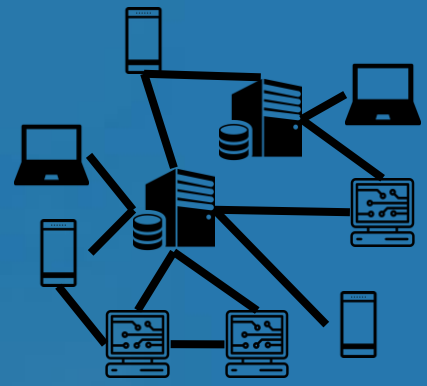
Parameterized Verification 2022

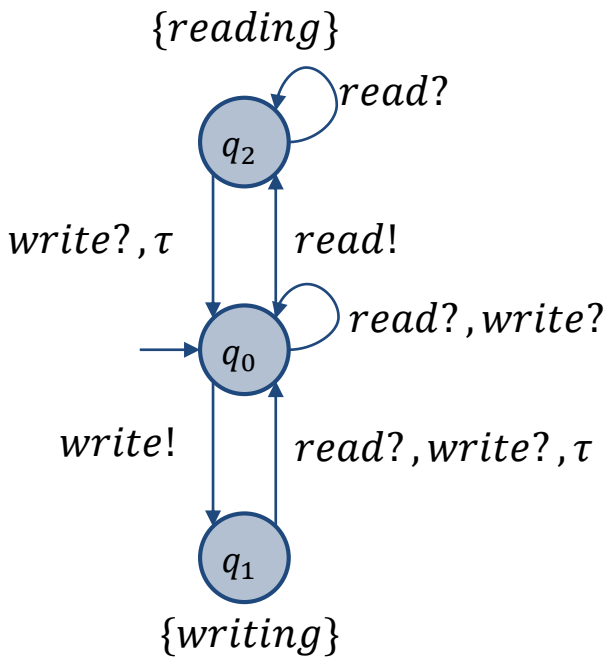
Lecture 5







Example: Simple Reader-Writer in PR System?



Safety properties of interest:

- never two (or more) processes in a $writing$ state at the same time
- never a process in a $reading$ state if there is also a process in a $writing$ state

First try: replace BC actions with PR actions

Satisfies both properties for system with two processes

But not with three or more:

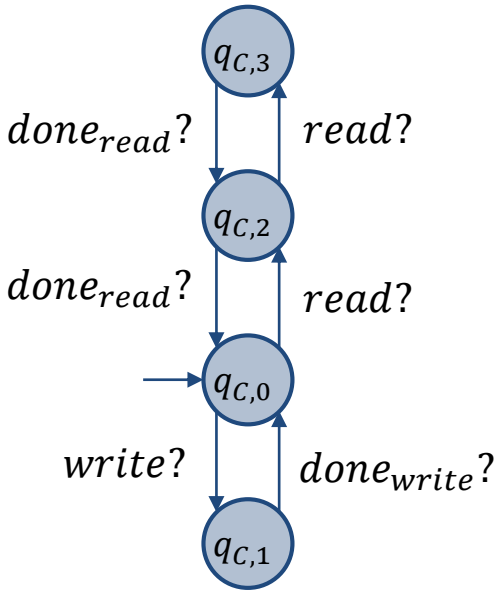
if there is a process p in q_1 and another process p' wants to move from q_0 to q_1, then p' can synchronize with p, but it can also synchronize with any other process in the system, which results in both p and p' being in q_1 at the same time

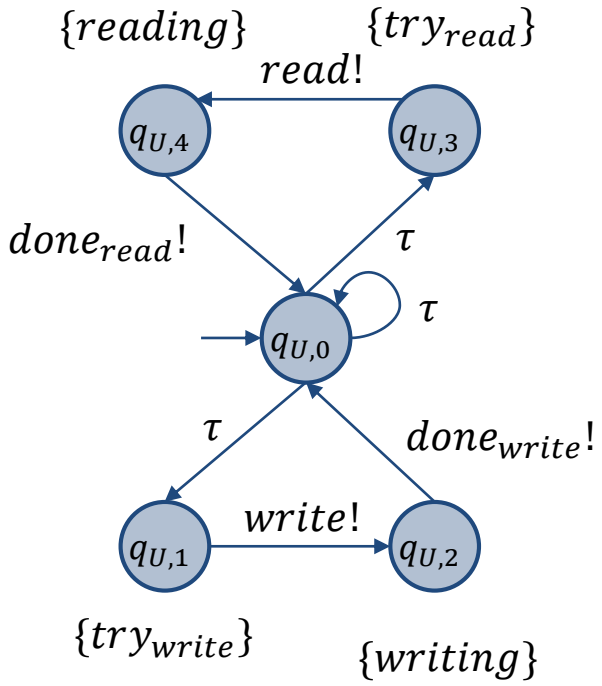
Advanced Reader-Writer in PR System with Controller Process

Suppose we want to guarantee an additional liveness property

"every process that wants to write will eventually be in a writing state",

and similarly for reading. To model this, add states that signal "wants to write" and "wants to read":





Then we can express our properties as

"every process that enters a try_{write} state will eventually enter a $writing$ state",

and similarly for $reading$

Vector Addition System with States

A vector addition system with states (VASS) is a tuple (Q, q_0, A, T), where

- Q is a finite set of states
- $q_0 \in Q$ is an initial state
- $A \subset \mathbb{Z}^m$ is a finite set of actions
- $T \subseteq Q \times A \times Q$ is a transition relation

We call a tuple $(c, q) \in \mathbb{N}_0^m \times Q$ a configuration of the VASS, and $c \in \mathbb{N}_0^m$ the configuration vector.

Then the configuration space of a VASS (Q, q_0, A, T) is the (possibly infinite-state) transition system $(\mathbb{N}_0^m \times Q, (\vec{0}, q_0), A, \Delta)$, where

Δ contains a transition $(c, q) \xrightarrow{v} (c', q')$ if $c' = c + v$ and $(q, v, q') \in T$ (where $c' \in \mathbb{N}_0^m$)

Representing PR Systems as VASSs

Consider the PR system with controller process $C = (Q_C, q_{C,0}, \Sigma, \delta_C, \lambda_C)$ and user processes $U = (Q_U, q_{U,0}, \Sigma, \delta_U, \lambda_U)$, and let $m = |Q_U|$.

Let $u_i \in \mathbb{Z}^m$ be the unit vector with $u_i(i) = 1$, let $v_{i,j} = -u_i + u_j$ and $v_{i,j,k,l} = -u_i + u_j - u_k + u_l$.

Idea: use a VASS to represent the parameterized PR system, similar to the counter representation for BC protocols.

The PR system can be represented as the VASS with $A \subset \mathbb{Z}^m$, $Q = Q_C \cup \{q_{init}\}$, $q_0 = q_{init}$, and

$$A = \{\vec{0}\} \cup \{u_0\} \cup \{v_{i,j}\}_{q_i,q_j \in Q_U} \cup \{v_{i,j,k,l}\}_{q_i,q_j,q_k,q_l \in Q_U},$$

with T defined on the next slide.

Checking Safety Properties of PR Systems

Consider a PR system with controller process $C = (Q_C, q_{C,0}, \Sigma, \delta_C, \lambda_C)$, where $\lambda_C: Q_C \to 2^{AP}$, and user processes $U = (Q_U, q_{U,0}, \Sigma, \delta_U, \lambda_U)$.

To check safety properties of the controller process:

1. let $AP' = AP \cup \{init\}$ for a fresh atomic proposition $init$, and extend λ_C to $Q_C \cup \{q_{init}\}$ by letting $\lambda'_C(q_{init}) = \{init\}$ and $\lambda'_C(q_C) = \lambda_C(q_C)$ for $q_C \in Q_C$
2. as seen before, encode desired property into an automaton $A = (Q_A, q_{A,0}, \Sigma_A, \delta_A, F_A)$ with $\Sigma_A = 2^{AP'}$ that reads state labels of controller and accepts all paths that violate the property. For this automaton, let $(q_{A,0}, \{init\}, q_{A,0})$ be the only transition on $\{init\}$ in δ_A
3. instead of the VASS $(\{q_{init}\} \cup Q_C, q_{init}, A, T)$, consider the VASS $\left((\{q_{init}\} \cup Q_C) \times Q_A, (q_{init}, q_{A,0}), A, \Delta\right)$
   where $((q_C, q_A), v, (q'_C, q'_A)) \in \Delta$ if $(q_C, v, q'_C) \in T$ and $(q_A, \lambda'_C(q_C), q'_A) \in \delta_A$
4. check for reachability of a configuration $(c, (q_C, q_A))$ where $q_A \in F_A$

Checking Safety Properties of PR Systems (IV)

Why did we change the construction such that U starts in q_{init} instead of C?

Because otherwise the automaton would read the state label of $q_{U,0}$ in every step of the initialization phase. We want to start it reading state labels (other than $init$) only when initialization is over.

In a similar way as seen on the last slides, we can check safety properties of the controller process and any fixed number of user processes by

- defining an autmaton that reads the labels of multiple processes
- including all of these processes explicitly in the state space Q of the VASS (such that they stay in a special initialization state until the initialization is over)

Checking Liveness Properties of PR Systems

Let $C \parallel U^n$ be the PR system with controller process C and n user processes U

Let f be a liveness property of C and $A_{\neg f}$ an automaton that accepts all runs that do not satisfy f

Let $VASS(C, U^i, A_{\neg f})$ be the configuration space of the VASS constructed from C, U and $A_{\neg f}$, with i explicit copies of U. We call a configuration of this VASS an accepting configuration if it contains an accepting state of $A_{\neg f}$

Lemma: There is an $n \in \mathbb{N}$ and a run of $C \parallel U^n$ that does not satisfy f
iff $VASS(C, U^0, A_{\neg f})$ has an infinite run that contains infinitely many accepting configurations.

Proof idea:
$\Rightarrow$: follows from the ability of the VASS to „load“ a system of size n and correctly simulate the run

$\Leftarrow$: follows from the fact that the VASS has to „load“ a system of some size n and then correctly simulates a run (and from the properties of $A_{\neg f}$)

Checking Liveness Properties of PR Systems (III)

Lemma: $VASS(C, U^i, A)$ has an infinite run with infinitely many accepting configurations iff it has a finite path of the form $\alpha\beta$ with the following properties:

1. α starts in the initial configuration
2. β is a cycle, i.e., it starts and ends in the same configuration
3. a final configuration appears in β

Proof idea: follows from the fact that there are only finitely many accepting states of A, and, even though $VASS(C, U^i, A)$ is an infinite-state system, after the initialization it simulates a system of fixed size n. Therefore, any infinite run must contain repetitions, which implies that we can find a run with the desired properties.

Checking Liveness Properties of PR Systems (IV)

Let $p = \left|(Q_C \cup \{q_{init}\}) \times (Q_U)^i \times Q_A\right|$ and $m = |Q_U|$.

Lemma: There exists a finite path of $VASS\left(C, U^i, A\right)$ of the form $\alpha\beta$ with the properties from the preceding Lemma iff there exists such a path of length at most $2^{k \cdot p \cdot \log(p) \cdot 2^{k \cdot m \cdot \log(m)}}$, for some constant k.

Theorem: The PMCP for PR systems with finite-state processes and properties defined as Büchi automata is decidable (in time exponential in $|Q_C|$ and doubly exponential in $|A|$ and $|Q_U|$).

Corollary: There exists a cutoff for the PMCP of PR systems where U, C and A are of bounded size.

(or, alternatively: the cutoff depends on the size of U, C and A)

Preview and Learning Goals for this Lecture

Topics:

- (Un)Decidability of Verification Problems
- Undecidability of PMCP for Liveness Properties of BC Protocols

Techniques:

- Reduction of Halting Problem for Two Counter Machines (2CMs) to a PMCP
- Simulating 2CMs (up to some limitations) in BC protocols

Background:
(Un)Decidability of Verification Problems

Decision Problems, Checking Safety/Reachability

A decision problem is a yes-or-no question, defined over an infinite set of inputs I.

Examples: Is $n \in \mathbb{N}$ a prime number? Does a given program terminate on input $i \in I$?

A decision problem is decidable if there exists an algorithm that can compute, for any input $i \in I$, whether the correct answer is yes or no.

Reachability of $R \subseteq Q$ in an LTS defines a decision problem:

- the set of inputs is the set of all LTSs and subsets $R \subseteq Q$
- the answer is yes if and only if R is reachable in the given LTS.

Reachability of R in an LTS is decidable. It can be checked by explicit search (e.g., DFS or BFS) over the states of the LTS.

Proofs of Decidability and Undecidability

To prove decidability of a decision problem, we have to show that there exists an algorithm that always produces the correct answer (and is guaranteed to terminate). Such an algorithm is called a decision procedure for this problem.

In many cases, a proof of decidability also allows to bound the complexity of the decision problem. I.e., we are not only guaranteed to get a correct answer, but for any given input we can bound the time needed to find the answer.

A decision problem is undecidable if there exists no algorithm that always computes the correct answer.

A proof of undecidability usually is a proof by contradiction, in many times by reducing a problem that is known to be undecidable to the given decision problem.

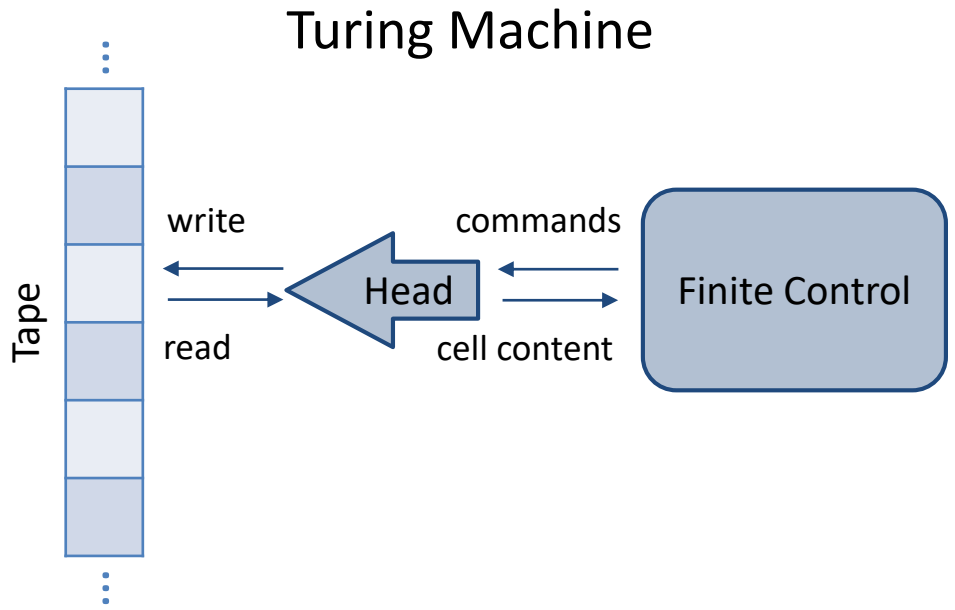
A decision problem is semi-decidable if there exists a semi-algorithm (i.e., an algorithm that is not guaranteed to terminate) that always gives the correct answer if it terminates, and is guaranteed to terminate whenever the correct answer is “yes”. Such a semi-algorithm is called a semi-decision procedure for this problem.

Undecidability of the Halting Problem

Halting Problem (General Form):
Given the description of an arbitrary program and a finite input, decide whether the program will terminate or not.

Halting Problem (Turing Machines):

Given a Turing machine with arbitrary tape contents, decide whether the Turing Machine will terminate or not.



Theorem [Turing, 1936]:

The halting problem for Turing Machines is undecidable.

Proof idea: suppose we have an algorithm that always gives the right answer. Knowing the algorithm, we can construct a Turing machine and an input such that the machine halts, but the algorithm predicts it will not halt.

A Simpler Model: Two-Counter Machines

A two-counter machine (2CM) is a tuple (Q, q_0, A, δ), where

- Q is a finite set of states
- $q_0 \in Q$ is the initial state
- $q_h \in Q$ is the halting state
- $A = \{inc(c_i), dec(c_i), zero(c_i) \mid i \in \{1,2\}\}$ is the set of actions
- $\delta \subseteq Q \times A \times Q$ is the set of commands

A configuration of a 2CM is a tuple $(q, v_1, v_2) \in Q \times \mathbb{N}_0 \times \mathbb{N}_0$. It is a halting configuration if $q = q_h$.

The configuration graph of a 2CM (Q, q_0, A, δ) is the (infinite-state) transition system $(Q \times \mathbb{N}_0 \times \mathbb{N}_0, (q_0, 0,0), A, \Delta)$, where $\big((q, v_1, v_2), a, (q', v_1', v_2')\big) \in \Delta$ if $(q, a, q') \in \delta$ and

- if $a = inc(c_i)$ then $v_i' = v_i + 1$ and $v_j' = v_j$ for $j \neq i$
- if $a = dec(c_i)$ then $v_i > 0, v_i' = v_i - 1$ and $v_j' = v_j$ for $j \neq i$
- if $a = zero(c_i)$ then $v_i = v_i' = 0$ and $v_j' = v_j$ for $j \neq i$

Undecidability of Halting Problem for 2CMs

We say that a 2CM halts if a halting configuration is reachable in its configuration graph.

The halting problem for 2CMs is to decide, given a 2CM, whether it will halt.

Theorem [Minsky 1967]:

The halting problem for 2CMs is undecidable.

Minsky in fact proved undecidability of the halting problem for a slightly different machine model, also called Minsky machines. The main differences to our model are:

- our model allows non-determinism, Minsky machines are deterministic
- our model allows deadlocks (i.e., non-halting configurations without outgoing transitions), Minsky machines do not

Any Minsky machine can be translated into a 2CM, such that the Theorem above follows from Minsky's original undecidability proof.

Undecidability of Verification Problems

Note: since the configuration graph of a 2CM is a transition system, the halting problem for 2CMs can be seen as a model checking problem (over an infinite state space), where the specification is reachability of a halting configuration.

Therefore, we can use the result of Minsky to prove undecidability of verification problems if the halting problem for 2CMs can be reduced to solving the verification problem.

To this end, we need to show that

- we can simulate the behavior of arbitrary 2CMs in our system model, and
- we can express the reachability of halting configurations in our specification language.

Because of their simpler structure, such a proof is much easier for 2CMs than e.g. for Turing machines.

Example: Undecidability of Reachability in Infinite-State Systems/Programs

Consider programs that compute on unbounded integers, and allow (at least) the following operations:

- addition and subtraction of constants
- branching based on $if - then - else$, including non-deterministic branching
- comparison of integer variables against 0 (i.e., $v = 0$ and $v > 0$)
- loops (or jumps to program locations)

To show that this class of programs can simulate 2CMs, we need to show that for every 2CM there is a program s.t.

- there is a mapping from configurations of the 2CM to states of the program
- transitions in the configuration graph of the 2CM can be simulated in the program s.t. this mapping is respected (in particular, conditions and effects of actions $inc(c_i), dec(c_i), zero(c_i)$ can be simulated)

The mapping in this case is straightforward - for every 2CM, a program with two integer variables v_1 and v_2 is sufficient, with the obvious variable valuations. Transitions are simulated in the following way:

- $inc(c_i)$ and $dec(c_i)$ are simulated by addition or subtraction of constant 1 from the corresponding integer variable, with an additional check for $v_i > 0$ in case of subtraction
- $zero(c_i)$ is simulated by an $if - then - else$ construct with a check for $v_i = 0$, that goes to the successor state in the positive case and to a special sink state in the negative case

Example: Undecidability of Reachability in Infinite-State Systems/Programs

Consider programs that compute on unbounded integers, and allow (at least) the following operations:

- addition and subtraction of constants
- branching based on $if - then - else$, including non-deterministic branching
- comparison of integer variables against 0 (i.e., $v = 0$ and $v > 0$)
- loops (or jumps to program locations)

To show that reachability of a halting configuration can be expressed in our specification language:

- model a halting configuration of the 2CM as a $return$ statement of the program
- then the halting problem for the 2CM is equivalent to checking reachability of a $return$ statement

Thus, if we could decide whether a return statement is reachable for arbitrary programs from this class, then we could also decide the halting problem for 2CMs. This is in contradiction to the result of Minsky.

Undecidability of Parameterized Model Checking Problems

Instead of proving undecidability of model checking or halting problems, we want to consider PMCPs.

To this end, we want to reduce the question “does an arbitrary 2CM reach a halting configuration?” to the question “does a property hold for all instances of a parameterized system?”.

Note that we can decide this PMCP if and only if we can decide the dual problem:

> “does there exist an instance of a parameterized system for which the property does not hold?”

Looking at this dual problem, we can see why, even though every instance of a parameterized system is finite-state, we may be able to reduce the halting problem of 2CMs to the PMCP:

- the 2CM halts iff there is a finite run in the configuration space of the 2CM that ends in a halting configuration
- on this path, the 2CM visits finitely many configurations, therefore counter values on the path are bounded
- thus, the halting path can possibly be simulated by an instance of the parameterized system, even if that instance only supports a finite number of counter values
- since we consider all instances, we may be able to construct parameterized systems such that whatever counter value is necessary for a halting run, there is an instance of the system that supports it

Undecidability of Parameterized Model Checking Problems (II)

That is, undecidability proofs for PMCPs are similar to undecidability proofs for other verification problems, but are usually parameterized in a bound on the counter values.

I.e., for any given bound on the counter values we have to show that there exists an instance of the parameterized system such that:

1. there is a mapping from configurations of the bounded 2CM to states of this instance
2. transitions in the configuration graph of the bounded 2CM are faithfully simulated in this instance (in particular, conditions and effects of the actions $inc(c_i), dec(c_i), zero(c_i)$)

In addition, we need to be able to express unreachability of a halting state in the 2CM as a property of our parameterized system.

In the following , we show how this can be done for BC protocols with liveness properties.

Undecidability of PMCP for Liveness Properties of BC Protocols

Undecidability of the PMCP for Liveness Properties of BC Protocols

Theorem [Esparza et al. 1999]:

The PMCP for liveness properties of BC protocols is undecidable.

Proof idea:

By reduction of the halting problem for 2CMs to the PMCP.
Outline:

1. given a 2CM, define a BC protocol such that, for any fixed bound on counter values, there exists an instance such that:
   a. there is a mapping between configurations of the 2CM and states of the instance
   b. transitions in the configuration graph of the 2CM are (up to some exceptions) faithfully simulated in this instance, as long as the counter values do not exceed the bound
2. define a liveness property such that the property is violated in some instance of the BC protocol iff the 2CM can reach a halting configuration

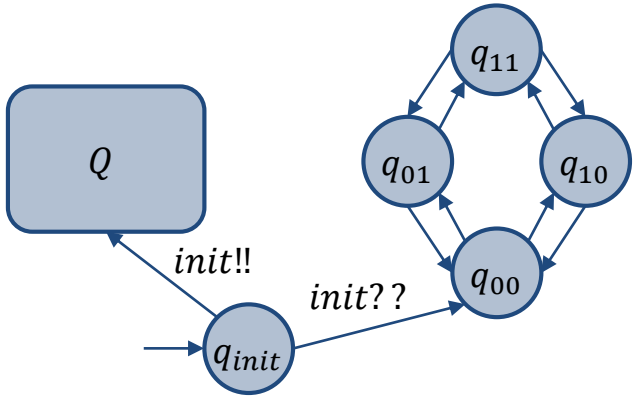
Mapping a Bounded 2CM to an Instance of a BC Protocol

A configuration (q, v_1, v_2) of a 2CM consists of

- a state $q \in Q$
- values $v_1, v_2 \in \mathbb{N}_0$ of the counters

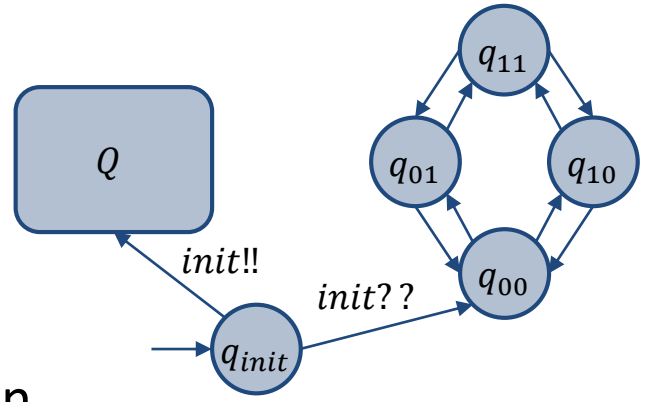
Idea: model finite state space Q explicitly and simulate it
as the state of one process, and model the value of each counter
by a unary encoding with one Boolean variable in each other process

I.e., the local state space of a process roughly looks like this:
(we will see that we need some auxiliary states for the simulation)



Mapping a Bounded 2CM to an Instance of a BC Protocol (II)

The local state space of a process roughly looks like this:



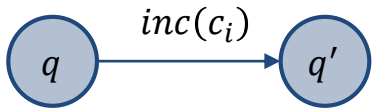
Then, there is a mapping of configurations of a 2CM with bound k on counter values to states of an instance of this BC protocol with $k+1$ processes:

Note that after the initial action $init$, exactly one process (the control process) is in the part Q of the state space, all others (the storage processes) are in q_{00}, q_{01}, q_{10}, or q_{11}.

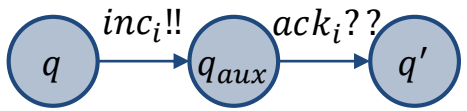
A configuration (q, v_1, v_2) of the 2CM corresponds to any state of this instance where the control process is in state q, there are v_1 storage processes in q_{10} or q_{11}, and v_2 storage processes in q_{01} or q_{11}.

Since we have k storage processes, every configuration with counter values up to k has at least one corresponding state in the instance of the BC protocol.

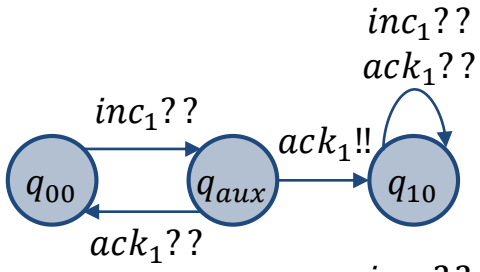
Simulating Increment and Decrement Actions

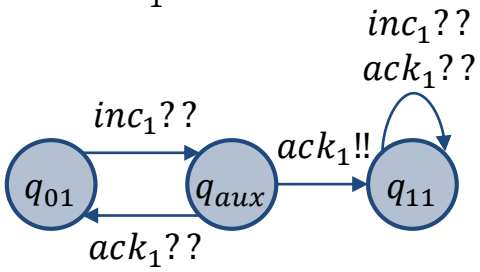


transition of the 2CM



corresponding transitions of the control process

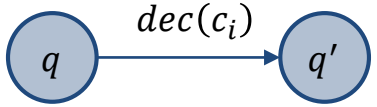


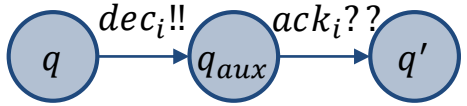


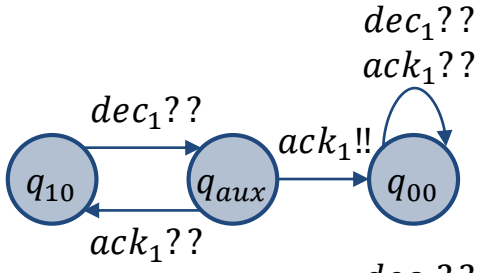
corresponding transitions of storage processes (shown for $i = 1$)

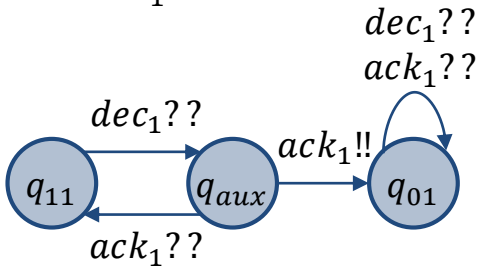
each state named q_{aux} is a fresh auxiliary state

note: can lead to deadlocks if no storage process can go to q_{aux}



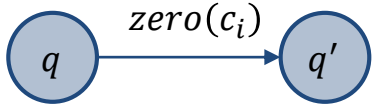




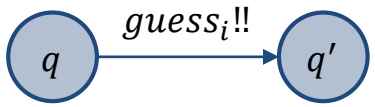


if we started from corresponding states, then after execution of the transitions in both systems, we are again in corresponding states

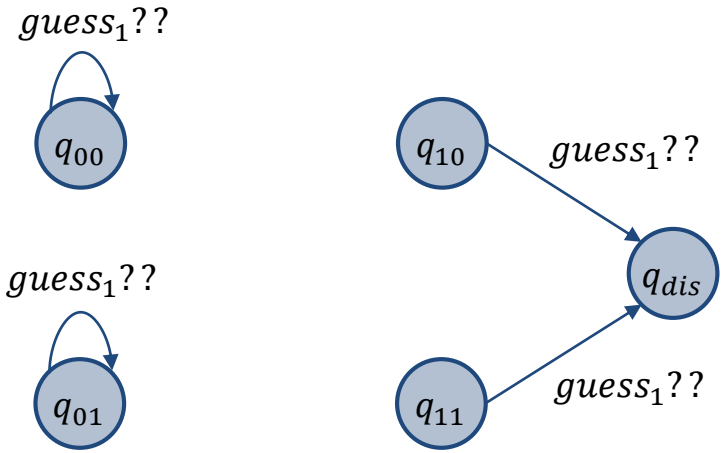
Simulating Test for Zero



transition of the 2CM



corresponding transitions of the control process



corresponding transitions of storage processes (shown for $i = 1$)

Test for zero cannot be simulated faithfully:

- that would require **all** storage processes to send a message to the control process
- but the control process can never know when it has received all messages

Therefore, we let the control process guess whether a counter is currently zero, and all processes that have a non-zero value go into state q_{dis} where they are disabled for the rest of the run.

If we started in corresponding states, then after execution of the transitions in both systems, we are either in corresponding states or at least one storage process has moved into q_{dis}.

How to Build a Proof based on Semi-Faithful Tests for Zero

Since the controller does not get feedback from storage processes in case the guess is wrong, the parameterized system instance may enter a state that corresponds to a halting configuration of the 2CM even if this configuration is not reachable in the 2CM:

$$(q_0, 0,0) \xrightarrow{inc(c_1)} (q_1, 1,0) \xrightarrow{zero(c_1)} (q_h, 1,0)$$

I.e., our simulation may reach q_h even though it is not reachable in the 2CM, but to do so it needs to "sacrifice" one or more processes.

Therefore, we can achieve the desired goal by considering

- a modified system that moves back to the initial state after reaching a state that corresponds to a halting configuration
- the liveness property

  "the system does not visit a state that corresponds to a halting configuration infinitely often"

How to Build a Proof based on Semi-Faithful Tests for Zero (II)

We consider

- a modified system that moves back to the initial state after reaching a state that corresponds to a halting configuration
- the liveness property

"the system does not visit a state that corresponds to a halting configuration infinitely often"

Claim:
there is a path of the parameterized system that violates the property iff there is a halting run of the 2CM

Proof:
⇒: If the property is violated on a run, then there must be at least one global state that is visited infinitely often.
We can conclude that all guesses after the first occurrence of this state have been correct.
Thus, there must be a path from the initial configuration to a halting configuration in the 2CM.

⇐: On the other hand, if there exists a path from the initial configuration to a halting configuration in the 2CM, then we can construct an infinite run of an instance of the parameterized system that visits a corresponding state infinitely often, violating the property.

Summary: Undecidability of the PMCP for Liveness Properties of BC Protocols

We have shown that the PMCP for liveness properties of BC protocols is undecidable by

1. defining a BC protocol such that any run of the 2CM with bounded counter values is simulated faithfully, except for guesses in zero tests
2. modifying the BC protocol such that it resets to the initial state whenever it reaches a state that corresponds to a halting configuration of the 2CM, except for processes that have been disabled due to incorrect guesses
3. defining a liveness property such that the property is violated in some instance of the modified BC protocol iff the 2CM can reach a halting configuration

Now, if we could decide the PMCP for liveness properties of BC protocols, then we could decide the halting problem for 2CMs. I.e., we have reduced the (undecidable) halting problem for 2CMs to our PMCP, thus showing that this PMCP must be undecidable.

Why are Safety Properties not Sufficient for Undecidability?

Note that the given proof does not work if we restrict specifications to safety properties. This is due to several restrictions of the model and the specifications we consider:

- for the zero test, the controller needs to know whether any storage process is in a non-zero state (for a given counter), but this cannot be directly communicated
- also, knowledge about zero-valued counters cannot be encoded into the specification, e.g., by requiring that $guess_i$ only happens if $v_i = 0$, because our specifications only support properties of a fixed number of processes
- if we consider the modified system and safety properties, then
  - reachability of a state that corresponds to a halting configuration for a fixed number of times can always be violated by a run including some number of wrong guesses
  - if we want to detect wrong guesses, this can neither be communicated nor be excluded through the specification, for the same reasons as for the zero test itself

Today: Summary

- (Un)Decidability of Verification Problems (proof by reduction to halting problem of 2CMs)
- Undecidability of PMCP for Liveness Properties of BC Protocols (how to simulate a 2CM)

Next time: guarded protocols, i.e., systems that communicate via global transition guards