Johannes Mey[1], René Schöne[1], Uwe Aßmann[1], Niklas Fors[2], Görel Hedin[2]

[1] Technische Universität Dresden
[2] Lund University

Reference Attribute Grammars with JastAdd

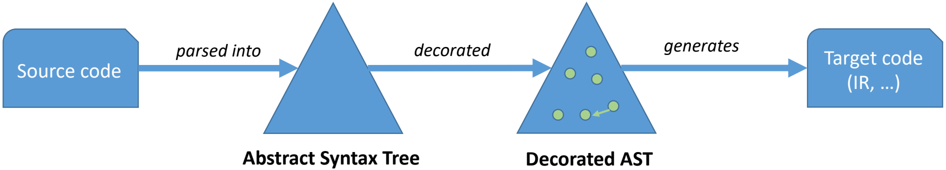
Dresden, January 14, 2021

Attribute Grammars (AGs)

Attributes

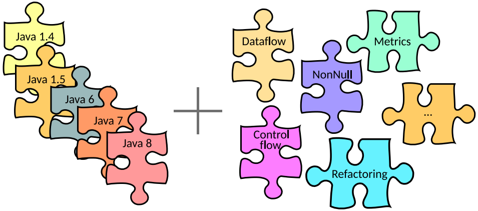
- compute *derived* properties of nodes in abstract syntax tree
- proposed by Donald Knuth in 1968
- *references* **(RAGs)** simplify navigation in attribute definitions

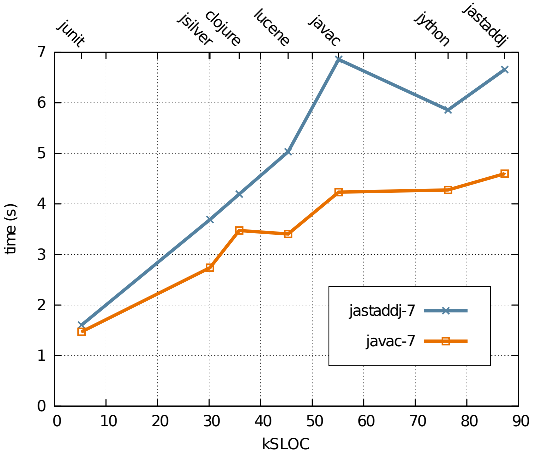
Today's Focus: JastAdd RAG system



JastAdd Applications

ExtendJ





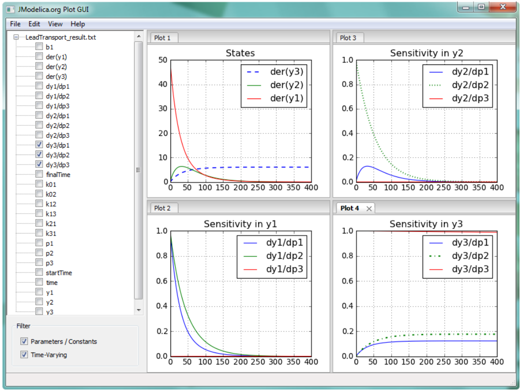
— Java 8 compiler with many extensions

— performance compared to OpenJDK

JastAdd Applications

JModelica

- open source modelica compiler
- maintained and used by company Modelon

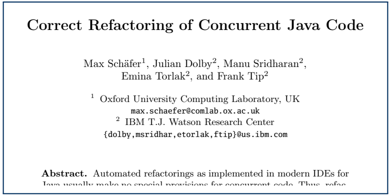


JModelica User Guide, jmodelica.org

Refactoring and JastAdd

Previous work by Max Schäfer

- papers at ECOOP, OOPSLA, ICSE, . . .
- dissertation
- JastAdd refactoring tool JRRT
  `https://code.google.com/archive/p/jrrt/`



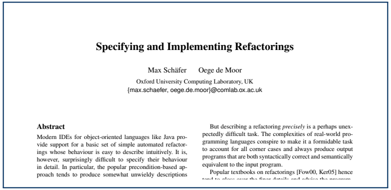
Correct Refactoring of Concurrent Java Code

Max Schäfer[1], Julian Dolby[2], Manu Sridharan[2],
Emina Torlak[2], and Frank Tip[2]

[1] Oxford University Computing Laboratory, UK
max.schaefer@comlab.ox.ac.uk
[2] IBM T.J. Watson Research Center
{dolby,msridhar,etorlak,ftip}@us.ibm.com

Abstract. Automated refactorings as implemented in modern IDEs for Java usually make no special provisions for concurrent code. Thus, refac-

[Schäfer et al., 2010]



Specifying and Implementing Refactorings

Max Schäfer Oege de Moor
Oxford University Computing Laboratory, UK
{max.schaefer, oege.de.moor}@comlab.ox.ac.uk

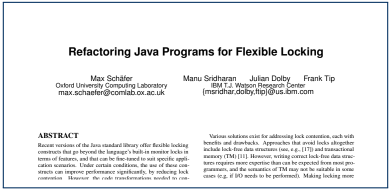
Abstract

Modern IDEs for object-oriented languages like Java provide support for a basic set of simple automated refactorings whose behaviour is easy to describe intuitively. It is, however, surprisingly difficult to specify their behaviour in detail. In particular, the popular precondition-based approach tends to produce somewhat unwieldy descriptions

But describing a refactoring *precisely* is a perhaps unexpectedly difficult task. The complexities of real-world programming languages conspire to make it a formidable task to account for all corner cases and always produce output programs that are both syntactically correct and semantically equivalent to the input program.

Popular textbooks on refactorings [Fow00, Ker05] hence tend to gloss over the finer details and advise the program-

[Schaefer and de Moor, 2010]



Refactoring Java Programs for Flexible Locking

Max Schäfer
Oxford University Computing Laboratory
max.schaefer@comlab.ox.ac.uk

Manu Sridharan Julian Dolby Frank Tip
IBM T.J. Watson Research Center
{msridhar,dolby,ftip}@us.ibm.com

ABSTRACT

Recent versions of the Java standard library offer flexible locking constructs that go beyond the language's built-in monitor locks in terms of features, and that can be fine-tuned to suit specific application scenarios. Under certain conditions, the use of these constructs can improve performance significantly, by reducing lock contention. However, the code transformations needed to con-

Various solutions exist for addressing lock contention, each with benefits and drawbacks. Approaches that avoid locks altogether include lock-free data structures (see, e.g., [17]) and transactional memory (TM) [11]. However, writing correct lock-free data structures requires more expertise than can be expected from most programmers, and the semantics of TM may not be suitable in some cases (e.g, if I/O needs to be performed). Making locking more

[Schafer et al., 2011]

JastAdd Exercise 1

Presentation

- a *practical* introduction to RAGs
- the JastAdd compiler and its infrastructure

Homework

- simple expression language
- construction of a small grammar
- writing some attributes

Reference Attribute Grammars

An introduction to

- grammar specification
- syntax trees
- attribute specification

Reference Attribute Grammars

An introduction to

- grammar specification
- syntax trees
- attribute specification

⚠ Disclaimer

- focus on the JastAdd understanding
- no parsing, only syntax trees

JastAdd Grammar

Elements:

Nonterminals	`A   B   SomeName`
Terminals/Tokens	`<X:int>` `<Y>` (default type `String`)

Production rules:

Child nodes	`A ::= C First:B Second:B;`
List/optional children	`B ::= C* [MyD:D];`
Terminals	`C ::= <TerminalSymbol:String>;`
Abstract nonterminals	`abstract E ::= <Name>;`

Inheritance	`F : E ::= <Value:int>;`
	`G : E ::= H <Value:float>;`

Empty productions	`H ::= /* right side can be empty! */;`
	`H; // also valid`

JastAdd Grammar

Generated interface for nonterminals:

regular nonterminal	`class A { /* */ }`
abstract nonterminal	`abstract class E { /* */ }`

Generated child accessors (within nonterminal class):

unnamed child	`public C getC() { /* */ }`
named child	`public C getMyChild() { /* */ }`
list children	`public C getC(int index) { /* */ }`
optional child	`public boolean hasMyD() { /* */ }`
terminal	`public String getTerminalSymbol() { /* */ }`

JastAdd Attributes

Attributes

- proposed by Donald Knuth [Knuth, 1968]
- computed properties of tree
- side-effect free
- *declaration* and *definition*
- different types with different information flow

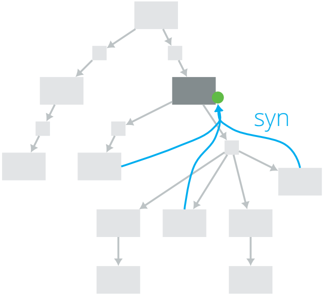
JastAdd Attributes
Synthesized Attributes

Information from the subtree: *synthesized* attributes

- Must be defined for declared type
- If type is abstract for all non-abstract sub-types
- Example:

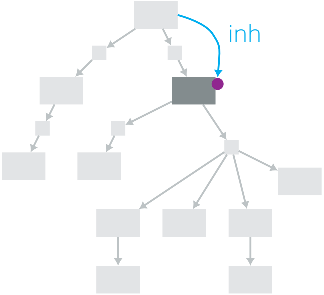
```
syn boolean TrackElement.isSegment();

// attribute equations
eq Switch.isSegment()  { return false; }
eq Segment.isSegment() { return true;  }
```



JastAdd Attributes

Inherited Attributes



Information from parent: *inherited* attributes

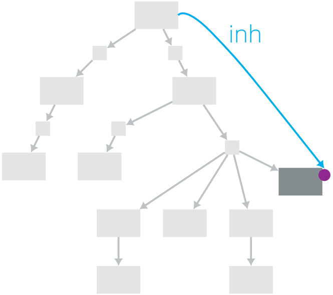
- Must be defined on an ancestor
- Example:

```
inh Region Element.containingRegion();

// attribute equation
eq Region.getElement(int index).containingRegion() = this;
```

JastAdd Attributes

Inherited Attributes



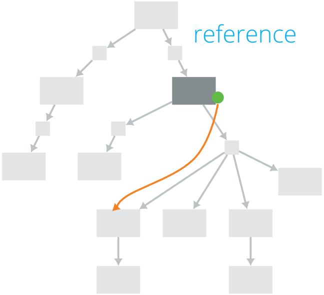
Information from parent: *inherited* attributes

- Must be defined on an ancestor
- Example:

```
inh Region Element.containingRegion();

// attribute equation
eq Region.getElement(int index).containingRegion() = this;
```

JastAdd Attributes
Reference Attributes



Existing nodes as attribute values

- can be any type of attribute (syn, inh, coll)
- Example:

```
inh Region Element.containingRegion();

// attribute equation
eq Region.getElement(int index).containingRegion() = this;
```

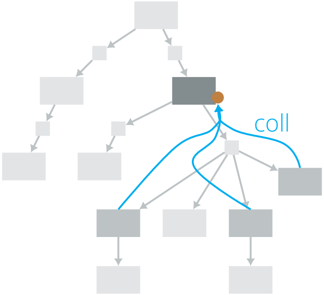
JastAdd Attributes
Collection Attributes

Collecting information: *collection* attributes

- Must be defined for declared type
- If type is abstract for all non-abstract sub-types
- Example:

```
coll Set<Element> Region.coolElements() [new HashSet];

// contribution to collection
Element contributes this
  when isCool()
  to Region.coolElements();
```



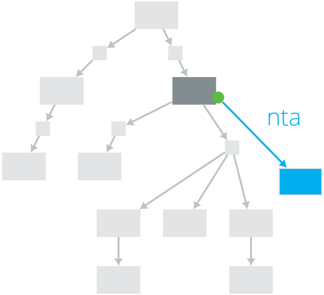
JastAdd Attributes
Nonterminal Attributes

Building new subtrees: *nonterminal* attributes

- Also: *higher order attribute*
- Subtrees must be **new** objects!
- Example:

```
// grammar excerpt
A ::= /* ... */;
B ::= <Name:String>;
// declaration
syn nta B A.getB();

// attribute equation
eq A.getB() {
  B b = new B();
  b.setName("Boaty McBoatface");
  return b;
}
```



JastAdd Attributes

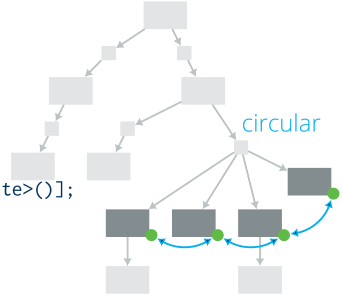
Circular Attributes

Fix-Point Computation: *circular* attributes

- can call itself
- computed iteratively
- example:

```
syn Set<State> State.reachable() circular [new HashSet<State>()];

eq State.reachable() {
  HashSet<State> result = new HashSet<State>();
  for (State s : successors()) {
    result.add(s);
    result.addAll(s.reachable());
  }
  return result;
}
```



Attributes in JastAdd

synthesized: information from subtree

inherited: information from parents

reference: any kind of attribute can be reference; points to other nonterminal

collection: information from nodes of certain type

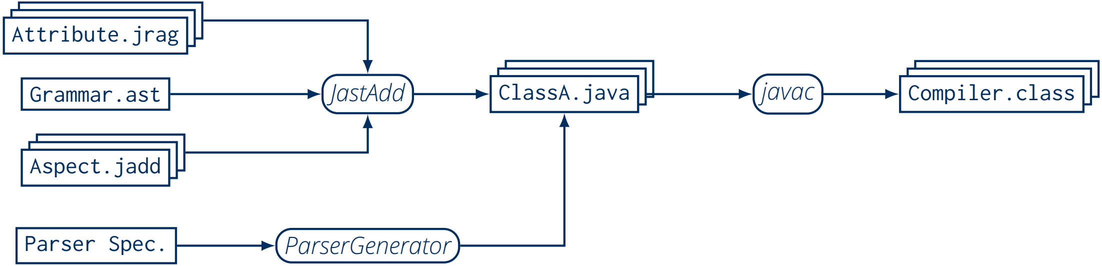
nonterminal: can be synthesized or inherited; compute new subtrees

circular: any kind of attribute can be circular; iterative fixpoint computation

The JastAdd System

RAG to Java Code Generation

- nonterminals → classes
- attributes → methods
- additional magic



Other JastAdd Features

Attribute Evaluation

- caching of attribute values
  - attribute values are memoized
  - configurable on per-equation level
- incremental attribute evaluation
  - dynamic attribute dependency graph

Aspect-Oriented Programming Features

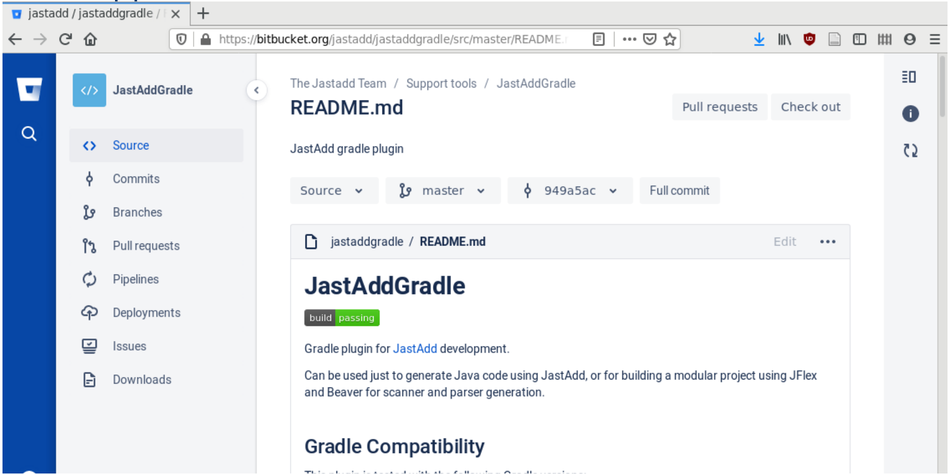
- additional methods can be woven into classes
- methods and attributes can be refined

Other nice features

- debugging and tracing support

Build Tools: JastAddGradle

Tool Support


jastadd / jastaddgradle / F
https://bitbucket.org/jastadd/jastaddgradle/src/master/README.
JastAddGradle
Source
Commits
Branches
Pull requests
Pipelines
Deployments
Issues
Downloads
The Jastadd Team / Support tools / JastAddGradle
Pull requests
Check out
Source
master
949a5ac
Full commit
jastaddgradle / README.md
Edit
build passing

README.md

JastAdd gradle plugin

JastAddGradle

Gradle plugin for JastAdd development.

Can be used just to generate Java code using JastAdd, or for building a modular project using JFlex and Beaver for scanner and parser generation.

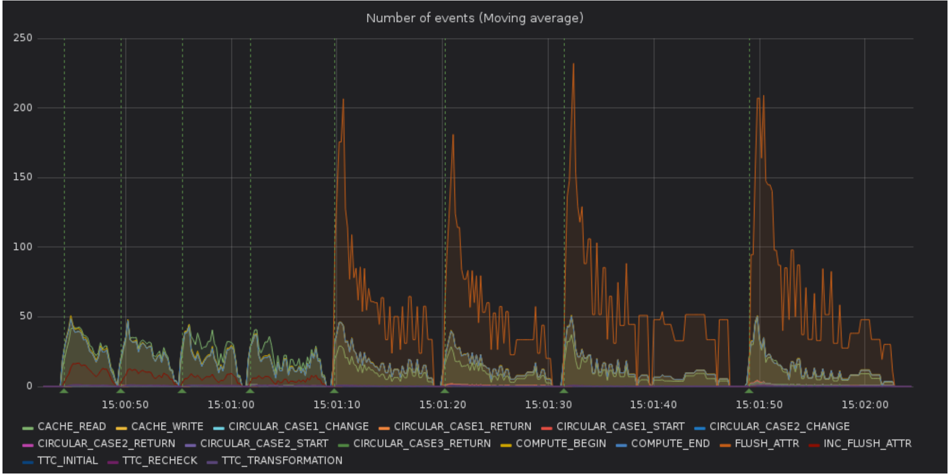
Gradle Compatibility

This plugin is tested with the following Gradle versions:

JastAdd Gradle plugin

Tracing API

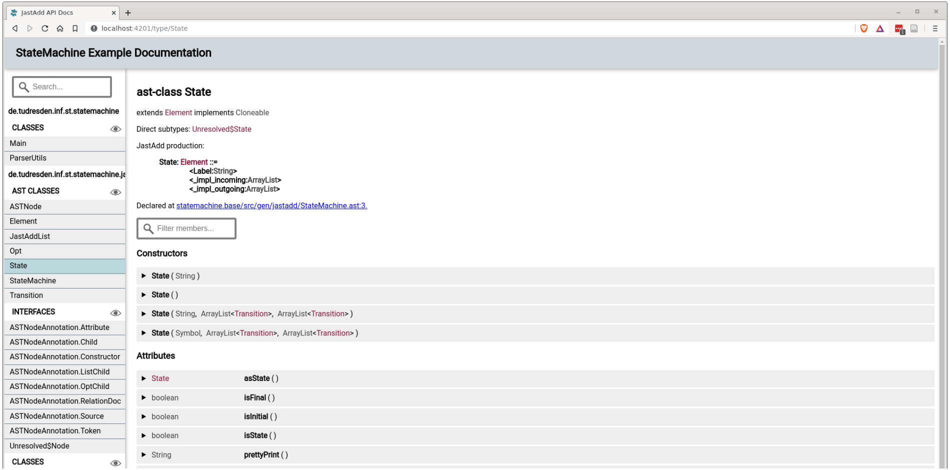
Tool Support



Grafana visualization of events created by the JastAdd tracing API

Documentation Generation: RAGdoc

Tool Support



StateMachine Example Documentation

Search...

de.tudresden.inf.st.statemachine

CLASSES

- Main
- ParserUtils

de.tudresden.inf.st.statemachine.ja

AST CLASSES

- ASTNode
- Element
- JastAddList
- Opt
- State
- StateMachine
- Transition

INTERFACES

- ASTNodeAnnotation.Attribute
- ASTNodeAnnotation.Child
- ASTNodeAnnotation.Constructor
- ASTNodeAnnotation.ListChild
- ASTNodeAnnotation.OptChild
- ASTNodeAnnotation.RelationDoc
- ASTNodeAnnotation.Source
- ASTNodeAnnotation.Token
- Unresolved$Node

CLASSES

ast-class State

extends Element implements Cloneable

Direct subtypes: Unresolved$State

JastAdd production:

```
State: Element ::=
        <Label:String>
        <_impl_incoming:ArrayList>
        <_impl_outgoing:ArrayList>
```

Declared at statemachine.base/src/gen/jastadd/StateMachine.ast:3.

Filter members...

Constructors

- ▶ **State** (String)
- ▶ **State** ( )
- ▶ **State** (String, ArrayList<Transition>, ArrayList<Transition>)
- ▶ **State** (Symbol, ArrayList<Transition>, ArrayList<Transition>)

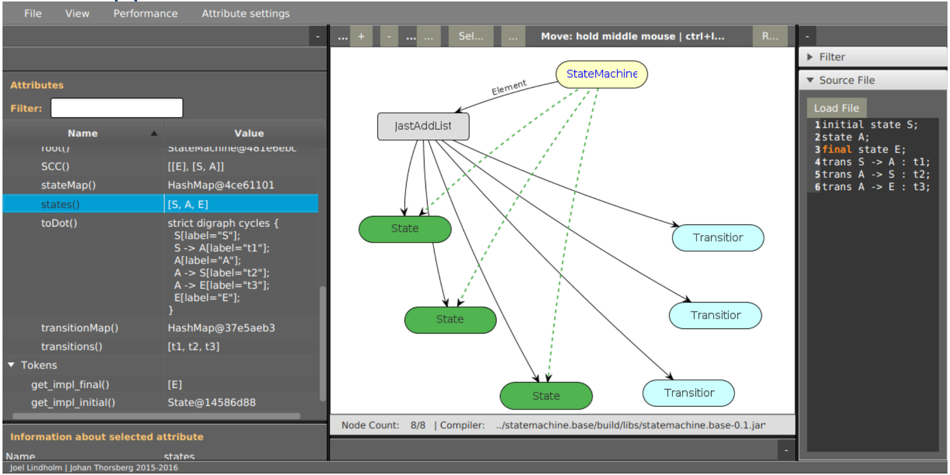
Attributes

▶ State	**asState** ( )
▶ boolean	**isFinal** ( )
▶ boolean	**isInitial** ( )
▶ boolean	**isState** ( )
▶ String	**prettyPrint** ( )

RAGdoc documentation including links to source code

Visualization and Debugging: DrAST

Tool Support



Attributes

Filter:

Name	Value
root()	StateMachine@481e6ebc
SCC()	[[E], [S, A]]
stateMap()	HashMap@4ce61101
states()	[S, A, E]

toDot()	strict digraph cycles {
	S[label="S"];
	S -> A[label="t1"];
	A[label="A"];
	A -> S[label="t2"];
	A -> E[label="t3"];
	E[label="E"];
	}

transitionMap()	HashMap@37e5aeb3
transitions()	[t1, t2, t3]
Tokens	
get_impl_final()	[E]
get_impl_initial()	State@14586d88

```
initial state S;
state A;
final state E;
trans S -> A : t1;
trans A -> S : t2;
trans A -> E : t3;
```

Node Count: 8/8 | Compiler: ../statemachine.base/build/libs/statemachine.base-0.1.jar

Information about selected attribute

Name states

Joel Lindholm | Johan Thorsberg 2015-2016

DrAST visualization with computed attributes

JastAdd
Important Information

JastAdd

- Website with reference manual and bibliography `www.jastadd.org`
- Source code `https://bitbucket.org/jastadd/jastadd2`

Build tool support

- gradle/maven/…packages:
  - `org.jastadd:jastadd`, `org.jastadd:jastaddparser`, `org.jastadd:jastaddgradle`
- gradle plugin: `https://bitbucket.org/jastadd/jastaddgradle/`

DrAST

- code and doc: `https://bitbucket.org/jastadd/drast/`

RagDoc

- code and doc: `bitbucket.org/extendj/ragdoc-builder/` , `bitbucket.org/extendj/ragdoc-view/`

Questions so far?

References I

Knuth, D. E. (1968).
Semantics of context-free languages.
Mathematical systems theory, 2(2):127–145.

Schaefer, M. and de Moor, O. (2010).
Specifying and implementing refactorings.
In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 286–301. ACM.
event-place: Reno/Tahoe, Nevada, USA.

Schafer, M., Sridharan, M., Dolby, J., and Tip, F. (2011).
Refactoring java programs for flexible locking.
In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 71–80.
ISSN: 0270-5257.

Schäfer, M., Dolby, J., Sridharan, M., Torlak, E., and Tip, F. (2010).
Correct refactoring of concurrent java code.
In D'Hondt, T., editor, *ECOOP 2010 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 225–249. Springer.