Formal Verification of Software

Sabine Broda

Department of Computer Science/FCUP

12 de Novembro de 2014

Formal Verification of Software

- Area of computer science that studies and applies mathematical methods for proving the correctness of software systems/programs with respect to a formal specification or property.
- The dissemination and importance of the role of information systems in essential sectors of society is continually increasing, demanding more and more for the certification/guarantee of their reliability.
- This is particularly crucial when *critical systems* are concerned, such as traffic control, nuclear or medical equipment.
- Some facts:
  - the cost of maintaining software is about 66% of its total cost;
  - a software specification error is about 20 times more costly to repair if detected after production than before.

Formal Verification of Software

- Area of computer science that studies and applies mathematical methods for proving the correctness of software systems/programs with respect to a formal specification or property.
- The dissemination and importance of the role of information systems in essential sectors of society is continually increasing, demanding more and more for the certification/guarantee of their reliability.
- This is particularly crucial when *critical systems* are concerned, such as traffic control, nuclear or medical equipment.
- Some facts:
  - the cost of maintaining software is about 66% of its total cost;
  - a software specification error is about 20 times more costly to repair if detected after production than before.

Formal Verification of Software

- Area of computer science that studies and applies mathematical methods for proving the correctness of software systems/programs with respect to a formal specification or property.
- The dissemination and importance of the role of information systems in essential sectors of society is continually increasing, demanding more and more for the certification/guarantee of their reliability.
- This is particularly crucial when *critical systems* are concerned, such as traffic control, nuclear or medical equipment.
- Some facts:
  - the cost of maintaining software is about 66% of its total cost;
  - a software specification error is about 20 times more costly to repair if detected after production than before.

Formal Verification of Software

- Area of computer science that studies and applies mathematical methods for proving the correctness of software systems/programs with respect to a formal specification or property.
- The dissemination and importance of the role of information systems in essential sectors of society is continually increasing, demanding more and more for the certification/guarantee of their reliability.
- This is particularly crucial when *critical systems* are concerned, such as traffic control, nuclear or medical equipment.
- Some facts:
  - the cost of maintaining software is about 66% of its total cost;
  - a software specification error is about 20 times more costly to repair if detected after production than before.

Formal Verification of Software

- Area of computer science that studies and applies mathematical methods for proving the correctness of software systems/programs with respect to a formal specification or property.
- The dissemination and importance of the role of information systems in essential sectors of society is continually increasing, demanding more and more for the certification/guarantee of their reliability.
- This is particularly crucial when *critical systems* are concerned, such as traffic control, nuclear or medical equipment.
- Some facts:
    - the cost of maintaining software is about 66% of its total cost;
    - a software specification error is about 20 times more costly to repair if detected after production than before.

Formal Verification of Software

- Area of computer science that studies and applies mathematical methods for proving the correctness of software systems/programs with respect to a formal specification or property.
- The dissemination and importance of the role of information systems in essential sectors of society is continually increasing, demanding more and more for the certification/guarantee of their reliability.
- This is particularly crucial when *critical systems* are concerned, such as traffic control, nuclear or medical equipment.
- Some facts:
  - the cost of maintaining software is about 66% of its total cost;
  - a software specification error is about 20 times more costly to repair if detected after production than before.

Formal Verification of Software

- Area of computer science that studies and applies mathematical methods for proving the correctness of software systems/programs with respect to a formal specification or property.
- The dissemination and importance of the role of information systems in essential sectors of society is continually increasing, demanding more and more for the certification/guarantee of their reliability.
- This is particularly crucial when *critical systems* are concerned, such as traffic control, nuclear or medical equipment.
- Some facts:
    - the cost of maintaining software is about 66% of its total cost;
    - a software specification error is about 20 times more costly to repair if detected after production than before.

In this talk

- Illustration of two standard approaches to formal verification by example:
    - Model checking;
    - Deductive program verification.
- Recent research results on the use of several extensions of Kleene algebra to verification:
    - KAT (Kleene Algebra with Tests);
    - SKA(T) (Synchronous Kleene Algebra with and without Tests).

In this talk

- Illustration of two standard approaches to formal verification by example:
  - Model checking;
  - Deductive program verification.
- Recent research results on the use of several extensions of Kleene algebra to verification:
  - KAT (Kleene Algebra with Tests);
  - SKA(T) (Synchronous Kleene Algebra with and without Tests).

In this talk

- Illustration of two standard approaches to formal verification by example:
    - Model checking;
    - Deductive program verification.
- Recent research results on the use of several extensions of Kleene algebra to verification:
    - KAT (Kleene Algebra with Tests);
    - SKA(T) (Synchronous Kleene Algebra with and without Tests).

In this talk

- Illustration of two standard approaches to formal verification by example:
    - Model checking;
    - Deductive program verification.
- Recent research results on the use of several extensions of Kleene algebra to verification:
    - KAT (Kleene Algebra with Tests);
    - SKA(T) (Synchronous Kleene Algebra with and without Tests).

In this talk

- Illustration of two standard approaches to formal verification by example:
    - Model checking;
    - Deductive program verification.
- Recent research results on the use of several extensions of Kleene algebra to verification:
    - KAT (Kleene Algebra with Tests);
    - SKA(T) (Synchronous Kleene Algebra with and without Tests).

Two (of many) different approaches

Model Checking

- technique for the verification of (finite-state) reactive/concurrent systems;
- systems are represented by a transition system (the model);
- specifications/properties are expressed by formulae of temporal logic (LTL/CTL);
- a model checker (efficient symbolic algorithm) decides if the specification is true in the model;
- if a property is not valid, a counterexample is exhibited;
- in general this method is automatic for finite models;
- major drawback: state space explosion.

Two (of many) different approaches

Model Checking

- technique for the verification of (finite-state) reactive/concurrent systems;
- systems are represented by a transition system (the model);
- specifications/properties are expressed by formulae of temporal logic (LTL/CTL);
- a model checker (efficient symbolic algorithm) decides if the specification is true in the model;
- if a property is not valid, a counterexample is exhibited;
- in general this method is automatic for finite models;
- major drawback: state space explosion.

Two (of many) different approaches

Model Checking

- technique for the verification of (finite-state) reactive/concurrent systems;
- systems are represented by a transition system (the model);
- specifications/properties are expressed by formulae of temporal logic (LTL/CTL);
- a model checker (efficient symbolic algorithm) decides if the specification is true in the model;
- if a property is not valid, a counterexample is exhibited;
- in general this method is automatic for finite models;
- major drawback: state space explosion.

Two (of many) different approaches

Model Checking

- technique for the verification of (finite-state) reactive/concurrent systems;
- systems are represented by a transition system (the model);
- specifications/properties are expressed by formulae of temporal logic (LTL/CTL);
- a model checker (efficient symbolic algorithm) decides if the specification is true in the model;
- if a property is not valid, a counterexample is exhibited;
- in general this method is automatic for finite models;
- major drawback: state space explosion.

Two (of many) different approaches

Model Checking

- technique for the verification of (finite-state) reactive/concurrent systems;
- systems are represented by a transition system (the model);
- specifications/properties are expressed by formulae of temporal logic (LTL/CTL);
- a model checker (efficient symbolic algorithm) decides if the specification is true in the model;
- if a property is not valid, a counterexample is exhibited;
- in general this method is automatic for finite models;
- major drawback: state space explosion.

Two (of many) different approaches

Model Checking

- technique for the verification of (finite-state) reactive/concurrent systems;
- systems are represented by a transition system (the model);
- specifications/properties are expressed by formulae of temporal logic (LTL/CTL);
- a model checker (efficient symbolic algorithm) decides if the specification is true in the model;
- if a property is not valid, a counterexample is exhibited;
- in general this method is automatic for finite models;
- major drawback: state space explosion.

Two (of many) different approaches

Model Checking

- technique for the verification of (finite-state) reactive/concurrent systems;
- systems are represented by a transition system (the model);
- specifications/properties are expressed by formulae of temporal logic (LTL/CTL);
- a model checker (efficient symbolic algorithm) decides if the specification is true in the model;
- if a property is not valid, a counterexample is exhibited;
- in general this method is automatic for finite models;
- major drawback: state space explosion.

Two (of many) different approaches

Model Checking

- technique for the verification of (finite-state) reactive/concurrent systems;
- systems are represented by a transition system (the model);
- specifications/properties are expressed by formulae of temporal logic (LTL/CTL);
- a model checker (efficient symbolic algorithm) decides if the specification is true in the model;
- if a property is not valid, a counterexample is exhibited;
- in general this method is automatic for finite models;
- major drawback: state space explosion.

Two (of many) different approaches

Model Checking

- technique for the verification of (finite-state) reactive/concurrent systems;
- systems are represented by a transition system (the model);
- specifications/properties are expressed by formulae of temporal logic (LTL/CTL);
- a model checker (efficient symbolic algorithm) decides if the specification is true in the model;
- if a property is not valid, a counterexample is exhibited;
- in general this method is automatic for finite models;
- major drawback: state space explosion.

Two (of many) different approaches

(Deductive) Program Verification

- Based on Hoare Logic, introduced by C.A.R. Hoare in 1969 for reasoning about the correctness of imperative, sequential programs;
- Deductive system that can be used to assert the correctness of a program with respect to a given specification by constructing a derivation.
- Rules in the inference system can be applied if some side-conditions are satisfied (proof obligations) that have to be checked by some automatic theorem prover or some proof assistent (automatic/semi-automatic).

Two (of many) different approaches

(Deductive) Program Verification

- Based on Hoare Logic, introduced by C.A.R. Hoare in 1969 for reasoning about the correctness of imperative, sequential programs;
- Deductive system that can be used to assert the correctness of a program with respect to a given specification by constructing a derivation.
- Rules in the inference system can be applied if some side-conditions are satisfied (proof obligations) that have to be checked by some automatic theorem prover or some proof assistent (automatic/semi-automatic).

Two (of many) different approaches

(Deductive) Program Verification

- Based on Hoare Logic, introduced by C.A.R. Hoare in 1969 for reasoning about the correctness of imperative, sequential programs;
- Deductive system that can be used to assert the correctness of a program with respect to a given specification by constructing a derivation.
- Rules in the inference system can be applied if some side-conditions are satisfied (proof obligations) that have to be checked by some automatic theorem prover or some proof assistent (automatic/semi-automatic).

Two (of many) different approaches

(Deductive) Program Verification

- Based on Hoare Logic, introduced by C.A.R. Hoare in 1969 for reasoning about the correctness of imperative, sequential programs;
- Deductive system that can be used to assert the correctness of a program with respect to a given specification by constructing a derivation.
- Rules in the inference system can be applied if some side-conditions are satisfied (proof obligations) that have to be checked by some automatic theorem prover or some proof assistent (automatic/semi-automatic).

Two (of many) different approaches

(Deductive) Program Verification

- Based on Hoare Logic, introduced by C.A.R. Hoare in 1969 for reasoning about the correctness of imperative, sequential programs;
- Deductive system that can be used to assert the correctness of a program with respect to a given specification by constructing a derivation.
- Rules in the inference system can be applied if some side-conditions are satisfied (proof obligations) that have to be checked by some automatic theorem prover or some proof assistent (automatic/semi-automatic).

Model Checking: an Example

Mutual Exclusion

When concurrent processes share a resource (such as a file on a disk or a database entry), it may be necessary to ensure that they do not have access to it at the same time.

For this one has to identify certain *critical sections* of each process' code and arrange that only one process can be in its critical section at a time.

We will design a model ($\mathcal{M}$) of a system with two processes and specify several properties to be satisfied.

Model Checking: an Example

Mutual Exclusion

When concurrent processes share a resource (such as a file on a disk or a database entry), it may be necessary to ensure that they do not have access to it at the same time.

For this one has to identify certain *critical sections* of each process' code and arrange that only one process can be in its critical section at a time.

We will design a model ($\mathcal{M}$) of a system with two processes and specify several properties to be satisfied.

Model Checking: an Example

Mutual Exclusion

When concurrent processes share a resource (such as a file on a disk or a database entry), it may be necessary to ensure that they do not have access to it at the same time.

For this one has to identify certain *critical sections* of each process' code and arrange that only one process can be in its critical section at a time.

We will design a model ($\mathcal{M}$) of a system with two processes and specify several properties to be satisfied.

Model Checking: an Example

Mutual Exclusion

When concurrent processes share a resource (such as a file on a disk or a database entry), it may be necessary to ensure that they do not have access to it at the same time.

For this one has to identify certain *critical sections* of each process' code and arrange that only one process can be in its critical section at a time.

We will design a model ($\mathcal{M}$) of a system with two processes and specify several properties to be satisfied.

An Example: Mutual Exclusion

Each process i might :

- n_i be in it's non-critical state
- t_i trying to enter it's critical state, or
- c_i be in it's critical state

Each individual process undergoes transitions in the cycle $n_i \rightarrow t_i \rightarrow c_i \rightarrow n_i \rightarrow \cdots$, but the two processes interleave with each other.

The two processes start off in their non-critical sections (global state s_0).

An Example: Mutual Exclusion

Each process i might :

- n_i be in it's non-critical state
- t_i trying to enter it's critical state, or
- c_i be in it's critical state

Each individual process undergoes transitions in the cycle $n_i \rightarrow t_i \rightarrow c_i \rightarrow n_i \rightarrow \cdots$, but the two processes interleave with each other.

The two processes start off in their non-critical sections (global state s_0).

An Example: Mutual Exclusion

Each process i might :

- n_i be in it's non-critical state
- t_i trying to enter it's critical state, or
- c_i be in it's critical state

Each individual process undergoes transitions in the cycle $n_i \rightarrow t_i \rightarrow c_i \rightarrow n_i \rightarrow \cdots$, but the two processes interleave with each other.

The two processes start off in their non-critical sections (global state s_0).

An Example: Mutual Exclusion

Each process i might :

- n_i be in it's non-critical state
- t_i trying to enter it's critical state, or
- c_i be in it's critical state

Each individual process undergoes transitions in the cycle $n_i \rightarrow t_i \rightarrow c_i \rightarrow n_i \rightarrow \cdots$, but the two processes interleave with each other.

The two processes start off in their non-critical sections (global state s_0).

An Example: Mutual Exclusion

Each process i might :

n_i be in it's non-critical state

t_i trying to enter it's critical state, or

c_i be in it's critical state

Each individual process undergoes transitions in the cycle $n_i \rightarrow t_i \rightarrow c_i \rightarrow n_i \rightarrow \cdots$, but the two processes interleave with each other.

The two processes start off in their non-critical sections (global state s_0).

An Example: Mutual Exclusion

Each process *i* might :

- n_i be in it's non-critical state
- t_i trying to enter it's critical state, or
- c_i be in it's critical state

Each individual process undergoes transitions in the cycle $n_i \rightarrow t_i \rightarrow c_i \rightarrow n_i \rightarrow \cdots$, but the two processes interleave with each other.

The two processes start off in their non-critical sections (global state s_0).

An Example: Mutual Exclusion

Each process *i* might :

n_i be in it's non-critical state

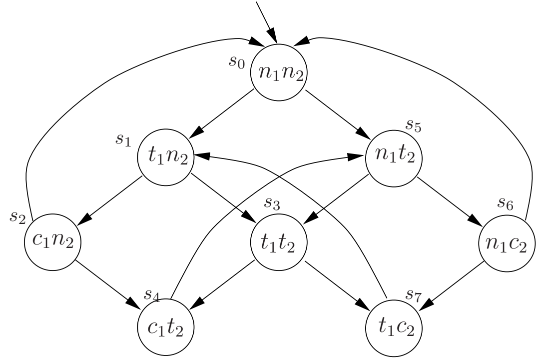
t_i trying to enter it's critical state, or

c_i be in it's critical state

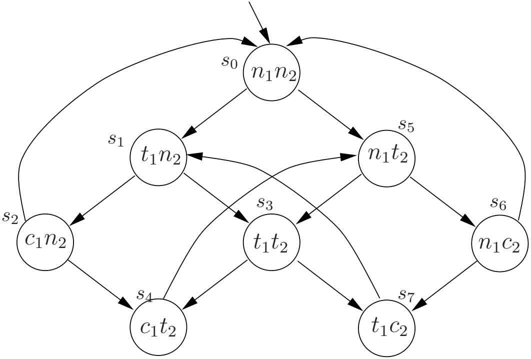
Each individual process undergoes transitions in the cycle $n_i \rightarrow t_i \rightarrow c_i \rightarrow n_i \rightarrow \cdots$, but the two processes interleave with each other.

The two processes start off in their non-critical sections (global state s_0).

A first-attempt model for mutual exclusion



A first-attempt model for mutual exclusion



Two expected properties expressed as LTL-formulae

Safety Only one process is in its critical section at any time.
$\varphi : G\neg(c_1 \ \wedge \ c_2)$

Liveness Whenever any process requests to enter its critical section, it will eventually be permitted to do so.
$\psi : G(t_1 \rightarrow Fc_1) \wedge G(t_2 \rightarrow Fc_2)$

Here G and F are temporal conectives of LTL (Linear Temporal Logic) such that,

- $\mathcal{M}, s_0 \models G\theta$ iff for every execution path starting in s_0 the formula θ is globally true (i.e. in all states);
- $\mathcal{M}, s_0 \models F\theta$ iff for every execution path starting in s_0 the formula θ is sometime in the future true (i.e. in some state).

Two expected properties expressed as LTL-formulae

Safety Only one process is in its critical section at any time.
$\varphi : G\neg(c_1 \wedge c_2)$

Liveness Whenever any process requests to enter its critical section, it will eventually be permitted to do so.
$\psi : G(t_1 \rightarrow Fc_1) \wedge G(t_2 \rightarrow Fc_2)$

Here G and F are temporal conectives of LTL (Linear Temporal Logic) such that,

- $\mathcal{M}, s_0 \models G\theta$ iff for every execution path starting in s_0 the formula θ is globally true (i.e. in all states);
- $\mathcal{M}, s_0 \models F\theta$ iff for every execution path starting in s_0 the formula θ is sometime in the future true (i.e. in some state).

Two expected properties expressed as LTL-formulae

Safety Only one process is in its critical section at any time.
$\varphi : G\neg(c_1 \wedge c_2)$

Liveness Whenever any process requests to enter its critical section, it will eventually be permitted to do so.
$\psi : G(t_1 \rightarrow Fc_1) \wedge G(t_2 \rightarrow Fc_2)$

Here G and F are temporal conectives of LTL (Linear Temporal Logic) such that,

- $\mathcal{M}, s_0 \models G\theta$ iff for every execution path starting in s_0 the formula θ is globally true (i.e. in all states);
- $\mathcal{M}, s_0 \models F\theta$ iff for every execution path starting in s_0 the formula θ is sometime in the future true (i.e. in some state).

Two expected properties expressed as LTL-formulae

Safety	Only one process is in its critical section at any time. $\varphi : G\neg(c_1 \ \wedge \ c_2)$
Liveness	Whenever any process requests to enter its critical section, it will eventually be permitted to do so. $\psi : G(t_1 \rightarrow Fc_1) \wedge G(t_2 \rightarrow Fc_2)$

Here G and F are temporal conectives of LTL (Linear Temporal Logic) such that,

- $\mathcal{M}, s_0 \models G\theta$ iff for every execution path starting in s_0 the formula θ is globally true (i.e. in all states);
- $\mathcal{M}, s_0 \models F\theta$ iff for every execution path starting in s_0 the formula θ is sometime in the future true (i.e. in some state).

Two expected properties expressed as LTL-formulae

Safety Only one process is in its critical section at any time.
$\varphi : G\neg(c_1 \ \wedge \ c_2)$

Liveness Whenever any process requests to enter its critical section, it will eventually be permitted to do so.
$\psi : G(t_1 \rightarrow Fc_1) \wedge G(t_2 \rightarrow Fc_2)$

Here G and F are temporal conectives of LTL (Linear Temporal Logic) such that,

- $\mathcal{M}, s_0 \models G\theta$ iff for every execution path starting in s_0 the formula θ is globally true (i.e. in all states);
- $\mathcal{M}, s_0 \models F\theta$ iff for every execution path starting in s_0 the formula θ is sometime in the future true (i.e. in some state).

Two expected properties expressed as LTL-formulae

Safety	Only one process is in its critical section at any time. $\varphi : G\neg(c_1 \wedge c_2)$
Liveness	Whenever any process requests to enter its critical section, it will eventually be permitted to do so. $\psi : G(t_1 \rightarrow Fc_1) \wedge G(t_2 \rightarrow Fc_2)$

Here G and F are temporal conectives of LTL (Linear Temporal Logic) such that,

- $\mathcal{M}, s_0 \models G\theta$ iff for every execution path starting in s_0 the formula θ is globally true (i.e. in all states);
- $\mathcal{M}, s_0 \models F\theta$ iff for every execution path starting in s_0 the formula θ is sometime in the future true (i.e. in some state).

Two expected properties expressed as LTL-formulae

Safety Only one process is in its critical section at any time.
$\varphi : G\neg(c_1 \ \wedge \ c_2)$

Liveness Whenever any process requests to enter its critical section, it will eventually be permitted to do so.
$\psi : G(t_1 \rightarrow Fc_1) \wedge G(t_2 \rightarrow Fc_2)$

Here G and F are temporal conectives of LTL (Linear Temporal Logic) such that,

- $\mathcal{M}, s_0 \models G\theta$ iff for every execution path starting in s_0 the formula θ is globally true (i.e. in all states);
- $\mathcal{M}, s_0 \models F\theta$ iff for every execution path starting in s_0 the formula θ is sometime in the future true (i.e. in some state).

Two expected properties expressed as LTL-formulae

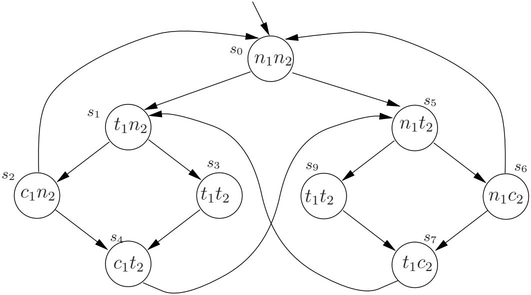
Safety	Only one process is in its critical section at any time. $\varphi : G\neg(c_1 \ \wedge \ c_2)$
Liveness	Whenever any process requests to enter its critical section, it will eventually be permitted to do so. $\psi : G(t_1 \rightarrow Fc_1) \wedge G(t_2 \rightarrow Fc_2)$

Here G and F are temporal conectives of LTL (Linear Temporal Logic) such that,

- $\mathcal{M}, s_0 \models G\theta$ iff for every execution path starting in s_0 the formula θ is globally true (i.e. in all states);
- $\mathcal{M}, s_0 \models F\theta$ iff for every execution path starting in s_0 the formula θ is sometime in the future true (i.e. in some state).

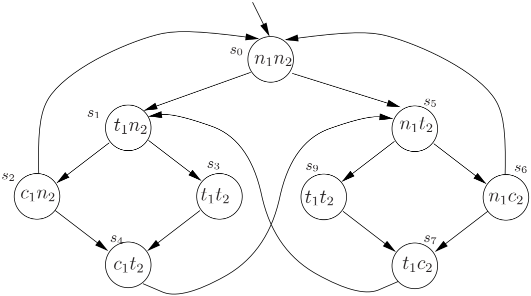
A second model satisfying both properties

In order for *liveness* to be true it is sufficient to divide the state labelled with $t_1 t_2$ into two different states:



A second model satisfying both properties

In order for *liveness* to be true it is sufficient to divide the state labelled with $t_1 t_2$ into two different states:



A third property expressed as a CTL-formula

Non-blocking A process can always request to enter its critical section.

$$AG(n_1 \rightarrow EFt_1) \wedge AG(n_2 \rightarrow EFt_2)$$

Here AG and EF are temporal conectives of CTL (Computation Tree Logic) such that,

- $\mathcal{M}, s \models$ AG θ iff for **A**ll computation paths beginning in s the property θ holds **G**lobally (i.e. in all states);
- $\mathcal{M}, s \models$ EF θ iff there **E**xists a computation paths beginning in s such that θ holds in some **F**uture state.

Note: this property cannot be expressed in LTL!

A third property expressed as a CTL-formula

Non-blocking A process can always request to enter its critical section.

$$\mathrm{AG}(n_1 \rightarrow \mathrm{EF}t_1) \wedge \mathrm{AG}(n_2 \rightarrow \mathrm{EF}t_2)$$

Here AG and EF are temporal conectives of CTL (Computation Tree Logic) such that,

- $\mathcal{M}, s \models \mathrm{AG}\ \theta$ iff for **A**ll computation paths beginning in s the property θ holds **G**lobally (i.e. in all states);
- $\mathcal{M}, s \models \mathrm{EF}\ \theta$ iff there **E**xists a computation paths beginning in s such that θ holds in some **F**uture state.

Note: this property cannot be expressed in LTL!

A third property expressed as a CTL-formula

Non-blocking A process can always request to enter its critical section.

$$\mathrm{AG}(n_1 \to \mathrm{EF}t_1) \wedge \mathrm{AG}(n_2 \to \mathrm{EF}t_2)$$

Here AG and EF are temporal conectives of CTL (Computation Tree Logic) such that,

- $\mathcal{M}, s \models \mathrm{AG}\,\theta$ iff for **A**ll computation paths beginning in s the property θ holds **G**lobally (i.e. in all states);
- $\mathcal{M}, s \models \mathrm{EF}\,\theta$ iff there **E**xists a computation paths beginning in s such that θ holds in some **F**uture state.

Note: this property cannot be expressed in LTL!

A third property expressed as a CTL-formula

Non-blocking A process can always request to enter its critical section.

$$AG(n_1 \to EF t_1) \wedge AG(n_2 \to EF t_2)$$

Here AG and EF are temporal conectives of CTL (Computation Tree Logic) such that,

- $\mathcal{M}, s \models AG\ \theta$ iff for **A**ll computation paths beginning in s the property θ holds **G**lobally (i.e. in all states);
- $\mathcal{M}, s \models EF\ \theta$ iff there **E**xists a computation paths beginning in s such that θ holds in some **F**uture state.

Note: this property cannot be expressed in LTL!

A third property expressed as a CTL-formula

Non-blocking A process can always request to enter its critical section.

$$AG(n_1 \rightarrow EF t_1) \wedge AG(n_2 \rightarrow EF t_2)$$

Here AG and EF are temporal conectives of CTL (Computation Tree Logic) such that,

- $\mathcal{M}, s \models AG\, \theta$ iff for **A**ll computation paths beginning in s the property θ holds **G**lobally (i.e. in all states);
- $\mathcal{M}, s \models EF\, \theta$ iff there **E**xists a computation paths beginning in s such that θ holds in some **F**uture state.

Note: this property cannot be expressed in LTL!

A third property expressed as a CTL-formula

Non-blocking A process can always request to enter its critical section.

$$\mathrm{AG}(n_1 \to \mathrm{EF}t_1) \wedge \mathrm{AG}(n_2 \to \mathrm{EF}t_2)$$

Here AG and EF are temporal conectives of CTL (Computation Tree Logic) such that,

- $\mathcal{M}, s \models \mathrm{AG}\,\theta$ iff for **A**ll computation paths beginning in s the property θ holds **G**lobally (i.e. in all states);
- $\mathcal{M}, s \models \mathrm{EF}\,\theta$ iff there **E**xists a computation paths beginning in s such that θ holds in some **F**uture state.

Note: this property cannot be expressed in LTL!

SMV code for mutual exclusion

```
MODULE main
   VAR
      pr1: process prc(pr2.st, turn, 0);
      pr2: process prc(pr1.st, turn, 1);
      turn: boolean;
   ASSIGN
      init(turn) := 0;
   -- safety
   LTLSPEC  G!((pr1.st = c) & (pr2.st = c))
   -- liveness
   LTLSPEC  G((pr1.st = t) -> F (pr1.st = c))
   LTLSPEC  G((pr2.st = t) -> F (pr2.st = c))
```

SMV code for mutual exclusion

```
MODULE main
   VAR
      pr1: process prc(pr2.st, turn, 0);
      pr2: process prc(pr1.st, turn, 1);
      turn: boolean;
   ASSIGN
      init(turn) := 0;
   -- safety
   LTLSPEC  G!((pr1.st = c) & (pr2.st = c))
   -- liveness
   LTLSPEC  G((pr1.st = t) -> F (pr1.st = c))
   LTLSPEC  G((pr2.st = t) -> F (pr2.st = c))
```

SMV code for mutual exclusion

```
MODULE prc(other-st, turn, myturn)
   VAR
      st: {n, t, c};
   ASSIGN
      init(st) := n;
      next(st) :=
         case
            (st = n)                                    : {t,n};
            (st = t) & (other-st = n)                   : c;
            (st = t) & (other-st = t) & (turn = myturn): c;
            (st = c)                                    : {c,n};
            1                                           : st;
         esac;
      next(turn) :=
         case
            turn = myturn & st = c : !turn;
            1                      : turn;
         esac;
   FAIRNESS running
   FAIRNESS  !(st = c)
```

Model checking algorithms

- There is a variety of model checking tools, such as SMV, Murphy, SPIN, Kronos, Design/CPN, etc.
- But what kind of algorithms and mathematical structures are at the core of these tools?

Model checking algorithms

- There is a variety of model checking tools, such as SMV, Murphy, SPIN, Kronos, Design/CPN, etc.
- But what kind of algorithms and mathematical structures are at the core of these tools?

Model checking algorithms with automata

Problem: Given an LTL-formula ϕ, a model $\mathcal{M}$ and a state s, check if $\mathcal{M}, s \models \phi$.

Approach:

- Construct an automata $\mathcal{A}_{\neg\phi}$ that accepts a computation path π iff $\pi \models \neg\phi$, i.e. $\pi \not\models \phi$.
- Represent $(\mathcal{M}, s)$ by an automata $\mathcal{A}_{\mathcal{M},s}$ (that accepts exactly the computation paths in $\mathcal{M}$ that start in s).
- Check if $\mathcal{L}(\mathcal{A}_{\neg\phi}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{M},s}) = \emptyset$. In this case one has $\mathcal{M}, s \models \phi$, otherwise, every path belonging to the intersection of the two languages can be exhibited as a counter-example.

Model checking algorithms with automata

Problem: Given an LTL-formula ϕ, a model $\mathcal{M}$ and a state s, check if $\mathcal{M}, s \models \phi$.

Approach:

- Construct an automata $\mathcal{A}_{\neg\phi}$ that accepts a computation path π iff $\pi \models \neg\phi$, i.e. $\pi \not\models \phi$.
- Represent $(\mathcal{M}, s)$ by an automata $\mathcal{A}_{\mathcal{M},s}$ (that accepts exactly the computation paths in $\mathcal{M}$ that start in s).
- Check if $\mathcal{L}(\mathcal{A}_{\neg\phi}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{M},s}) = \emptyset$. In this case one has $\mathcal{M}, s \models \phi$, otherwise, every path belonging to the intersection of the two languages can be exhibited as a counter-example.

Model checking algorithms with automata

Problem: Given an LTL-formula ϕ, a model $\mathcal{M}$ and a state s, check if $\mathcal{M}, s \models \phi$.

Approach:

- Construct an automata $\mathcal{A}_{\neg\phi}$ that accepts a computation path π iff $\pi \models \neg\phi$, i.e. $\pi \not\models \phi$.
- Represent $(\mathcal{M}, s)$ by an automata $\mathcal{A}_{\mathcal{M},s}$ (that accepts exactly the computation paths in $\mathcal{M}$ that start in s).
- Check if $\mathcal{L}(\mathcal{A}_{\neg\phi}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{M},s}) = \emptyset$. In this case one has $\mathcal{M}, s \models \phi$, otherwise, every path belonging to the intersection of the two languages can be exhibited as a counter-example.

Model checking algorithms with automata

Problem: Given an LTL-formula ϕ, a model $\mathcal{M}$ and a state s, check if $\mathcal{M}, s \models \phi$.

Approach:

- Construct an automata $\mathcal{A}_{\neg\phi}$ that accepts a computation path π iff $\pi \models \neg\phi$, i.e. $\pi \not\models \phi$.
- Represent $(\mathcal{M}, s)$ by an automata $\mathcal{A}_{\mathcal{M},s}$ (that accepts exactly the computation paths in $\mathcal{M}$ that start in s).
- Check if $\mathcal{L}(\mathcal{A}_{\neg\phi}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{M},s}) = \emptyset$. In this case one has $\mathcal{M}, s \models \phi$, otherwise, every path belonging to the intersection of the two languages can be exhibited as a counter-example.

Model checking algorithms with automata

- Computation paths are represented by infinite sequences of states. Ex.: $\{n_1, n_2\}\{n_1, t_2\}\{t_1, t_2\}\{t_1, c_2\}\{t_1, n_2\}\dots$
- Büchi automata are a type of automata that extends finite automata to process (accept/reject) infinite words (different acceptance criteria).
- In an alternating automaton the transition function is a partial function $\delta : S \times \Sigma \longrightarrow \mathcal{B}^+(S)$, where $\mathcal{B}^+(S)$ is the set of boolean formulas built from elements in S and conectives $\vee$ (representing existential choice) and $\wedge$ (representing universal choice). Ex.: $\delta(s, a) = (s_1 \wedge s_2) \vee (s_3 \wedge s_4)$.
- Alternating Büchi automata have the same expressive power as nondeterministic Büchi automata, but are much more succint (alternating Büchi automaton → exponential blow-up → nondeterministic Büchi automaton).

Model checking algorithms with automata

- Computation paths are represented by infinite sequences of states. Ex.: $\{n_1, n_2\}\{n_1, t_2\}\{t_1, t_2\}\{t_1, c_2\}\{t_1, n_2\}\dots$
- Büchi automata are a type of automata that extends finite automata to process (accept/reject) infinite words (different acceptance criteria).
- In an alternating automaton the transition function is a partial function $\delta : S \times \Sigma \longrightarrow \mathcal{B}^+(S)$, where $\mathcal{B}^+(S)$ is the set of boolean formulas built from elements in S and conectives $\vee$ (representing existential choice) and $\wedge$ (representing universal choice). Ex.: $\delta(s, a) = (s_1 \wedge s_2) \vee (s_3 \wedge s_4)$.
- Alternating Büchi automata have the same expressive power as nondeterministic Büchi automata, but are much more succint (alternating Büchi automaton → exponential blow-up → nondeterministic Büchi automaton).

Model checking algorithms with automata

- Computation paths are represented by infinite sequences of states. Ex.: $\{n_1, n_2\}\{n_1, t_2\}\{t_1, t_2\}\{t_1, c_2\}\{t_1, n_2\} \ldots$
- Büchi automata are a type of automata that extends finite automata to process (accept/reject) infinite words (different acceptance criteria).
- In an alternating automaton the transition function is a partial function $\delta : S \times \Sigma \longrightarrow \mathcal{B}^+(S)$, where $\mathcal{B}^+(S)$ is the set of boolean formulas built from elements in S and conectives $\vee$ (representing existential choice) and $\wedge$ (representing universal choice). Ex.: $\delta(s, a) = (s_1 \wedge s_2) \vee (s_3 \wedge s_4)$.
- Alternating Büchi automata have the same expressive power as nondeterministic Büchi automata, but are much more succint (alternating Büchi automaton → exponential blow-up → nondeterministic Büchi automaton).

Model checking algorithms with automata

- Computation paths are represented by infinite sequences of states. Ex.: $\{n_1, n_2\}\{n_1, t_2\}\{t_1, t_2\}\{t_1, c_2\}\{t_1, n_2\}\dots$
- Büchi automata are a type of automata that extends finite automata to process (accept/reject) infinite words (different acceptance criteria).
- In an alternating automaton the transition function is a partial function $\delta : S \times \Sigma \longrightarrow \mathcal{B}^+(S)$, where $\mathcal{B}^+(S)$ is the set of boolean formulas built from elements in S and conectives $\vee$ (representing existential choice) and $\wedge$ (representing universal choice). Ex.: $\delta(s, a) = (s_1 \wedge s_2) \vee (s_3 \wedge s_4)$.
- Alternating Büchi automata have the same expressive power as nondeterministic Büchi automata, but are much more succint (alternating Büchi automaton → exponential blow-up → nondeterministic Büchi automaton).

Model checking algorithms with automata

- Computation paths are represented by infinite sequences of states. Ex.: $\{n_1, n_2\}\{n_1, t_2\}\{t_1, t_2\}\{t_1, c_2\}\{t_1, n_2\}\ldots$
- Büchi automata are a type of automata that extends finite automata to process (accept/reject) infinite words (different acceptance criteria).
- In an alternating automaton the transition function is a partial function $\delta : S \times \Sigma \longrightarrow \mathcal{B}^+(S)$, where $\mathcal{B}^+(S)$ is the set of boolean formulas built from elements in S and conectives $\vee$ (representing existential choice) and $\wedge$ (representing universal choice). Ex.: $\delta(s, a) = (s_1 \wedge s_2) \vee (s_3 \wedge s_4)$.
- Alternating Büchi automata have the same expressive power as nondeterministic Büchi automata, but are much more succint (alternating Büchi automaton → exponential blow-up → nondeterministic Büchi automaton).

Model checking algorithms with OBDD's

- An OBDD (Ordered Binary Decision Diagram) is a data structure used to represent boolean functions, providing compact representations for sets or relations.
- Given a model $\mathcal{M}$, sets of states of $\mathcal{M}$, as well as the transition relation of $\mathcal{M}$ can be encoded by OBDD's.
- An algorithm for deciding CTL-logic (the labelling algorithm) can operate directly on the encodings (OBDD's).
- SMV programs can be compiled directly into OBDD's without having to go via intermediate representations (bigger size).

Model checking algorithms with OBDD's

- An OBDD (Ordered Binary Decision Diagram) is a data structure used to represent boolean functions, providing compact representations for sets or relations.
- Given a model $\mathcal{M}$, sets of states of $\mathcal{M}$, as well as the transition relation of $\mathcal{M}$ can be encoded by OBDD's.
- An algorithm for deciding CTL-logic (the labelling algorithm) can operate directly on the encodings (OBDD's).
- SMV programs can be compiled directly into OBDD's without having to go via intermediate representations (bigger size).

Model checking algorithms with OBDD's

- An OBDD (Ordered Binary Decision Diagram) is a data structure used to represent boolean functions, providing compact representations for sets or relations.
- Given a model $\mathcal{M}$, sets of states of $\mathcal{M}$, as well as the transition relation of $\mathcal{M}$ can be encoded by OBDD's.
- An algorithm for deciding CTL-logic (the labelling algorithm) can operate directly on the encodings (OBDD's).
- SMV programs can be compiled directly into OBDD's without having to go via intermediate representations (bigger size).

Model checking algorithms with OBDD's

- An OBDD (Ordered Binary Decision Diagram) is a data structure used to represent boolean functions, providing compact representations for sets or relations.
- Given a model $\mathcal{M}$, sets of states of $\mathcal{M}$, as well as the transition relation of $\mathcal{M}$ can be encoded by OBDD's.
- An algorithm for deciding CTL-logic (the labelling algorithm) can operate directly on the encodings (OBDD's).
- SMV programs can be compiled directly into OBDD's without having to go via intermediate representations (bigger size).

Model checking algorithms with OBDD's

- An OBDD (Ordered Binary Decision Diagram) is a data structure used to represent boolean functions, providing compact representations for sets or relations.
- Given a model $\mathcal{M}$, sets of states of $\mathcal{M}$, as well as the transition relation of $\mathcal{M}$ can be encoded by OBDD's.
- An algorithm for deciding CTL-logic (the labelling algorithm) can operate directly on the encodings (OBDD's).
- SMV programs can be compiled directly into OBDD's without having to go via intermediate representations (bigger size).

Verification of sequential programs

- *Hoare logic* is the fundamental formalism (Hoare, 1969) for reasoning about the correctness of imperative programs.
- It builds on first-order logic, dealing with the notion of correctness of a program w.r.t. a given specification.
- The specification consists of a *precondition* and a *postcondition*.
- Correctness of a program is asserted by constructing a derivation in the inference system of Hoare logic.
- While doing so, onde must identify an *invariant* for every loop in the program.
- In the system presented here loop invariants are given beforehand by the programmer as an input to the program verification process (and not invented during the construction of the proof).
- A program can only be proved correct if it is correctly annotated.

Verification of sequential programs

- *Hoare logic* is the fundamental formalism (Hoare, 1969) for reasoning about the correctness of imperative programs.
- It builds on first-order logic, dealing with the notion of correctness of a program w.r.t. a given specification.
- The specification consists of a *precondition* and a *postcondition*.
- Correctness of a program is asserted by constructing a derivation in the inference system of Hoare logic.
- While doing so, onde must identify an *invariant* for every loop in the program.
- In the system presented here loop invariants are given beforehand by the programmer as an input to the program verification process (and not invented during the construction of the proof).
- A program can only be proved correct if it is correctly annotated.

Verification of sequential programs

- *Hoare logic* is the fundamental formalism (Hoare, 1969) for reasoning about the correctness of imperative programs.
- It builds on first-order logic, dealing with the notion of correctness of a program w.r.t. a given specification.
- The specification consists of a *precondition* and a *postcondition*.
- Correctness of a program is asserted by constructing a derivation in the inference system of Hoare logic.
- While doing so, onde must identify an *invariant* for every loop in the program.
- In the system presented here loop invariants are given beforehand by the programmer as an input to the program verification process (and not invented during the construction of the proof).
- A program can only be proved correct if it is correctly annotated.

Verification of sequential programs

- *Hoare logic* is the fundamental formalism (Hoare, 1969) for reasoning about the correctness of imperative programs.
- It builds on first-order logic, dealing with the notion of correctness of a program w.r.t. a given specification.
- The specification consists of a *precondition* and a *postcondition*.
- Correctness of a program is asserted by constructing a derivation in the inference system of Hoare logic.
- While doing so, onde must identify an *invariant* for every loop in the program.
- In the system presented here loop invariants are given beforehand by the programmer as an input to the program verification process (and not invented during the construction of the proof).
- A program can only be proved correct if it is correctly annotated.

Verification of sequential programs

- *Hoare logic* is the fundamental formalism (Hoare, 1969) for reasoning about the correctness of imperative programs.
- It builds on first-order logic, dealing with the notion of correctness of a program w.r.t. a given specification.
- The specification consists of a *precondition* and a *postcondition*.
- Correctness of a program is asserted by constructing a derivation in the inference system of Hoare logic.
- While doing so, onde must identify an *invariant* for every loop in the program.
- In the system presented here loop invariants are given beforehand by the programmer as an input to the program verification process (and not invented during the construction of the proof).
- A program can only be proved correct if it is correctly annotated.

Verification of sequential programs

- *Hoare logic* is the fundamental formalism (Hoare, 1969) for reasoning about the correctness of imperative programs.
- It builds on first-order logic, dealing with the notion of correctness of a program w.r.t. a given specification.
- The specification consists of a *precondition* and a *postcondition*.
- Correctness of a program is asserted by constructing a derivation in the inference system of Hoare logic.
- While doing so, onde must identify an *invariant* for every loop in the program.
- In the system presented here loop invariants are given beforehand by the programmer as an input to the program verification process (and not invented during the construction of the proof).
- A program can only be proved correct if it is correctly annotated.

Verification of sequential programs

- *Hoare logic* is the fundamental formalism (Hoare, 1969) for reasoning about the correctness of imperative programs.
- It builds on first-order logic, dealing with the notion of correctness of a program w.r.t. a given specification.
- The specification consists of a *precondition* and a *postcondition*.
- Correctness of a program is asserted by constructing a derivation in the inference system of Hoare logic.
- While doing so, onde must identify an *invariant* for every loop in the program.
- In the system presented here loop invariants are given beforehand by the programmer as an input to the program verification process (and not invented during the construction of the proof).
- A program can only be proved correct if it is correctly annotated.

Verification of sequential programs

- *Hoare logic* is the fundamental formalism (Hoare, 1969) for reasoning about the correctness of imperative programs.
- It builds on first-order logic, dealing with the notion of correctness of a program w.r.t. a given specification.
- The specification consists of a *precondition* and a *postcondition*.
- Correctness of a program is asserted by constructing a derivation in the inference system of Hoare logic.
- While doing so, onde must identify an *invariant* for every loop in the program.
- In the system presented here loop invariants are given beforehand by the programmer as an input to the program verification process (and not invented during the construction of the proof).
- A program can only be proved correct if it is correctly annotated.

System $\mathcal{H}_g$

$$\frac{}{\{\phi\}\ \mathsf{skip}\ \{\psi\}}\ \text{if} \models \phi \to \psi$$

$$\frac{}{\{\phi\}\ x := E\ \{\psi\}}\ \text{if} \models \phi \to \psi[E/x]$$

$$\frac{\{\phi\}\ C_1\ \{\eta\} \qquad \{\eta\}\ C_2\ \{\psi\}}{\{\phi\}\ C_1;\ C_2\ \{\psi\}}$$

$$\frac{\{\phi\ \wedge\ B\}\ C_1\ \{\psi\} \qquad \{\phi\ \wedge\ \neg B\}\ C_2\ \{\psi\}}{\{\phi\}\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \{\psi\}}$$

$$\frac{\{\eta\ \wedge\ B\}\ C\ \{\eta\}}{\{\psi\}\ \texttt{while}\ B\ \texttt{do}\ \{\eta\} C\ \{\phi\}}\ \text{se} \models \psi \to \eta\ \text{e} \models \eta\ \wedge\ \neg B \to \phi$$

System $\mathcal{H}_g$

$$\frac{}{\{\phi\}\ \mathsf{skip}\ \{\psi\}}\quad \text{if } \models \phi \to \psi$$

$$\frac{}{\{\phi\}\ x := E\ \{\psi\}}\quad \text{if } \models \phi \to \psi[E/x]$$

$$\frac{\{\phi\}\ C_1\ \{\eta\} \qquad \{\eta\}\ C_2\ \{\psi\}}{\{\phi\}\ C_1;\ C_2\ \{\psi\}}$$

$$\frac{\{\phi\ \wedge\ B\}\ C_1\ \{\psi\} \qquad \{\phi\ \wedge\ \neg B\}\ C_2\ \{\psi\}}{\{\phi\}\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \{\psi\}}$$

$$\frac{\{\eta\ \wedge\ B\}\ C\ \{\eta\}}{\{\psi\}\ \texttt{while}\ B\ \mathsf{do}\ \{\eta\}C\ \{\phi\}}\ \text{se } \models \psi \to \eta \text{ e } \models \eta\ \wedge\ \neg B \to \phi$$

System $\mathcal{H}_g$

$$\frac{}{\{\phi\}\,\mathsf{skip}\,\{\psi\}}\ \text{if} \models \phi \rightarrow \psi$$

$$\frac{}{\{\phi\}\, x := E\,\{\psi\}}\ \text{if} \models \phi \rightarrow \psi[E/x]$$

$$\frac{\{\phi\}\ C_1\ \{\eta\} \qquad \{\eta\}\ C_2\ \{\psi\}}{\{\phi\}\ C_1; C_2\ \{\psi\}}$$

$$\frac{\{\phi\ \wedge\ B\}\ C_1\ \{\psi\} \qquad \{\phi\ \wedge\ \neg B\}\ C_2\ \{\psi\}}{\{\phi\}\ \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \{\psi\}}$$

$$\frac{\{\eta\ \wedge\ B\}\ C\ \{\eta\}}{\{\psi\}\ \mathtt{while}\ B\ \mathtt{do}\ \{\eta\} C\ \{\phi\}}\ \text{se} \models \psi \rightarrow \eta \text{ e} \models \eta\ \wedge\ \neg B \rightarrow \phi$$

System $\mathcal{H}_g$

$$\frac{}{\{\phi\}\,\mathsf{skip}\,\{\psi\}}\ \text{if} \models \phi \rightarrow \psi$$

$$\frac{}{\{\phi\}\, x := E\, \{\psi\}}\ \text{if} \models \phi \rightarrow \psi[E/x]$$

$$\frac{\{\phi\}\, C_1\, \{\eta\} \qquad \{\eta\}\, C_2\, \{\psi\}}{\{\phi\}\, C_1; C_2\, \{\psi\}}$$

$$\frac{\{\phi \ \wedge\ B\}\, C_1\, \{\psi\} \qquad \{\phi \ \wedge\ \neg B\}\, C_2\, \{\psi\}}{\{\phi\}\, \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\, \{\psi\}}$$

$$\frac{\{\eta \ \wedge\ B\}\, C\, \{\eta\}}{\{\psi\}\, \mathtt{while}\ B\ \mathtt{do}\, \{\eta\} C\, \{\phi\}}\ \text{se} \models \psi \rightarrow \eta \text{ e} \models \eta \ \wedge\ \neg B \rightarrow \phi$$

System $\mathcal{H}_g$

$$\frac{}{\{\phi\}\,\mathsf{skip}\,\{\psi\}}\ \text{if} \models \phi \to \psi$$

$$\frac{}{\{\phi\}\, x := E\,\{\psi\}}\ \text{if} \models \phi \to \psi[E/x]$$

$$\frac{\{\phi\}\ C_1\ \{\eta\} \qquad \{\eta\}\ C_2\ \{\psi\}}{\{\phi\}\ C_1;\ C_2\ \{\psi\}}$$

$$\frac{\{\phi\ \wedge\ B\}\ C_1\ \{\psi\} \qquad \{\phi\ \wedge\ \neg B\}\ C_2\ \{\psi\}}{\{\phi\}\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \{\psi\}}$$

$$\frac{\{\eta\ \wedge\ B\}\ C\ \{\eta\}}{\{\psi\}\ \texttt{while}\ B\ \texttt{do}\ \{\eta\}C\ \{\phi\}}\ \text{se} \models \psi \to \eta \text{ e} \models \eta\ \wedge\ \neg B \to \phi$$

The weakest precondition strategy

- System $\mathcal{H}_g$ contains an implicit strategy for constructing a proof/derivation of an assertion $\{\phi\}C\{\psi\}$ in a deterministic way.
- During the construction of a proof, side conditions (verification conditions) are created, that have to be checked to hold by some proof tool.
- For the sequence rule $\{\phi\}C_1; C_2\{\psi\}$ an intermediate formula η has to be guessed. For this, the weakest precondition η verifying $\{\eta\}C_2\{\psi\}$ is used.

The weakest precondition strategy

- System $\mathcal{H}_g$ contains an implicit strategy for constructing a proof/derivation of an assertion $\{\phi\}C\{\psi\}$ in a deterministic way.
- During the construction of a proof, side conditions (verification conditions) are created, that have to be checked to hold by some proof tool.
- For the sequence rule $\{\phi\}C_1;C_2\{\psi\}$ an intermediate formula η has to be guessed. For this, the weakest precondition η verifying $\{\eta\}C_2\{\psi\}$ is used.

The weakest precondition strategy

- System $\mathcal{H}_g$ contains an implicit strategy for constructing a proof/derivation of an assertion $\{\phi\}C\{\psi\}$ in a deterministic way.
- During the construction of a proof, side conditions (verification conditions) are created, that have to be checked to hold by some proof tool.
- For the sequence rule $\{\phi\}C_1; C_2\{\psi\}$ an intermediate formula η has to be guessed. For this, the weakest precondition η verifying $\{\eta\}C_2\{\psi\}$ is used.

A VCGen algorithm: computation of the weakest preconditions of a program (wp)

Given a program C and a postcondition ϕ, one can compute $wp(C,\phi)$ by the following rules. Then, $\{wp(C,\phi)\}C\{\phi\}$ holds and, furthermore, if $\{\psi\}C\{\phi\}$ holds for some ψ then $\psi \rightarrow wp(C,\phi)$.

$$\begin{array}{rcl} wp(\mathtt{skip},\phi) & = & \phi \\ wp(x := E,\phi) & = & \phi[E/x] \\ wp(C_1; C_2,\phi) & = & wp(c_1, wp(C_2,\phi)) \\ wp(\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2,\phi) & = & (B \rightarrow wp(C_1,\phi) \\ & & \wedge\ (\neg B \rightarrow wp(C_2,\phi) \\ wp(\mathtt{while}\ B\ \mathtt{do}\ \{\eta\}C,\phi) & = & \eta \end{array}$$

A VCGen algorithm: computation of the weakest preconditions of a program (wp)

Given a program C and a postcondition ϕ, one can compute $wp(C, \phi)$ by the following rules. Then, $\{wp(C, \phi)\} C \{\phi\}$ holds and, furthermore, if $\{\psi\} C \{\phi\}$ holds for some ψ then $\psi \to wp(C, \phi)$.

$$\begin{array}{rcl}
wp(\mathtt{skip}, \phi) & = & \phi \\
wp(x := E, \phi) & = & \phi[E/x] \\
wp(C_1; C_2, \phi) & = & wp(c_1, wp(C_2, \phi)) \\
wp(\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, \phi) & = & (B \to wp(C_1, \phi) \\
 & & \quad \wedge (\neg B \to wp(C_2, \phi) \\
wp(\mathtt{while}\ B\ \mathtt{do}\ \{\eta\} C, \phi) & = & \eta
\end{array}$$

A VCGen algorithm: computation of the weakest preconditions of a program (wp)

Given a program C and a postcondition ϕ, one can compute $wp(C,\phi)$ by the following rules. Then, $\{wp(C,\phi)\}C\{\phi\}$ holds and, furthermore, if $\{\psi\}C\{\phi\}$ holds for some ψ then $\psi \to wp(C,\phi)$.

$$
\begin{array}{rcl}
wp(\mathtt{skip},\phi) & = & \phi \\
wp(x := E,\phi) & = & \phi[E/x] \\
wp(C_1;C_2,\phi) & = & wp(c_1, wp(C_2,\phi)) \\
wp(\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2,\phi) & = & (B \to wp(C_1,\phi) \\
 & & \ \wedge (\neg B \to wp(C_2,\phi) \\
wp(\mathtt{while}\ B\ \mathtt{do}\ \{\eta\}C,\phi) & = & \eta
\end{array}
$$

VCGen algorithm

First, function *VC* computes a set of verification conditions, without taking the precondition into acount:

$$\begin{array}{rcl}
VC(\texttt{skip}, \phi) & = & \emptyset \\
VC(x := E, \phi) & = & \emptyset \\
VC(C_1; C_2, \phi) & = & VC(C_1, wp(C_2, \phi)) \cup VC(C_2, \phi) \\
VC(\texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2, \phi) & = & VC(C_1, \phi) \cup VC(C_2, \phi) \\
VC(\texttt{while}\, B\, \texttt{do}\, \{\eta\} C, \phi) & = & \{(\eta \ \wedge \ B) \rightarrow wp(C, \eta)\} \cup \\
 & & \{(\eta \ \wedge \ \neg B) \rightarrow \phi\} \cup VC(C, \eta)
\end{array}$$

Finally, the precondition has to imply the weakest precondition, which is required for ϕ to hold after the execution of C:

$$VCG(\{\psi\} C \{\phi\}) = \{\psi \rightarrow wp(C, \phi)\} \cup VC(C, \phi)$$

VCGen algorithm

First, function *VC* computes a set of verification conditions, without taking the precondition into acount:

$$\begin{array}{rcl}
VC(\texttt{skip}, \phi) & = & \emptyset \\
VC(x := E, \phi) & = & \emptyset \\
VC(C_1; C_2, \phi) & = & VC(C_1, wp(C_2, \phi)) \cup VC(C_2, \phi) \\
VC(\texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2, \phi) & = & VC(C_1, \phi) \cup VC(C_2, \phi) \\
VC(\texttt{while}\, B\, \texttt{do}\, \{\eta\}\, C, \phi) & = & \{(\eta \ \wedge \ B) \rightarrow wp(C, \eta)\} \cup \\
 & & \{(\eta \ \wedge \ \neg B) \rightarrow \phi\} \cup VC(C, \eta)
\end{array}$$

Finally, the precondition has to imply the weakest precondition, which is required for ϕ to hold after the execution of C:

$$VCG(\{\psi\}C\{\phi\}) = \{\psi \rightarrow wp(C, \phi)\} \cup VC(C, \phi)$$

VCGen algorithm

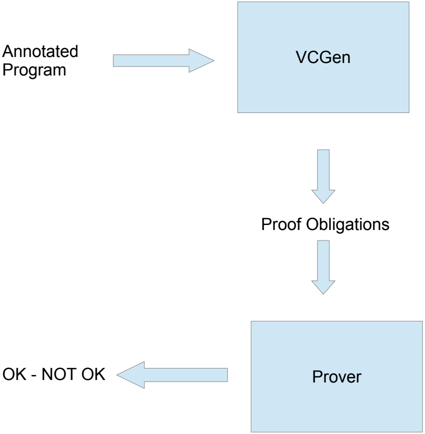
First, function *VC* computes a set of verification conditions, without taking the precondition into acount:

$$
\begin{array}{rcl}
VC(\texttt{skip}, \phi) & = & \emptyset \\
VC(x := E, \phi) & = & \emptyset \\
VC(C_1; C_2, \phi) & = & VC(C_1, wp(C_2, \phi)) \cup VC(C_2, \phi) \\
VC(\texttt{if}\, B \,\texttt{then}\, C_1 \,\texttt{else}\, C_2, \phi) & = & VC(C_1, \phi) \cup VC(C_2, \phi) \\
VC(\texttt{while}\, B \,\texttt{do}\, \{\eta\} C, \phi) & = & \{(\eta \ \wedge \ B) \to wp(C, \eta)\} \cup \\
 & & \{(\eta \ \wedge \ \neg B) \to \phi\} \cup VC(C, \eta)
\end{array}
$$

Finally, the precondition has to imply the weakest precondition, which is required for ϕ to hold after the execution of C:

$$VCG(\{\psi\} C \{\phi\}) = \{\psi \to wp(C, \phi)\} \cup VC(C, \phi)$$

General architecture of a program verification system



Annotated Program

VCGen

Proof Obligations

Prover

OK - NOT OK

Exemplo

Consider the following annotated program `fact`:

```
f:=1; i:=1;
while i<= n do {f = fact(i-1) and i <= n+1} {
      f:=f*i;
      i:=i+1;
}
```

We will compute

$$\mathrm{VCG}(\{n >= 0\}\texttt{fact}\{f = n!\})$$

using the following abbriations:

$$\theta = f = (i-1)! \wedge i \leq n+1 \quad \text{and} \quad C_w = f := f * i; i := i+1.$$

Exemplo

Consider the following annotated program `fact`:

```
f:=1; i:=1;
while i<= n do {f = fact(i-1) and i <= n+1} {
      f:=f*i;
      i:=i+1;
}
```

We will compute

$$\mathrm{VCG}(\{n >= 0\}\texttt{fact}\{f = n!\})$$

using the following abbriations:

$$\theta = f = (i-1)! \land i \leq n+1 \quad \text{and} \quad C_w = f := f * i; i := i+1.$$

$$
\begin{array}{rl}
& VC(\texttt{fact}, f = n!) \\
= & VC(f := 1; i := 1; wp(\texttt{while } i \leq n \texttt{ do}\{\theta\} C_w, f = n!)) \\
& \cup VC(\texttt{while } i \leq n \texttt{ do}\{\theta\} C_w, f = n!) \\
= & VC(f := 1; i := 1, \theta) \cup \{\theta \wedge i \leq n \rightarrow wp(C_w, \theta)\} \\
& \cup \{\theta \wedge i > n \rightarrow f = n!\} \cup VC(C_w, \theta) \\
= & VC(f := 1, wp(i := 1, \theta)) \cup VC(i := 1, \theta) \\
& \cup \{f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \rightarrow wp(f := f * i; i := i+1, \theta)\} \\
& \cup \{f = (i-1)! \wedge i \leq n+1 \wedge i > n \rightarrow f = n!\} \\
& \cup VC(f = f * i, wp(i := i+1, \theta)) \cup VC(i := i+1, \theta) \\
= & \emptyset \cup \emptyset \cup \{f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \\
& \qquad\qquad \rightarrow wp(f := f * i, f = (i+1-1)! \wedge i+1 \leq n+1)\} \\
& \cup \{f = (i-1)! \wedge i \leq n+1 \wedge i > n \rightarrow f = n!\} \cup \emptyset \cup \emptyset \\
= & \{f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \rightarrow f * i = (i+1-1)! \wedge i+1 \leq n+1, \\
& f = (i-1)! \wedge i \leq n+1 \wedge i > n \rightarrow f = n!\}
\end{array}
$$

$$
\begin{aligned}
& VCG(\{n \geq 0\}\texttt{fact}\{f = n!\}) \\
= \; & \{n \geq 0 \to wp(\texttt{fact}, f = n!)\} \cup VC(\texttt{fact}, f = n!) \\
= \; & \{n \geq 0 \to wp(f := 1; i := 1; wp(\texttt{while } i \leq n \texttt{ do}\{\theta\}C_w, f = n!)\} \\
& \{f = (i-1)! \land i \leq n+1 \land i \leq n \to f * i = (i+1-1)! \land i+1 \leq n+1, \\
& f = (i-1)! \land i \leq n+1 \land i \leq n \to f = n!\} \\
= \; & \{n \geq 0 \to wp(f := 1; i := 1; \theta)\} \\
& \{f = (i-1)! \land i \leq n+1 \land i > n \to f * i = (i+1-1)! \land i+1 \leq n+1, \\
& f = (i-1)! \land i \leq n+1 \land i > n \to f = n!\}
\end{aligned}
$$

The following proof obligations are generated:

1. $n \geq 0 \to 1 = (1-1)! \land 1 \leq n+1$
2. $f = (i-1)! \land i \leq n+1 \land i \leq n \to f * i = (i+1-1)! \land i+1 \leq n+1)$
3. $f = (i-1)! \land i \leq n+1 \land i > n \to f = n!$