System Validation 2007/2008 kw4
Lecture 3 - Wednesday, 7 May 2008

Building a Model Checker

Theo C. Ruys
http://www.cs.utwente.nl/~ruys/

University of Twente
Enschede - The Netherlands

SV #3
7 May 2008

Formal Methods & Tools

SV Lectures

#	date	topic	material
1	Mon 14 April	SPIN	[Gerth 1997, SPIN QuickRef, Hatcliff 2001]
	Wed 16 April	no lecture	
2	Mon 21 April	Linear Temporal Logic	[Merz 2000]
	Wed 23 April	no lecture	
3	Wed 7 May	Building a Model Checker	[Kattenbelt et.al. 2007]
4	Wed 14 May	Partial Order Reduction	[Peled 1999, Flanagan & Godefroid 2005]
5	Mon 19 May	Hashing	[Kuntz & Lampka 2004]
6	Mon 26 May	Compression	[Holzmann 1997]
	Mon 2 June	no lecture	
7	Mon 9 June	Software Verification	[Visser et.al. 2003, Ruys & Aan de Brugh 2007, Ball & Rajamani 2001]

Announcements

- Due to Whit Monday (‘tweede Pinksterdag’), the deadline for the SPIN exercises is postponed to

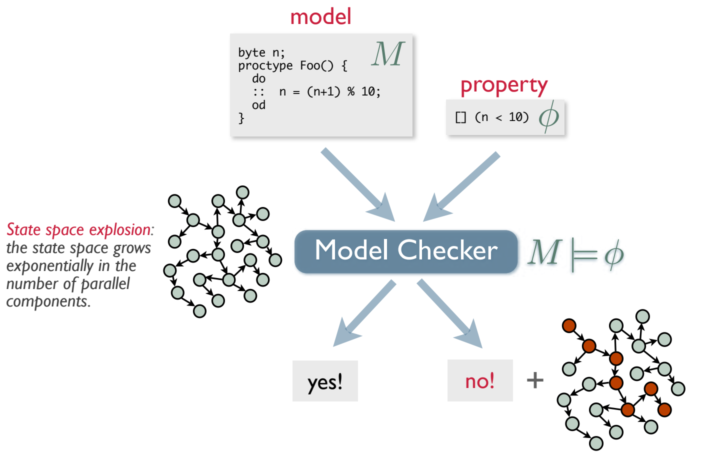
  Tuesday, 13 May 2008, 23.59h (was: Mon 12 May)

Overview of lecture 3

- Model Checking LTL
  - Kripke Structures
  - Büchi automaton
  - Model checking LTL by language inclusion
- Implementation of a model checker
  - Architecture, global algorithm
  - Layered Architecture
- ANTLR
- SUMO project

The ‘Model Checking LTL’ part is based upon [Wolper 2000] and presentations by Joost-Pieter Katoen and Ralf Huuck.

Model Checking



model

```
byte n;
proctype Foo() {
  do
  ::  n = (n+1) % 10;
  od
}
```

M

property

`[] (n < 10)` ϕ

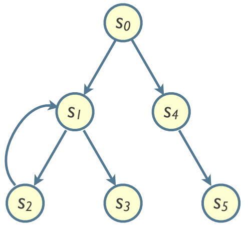
Model Checker $M \models \phi$

State space explosion: the state space grows exponentially in the number of parallel components.

yes!

no! +

Safety properties



Properties

- invariants
- (local) assertions
- deadlocks
- dead code

What about liveness properties?

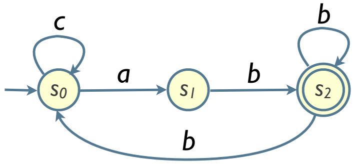
Depth First Search (DFS)

```
program check(M: model)
  foreach s in init(M) do
    dfs(s)
  od
end
```

```
proc dfs(s: state)
  if error(s) then report error fi
  add s to statespace
  foreach successor t of s do
    if t not in statespace then dfs(t) fi
  od
endproc
```

Finite State Automaton (1)

- A finite state automaton A is a tuple $(\Sigma, S, I, \rightarrow, F)$ where
  - Σ is an alphabet,
  - S is a finite set of states,
  - $I \subseteq S$ is the set of initial states,
  - $\rightarrow \subseteq S \times \Sigma \times S$ is a labelled transition relation,
  - $F \subseteq S$ is the set of accept states.

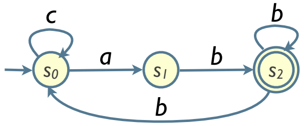


$\Sigma = \{a, b, c\}$
$S = \{s_0, s_1, s_2\}$
$I = \{s_0\}$
$F = \{s_2\}$

Finite State Automaton (2)

- Given a finite state automaton A is a tuple $(\Sigma, S, I, \rightarrow, F)$.
  - A run is a finite sequence of states $\sigma = s_0 s_1 \ldots s_n$ such that $s_0 \in I$ and $s_i \xrightarrow{a_i} s_{i+1}$ for all $0 \leq i < n$ for some $a_i \in \Sigma$.
  - Run σ is called accepted by A iff $s_n \in F$.
  - A finite word $w = a_0 a_1 \ldots a_{n-1} \in \Sigma^*$ is accepted by A iff there exists an accepting run $\sigma = s_0 s_1 \ldots s_n$ such that $s_i \xrightarrow{a_i} s_{i+1}$ for all $0 \leq i < n$.
  - The language accepted by A, denoted by $L(A)$, is the set of finite words accepted by A, i.e. $L(A) = \{ w \in \Sigma^* \mid w \text{ is accepted by } A \}$.

some accepted runs:
cab
cabbcab
cabbbbbcabbbb



$L(A) = c^*ab+(bc^*ab+)^*$

Model Checking LTL

- Model Checking Problem: M |= ϕ
  - M is given as a Kripke structure
  - ϕ is given in temporal logic
- Idea: model checking as language inclusion checking
  - Encode M as an automaton, which accepts $L(M)$
  - Encode ϕ as an automaton, which accepts $L(\phi)$
  - Check: $L(M) \subseteq L(\phi)$

Kripke Structure

- A Kripke structure K is a tuple $(S, I, R, Label)$ where
  - S is a countable set of states,
  - $I \subseteq S$ is the set of initial states,
  - $R \subseteq S \times S$ is a transition relation satisfying,
    - Every state has a successor
      $\forall s \in S\,.\,(\exists s' \in S\,.\,(s, s') \in R)$
  - $Label: S \rightarrow 2^{AP}$ is an interpretation function on S.

Kripke to Automaton

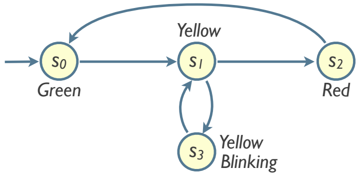
- Let Kripke Structure $K = (S, I, R, Label)$ with $Label: S \rightarrow 2^{AP}$.
- The corresponding automaton $A = (\Sigma, S', I', \rightarrow, F')$, where
  - $\Sigma = 2^{AP}$
  - $S' = S \cup \{s\}$, with $s \notin S$
  - $I' = \{s\}$
  - $F = S'$
  - $\rightarrow$ is the smallest relation satisfying
    $s \xrightarrow{\alpha} s'$ iff $s' \in I$ and $\alpha = Label(s')$
    $s' \xrightarrow{\alpha} s''$ iff $(s', s'') \in R$ and $\alpha = Label(s'')$

Thus:

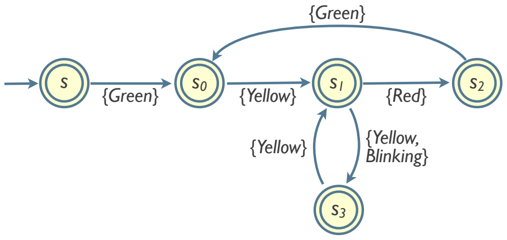
- Add an additional initial node s to A.
- Propositions p are attached to incoming edges.
- All nodes in A are accepting.

Runs through A are now words of "sets of the propositions" of K.

Example



A Kripke structure of a traffic light.



The corresponding automaton.

Semantics of LTL

- A path in K is an infinite sequence of states $\sigma = s_0\, s_1\, s_2 \ldots$ with $(s_i, s_{i+1}) \in R$.
- The semantics of LTL is defined as follows:
  - $\sigma \models a$ iff $a \in Label(\sigma[0])$
  - $\sigma \models \neg\Phi$ iff not $(\sigma \models \Phi)$
  - $\sigma \models \Phi \vee \Psi$ iff $(\sigma \models \Phi)$ or $(\sigma \models \Psi)$
  - $\sigma \models X\,\Phi$ iff $\sigma^1 \models \Phi$
  - $\sigma \models \Phi\, U\, \Psi$ iff $\exists j \geq 0\, .\, (\sigma^j \models \Psi$ and $(\forall 0 \leq k < j\, .\, \sigma^k \models \Phi))$

where $\sigma[i]$ denotes the i-th state in the path σ
and σ^i denotes the suffix of σ by removing the first i states

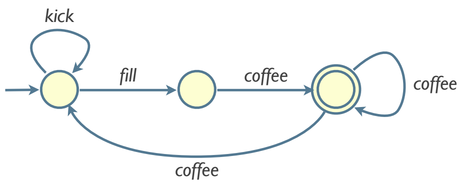
---

Büchi automata

- A Büchi automaton has the same ingredients as the finite state automaton. Only the acceptance condition is different.
- A infinite trace is accepted by a Büchi automaton when it visits an accept state infinitely often.
- Infinite words (or ω-words) are sequences of symbols isomorphic to the natural numbers. Precisely, an infinite word over an alphabet Σ is a mapping $w\colon \mathrm{N} \rightarrow \Sigma$.

Because Σ is finite, this means that certain (sequences of) symbols will be repeated infinitely often.

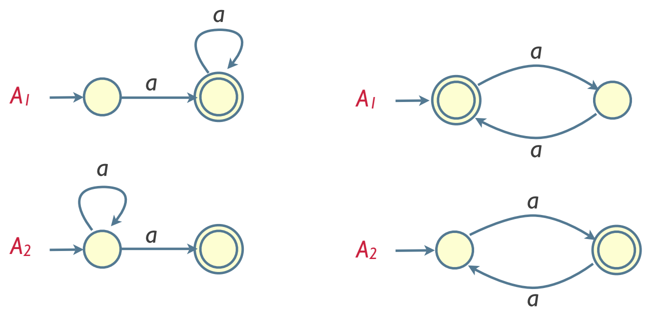
Example (1)



$L_\omega(A)$ = $kick^* . fill . coffee . (coffee \mid coffee . kick^* . fill . coffee)^\omega$

= $kick^* . fill . (coffee . (- \mid kick^* . fill))^\omega$

Example (2)



$L(A_1) = L(A_2)$
$L_\omega(A_1) \neq L_\omega(A_2)$

$L(A_1) \neq L(A_2)$
$L_\omega(A_1) = L_\omega(A_2)$

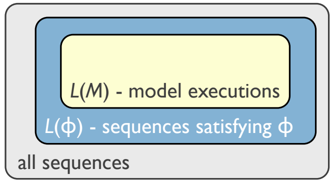
Model Checking LTL

reminder

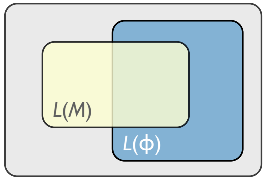
- Model Checking Problem: $M \models \phi$
  - M is given as a Kripke structure
  - ϕ is given in temporal logic
- Idea: model checking as language inclusion checking
  - Encode M as an automaton, which accepts $L(M)$
  - Encode ϕ as an automaton, which accepts $L(\phi)$
  - Check: $L(M) \subseteq L(\phi)$

Language Inclusion

$L(M) \subseteq L(\phi)$

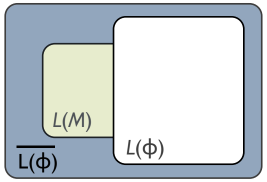


$L(M) \not\subseteq L(\phi)$



So to check that $L(M) \subseteq L(\phi)$,
we can equivantly check:
$L(M) \cap \overline{L(\phi)} = \emptyset$, or
$A_M \cap \overline{A_\phi} = \emptyset$

This is called emptiness checking.



$L(M) \cap \overline{L(\phi)} \neq \emptyset$

Problems to solve

- So we need to check $L(M) \cap \overline{L(\phi)} = \emptyset$
  or equivalently: $A_M \cap \overline{A_\phi} = \emptyset$
- Problems to solve:
  1. How to intersect two automata?
  2. How to complement an automaton?
  3. How to check for emptiness of an automaton?
  4. How to translate a LTL formula to an automaton?

Complementation

$\overline{A_\phi}$

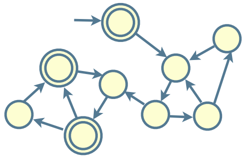
- Complementation of automata is hard!
- But if we know how to translate an LTL formula ϕ to a Büchi automaton, we can:
  - Build an automaton A for ϕ, and complement A, or
  - Negate the property, obtaining $\neg\phi$.
    (i.e. the sequences that should never occur).
    And then build an automaton for $\neg\phi$, i.e. $A_{\neg\phi}$.

    We choose this option, so we do not have to bother with complementation.

Emptiness checking (1)

- We need to check if there exists an accepting run (passes through an accepting state infinitely often). If no such run exists, then $L_\omega(A) = \varnothing$.

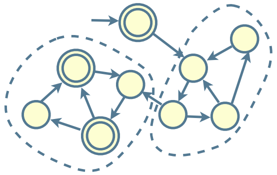
- Formally: $L_\omega(A) \neq \varnothing$ iff there exists $s_0 \in I$ and $s' \in F$ such that s' is reachable from s_0 and s' is reachable from s'.



L(A) is non-empty = A has a reachable cycle that contains an accept state.

Emptiness checking (2)

- Nested DFS
  - If there is an accepting run, then it contains at least one accept state an infinite # of times.
    - *This state must appear in a cycle.*
  - So, find a reachable accepting state (DFS) on a cycle.
    - *How to detect a cycle?*
- Find a reachable strongly connected component (SCC) with an accepting node.



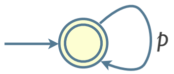
LTL to Büchi

- For any LTL-formula Φ, a Büchi automaton A_Φ over alphabet $\Sigma = 2^{AP}$ can be constructed such that
  $L_\omega(A_\Phi) = \{ \sigma \in (2^{AP})^\omega \mid \sigma \models \Phi \}$
- A_Φ accepts all traces satisfying Φ.
- The number of states in A_Φ is in $O(2^{|\Phi|})$.

[Wolper, Vardi and Sistla, 1983]

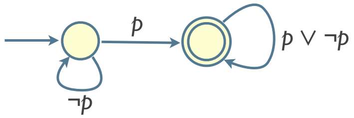
See [Wolper 2000] for details.

Intuition (1)



$L_\omega = p^\omega$

LTL formula: G p

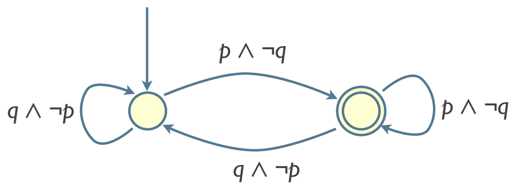


$L_\omega = (\neg p)^* p\ (p \vee \neg p)^\omega$

LTL formula: F p

Intuition (2)

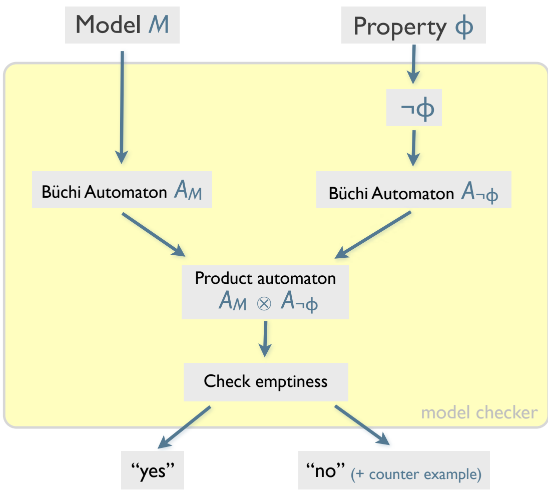
A_ϕ



$$L_\omega = ((q \wedge \neg p)^* (p \wedge \neg q))^\omega$$

LTL formula: $\mathbf{G}\ ((q \wedge \neg p)\ \mathbf{U}\ (p \wedge \neg q))$

Summary



Model M

Property ϕ

$\neg\phi$

Büchi Automaton A_M

Büchi Automaton $A_{\neg\phi}$

Product automaton
$A_M \otimes A_{\neg\phi}$

Check emptiness

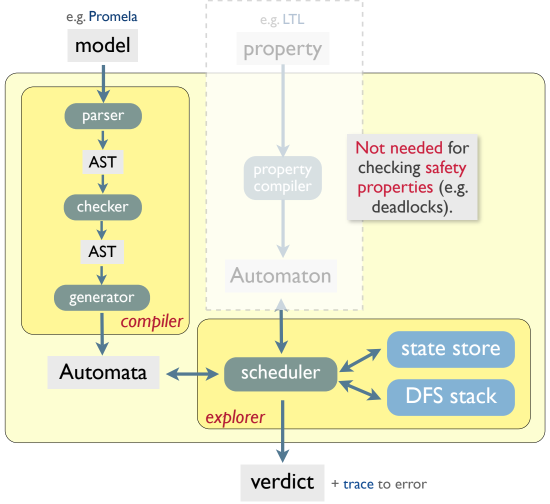
model checker

"yes"

"no" (+ counter example)

Architecture of a straightforward, on-the-fly model checker

A r c h i t e c t u r e



e.g. Promela
model

e.g. LTL
property

parser
AST
checker
AST
generator
compiler

property compiler

Not needed for checking safety properties (e.g. deadlocks).

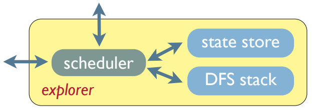
Automaton

Automata

scheduler
state store
DFS stack
explorer

verdict + trace to error

Global algorithm - DFS



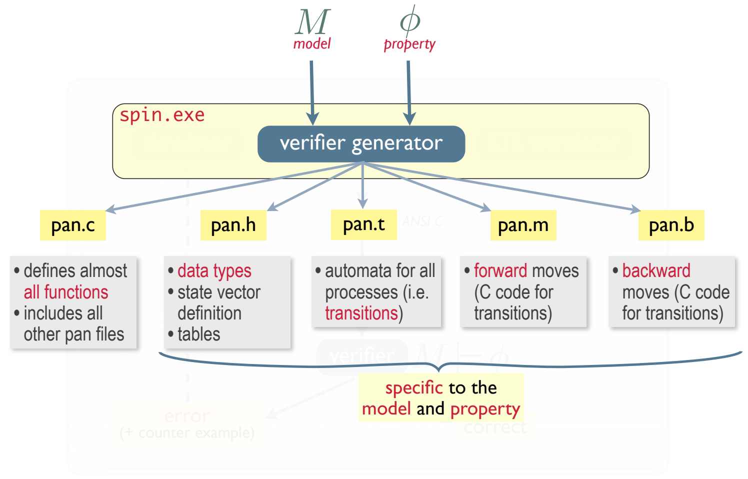
```
proc dfs($s_0$: state)
  push $s_0$ on stack S
  while S is not empty do
    s ← top(S)
    if error(s) then report error fi
    add s to state store
    foreach successor t of s do
      if t not in state store then
        push t on stack S
      fi
    od
  od
endproc
```

For the state store, typically a hash table is used:

- addition is fast
- query is fast

Other, more effective (iterative) implementations are possible.

SPIN (1)



- defines almost all functions
- includes all other pan files

- data types
- state vector definition
- tables

- automata for all processes (i.e. transitions)

- forward moves (C code for transitions)

- backward moves (C code for transitions)

specific to the model and property

SPIN (2)

- SPIN supports many optimisations to tune the verification.
  - partial order reduction (`-DNOREDUCE`)
  - bitstate hashing (`-DBITSTATE`)
  - hash compaction (`-DHC`)
  - safety verification run (`-DSAFETY`)
  - minimised automaton (`-DMA`)
  - multi-core verification (`-DCORE`)
  - state collapsion (`-DCOLLAPSE`)
  - breath first search (`-DBFS`)
  - verbose debugging printing (`-DVERBOSE`)
  - ...

The effectiveness of these advanced and powerful options account for the popularity of SPIN.

Most (if not all) of these features are controlled using C compiler options.

Beware of feature interaction!

```
#if NCORE>1 && defined(FULL_TRAIL)
                if (upto > 0)
                {   Pop_Stack_Tree();
                }
#endif
                  goto Up;
                }
#ifdef CHECK
                printf("not seed\n");
#endif
            }
#endif
            if (!(trpt->tau&8)) /* if no atomic move */
            {
#ifdef BITSTATE
#ifdef CNTRSTACK
                II = bstore((char *)&now, vsize);
                trpt->j6 = j1; trpt->j7 = j2;
                JJ = LL[j1] && LL[j2];
#else
#ifdef FULLSTACK
                JJ = onstack_now();
#else
#ifndef NOREDUCE
                JJ = II; /* worstcase guess for p.o. */
#endif
#endif
                II = bstore((char *)&now, vsize);
#endif
#ifdef MA
                II = gstore((char *)&now, vsize, 0);
#ifndef FULLSTACK
                JJ = II;
#else
                JJ = (II == 2)?1:0;
#endif
```

(typical) fragment of C code in pan.c

The resulting verifier is fast and tuned to the maximum.

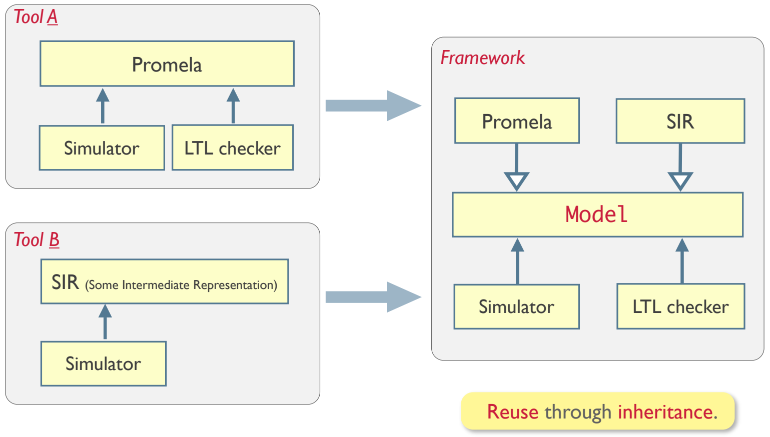
But hard to maintain (spaghetti-code) and impossssible to reuse.

S P I N (3)

Modular approach [Kattenbelt et.al. 2007]

- Motivation: model checkers are specialised.
  - Reusing functionality requires model transformations.
  - Most tools use their own formalism.
  - Typically built from scratch.
- Idea
  - Implement functionality generically.
  - Reusable functionality for different models.
  - Focus on explicit-state model checking.

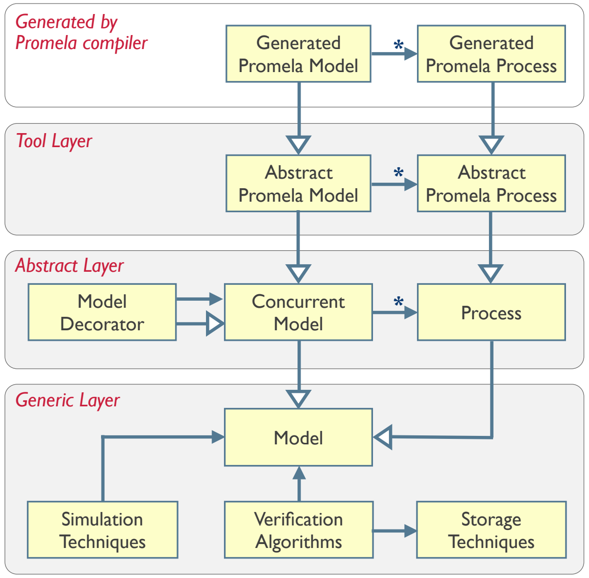
Concept



Reuse through inheritance.

[de Jonge 2008]

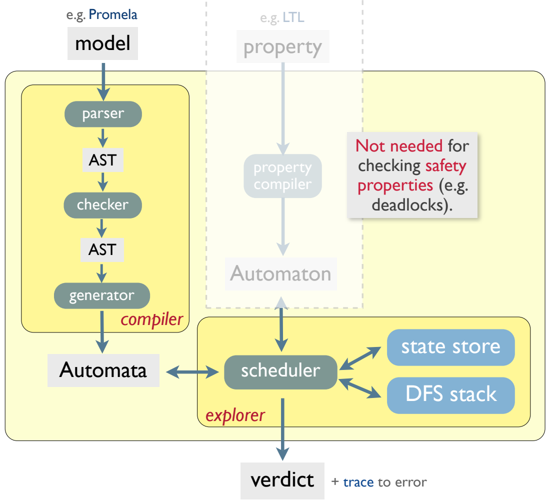
SpinJ



SpinJ

Architecture of a straightforward, on-the-fly model checker

Architecture

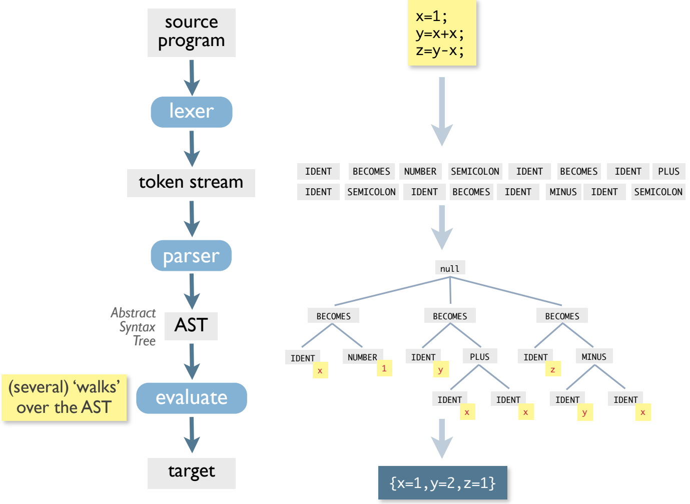


Not needed for checking safety properties (e.g. deadlocks).

How does a compiler work?

ANTLR

(1)



(several) 'walks' over the AST

ANTLR (2)

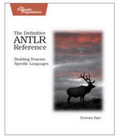
http://www.antlr.org/

- ANTLR
  - input: language descriptions using EBNF grammar
  - output: recognizer for the language
- ANTLR can build recognizers for three kinds of input:
  - character streams (i.e. by generating a scanner)
  - token streams (i.e. by generating a parser)
  - node streams (i.e. by generating a tree walker)

  ANTLR uses the same syntax for all its recognizer descriptions.
- ANTLR 3.x
  - LL(*) compiler generator
  - generates recognizers in Java, C++, C#, Python, etc.

ANTLR (3)

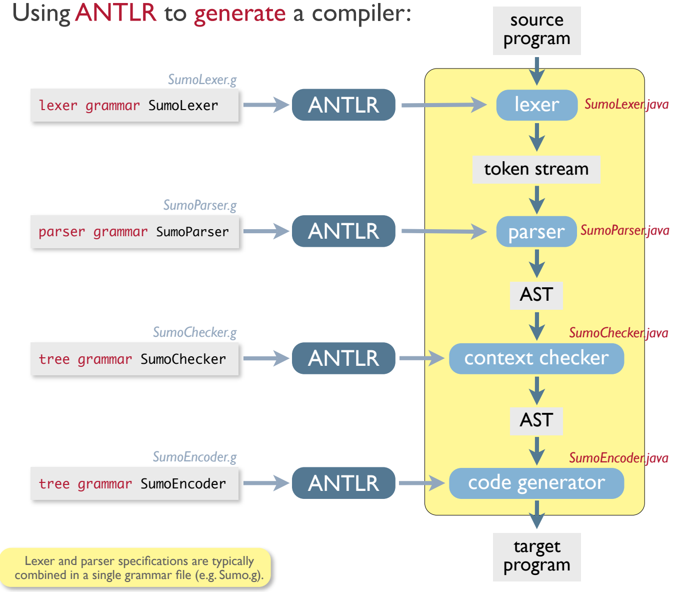
- Documentation on ANTLR
  - Getting started with ANTLR3:
    http://www.antlr.org/wiki/display/ANTLR3/FAQ+-+Getting+Started
  - See also the course on Compiler Construction (Vertalerbouw)
    http://fmt.cs.utwente.nl/courses/vertalerbouw
    - Lecture 4 gives an extensive introduction to ANTLR3.
    - See also the laboratory ('practicum') files of week 3 for an complete compiler, checker and interpreter for a small language (i.e. Calc).
- Book
  - Terence Parr.
    The Definitive ANTLR Reference.
    Pragmatic Bookshelf, 2007.



Hard copy: $36.95
PDF: $24.00

not really needed for this course though

Using ANTLR to generate a compiler:



Lexer and parser specifications are typically combined in a single grammar file (e.g. Sumo.g).

A N T L R (4)

ANTLR (5)

```
grammar Vars;

options {
    k=1;
    output=AST;
}

tokens {
    BECOMES     =   '='   ;
    SEMICOLON   =   ';'   ;
    PLUS        =   '+'   ;
    MINUS       =   '-'   ;
}

program     :   assign+ EOF!                                ;
assign      :   IDENTIFIER BECOMES^ expr SEMICOLON!         ;
expr        :   operand ((PLUS^ | MINUS^) operand)*         ;
operand     :   IDENTIFIER | NUMBER                         ;

IDENTIFIER  :   LETTER (LETTER | DIGIT)*                    ;
NUMBER      :   DIGIT+                                      ;
WS          :   (' ' | '\t' | '\f' | '\r' | '\n')+
                    { $channel=HIDDEN; }
            ;

fragment DIGIT  :   ('0'..'9')      ;
fragment LOWER  :   ('a'..'z')      ;
fragment UPPER  :   ('A'..'Z')      ;
fragment LETTER :   LOWER | UPPER   ;
```

generate AST

```
x=1;
y=x+x;
z=y-x;
```

In ANTLR 3, the lexer and scanner specification can be conveniently combined.

$\alpha \mid \beta$	either α or β
$\alpha+$	one or more occurences of α
$\alpha*$	zero or more occurences of α

Annotations for building the AST	
T^	make T the root of this rule
T!	discard T

parser rules
(start with lowercase letter)

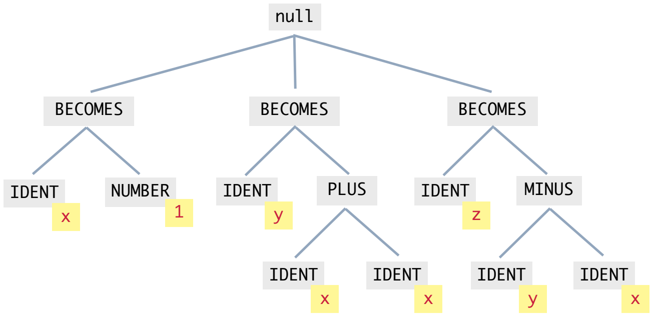
lexer rules (tokens)
(start with UPPERCASE letter)

fragment rules are not turned into tokens

ANTLR (6)

```
x=1;
y=x+x;
z=y-x;
```

```
program     :   assign+ EOF!                                  ;
assign      :   IDENTIFIER BECOMES^ expr SEMICOLON!           ;
expr        :   operand ((PLUS^ | MINUS^) operand)*           ;
operand     :   IDENTIFIER | NUMBER                           ;
```



ANTLR (7)

```
tree grammar VarsWalker;

options {
    tokenVocab=Vars;
    ASTLabelType=CommonTree;
}

@members {
    private SortedMap<String,Integer> store
        = new TreeMap<String,Integer>();
}

program :   assign+
            {   System.out.println(store.toString());   }
        ;
assign  :   ^(BECOMES id=IDENTIFIER val=expr)
            {   store.put($id.text,val);    }
        ;
expr returns [int val]
        :   z=operand                   { val=z;    }
        |   ^(PLUS  x=expr y=expr)      { val=x+y;  }
        |   ^(MINUS x=expr y=expr)      { val=x-y;  }
        ;
operand returns [int val]
        :   id=IDENTIFIER
            {   if (!store.containsKey($id.text))
                    store.put($id.text, 0);
                val = store.get($id.text);          }
        |   n=NUMBER
            {   val = Integer.parseInt($n.text);    }
        ;
```

Parser grammar

```
program     :   assign+ EOF!                                  ;
assign      :   IDENTIFIER BECOMES^ expr SEMICOLON!           ;
expr        :   operand ((PLUS^ | MINUS^) operand)*           ;
operand     :   IDENTIFIER | NUMBER                           ;
```

The original tokens are used to identify the tree nodes.

A tree parser (walker) walks over a flattened representation of the AST: a tree node stream.

Matches a tree whose root is a PLUS token with two children that match the expr rule.

Some additional code is needed to connect the lexer, parser and tree parser, of course. See the Calc (or SUMO) source files for details.

SUMO project (1)

- Develop a state space explorer for the modelling language SUMO, a subset of Promela. The explorer should check for safety properties.

SUMO = Simple but Useful MOdelling Language

  - input: system description in SUMO
  - output
    - no errors
    - error: deadlock / assertion violation + trace leading to error state

```
short variables
channels (capacity 1)
active proctype
expressions
assignment
send (!)
receive (?)
assert
if
do
break
```

- Implementation language: Java.
- The state space explorer should be able to be compiled and executed on a standard Unix/Linux system.

SUMO project (2)

- Basic grade for SUMO project:

grade	*status of implementation*
0	does not compile
≤ 5	does not work correctly on all test files
≥ 6	works correctly on all test files
6	30% slowest implementations
7	40% average implementations
8	30% fastest implementations
9	*the* fastest implementation

Note the ≤: even implementations that compile and do 'something' might be rewarded with 0.

Bonus points

+0.5	shortest counterexample
+0.5	state compression
+0.5	bitstate hashing + hash compaction
+1.0	other Promela features (max +1.0!)
+1.5	partial order reduction
+2.0	LTL model checking

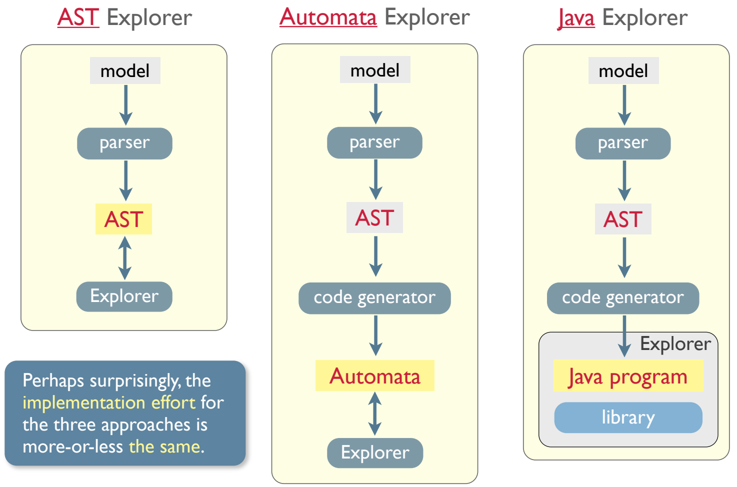
Beware: grade *might* be lowered due to flawed design, inefficient implementation, bad programming style, missing test results, etc.

SUMO project (3)

- Working in groups of preferably two (max) students.
- Deadline: Wednesday, 11 June 2008 23:59h (was: Mon 9 June)
- Project should be emailed as a zip-file, containing:
  - source code of the project
  - test files and results
  - small report as PDF-document (≤ 5 pages), describing the architecture, design and implementation
- Full description of "SUMO project" can be downloaded from the SV website on Tuesday, 13 May 2008.

An ANTLR3 grammar of the SUMO language will be provided.

Possible Approaches (1)



Perhaps surprisingly, the implementation effort for the three approaches is more-or-less the same.

Possible Approaches (2)

Apple MacBook (June 2006).
2Ghz Intel Core Duo, 2Gb RAM.
Mac OS X 10.4.11, Java 1.5.

Comparison (in terms of speed)

On basis of a very limited, light weight benchmark set.

tool / implementation	*language*	*states / sec*
SPIN 4.2.9	C	$340 \cdot 10^3$
NIPS 1.2.7	C	$190 \cdot 10^3$
SpinJ (July 2007)	Java	$120 \cdot 10^3$
(fastest) AST Explorer	Java	$20 \cdot 10^3$
(fastest) Automata Explorer	Java	$80 \cdot 10^3$
(fastest) Java Explorer	Java	$200 \cdot 10^3$
JPF / MoonWalker	Java / C#	$< 5 \cdot 10^3$

Limited benchmark set consisting of 7 SUMO-like models: $120 \cdot 10^3$ < # states < $1200 \cdot 10^3$

Since the 'competition' element, the performance of all explorers have improved. Before, the slowest explorer could visit less than 100 states/sec.

Typically. On different benchmarks, of course.