# 22. Concrete Interpretation and Abstract Interpretation

Prof. Dr. rer. nat. Uwe Aßmann

Institut für Software- und Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

TU Dresden

http://st.inf.tu-dresden.de

Version 12-1.4, 21.11.12

1) Abstract Interpretation (AI)
2) Iteration in Abstract Interpreters
3) Attribute Grammars for Interpreters on Syntax Trees

---

## Obligatory Literature

- David Schmidt. Tutorial Lectures on Abstract Interpretation. (Slide set 1.) International Winter School on Semantics and Applications, Montevideo, Uruguay, 21-31 July 2003.
  - http://santos.cis.ksu.edu/schmidt/Escuela03/home.html
- List of analysis tools
  - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

---

## Other Resources

- Selective reading:
  - Neil D. Jones and Flemming Nielson. 1995. Abstract interpretation: a semantics-based tool for program analysis. In Handbook of logic in computer science (vol. 4), S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum (Eds.). Oxford University Press, Oxford, UK 527-636.
    - http://dl.acm.org/citation.cfm?id=218637
  - Michael Schwartzbach's Tutorial on Program Analysis
    - http://lara.epfl.ch/dokuwiki/_media/sav08:schwartzbach.pdf
- Patrick Cousot's web site on A.I. http://www.di.ens.fr/~cousot/AI/
- [CC92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In U. Kastens and P. Pfahler, editors, Proceedings of the International Conference on Compiler Construction (CC), volume 641 of Lecture Notes in Computer Science, pages 125-140, Heidelberg, October 1992. Springer.
- [Kam/Ullmann] John B. Kam and Jeffery D. Ullmann. Global data flow analysis and iterative algorithms. Journal of the ACM, 23:158-171, 1976.

---

## Literature on Attribute Grammars

- Knuth, D. E. 1968. „Semantics of context-free languages“. Theory of Computing Systems 2 (2): 127–145.
- Paakki, Jukka. 1995. „Attribute grammar paradigms—a high-level methodology in language implementation“. ACM Comput. Surv. 27 (2) (Juni): 196–255.
- Hedin, Görel. 2000. „Reference Attributed Grammars“. Informatica (Slovenia) 24 (3): 301–317.
- Boyland, John T. 2005. „Remote attribute grammars“. Journal of the ACM 52 (4) (Juli): 627–687.
- Bürger, Christoff, Sven Karol, Christian Wende, und Uwe Aßmann. 2011. „Reference Attribute Grammars for Metamodel Semantics“. In Software Language Engineering,LNCS 6563:22–41.
- 
- Examples on: www.jastemf.org

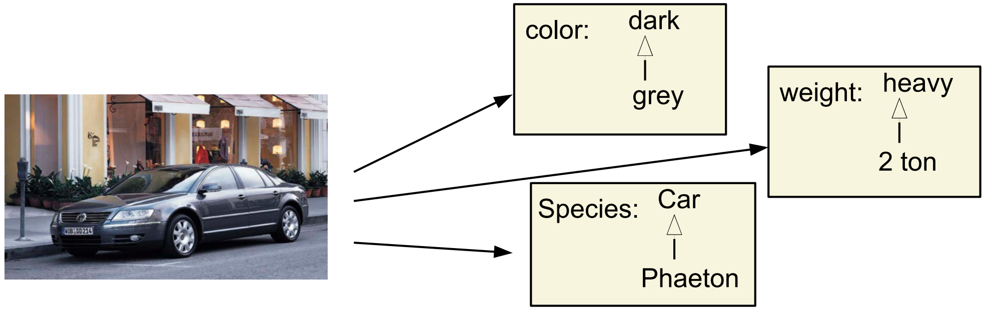
# 22.1 Abstract Interpretation (A.I.)

---

## What is Abstraction?

**Abstraction** is the neglection of unnecessary detail.
(**Abstraktion** ist das Weglassen von unnötigen Details)

- A thing of the world can be abstracted differently
- This generates mappings from a concrete domain (D) to abstract domains (D#)



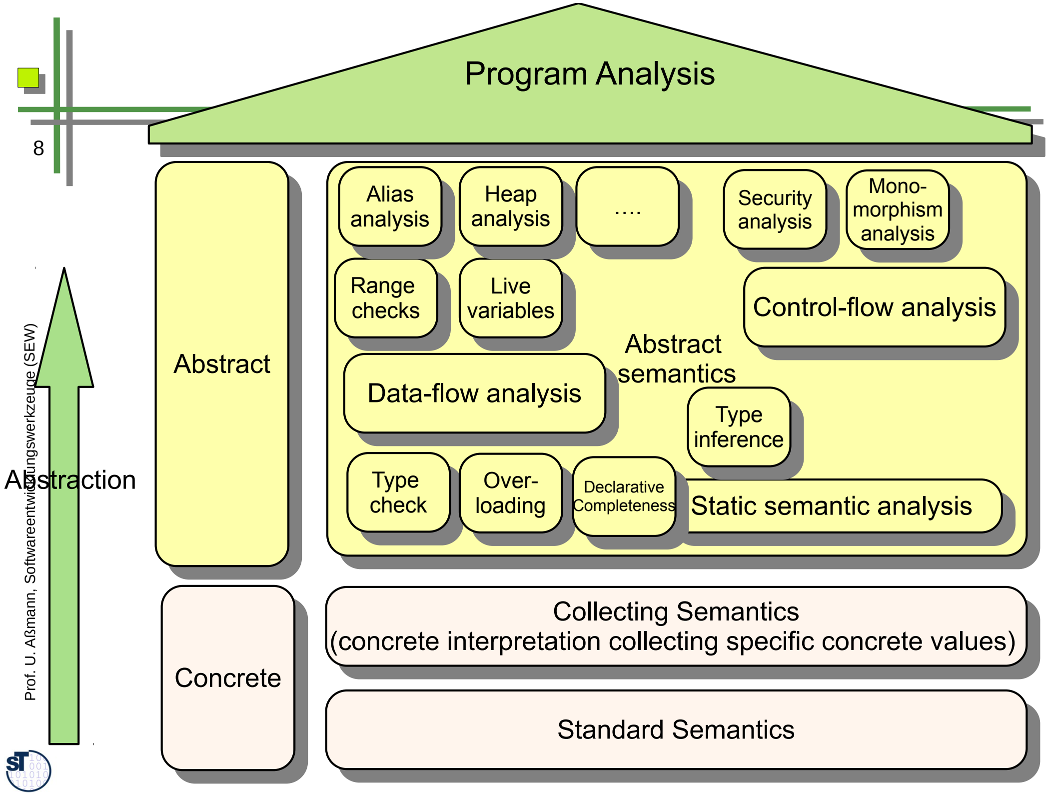
[VW factory]

---

## Interpretation and Semantics of Programs

- Given a fixed set of input values, a program has a **concrete standard semantics (dynamic semantics)**.
  - Denotational semantics (result semantics):
    - The output values
  - Operational semantics (interpretative semantics):
    - The set of traces of the execution
    - The set of states in the execution traces
  - Axiomatic semantics:
    - The set of all true predicates at each execution point
- A **collecting semantics** selects a subset of interest from the standard semantics, in preparation of the abstract interpretation.
  - The values of the semantics stay concrete.
- An **abstract interpretation** interprets on the **abstract semantics**, an abstraction of the the collecting semantics

---



Program Analysis

Abstract

Alias analysis

Heap analysis

....

Security analysis

Mono-morphism analysis

Range checks

Live variables

Control-flow analysis

Abstract semantics

Data-flow analysis

Type inference

Type check

Over-loading

Declarative Completeness

Static semantic analysis

Abstraction

Concrete

Collecting Semantics
(concrete interpretation collecting specific concrete values)

Standard Semantics

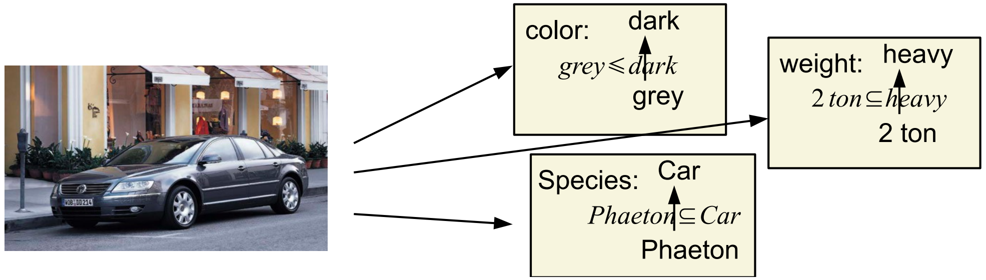
# What is an Interpreter?

- An **interpreter** executes a program on a set of input data and realizes an operational semantics
- For all metaclasses of the language, interpretation functions have to be given

---

# Abstract Interpretation

- ***Abstract interpretation** is static symbolic execution* of the program with *abstract symbolic* values
  - Since the values cannot be concrete we must abstract them to "easier" values, i.e., simpler domains of *finite* count, height, or breadth
- Values are taken from the *abstract domains* (called D#)
  - complete partial orders (cpo, with “or” or “subset”),
  - semi-lattices (cpo with some top elements) or
  - lattices (semi-lattice with top and bottom element)
- The suprenum operation of the cpo expresses the “unknown”, i.e., the unknown decisions at control flow decision points (if's)

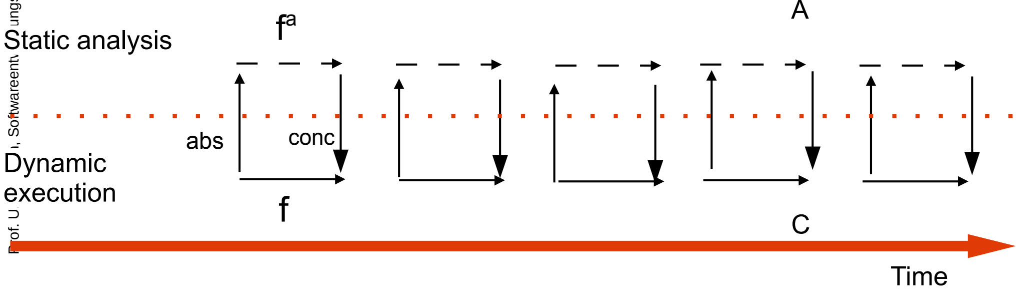


[VW factory]

---

# Functions for Abstract Interpretation

- f: C → C, run-time semantics of the program (**interpreter**)
- abs: C → A, **abstraction function** from concrete to abstract
- conc: A → C, **concretization function** from abstract to concrete
- $f^a$:A → A, **abstract interpretation function** (abstract semantic function, abstract interpreter, flow/transfer function)
- $f^a$ is like a *shadow* of f



---

# More Precisely: Abstract Interpreters are Sets of Abstract Interpretation Functions

- For an abstract interpretation, for all node types 1..k in the control flow graph (or metaclasses in the language), set up *interpretation functions (transfer functions),* each for one statement of the program
  - They form the core of the abstract interpreter

Real interpreter functions

$$f_n : C \to C$$
$$\Leftrightarrow$$
$$f_1 : C \to C$$
$$\dots$$
$$f_k : C \to C$$

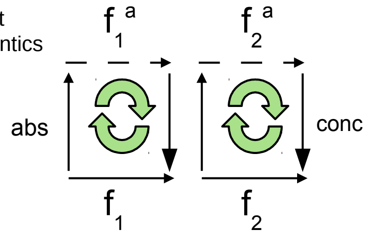
Abstract interpreter functions (transfer functions)

$$\{ f^a_n : A \to A \}$$
$$\Leftrightarrow$$
$$f^a_1 : A \to A$$
$$\dots$$
$$f^a_k : A \to A$$

# The Iron Law of Abstract Interpretation

The abstract interpretation must be correct, i.e., faithfully abstracting the run-time behavior of the program („reality proof“)

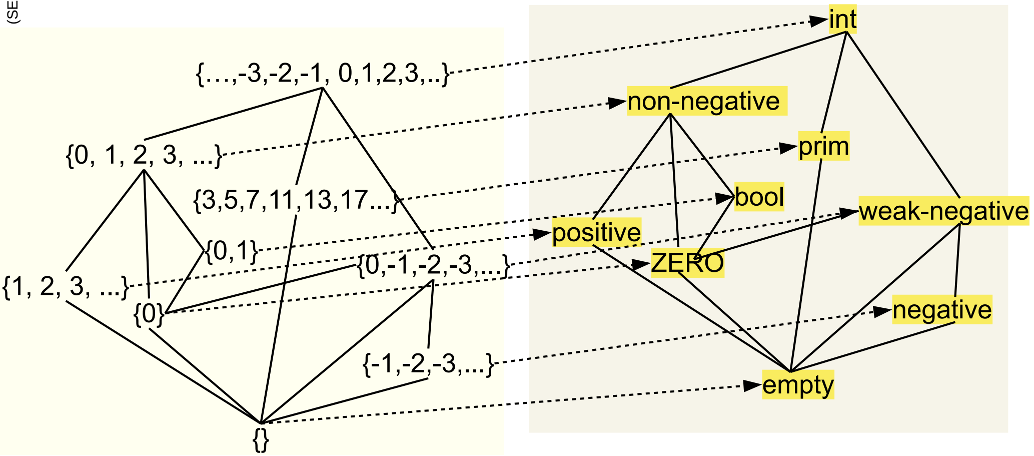
- Abs (abstraction function), conc (concretization function), and $f^a$ (abstract interpretation function) must form a commuting diagram
  - The abstract interpretation should deliver all correct values, but may be more
  - They must be "interchangeable", formally: a Gaulois connection
- The interpretation must be a subset of the abstract interpretation:
- f \subset conc ◦ fa ◦ abs
  - The concrete semantics must be a subset of the concretization of the abstract semantics (conservative approximation)
  - The abstract semantic value must be a superset of the concrete semantic value after application of the transfer function
  - The concrete value of f must be a subset of the abstracted value after application of the transfer function



---

# Ex. Concrete and Abstract Values over int

- A program variable v has a value from a concrete domain C (here Integers)
- At a point in the program, v can be typed by a subset of C
- This concrete domain C is mapped to symbolic abstract domain A
  - Here: subsets of C=int to symbolic A=”abstract symbolic sets over ints”
  - Top means *any-concrete-value*, bottom means *none*
  - cpo suprenum operation *meet:* unioning all subsets

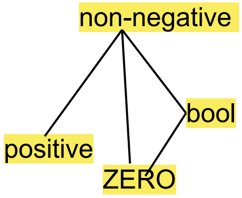


---

# Law of Join of Control Flow

When the abstract interpreter does not know what the type of a variable will be from 2 or n incoming paths, it takes the suprenum in the abstract domain

- In a *join point* of the control flow (at the end of an If, Switch, While, Loop), an abstract interpreter will not know from which incoming path it should select the value
  - If: two paths
  - Switch: finitely many paths
  - While, Loop: infinitely many paths
- In order to proceed, the interpreter chooses the *suprenum* of the values of all paths (meet over all incoming paths)
- Ex.: in a Switch the values are ZERO, bool, positive.
  - The interpreter will choose “non-negative”, to cover all.



---

# Ubiquituous A.I.

- Any program in any programming or specification language can be interpreted abstractly, if a collecting semantics is given.
- Examples:
  - A.I. of embedded C programs
  - A.I. of Prolog rule sets
  - A.I. of ECA-rule bases
  - A.I. of state machines (looks like model checking, see later)
  - A.I. of Petri Nets
- Quality analyses:
  - Worst case execution time analysis (WCETA)
  - Worst case energy analysis (WCENA)
  - Security analysis
- Functional analysis
  - Value analysis (“data-flow analysis”)
  - Range check analysis, null check analysis
  - Heap analysis, alias analysis

# 22.2 Iteration of Abstract Interpreters (Intra- and Interprocedural)

---

## Example: Interpretation of a Procedure with a Worklist Algorithm

- Iteration can be done *forward* over a worklist that contains “nodes not finished”

```
worklist := nodes;
WHILE (worklist != NULL) DO
SELECT n:node FROM worklist;
// forward propagation from predecessors to n
            FORALL p in n.ControlFlowGraph.predecessors
               X := meet( f^a(p) );
            // test fixpoint condition
            IF (X != value(n)) THEN
               value(n) = X;
               worklist += n.ControlFlowGraph.successors;
           END
END
```

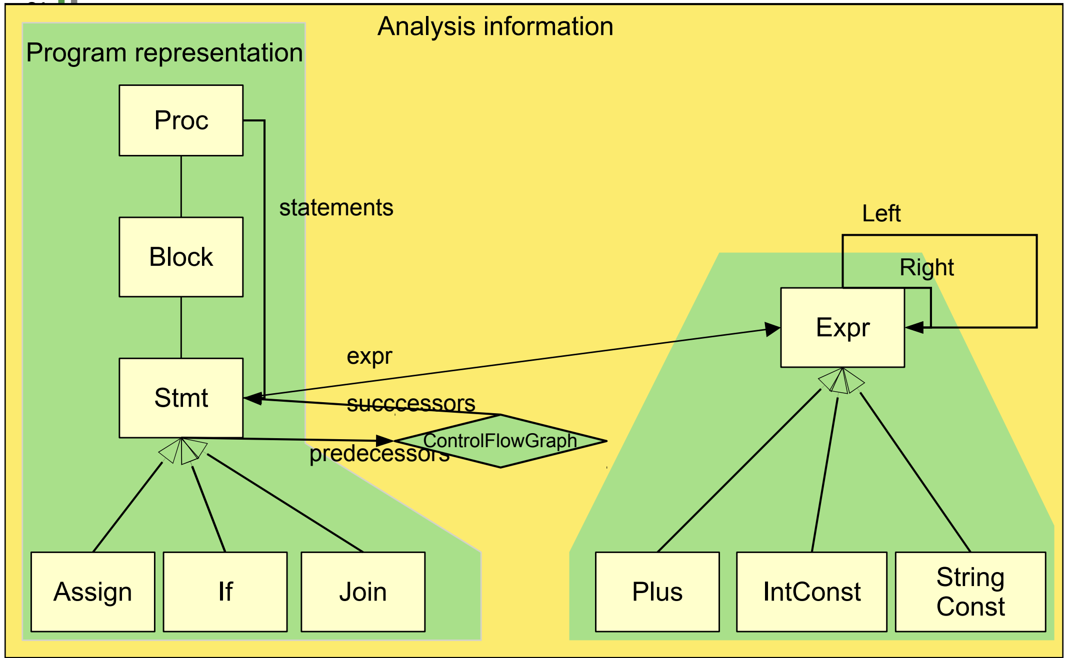
---

## Building Abstract Interpreters

- Works basically with Design Pattern “Interpreter”, as from the Gamma book
- What has to be modeled:
  - A model of the program (program representation), with Class, Proc, Stmt, Expr, etc
  - A model of the analysis information
    - ControlFlowGraph: has inserted Join nodes representing control flow joins in If#s and While's
    - AbstractValue domains: e.g., abstract integers, abstract intervals and ranges, abstract heap configurations
    - Environments binding variables to abstract values

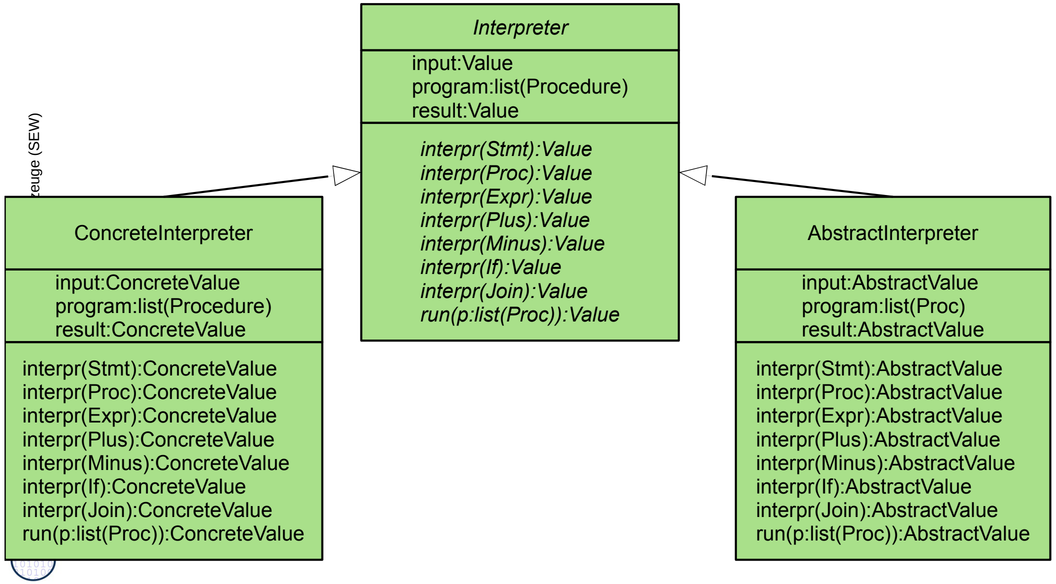
---

## A Simple Program (Code) Model (Schema) in MOF



# An OO Design of an Interpreter of a Programming Language

- Concrete and abstract interpreters are “twins”, i.e., have the same interface but working on concrete vs abstract values



*Interpreter*

input:Value
program:list(Procedure)
result:Value

*interpr(Stmt):Value*
*interpr(Proc):Value*
*interpr(Expr):Value*
*interpr(Plus):Value*
*interpr(Minus):Value*
*interpr(If):Value*
*interpr(Join):Value*
*run(p:list(Proc)):Value*

ConcreteInterpreter

input:ConcreteValue
program:list(Procedure)
result:ConcreteValue

interpr(Stmt):ConcreteValue
interpr(Proc):ConcreteValue
interpr(Expr):ConcreteValue
interpr(Plus):ConcreteValue
interpr(Minus):ConcreteValue
interpr(If):ConcreteValue
interpr(Join):ConcreteValue
run(p:list(Proc)):ConcreteValue

AbstractInterpreter

input:AbstractValue
program:list(Proc)
result:AbstractValue

interpr(Stmt):AbstractValue
interpr(Proc):AbstractValue
interpr(Expr):AbstractValue
interpr(Plus):AbstractValue
interpr(Minus):AbstractValue
interpr(If):AbstractValue
interpr(Join):AbstractValue
run(p:list(Proc)):AbstractValue

# Example: Interpretation of a Procedure with a Worklist Algorithm

- Simplified assumption: one value per statement is computed by the abstract interpreter.
- The value at the return statement of the interpreted procedure is the final result of the abstract interpretation

```
CLASS AbstractInterpreter EXTENDS Interpreter {
…
 FUNCTION interpr(p:Procedure):AbstractValue {
  worklist:list(Statement) := p.statements;
  WHILE (worklist != NULL) {
   SELECT current:Statement FROM worklist;
   // forward propagation from current.predecessors to
current
   FORALL pred in current.ControlFlowGraph.predecessors {
    NewValue := meet( pred.value );
   }
   // test whether fixpoint is reached
   IF (NewValue != current.value) {
    current.value = NewValue;
    worklist += current.ControlFlowGraph.successors;
   }
  }
  RETURN p.statements.last.value;
 }
}
```

# Intraprocedural Coincidence Theorem

[Kam/Ullman] Intraprocedural Coincidence Theorem:

The maximum fixpoint of an iterative evaluation of the system of abstract-interpretation functions fn at a node N
is equal
to the value of the meet over all paths to a node n (MOP(n))

- Forall n:Node: $MFP(n,f_n) = MOP(n,f_n)$
- The theorem means, that no matter how the abstract-interpretation functions are iterated over a procedure, if they stop at a fixpoint, they stop at the meet over all paths
  - Any iteration algorithm can be used to reach the abstract values at each node (i.e., the maximal fixpoint of the function system)
  - The paths through a procedure need not be formed (there may be infinitely many), instead, free iteration can be used until the fixpoint is found (until termination of the iteration)

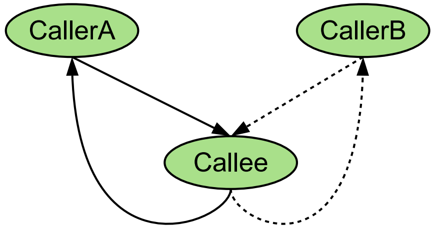
# Example: Backward Interpretation with Worklist Algorithm

- Iteration can be done with many strategies
- E.g., iterating *backward* over a worklist that contains “nodes not finished”
- Other alternatives: innermost-outermost, lazy, etc.

```
CLASS AbstractInterpreter EXTENDS Interpreter {
…
 FUNCTION interpr(p:Procedure):AbstractValue {
  worklist:list(Statement) := p.statements;
  WHILE (worklist != NULL) {
   SELECT current:Statement FROM worklist;
   // backward propagation from current.successors to current
   FORALL succ in current.ControlFlowGraph.successors {
    NewValue := meet( succ.value );
   }
   // test whether fixpoint is reached
   IF (NewValue != current.value) {
    current.value = NewValue;
    worklist += current.ControlFlowGraph.predecessors;
   }
  }
  RETURN p.statements.last.value;
 }
}
```

## Interprocedural Control Flow Graphs and Valid Paths

- Flow Functions f# can be on Nodes f#(n), or on Edges f#(e)
- **Interprocedural edges** are call edges from caller to callee
- **Local edges** are within a procedure from "call" to "return"
- Problem: not all interprocedural paths will be taken at the run time of the program
  - Call and return are *symmetric*
  - From whereever I enter a procedure, to there I leave
- An **interprocedurally valid path** respects the symmetry of call/return



---

## Interprocedural Problems

- Non-valid interprocedural paths invalidate the coincidence for the interprocedural case
- Knoop found a restricted one [CC92]:
  - No global parameters of functions
  - Restricted return behavior

---

## Abstract Interpretation on Other Languages

- A.I can be applied also to other languages on M2:
  - Query languages, also logic languages
  - Constraint languages
  - Transformation languages (term and graph rewrite languages)

---

# 22.3 Attribute Grammars for Interpreters on Syntax Trees

- Interpretation and abstract interpretation on syntax trees

# Attribute Grammars (AG)

- An **attribute grammar** describes an interpreter on a syntax tree (a hierarchical program representation)
  - The syntax tree is described by a context-free grammar (e.g., in EBNF)
  - The nodes of the program in the syntax tree are augmented with values, **attributes**. The resulting data structure is called **attributed syntax tree (AST)**
    - Graph representations are not possible in pure AGs
  - There is a set of **attribution rules (attribute equations)** which define interpretation functions on the syntax tree
  - Usually, the rules are interpreted with recursion along the syntax tree
- *An attribute grammar describes an abstract interpreter*, if the values are from an abstract domain (e.g., from a type system, interval ranges, etc.)
  - Then, the set of **attribution rules (attribute equations)** define abstract interpretation functions on the syntax tree
- Because the underlying program representation is hierarchic, often
  - AG-based interpreters can be proven to terminate
  - can be compiled to code, instead of interpreted (pretty fast)

AG-based abstract interpreters can analyze syntax trees by abstract interpretation

---

# Reference Attribute Grammars (RAG)

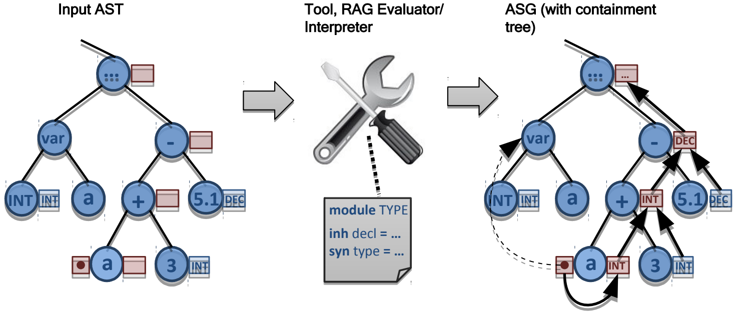
- A **reference attribute grammar** describes an interpreter on a syntax tree with references to other branches (an overlay graph)
  - The syntax tree is described by a context-free grammar (e.g., in EBNF or XSD)
  - The references are described separately (e.g., links in XSD)
    - Graph representations *are* possible in pure AGs
  - The nodes of the program in the syntax tree are augmented with values, **attributes**
  - There is a set of **attribution rules (attribute equations)** which define interpretation functions on the syntax tree
  - Usually, the rules are interpreted with recursion along the syntax tree *plus* side recursions along the references
- *A reference attribute grammar describes an abstract interpreter*, if the values are from an abstract domain (e.g., from a type system, interval ranges, etc.)
  - Then, the set of **attribution rules (attribute equations)** define abstract interpretation functions on the syntax tree

RAG-based abstract interpreters can analyse and interpret models

---

# What is a Reference Attribute Grammar (RAG)?

- **Attributes compute static semantics over syntax trees** [Knuth68]
  - Basis: (context-free) grammars + attributes + semantic functions
- **Attribute types:**
  - **Inherited attributes** (inh): Top-down value dataflow/computation (IN-parameters)
  - **Synthesized attributes** (syn): Bottom-up value dataflow/computation (OUT)
  - **Collection attributes** (coll): Collect values freely distributed over the AST
  - **Reference attributes**: Compute references to exisiting nodes in the AST
- **Tool:** www.jastadd.org

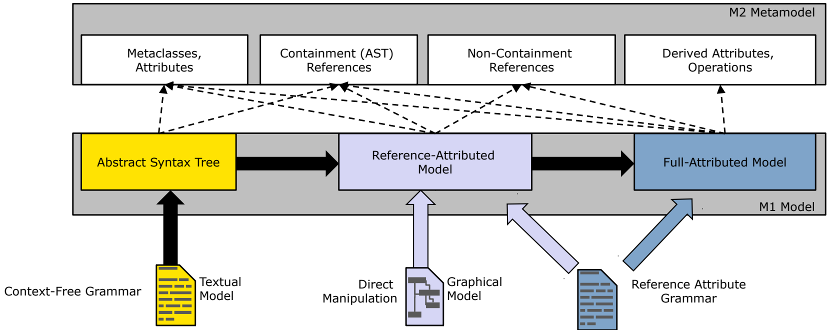


module TYPE
inh decl = ...
syn type = ...

---

# EMOF and Reference Attribute Grammars

- Ecore (EMOF) models are $\underbrace{\text{ASTs}}_{\text{syntactic interface}}$ with $\underbrace{\text{cross-references and derived information!}}_{\text{semantic interface}}$
- Ecore (EMOF) metamodels can be built around a **tree**-based abstract syntax used by
  - Tree iterators, tree editors, transformation tools, interpreters
  - Tools use the tree structure to derive all other information (e.g., resolving cross references, partial interpretation)
  - Graphical editors use the tree structure to manage user created object hierarchies, cross references and values therein and to compute read-only information (e.g., cross references, derived values)



# EMOF and Reference Attribute Grammars

- EMOF models are ASTs with cross-references and derived information!
- **Tool:** www.jastemf.org

| AST in Ecore | AST in RAGs |
|---|---|
| EClass | AST Node Type |
| EReference[containment] | Nonterminal |
| EAttribute[non-derived] | Terminal |

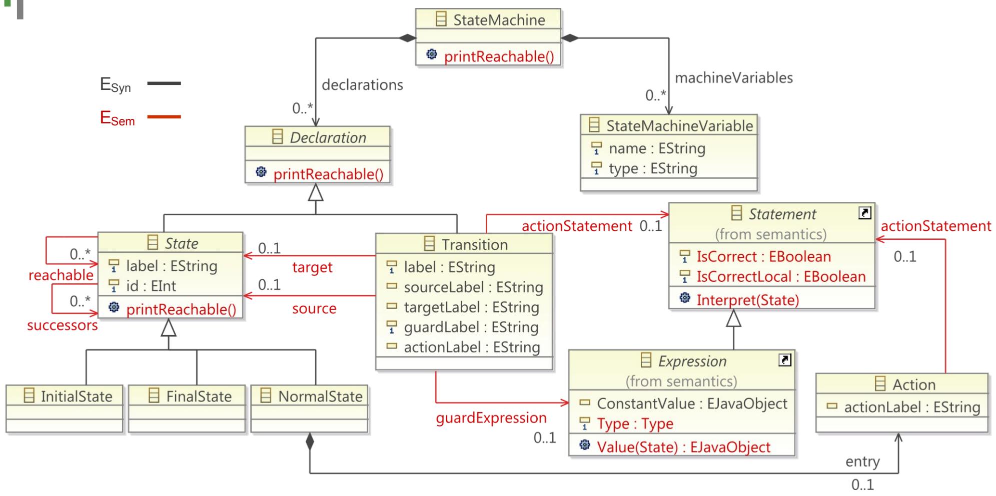
$E_{Syn}$

| Semantics Interface in Ecore | Semantics in RAGs |
|---|---|
| EAttribute[derived] | [synthesized\|inherited] attribute |
| EAttribute[derived,multiple] | collection attribute |
| EReference[non-containment] | collection attribute, reference attribute |
| EOperation[side-effect free] | [synthesized\|inherited] attribute |
| EReference[containment,derived] | Nonterminal  attribute |

$E_{Sem}$

---

# Example: Statechart Metamodel in EMOF

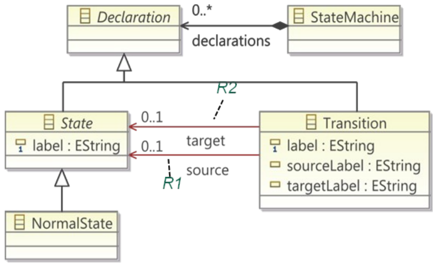


(Ecore-based, extended version of Statemachine example in Hedin, G.: Generating Language Tools with JastAdd. In: GTTSE '09. LNCS,Springer (2010), see also www.jastemf.org)

---

# Example: Statechart Metamodel Name Analysis

**AST specification (partial):**



**abstract** State:Declaration ::= <label:String>;
NormalState:State;
Transition:Declaration ::=<label:String>
<sourceLabel:String><targetLabel:String>;

**Attribution example:**

**syn lazy** State Transition.source() = lookup(getSourceLabel()); *// R1*
**syn lazy** State Transition.target() = lookup(getTargetLabel()); *// R2*
**inh** State Declaration.lookup(String label); *// R3*
**eq** StateMachine.getDeclarations(int i).lookup(String label) { … } *// R4*
**syn** State Declaration.localLookup(String label) =
(label==getLabel()) ? this : null; *// R5*

(Ecore-based, extended version of Statemachine example in Hedin, G.: Generating Language Tools with JastAdd. In: GTTSE '09. LNCS,Springer (2010), see also www.jastemf.org)

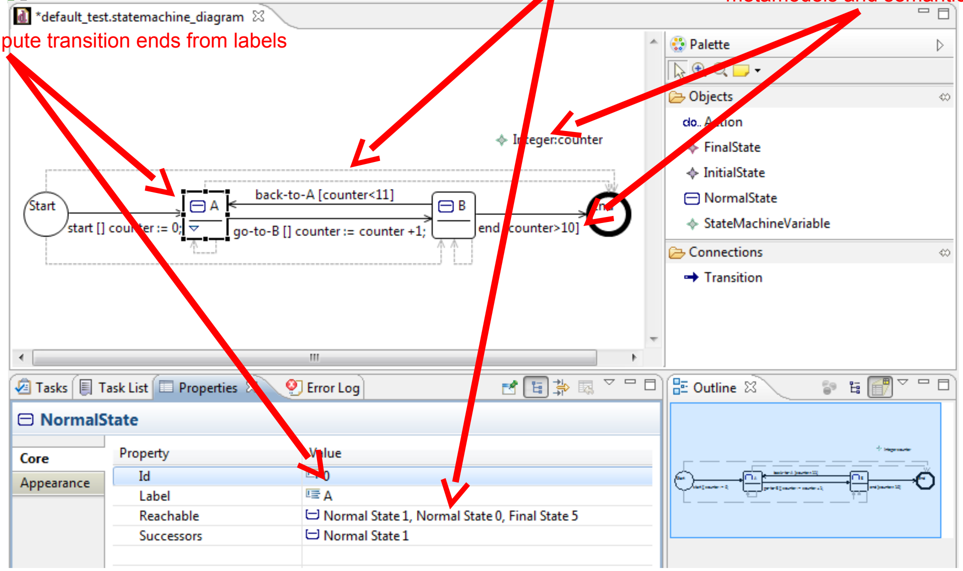
---

# Example: Statechart Runtime

compute closure

reuse of
metamodels and semantics

compute transition ends from labels



| Property | Value |
|---|---|
| Id | 0 |
| Label | A |
| Reachable | Normal State 1, Normal State 0, Final State 5 |
| Successors | Normal State 1 |

# The End

- Some slides are courtesy to Sven Karol

- Explain the differences of an interpreter and an abstract interpreter
- What are the differences of an abstract interpreter and an attribute grammar?
- Why is a reference attribute grammar more expressive than a pure AG?
- What happens at a control-flow join during an abstract interpretation?