MASTER'S THESIS 2019

Contributions to Parallelizing JastAdd Compilers

Joachim Wedin

ISSN 1650-2884
LU-CS-EX 2019-26

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2019-26

Contributions to Parallelizing JastAdd Compilers

Joachim Wedin

Contributions to Parallelizing JastAdd Compilers

Joachim Wedin
joachim.wedin@gmail.com

September 19, 2019

Master's thesis work carried out at Modelon AB.

Supervisors: Jesper Öqvist, jesper.oqvist@cs.lth.se
Christoph Reichenbach, christoph.reichenbach@cs.lth.se
Axel Mårtensson, axel.martensson@modelon.com

Examiner: Görel Hedin, gorel.Hedin@cs.lth.se

Abstract

Reference Attribute Grammars (RAGs), an extension to canonical attribute grammars, allow attribute to be references to any node in the syntax tree. In previous work, RAGs were extended to support concurrent attribute evaluation.

This thesis investigates the possibility of applying these extensions to existing RAG compilers, such as JModelica.org, to improve compilation speed. We focus on one of the requirements for concurrent attribute evaluation, which is that attributes must be observationally pure. Our endeavours lead to the formulation of a new definition of functional purity, useful for parallelizing RAG compilers. We present a new analysis which combines simplified code generation for the JastAdd meta-compiler with an existing purity analysis tool extended with further capabilities. We show that our simplified code generation based purity analysis reduces, on average, the number of warnings by 60% and analysis time by 45% for ExtendJ and JModelica.org.

Keywords: Parallelization, Concurrency, JastAdd, Reference Attribute Grammars, JModelica.org, Functional Purity Analysis

Acknowledgements

I would like to express my deep gratitude to my supervisors Jesper, Christoph and Axel for making this project possible and for their invaluable guidance throughout the project. I also want to thank Jiachen for allowing me to use his tool for the project.

To my partner Larissa, I would never have been able to do this without your love and support. Your encouragements in tough moments were very important to me and will always be remembered. And to my son Gustav, thank you for giving me the endless joy of being a father.

Finally, I would like to thank all the people at Modelon AB who made this project possible. You were all very nice to me and gave me all the support I could of asked for.

Contents

1 Introduction **7**

2 Background **9**
2.1 Reference Attribute Grammars . . . 9
2.2 The JastAdd Meta-Compiler . . . 10
2.2.1 Higher Order Attributes . . . 10
2.2.2 Rewrites . . . 11
2.2.3 Collection Attributes . . . 11
2.2.4 Circular Attributes . . . 12
2.2.5 Concurrent Attribute Evaluation . . . 12
2.3 Race Conditions . . . 12
2.4 Thread Safety . . . 13
2.5 Functional Purity . . . 15
2.6 ExtJPureChecker . . . 19
2.7 Purano . . . 20

3 Parallelization in JModelica.org **21**
3.1 ExtJPureChecker . . . 21
3.2 JastAdd-Purity-Analyzer . . . 24
3.2.1 Rewrites . . . 27
3.2.2 Higher Order Attributes . . . 29
3.2.3 Collection Attributes . . . 29
3.2.4 Circular Attributes . . . 30
3.2.5 Analysis by Purano . . . 31
3.2.6 Extensions to Purano . . . 33
3.3 Maintaining Separate ASTs . . . 36

4 Evaluation **39**
4.1 JModelica.org Purity Survey . . . 39
4.1.1 Memoization . . . 40

4.1.2 Aborting Fixed Point Iteration 40
4.1.3 Accessing Inherited Attributes 41
4.1.4 Impure Rewrite Equations 41
4.1.5 Rewriting Final Nodes 42
4.1.6 Logging 42
4.1.7 Error Checking 42
4.1.8 Constant Propagation 43
4.2 Purity Analyzis of JastAdd Attributes 43
4.3 Compile Time in JModelica.org 45

5 Conclusion and Future Work **47**
5.1 Conclusion 47
5.2 Future Work 48

Bibliography **53**

Chapter 1
Introduction

Compilation speed is important for many applications [20, 3, 17, 19]. Slow compilation can, for example, lead to delays in testing which wastes time and increases latency in finding compile errors and warnings. The latter is especially problematic since these procedures are typically used in IDE productivity features. At best, increased latency will just make the IDE feel slow. At worst, increased latency leads to the IDE features being considered useless because the latency is unacceptable.

As the improvement rate of processor clock speeds is slowing down the trend in the computer industry has, in recent years, been to improve performance with more parallelism. Modern processors are successively gaining an increased number of cores. Therefore, many companies and researchers are utilizing parallel processing on multi-core architectures in order to speed up computations [21]. Similarly, the goal of this thesis is to investigate possibilities of utilizing parallelism to speed up JastAdd compilers, such as JModelica.org.

JModelica.org is an open source compiler for the Modelica Language [22]: a declarative, object oriented and equation based language used for simulating complex physical systems. The JModelica.org compiler is implemented with the JastAdd meta-compiler [11] which supports reference attribute grammars (RAGs) and is designed to support extensible implementations of compilers. JModelica.org has been extended with optimization features as a JastAdd compiler extension. Other examples of extensible compilers built with JastAdd include ExtendJ [25], Fuji [16] and abc [2].

Recent extensions to RAGs include concurrent attribute evaluation [26]. These contributions have opened the door for parallelizing JastAdd compilers like JModelica.org. One of the requirements for concurrent attribute evaluation is that attributes are observationally pure [23] so that attribute evaluation can safely be reordered and parallelized. In order to apply concurrent attribute evaluation to JModelica.org any existing purity-breaking side effects inside attributes need to be removed. While side effects should not exist in JastAdd attributes, the attribute

equations are written in plain Java code which does not prevent side effects. Side effects are sometimes unintentionally introduced due to accidental aliasing of references, for example.

In order to ensure side-effect freedom for JModelica.org, parts of this thesis focuses on analyzing attribute purity in JModelica.org and other JastAdd-based compilers. Furthermore, we develop a new definition of functional purity specifically for concurrent applications which removes some sources of concurrent side-effects which would otherwise be allowed (see Section 2.5). We have applied existing purity analysis tools to JModelica.org to investigate which attributes have side effects that would need to be replaced in order to parallelize the compiler. For analyzing purity in JModelica.org we have used the tools EPC[13] and Purano [35].

We also present a new analysis that combines our new simplified code generation for the JastAdd meta-compiler with a version of Purano that has been extended with further capabilities. The analysis is part of our new tool JastAdd-Purity-Analyzer that analyzes Reference Attribute Grammars for concurrent purity.

We also present a different approach for parallel attribute evaluation in JModelica.org where separate versions of the AST are maintained. Each thread that evaluates attributes owns a separate copy of the AST. The goal is to ensure that threads only operate on thread local data, which guarantees safe parallel attribute evaluation. We apply this approach to interactive Modelica editing where two separate ASTs are maintained to improve the responsiveness of the compiler. The research questions that the thesis aims to answer are the following:

- **RQ1**: Can compile time be reduced in a JastAdd compiler by parallelization?
- **RQ2**: How and where are concurrently impure attributes used in JModelica.org?
- **RQ3**: Can parts of JModelica.org be transformed to be thread safe?
- **RQ4**: How can JastAdd attributes be analyzed for concurrent purity?

Chapter 2
Background

In this chapter we will introduce some concepts that are important for this thesis. We begin by introducing Reference Attribute Grammars (RAGs) and the related tool JastAdd for building extensible compilers. Furthermore, we introduce some concepts for multi-thread programs, namely data races, race conditions and thread safety. Finally, we introduce functional purity and two existing functional purity analysis tools for Java, ExtJPureChecker and Purano.

2.1 Reference Attribute Grammars

In Attribute Grammars, originally introduced by Knuth [15], a context-free grammar is decorated with so-called attributes on nonterminals. A context-free grammar (CFG) defines all the possibles strings of a language. A CFG consists of terminal and nonterminal symbols, production rules and finally a starting symbol. A production rule is on the form $A \rightarrow B_1\ B_2\ \ldots\ B_n$ where A is a nonterminal and B_i where $i \in \{1 \ldots n\}$ can be either a terminal or a nonterminal symbol.

The attributes of an Attribute Grammar are directed equations computing semantic information about nodes in the Abstract Syntax Tree (AST). The AST is a tree representation of a sentence in a context-free language which has been derived using the production rules of a CFG. The AST incorporates, unlike the syntax tree, only the essential parts which are needed to represent the sentence that the AST corresponds to.

Knuth's work introduced *synthesized* and *inherited* attributes. A synthesized attribute equation is defined using only the node and other attributes on the node. Inherited attributes depend on attributes defined in its ancestors[1].

[1] It should be noted that here the term inherited has no relation to inheritance in object oriented programming

Reference Attribute Grammars (RAGs) extend Knuth's attribute grammars with reference valued attributes [7]. That is, attribute values are allowed to be references to any node in the AST. With reference valued attributes it becomes much easier to implement attributes with non-local dependencies, for example to compute (non-local) name bindings. The addition of reference attributes makes it feasible to build larger compilers using attribute grammars. There have been several large and complex compilers developed for languages like Modelica and Java based on RAGs [12, 25].

2.2 The JastAdd Meta-Compiler

JastAdd [9] is a meta-compiler supporting RAGs. JastAdd supports both declarative specification with RAGs and an imperative computation with normal Java code. JastAdd supports so called aspect oriented programming, meaning that code can be defined in a modular way in aspects.

An example of a JastAdd grammar with two node types, `A` and `B`, and a synthesized and inherited attribute is given below.

```
// Abstract Grammar
A ::= left:B right:B;
B;

// Attributes
aspect F {
    syn int A.x() = 0;
    inh int B.y();
    eq A.getLeft().y() = 1;
    eq A.getRight().y() = 2;
}
```

Here the abstract grammar specifies that there exists a node type `A` and that `A` has two children. The first child is named "left" and is of type `B`. The second child is named "right" and is also of type `B`. The abstract grammar also declares node type `B`.

The aspect above defines a synthesized (`syn`) attribute and an inherited (`inh`) attribute. Attribute `x` is a synthesized attribute defined on `A` nodes with an equation giving it the value 0. Attribute `y` is declared as an inherited attribute on the `B` node. The value of attribute `y` is defined by an ancestor, in this case a node of type `A`. Depending on the context the attribute `y` will have either the value 1 or 2. If `B` is the left child of `A` the value is 1 and if it is the right child the value is 2. This behaviour is visualized in Figure 2.1.

2.2.1 Higher Order Attributes

It is sometimes useful to dynamically compute nodes in the AST. This can be useful, for example, to reify implicit constructs or to build temporary simplified expressions. For this, we can use a type of attribute known as Higher Order Attributes (HOAs) [33].

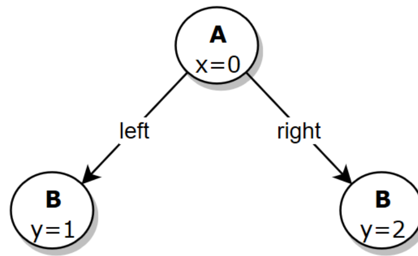

Figure 2.1: The value of the inherited attribute `y` depends on the position of the `B` node.

HOAs can be used to compute any subtree of the AST with an attribute. The computed subtree of an HOA acts just like any other part of the AST and is *higher-order* in the sense that is is an attribute value which itself can be attributed. HOAs are only computed when needed (since they can in general compute unbounded trees [18]). In JastAdd, a HOA is declared using the keyword `nta` (for Non-Terminal Attribute, another name for HOAs).

2.2.2 Rewrites

JastAdd has support for automatic tree rewriting using the rewrite mechanism described by Ekman et al. [4]. The initial AST built by the parser can often be simplified, for example by classifying variable names by context or constant expression folding. These transformations can be applied by using rewrites instead of using imperative tree transformations. A parsed node is automatically transformed the first time it is accessed after parsing.

Rewrites and HOAs are both used for transforming the AST and in previous work it was shown that circular HOAs are equivalent to rewrites [31]. A rewritable node can in JastAdd be rewritten through a chain of rewrites if there exists a chain of rewrite rules (rewriting from one node type to next, etc.). The chain of rewrites should be non circular in order to find a final node state. JastAdd will treat the final node as a non rewritable node. This is achieved with a boolean flag which indicates that the node should not be rewritten further.

2.2.3 Collection Attributes

JastAdd supports a special type of attribute called collection attributes. The value of a collection attribute is composed from contributions which are added by contributor nodes in the AST if the right conditions hold. Contributors are always descendants of a special node called the collection *root*. That is, only the sub tree of the collection root is searched for contributors. The collection root is typically the node that holds the collection attribute value. However, the collection root can be changed to be, for example, the root node of the AST. Some example usages of collection attributes are: gathering compile time errors, finding the call locations of a function or finding all the sub-classes of a class.

2.2.4 Circular Attributes

Some problems in compilers are circular by nature. For example, computing the reachability of a function, i.e., all the functions reachable from that function in the call graph. Reachability can be reflexive, which is the case for, e.g., recursive functions. Special support in the attribute evaluator is needed for this kind of circularly defined attributes because otherwise the evaluation never terminate. JastAdd supports the definition of circular attributes as long as the possible values of the computation can be arranged in a lattice of finite height [5].

2.2.5 Concurrent Attribute Evaluation

Öqvist and Hedin presented a new extension to RAGs that support concurrent attribute evaluation [26]. The new algorithms have been shown to be lock and wait-free. The implementation of the algorithms has the added limitation that they are not wait-free since a non wait-free implementation of a map is used.

The concurrent attribute evaluation requires that the base AST is not modified after parsing. Therefore, normal rewrites are not supported for concurrent attribute evaluation since rewrites modify the parsed AST. However, Söderberg et al. have shown that rewrites are equivalent to circular HOAs [31] and they also present an algorithm for translating rewrite to circular HOAs in JastAdd. Circular HOAs are safe for concurrent attribute evaluation because they are HOAs which only compute new parts of the AST and do not modify the base AST [26]. The concurrent attribute evaluation also requires that semantic functions are terminating, observationally pure and that circular attributes are computable. Similar assumptions are made for JastAdd without concurrent attribute evaluation [8].

2.3 Race Conditions

A race condition occurs when two or more threads execute order-dependent operations without guaranteeing the correct ordering of those operations. Data races are a kind of race condition which are caused by concurrent reads/writes of data (memory). Race conditions can make the execution of the program non deterministic and lead to corrupted data and crashes [24].

When two threads or more share a variable and they access the variable concurrently, there is a risk of a data race when the following conditions hold:

- At least one access is a write
- The variable is not protected with mutual exclusion

The effect of a data race is not predictable as it depends on the timing of the execution [30]. Data races can cause data corruption or program crashes. Errors caused by race conditions can be hard to reproduce and lead to strange results [36].

Suppose we have the following simple (and faulty) Java class representing accounts in a bank system:

```
public class Account {
    private int balance;

    private synchronized int getBalance() {
        return balance;
    }

    private synchronized void setBalance(int amount) {
        balance = amount;
    }

    public int withdraw(int amount) {
        int tmp = getBalance();
        if (amount <= tmp) {
            setBalance(tmp - amount);
            return amount;
        }
        return 0;
    }

    public void deposit(int amount) {
        int tmp = getBalance();
        setBalance(tmp + amount);
    }
}
```

Suppose now that we were to simulate this bank system with two actors: an account holder and an bank employee. The account holder is withdrawing 50 SEK from their account with a current balance of 100 SEK. At the same time the bank employee is depositing the account holders monthly salary of 20000 SEK to the account. When `withdraw` and `deposit` are called concurrently and with a certain ordering of events the effect of the deposit can be overwritten, causing the account holder to not receive their salary. A visualization of how this happens can be seen in Figure 2.2. The program is clearly free from data races: two threads can never access the balance variable at the same time in the same `Account` object because the only way to access it is through synchronized methods[2]. The program does however have race conditions, which is why the account holder never received their salary.

2.4 Thread Safety

When building concurrent programs, a key property is thread safety. As defined by Goetz [6], a thread safe class behaves correctly when accessed by multiple threads regardless of the possible interleaving of the execution and without any synchronization between the threads. Pradel et al. [28] use the same definition but extend it further to refer to a class with only thread safe methods. A thread will for a thread safe class only observe behaviour equivalent to a linearization of the function calls. A linearization here means a possible sequential ordering (as if in a single thread)

[2]Only one thread at a time is able to run a synchronized method on a single object instance.

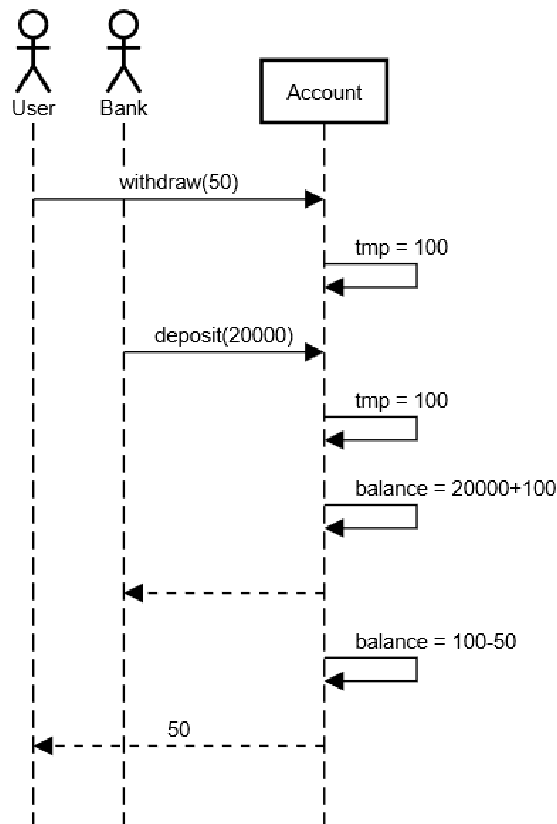

Figure 2.2: A sequence diagram showing an execution ordering that leads to the account holder in our bank system simulation losing their salary.

of all function calls from all the threads to a single object. The `Account` class defined in Section 2.3 is an example of a class that is not thread safe. We used the `Account` class in a bank system simulation and showed how calling `withdraw` and `deposit` concurrently could lead to an account holder losing their salary. There are two possible ways to linearize the calls `deposit` and `withdraw`, namely:

```
deposit → withdraw
withdraw → deposit
```

With the initial balance, withdrawal and deposit amounts as in Figure 2.2, both of these linearizations will lead to the final balance being 20050. Thus, the end result that was observed when calling the methods concurrently is not the result of a linearization of the function calls. Therefore, the `Account` is not thread safe.

There are different ways of writing thread safe code. All of the different approaches do, however, have one thing in common. They all, in their own way, ensure that code is free from race conditions. One way of dealing with race conditions is to use mutual exclusion. For example, the `Account` class would be made thread safe if the `withdraw` and `deposit` methods were synchronized. Making the methods synchronized would remove the existing race conditions. There would be no way for he threads to interleave since the methods would appear as atomic for the threads. Another way of achieving thread safety, by ensuring freedom from race

conditions, is by eliminating shared mutable data among threads. Race conditions can only occur when threads operate on common data which is mutable.

2.5 Functional Purity

An important property of a pure function is that it always returns the same value given the same inputs. Despite the importance of functional purity, there is no well established definition in the research community. The definitions found in literature have varying level of strictness.

Haiying [34] defines four different types of purity: strong, moderate, weak and once-impure. The following summary is by no means complete but highlights important aspects of these definitions. Strong purity requires that the function does not access the heap or static variables. The less restrictive definition, weakly pure, allows a function to write to and read from local objects on the heap but only read from pre-existing heap objects. Furthermore, an object is not allowed to escape from the function. The once-impure definition allows the function to be impure on the first invocation. A definition such as the strongly pure definition is of limited use for a language like Java. Generally a less restrictive definition is preferable.

Pearce [27] defines a pure method as a method that does not modify any field or array cell that existed before calling the method. Pearce also points out that any pure method must also only call pure methods. Yang et al. [35] define a function to be pure when it does not modify state outside of the object. They further divide pure functions into stateless and stateful: a stateless function being when the return value is only determined by the arguments, whereas a stateful function also depends on member fields. Naumann [23] defines the term observational purity which allows a method to modify preexisting objects as long as those changes are properly encapsulated. That is, side effects are allowed if they are not observable to the caller. In the analysis developed by Sălcianu and Rinard [29] a method is considered pure as long as it does not write to the heap, never invokes any impure methods and finally never escapes an object globally.

We will now present a summary of the definitions that we have previously mentioned. Some of the definitions never mention certain operations such as if calling native methods is allowed. Therefore, we have left these operations as undefined in our summary. The summary covers escaping objects, which we consider to occur when the reference of an object instantiated in the method is made visible outside the method execution context. We have left out reading and writing to the stack simply because it is allowed for all the definitions we found. Our summary of purity definitions from literature can be seen in Table 2.1.

In previous work [35] it has been pointed out that pure functions are good candidates for parallelization. For example, the strong purity definition is guaranteed to be thread safe because such a function only operates on state that not visible to the rest of the program. As we have previously pointed out, the strong purity definition is of limited use for real world Java programs. The strong purity definition would, for example, disallow memoization, a technique for only computing expensive operations once. Consider the following Java code

Table 2.1: A summary of the purity definitions from the literature. The left column contains either the name of the purity definition or the name of the author who established the definition. We mark properties with ⊥ if we were unable to find specific information or draw any conclusion about that particular operation. Operations marked with † are allowed in some cases but not all. If the operation is only allowed the first time the method is invoked it is denoted by 1st. Operations that are always allowed are marked with ✓and any operation without any marking is disallowed. The operations that we consider are: reading/writing of the heap, reading/writing of static fields, escaping an object, monitor operations, i.e., acquiring and releasing locks (synchronized methods do this), throwing exceptions and finally calling native methods.

	Heap		Static		Escape	Monitor	Exception	Native
	Read	Write	Read	Write				
Strong								
Moderate	†	†						
Weak	✓	†						
Once-impure	✓	†	1st	1st	1st	1st	1st	1st
Pearce	✓	†	✓		⊥	⊥	⊥	⊥
Naumann	✓	†	✓		⊥	⊥	⊥	⊥
Yang	✓	†	✓		⊥	⊥	⊥	⊥
Salcianu	✓		✓		†	⊥	⊥	⊥
Concurrent	✓	†	✓		✓	✓	✓	

```
int value;
boolean value_calcuated;
public int f() {
    if (!value_calculated) {
        value = expensiveCalculation();
        value_calculated = true;
    }
    return value;
}
```

The function `f` above would not be considered pure with the strong purity definition. On the other hand, the function would be considered pure when using the once-impure definition because it allows a method to write to the heap the first time it is invoked. However, calling the method concurrently could possibly cause race conditions. Thus, a function can be considered pure according to some of the existing purity definitions in academic writing without it being thread safe.

Therefore, it would be desirable to establish a definition for purity that can guarantee thread safety but that is not as restrictive as the strong purity definition. A first attempt of a purity definition that is less restrictive than the strong purity definition but that can guarantee thread safety could be the moderate purity definition. However, this definition is also far too restrictive to be useful in practice. Consider the Java code below.

```
public class A {
    public int x;
}
public int g(A a) {
    return a.x;
}
```

According to the moderate purity definition the function `g` is not pure since it reads from a preexisting heap object. The parameter `a` is already allocated on the heap before the method `g` is invoked, and reading from preexisting heap memory is not allowed with the moderate purity definition. Reading from a preexisting heap object can cause concurrency issues if another thread is writing to the same heap object. However, if no other thread *writes* to the heap object that `g` is reading from then the read operation is safe. It is important to realize that this does not mean that the function `g` is thread safe: if any other function writes to heap memory that `g` is reading from a data race can occur.

A more suitable definition could be the weakly pure definition which allows functions to read from preexisting heap objects. For the purposes of our work this definition is also too restrictive since it disallows escaping objects, preventing otherwise safe HOAs. Allowing an object to escape is no cause for concern as long as it not modified after escaping. For example, when defining a HOA in JastAdd the generated method escapes the HOA, which is expected behaviour.

In concurrent RAGs attribute equations are required to be observationally pure. Although Naumann's definition in principle disallows non-thread safe effects in a function (if a function is not thread safe it will, in some thread interleaving, cause an observable side effect), the side effects it disallows are predicated on the execution context of the function. In single-threaded execution a Naumann observationally

pure function is free to memoize its result and return a previously memoized value. However, in a concurrent setting memoization can cause race conditions if not done carefully. We believe that another purity definition would be a more suitable requirement. Below we give an example of a method which is observationally pure for single-threaded execution but not observationally pure in a multi-threaded setting. Suppose we defined the following JastAdd attribute

```
syn lazy int A.x() = 0;
```

The `lazy` keyword tells JastAdd that the attribute should be memoized. JastAdd would from this generate the following Java code

```
class A {
    protected int x_value;
    protected boolean x_computed = false;
    public int x() {
        if (x_computed) {
            return x_value;
        }
        x_value = 0;
        x_computed = true;
        return x_value;
    }
}
```

The method `x` is Naumann observationally pure. However, the method is not thread safe because it can lead to race conditions. The method modifies preexisting heap memory, namely the fields `x_value` and `x_computed`. These writes are to protected fields, which is why the method is considered observationally pure. However, that the fields are encapsulated has no relevance for thread safety: race conditions can still occur despite the fields being protected.

We can also observe that `x` has side effects (the writes to `x_value` and `x_computed`) even though the attribute definition does not. These side effects are added by JastAdd in the code generation and are not defined by a developer. If a functional purity analysis tool, like Purano (see section 2.7), was applied to `x` the tool could generate warnings for these side effects. Since the side effects are not defined by a developer it is not interesting to generate warnings for them. Therefore, this these types of side effects can be considered to be false positives.

We propose a new purity definition which we will henceforth refer to as *concurrent* purity. The goal is to impose as few restrictions as possible but still be able to guarantee thread safety if only concurrently pure functions are executed. To be able to precisely define concurrent purity we need to define some additional and related terms. Below we define the *effects* of a function and further divide this definition into *direct effects* and *indirect effects*.

Definition 2.5.1. (Effect) We define the *effects* of a method f to be all writes to a memory location that can be caused by invoking f.

Definition 2.5.2. (Side Effect) We define a *side effect* of a method f to be any effect that modifies a Java object or an array that existed prior to f being invoked.

Definition 2.5.3. (Direct/Indirect Side Effect) We define a *direct side effect* of a method to be any side effect that is not caused by invoking another method. Furthermore, we define a side effect that is not a direct side effect to be an *indirect side effect*.

It should be noted that we consider modifying a static field to always be a modification of a preexisting Java object. Our definition of concurrent purity disallows invoking native methods (directly or indirectly), i.e., methods that do not run in the JVM.

Definition 2.5.4. (Concurrently Pure/Impure) We define a method f to be *concurrently pure* if f does not have any side effects and does not invoke any native methods. A method that is not concurrently pure is *concurrently impure*.

We will from here on use the term concurrently pure interchangeably with pure and similarly use the term concurrently impure interchangeably with impure. Furthermore, we extend our definition of impure into *directly impure* and *indirectly impure* to be able to distinguish between the two.

Definition 2.5.5. (Directly/Indirectly Impure) We define a method f to be *directly impure* if f is a native method or if f has any direct side effects. Furthermore, we define any method that invokes native methods or has any indirect side effects to be *indirectly impure*.

2.6 ExtJPureChecker

ExtJPureChecker (EPC) is a purity analysis tool developed for the master's thesis by Johnsson [13]. The tool is an extension to the ExtendJ compiler. The analysis of EPC is, like JPure [27], based on Java annotations, i.e., in order for the tool to be able to perform its analysis functions have to be annotated with a special set of Java annotations. The annotations used by EPC are the same as JPure but with a few new ones for a more precise analysis of HOAs. To handle HOAs the tool uses a notion of freshness, like JPure. With this it possible to make modifications to a fresh (newly created) object without generating warnings (the result of the tool). This is crucial for RAGs that support HOAs because a HOA value is an AST node which must be modified while constructing it inside the HOA equation. The HOA value needs to be considered a fresh object which it is permitted to modify inside the attribute equation.

EPC reports warnings for methods violating their purity annotation. The analysis takes into account problems occurring directly in a method such as modifying a field or a static field but it also considers function calls. For example a method annotated as pure is not allowed to call an impure method. EPC uses a conservative approach: the analysis is based on a call graph constructed using a class hierarchy analysis (CHA) as defined by [32], In a CHA call graph a method is connected to all its called methods and all methods that override it.

To support analysis of JastAdd RAGs Johnsson has made a second tool which is an extension to JastAdd that automatically annotates the code generated by JastAdd.

2.7 Purano

Yang [35] developed a purity analysis tool for Java called Purano that infer the purity of functions by analyzing Java bytecode. Purano uses the ASM framework for analyzing bytecode [1].

Purano constructs a class diagram and then traverses all the methods of the class diagram. The class diagram incorporates information such as inheritance, override relationships and method invocations. When a function is called, Purano will consider all overrides of the method. This is a conservative approach because it is unlikely that all overriding functions are called during runtime. This prevents false negatives but also causes the analysis to produce more false positives.

Purano does not rely on any Java annotations: the analysis can be applied to any existing code base without additional work to add annotations. Furthermore, since Purano analyzes byte code and not the Java source code Purano is capable of analyzing external libraries for which it only has access to the byte code.

Purano has the ability to identify cache semantics in a function. That a function uses cache semantics can loosely be defined as using an object field for storing the result of a calculation, for a more precise definition see the original paper [35]. An example of a function with cache semantics is shown in the code below.

```
public A f() {
    // lazy initialization
    if (a == null) {
        a = new A();
    }
    return a;
}
```

Purano does not generate warnings for side effects related to cache semantics. There are other tools with similar capabilities like the analysis by Sălcianu and Rinard [29] but this is not automatic: they maintain a list of so called special functions, which are specified to the analyzer to be pure.

Chapter 3
Parallelization in JModelica.org

We have identified two approaches for parallelizing JastAdd compilers, such as JModelica.org. The key difference between the two is in how threads share data. The first approach attempts to utilize the concurrent attribute evaluation extension to RAGs to evaluate attributes in parallel. As long as attribute equations are concurrently pure the data is shared in a safe manner. Therefore, this approach allows data to be shared fully when these requirements are met. At the time of writing JModelica.org has some known cases of impure attributes and could potentially have additional impure attributes that are not known about. Thus, the challenge with parallelizing JModelica.org this way is twofold: A method for purity analyzing JModelica.org has to be developed and any impure attribute that is found has to be transformed to be pure. This approach did, in the end, not lead to parallelization of JModelica.org due the amount of work required to ensure safe parallel attribute evaluation. In Section 3.1 we present our attempts of purity analyzing JastAdd compilers using the tool EPC [13] and in Section 3.3 we present our new tool JastAdd-Purity-Analyzer (JPA) for purity analyzing JastAdd compilers.

In our second approach for parallelizing JastAdd compilers we have tried to minimize the amount of shared data between threads. The motivation for this is that threads that do not share any data do not suffer from race conditions. Thread shared data is minimized by maintaining separate ASTs, each thread having its own version. The threads can safely evaluate attributes in parallel since they never operate on the same data. In Section 3.3 we present our work related to applying this approach to JModelica.org.

3.1 ExtJPureChecker

In previous work, Johnsson developed ExtJPureChecker (EPC) for purity analysis of Java programs with a special code generator to support JastAdd compilers [13].

We have investigated the possibility of using EPC to purity analyze JModelica.org with the motivation being that such an analysis could possibly be used to verify concurrent purity.

Our attempts at purity analyzing JModelica.org with EPC lead to the tool giving a number of false negatives and too many false positives. The causes of these problems and ways of improving the analysis are presented below.

There are several examples in JModelica.org where internal JastAdd methods, i.e., methods that are not intended for compiler developers, are used. One such internal method is `setChild`, which is a method for building the AST. The `setChild` method is used outside of the intended internal usages in 29 places in JModelica.org. EPC annotates methods for building the AST, like `setChild`, with `@Ignore` as they are not intended to be used by developers [13]. Because the methods are annotated with `@Ignore`, warnings are not generated when they are used by a developer, even though they are impure. This makes the analysis simpler for the checker, but also assumes that the developer does not use any of these methods. This leads to a number of false negatives in JModelica.org due to the use of `setChild`.

It would be possible to fix the false negatives problem by changing the generated annotations in EPC. To show how the altered annotations work we will show an example with a JastAdd rewrite. Suppose that a compiler developer has created an abstract grammar with a rewritable node. Rewrites are triggered on demand, when accessing a child through the `getChild` method. A part of the call graph for this method and the annotations generated by EPC is shown in Figure 3.1.

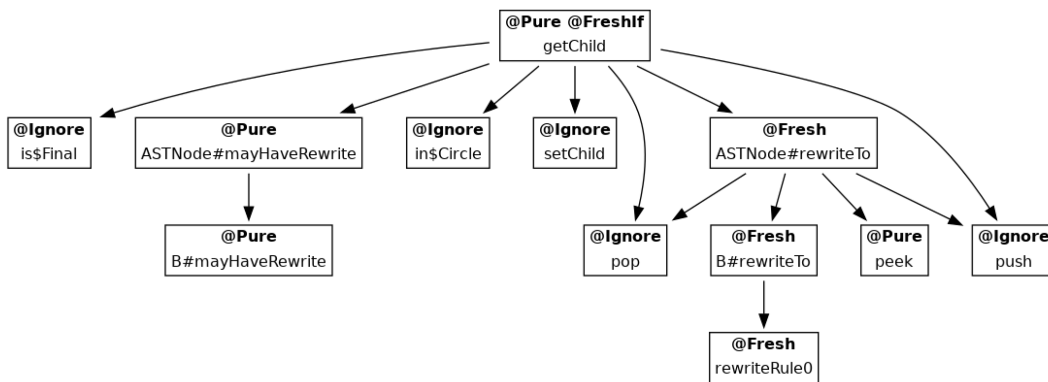

Figure 3.1: A part of the call graph for `getChild` and the corresponding generated annotations by EPC.

The figure shows how internal methods with side effects have been annotated with `@Ignore` and internal methods which are pure, such as `peek`, have been annotated with `@Pure`. This allows the parent of the rewritable node to call the `setChild` method, which it must to replace the old node with the new one, without generating a warning. The `rewriteTo` methods (the rewrite rules defined by developers) have been annotated with the `@Fresh` annotation to allow the rewrite methods to create and initialize the new node. This solution allows freshness to be analyzed in the sequence of method calls from `getChild` to `rewriteRule0`. Furthermore, any usage of internal methods with side effects are suppressed since they are ignored. The

downside is that, as was pointed out before, an attribute calling an internal method with a side effect such as `setChild` will not generate a warning.

Our goal was to let internal JastAdd methods that are known and expected to have side effects to be annotated with `@Local` and to filter out JastAdd methods that are of no interest to analyze by using the `@Ignore` annotation. A method annotated with `@Local` signifies that method may induce changes to the fields of the caller object [13]. An example of a method that can be ignored is the `ASTNode#mayHaveRewrite` (a JastAdd generated method) because it will always be pure. Therefore, its annotation can be changed from `@Pure` to `@Ignore`. Figure 3.2 shows the call graph with changed annotations.

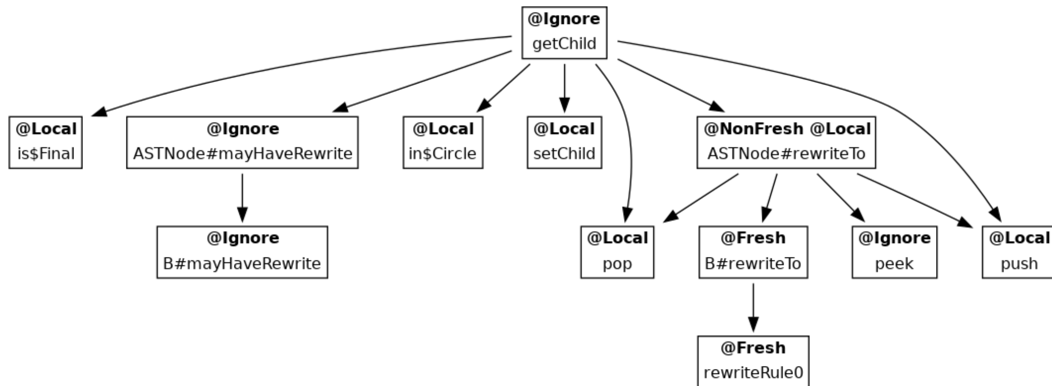

Figure 3.2: A part of the call graph for `getChild` along with a suggestion of a different set of annotations.

In this example, the only place a side effect can be added by a developer is in `rewriteTo` and the `rewriteRule0` method. It is of no real interest to analyze a method like, e.g., `getChild` because we actually already know how it behaves. For example, we know that it modifies the AST during a rewrite. However, this is intended behaviour and should not lead to a warning. A consequence of this approach is that a sequence of method calls containing one of these `@Ignore` annotations will stop the analyzer from analyzing the method body. This can potentially shorten the analysis time because the ignored methods no longer need to be analyzed. Likewise any call to an ignored method will not be analyzed.

There is a potential downside of this approach: suppose a developer has defined an attribute that calls the `getChild` method above and that the method `rewriteRule0` modifies a static field. The attribute is then impure since the method can trigger a rewrite which in turn causes the static field to change. With our suggested approach the defined attribute will inevitably be found pure because the side effect is "guarded" by ignored methods. This does however seem to be in line with how the checker operates: warnings are not generated for the entire sequence of method calls that can cause the side effect. Consider the code below:

```
public class A {
    public int i = 0;
    @Pure public void f() {
        g();
    }
```

```
    @Pure public void g() {
        h();
    }
    @Pure public void h() {
        i++;
    }
}
```

The analysis will in this case warn that `h` violates its annotation. Likewise a warning will be generated for `g`. However, the function `f` will not receive a warning. Thus, any sequence of methods calls will only generate warnings for the last two functions of the sequence. We believe this is a design choice with the motivation being that it will produce less redundant warnings from the tool. For a tool that is aimed at identifying where side effects occur this is perfectly reasonable design choice.

Unfortunately, for the purposes of this work this solution is not optimal: the function `f` is concurrently impure and should therefore be reported as a warning. To work around this, one could analyze the reachability of `f`, i.e., all reachable methods in the call graph from `f`, and then study the purity information for those methods.

The annotations generator tool for EPC does not generate annotations for everything. For example, plain Java methods as in the example below:

```
aspect F {
    public class A {
        private int i = 0;
        public void f() {
            i++;
        }
    }
}
```

The method `f` will in the generated code not receive any annotation. This might seem like a minor detail but as was discovered for JModelica.org this is not the case. We found over 500 class definitions, i.e., plain java definitions in the compiler. Every method in these classes will cause the tool to generate a warning (false positive) when they are called since they are assumed to be impure, due to the lacking annotation. A possible solution to this problem could be to let the tool treat all unannotated methods as if they were annotated with a `@Pure` annotation.

3.2 JastAdd-Purity-Analyzer

We were interested in finding ways of reducing the false positives rate when using Purano to analyze JastAdd-generated code. The primary benefit of Purano is that it does not rely on annotations. This could save substantial amounts of work that would otherwise be required to make the false positive rate low enough with EPC to be useful for an industrial compiler, such as JModelica.org.

A difficulty in purity analyzing a JastAdd grammar with a static analysis tool like Purano is that the analysis usually leads to a substantial amount of false positives.

Take for example when a compiler developer writes a JastAdd grammar with a HOA, JastAdd will insert the attribute value in the AST. Purano would in this case produce a warning that the AST has been modified. This is a false positive: we know that the higher order attribute is going to be placed in the AST and we do not want to report a warning for this. Thus, the major challenge in analyzing a JastAdd grammar with a tool like Purano is to filter out these false positives.

In order to reduce the false positives rate we developed a new tool which translates JastAdd code to Java code but removes side effects in generated parts of the code which are known to be safe (for memoizing attribute values, for example).

Our new tool JastAdd-Purity-Analyzer (JPA) generates Java code that is filtered from irrelevant side effects which is then analyzed by the purity analysis tool Purano. It should be noted that the generated Java code by JPA is not a functioning compiler and not intended to be run. The code only serves to be the statically analyzed by Purano. An overview of JPA can be seen in figure in Figure 3.3.

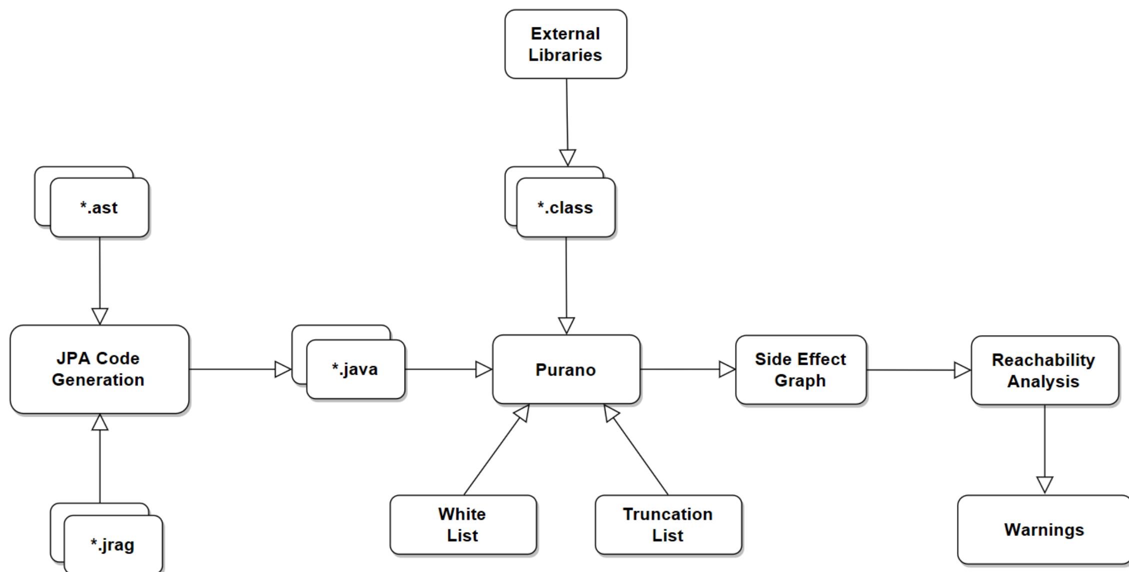

Figure 3.3: A high level overview of our tool JastAdd-Purity-Analyzer (JPA). The input consists of grammar files, external libraries, a white list and a truncation list. Any method in the white list will be assumed to be pure by Purano. The truncation list is used to stop the tool from traversing further when encountering methods in the list. The output is direct and indirect side effect warnings for the methods in the compiler Java code.

To demonstrate the usefulness of JPA we will now give a small example showing the difference between JPA and JastAdd code generation when an attribute has been declared as lazy, see below.

```
syn lazy int A.x() {
    return 0;
}
```

The corresponding code generated by JastAdd is shown below.

```
// JastAdd
public int x() {
    ASTState state = state();
    if (x_computed) {
        return x_value;
    }
    state().enterLazyAttribute(); // Indirect Side Effect
    x_value = x_compute(); // Direct Side Effect
    x_computed = true; // Direct Side Effect
    state().leaveLazyAttribute(); // Indirect Side Effect
    return x_value;
}
public int x_compute() {
    return 0;
}
```

When analyzing the attribute `x` we do not want to produce any warnings because the equation that defines `x` does not have any side effects. However, Purano will produce warnings for the generated code. Places in the code that could lead to a warning are shown in the code with a comment. Invoking the methods `enterLazyAttribute` and `leaveLazyAttribute` are considered indirect side effects because these methods increment/decrement a field. Their actions cancel when the method x returns, their effects cancel within the method. That is, when calling `x` sequentially it would not be possible to observe the effects of the `enterLazyAttribute` and `leaveLazyAttribute` methods as a change on the heap if the heap was compared before and after `x` was invoked. This is not true for the `x_computed` field though, when calling the method for the first time the field changes from false to true, and this effect would be visible on the heap after invoking the method. We have not seen any purity checking tool capable of identifying cases where side effects cancel.

The code generated by our tool JPA is shown below.

```
// JPA
public int x() {
    int x_value = x_compute();
    return x_value;
}
public int x_compute() {
    return 0;
}
```

As can be seen the calls to `enterLazyAttribute` and `leaveLazyAttribute` have been removed. The method no longer writes to the `x_computed` field and the `x_value` has been changed to a local variable, the JPA version of the method `x` is concurrently pure.

An important detail to JPA is that it does not remove all the side effects in JastAdd. It would be incorrect to do so: it would remove relevant purity problems and lead to false negatives when the code is analyzed. There are internal methods of JastAdd which are not supposed to be called by a developer, as pointed out in Section 3.1. These methods tend to be used anyway despite the intent being that they should not. An example of such a method is the `setChild` method generated by JastAdd. This method modifies the child vector of AST nodes (the children of

an AST node is stored in the child vector). Any attribute that uses this method to modify the AST is indirectly impure. Suppose now that this method was changed so that the child vector was no longer modified. Any attribute calling `setChild` would no longer be indirectly impure and thus not cause a warning. This is not the intent of JPA. The attribute should still be identified as being indirectly impure since it invokes `setChild`.

In the following sections we are going to show in detail how JPA handles RAG features like rewrites, higher order attributes, collection attributes and circular attributes. Each of the following cases is based on an example abstract grammar in which an attribute is defined in an impure way. To each case we provide a call graph highlighting how JPA differs from JastAdd in its code generation. These call graphs have been simplified for reader convenience (parts of the call graph we deem irrelevant are not shown).

The call graphs use a combination of color coding and labels to represent properties for the methods. A method that is colored blue and that is not marked with any label represents a method that is not directly impure. Methods that are directly impure are classified as being either added by a developer or by JastAdd. For example, when defining a HOA the code written by the developer is placed in the corresponding `_compute` method for the HOA. Any side effect in such a method is considered to have been added by a developer. Evaluating the HOA also leads to other methods being called, e.g., `setChild` to modify the AST (side effect) and we consider these side effects to have been added by JastAdd. A directly impure method where the side effect is added by JastAdd is colored in orange and marked with the label "JastAdd". Finally, a directly impure method where the side effects are added by a developer is colored in red and marked with a label "Developer". Each of the following cases are based on the following abstract grammar:

```
// Abstract Grammar
A ::= B;
B;
C;
aspect F {
    public static int A.i = 0;
}
```

For simplicity, all the side effects in the following cases will be the same, namely the incrementation of the static integer `i` in class A.

3.2.1 Rewrites

Suppose a developer has defined a rewrite as shown below.

```
rewrite B {
    when (A.i++ > 0) // Side Effect 1
        to C {
            A.i++; // Side Effect 2
            return new C();
        }
}
```

This implementation has two side effects: one in the when-statement of the rewrite (nr 1) and one in the block defining the new node of the rewrite (nr 2). As can be seen in Figure 3.4 the side effects added by the developer occur in the `rewriteTo` (nr 1) and `rewriteRule0` (nr 2) methods of `B`. Thus, to analyze the `getB` method we preserve all edges in the call graph leading to these two methods and remove all other edges (dashed in the figure).

This principle can be applied to any rewritable node, for example, `is$Final` or `setChild` are methods which can never lead to a side effect added by a developer. Therefore, the calls to these methods can safely be removed in the `getChild` method. The only way for the `getB` method in this case to become indirectly impure due to a side effect defined by a developer is if the `rewriteTo` method of the rewritable node is impure. The developer can cause this method to become directly impure, indirectly impure or possibly both. Therefore, the call graph should only preserve the call chain to `rewriteTo` method of the rewritable node. As a consequence, the `getChild` method generated by JPA only calls the `rewriteTo` method defined in the `ASTNode` class. This method is then connected to all other `rewriteTo` methods since they are overrides of that method.

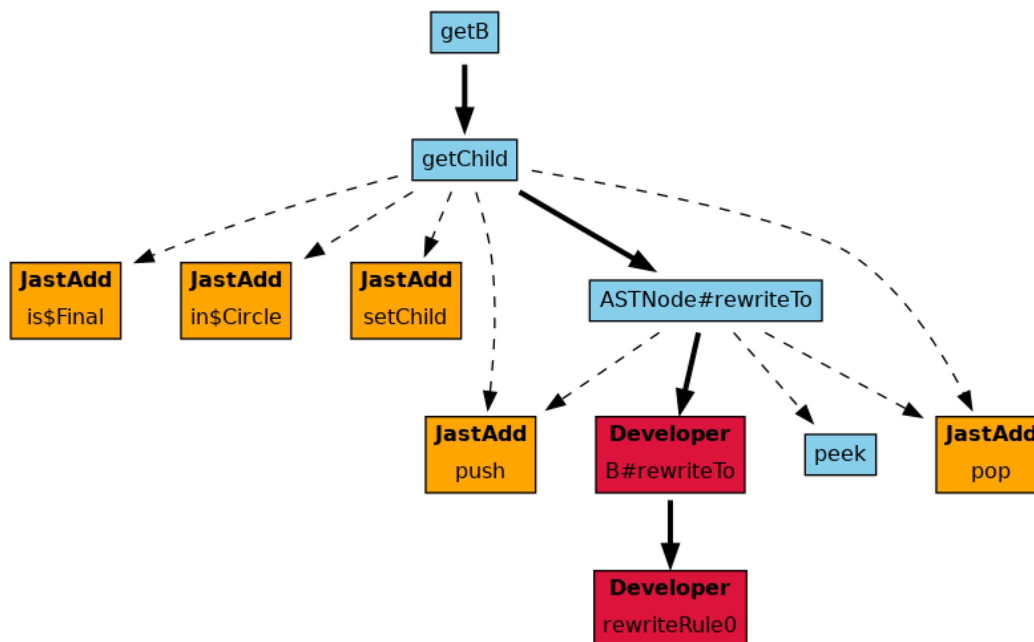

Figure 3.4: Part of the call graph for `getB` which accesses the rewritable node `B`. The graph shows where JastAdd and the developer have added side effects. The graph also shows removed methods calls, as dashed edges, in JPA code generation.

3.2.2 Higher Order Attributes

In the following code the example of a list HOA from the JastAdd webpage [11] has been used and a side effect has been added to the HOA definition.

```
syn nta C A.getCList() {
    A.i++; // Side Effect 3
    return new List<C>()
        .add(new C())
        .add(new C());
}
```

The call graph for the HOA as generated by JastAdd is shown in Figure 3.5. The side

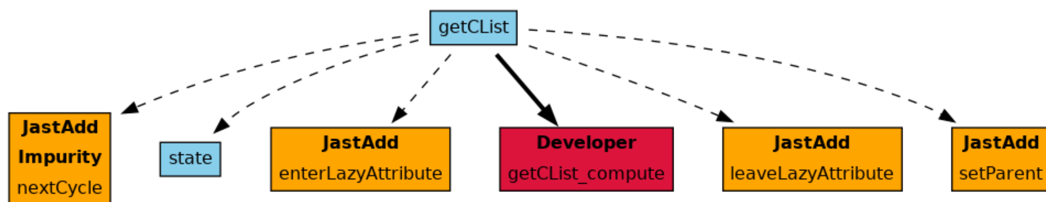

Figure 3.5: Part of the call graph for `getCList` which gets a HOA list attribute. The graph shows where JastAdd and the developer have added side effects. The graph also shows removed method calls, as dashed edges, in JPA code generation.

effect added by the developer (nr 3) is in the `getCList_compute` method. Making the `getCList` method call only the compute method will remove a large number of the side effects added by JastAdd. HOAs generated by JastAdd are treated as lazy attributes. As such there is always a value and boolean field for the HOA. In this case their names are `getCList_value` and `getCList_computed`. JPA does not cache the HOA and the method generated for the HOA will only call the corresponding compute method.

3.2.3 Collection Attributes

Suppose that a developer has defined the follwing collection attribute:

```
public class MySet<T> extends HashSet<T> {
    public MySet() {
        super();
        A.i++; // Side Effect 4
    }
    @Override
    public boolean add(T t) {
        A.i++; // Side Effect 5
        return super.add(t);
    }
}
```

```
coll Set<String> A.myCollection() [new MySet<>()];

B.contributes A.i++ > 0 ? "" : ""
    to A.myCollection(); // Side Effect 6
```

As can be seen in the code above, a new class `MySet` has also been defined where the constructor and the `add` method of the set have side effects. In the contribution statement the developer has also added a side effect. See Figure 3.6 for the call graph generated by JastAdd. The survey phase of the collection attribute, i.e., the phase where the contributors to the collection are collected, is completely removed in JPA code generation. The call to the survey method can never lead to side effects added by a developer and is therefore not useful to preserve. For collection attributes JPA preserves edges to the collection constructor and to the contribution statements to the collection. In the example above side effect 4 occurs in the `MySet` constructor, side effect 5 in the `B#contributeTo_A_c` node and finally side effect 6 in the `add` node.

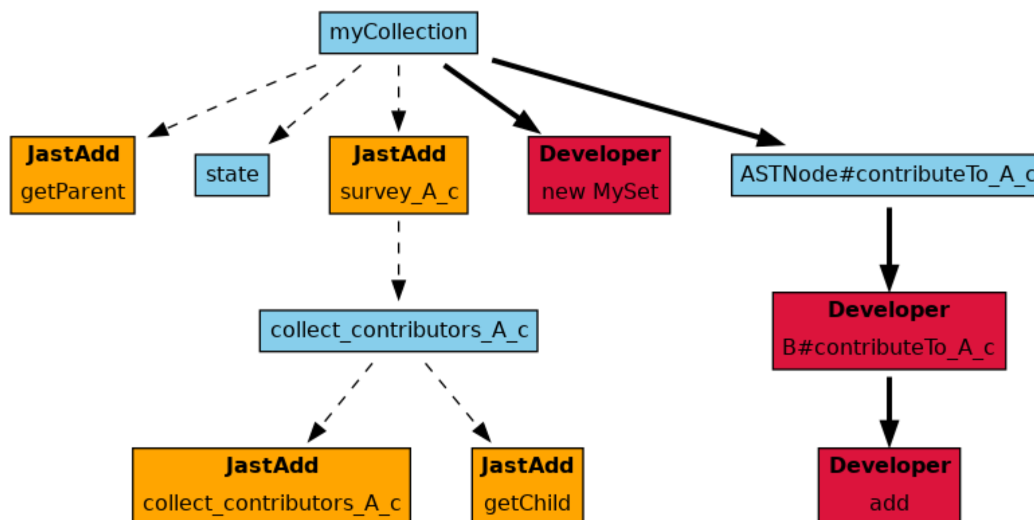

Figure 3.6: Part of the call graph for `myCollection`. The graph shows where JastAdd and the developer have added side effects. The graph also shows removed methods calls, as dashed edges, in JPA code generation.

3.2.4 Circular Attributes

The following code shows a circular attribute that contains two side effects.

```
syn int A.myCircular() circular [myCircularInit()];
eq A.myCircular() {
    A.i++; // Side Effect 7
    return 0;
}

syn int A.myCircularInit() = A.i++; // Side Effect 8
```

The call graph for the example can be seen in Figure 3.7. The side effects added

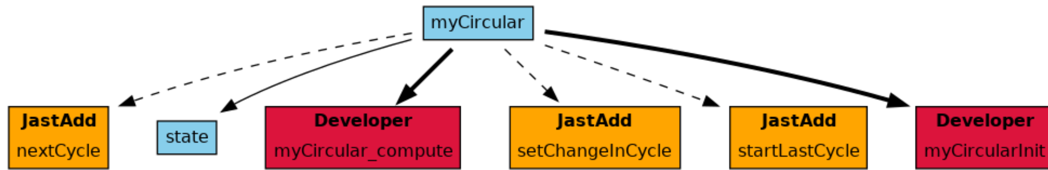

Figure 3.7: Part of the call graph for `myCircular`. The graph shows where JastAdd and the developer have added side effects. The graph also shows removed methods calls, as dashed edges, in JPA code generation.

by the developer are in the compute method and in the initialization of the circular attribute. Circular attributes in JastAdd use an initialized field and a value field, the code generated by JastAdd modifies these fields during evaluation (side effects). JPA removes these operations and in the end the attribute method only calls the corresponding compute method and the method initializing the attribute, unless of course the attribute is not initialized by a method.

3.2.5 Analysis by Purano

The first stage of JPA is to generate Java code that is filtered from irrelevant JastAdd side effects. Next, the JPA generated code is analyzed by the purity analysis tool Purano. Purano analyzes all the methods in the class diagram and for each method it produces warnings for both direct and indirect side effects. It should be noted that Purano does not use the terminology of direct and indirect side effects. Purano divides side effects to be either modification of fields, arguments and static fields. In the output of the tool this is denoted as `@Field`, `@Argument` and `@Static`, respectively.

Side effects in Purano have a reference named `from` that can either be null or point to another method. If the pointer is null the effect is according to our terminology a direct side effect, whereas if the pointer points to another method we consider it an indirect side effect. The analysis result is output as a summary for each method where both direct and indirect side effects are shown.

We can consider the analysis result by Purano to be a directed graph. Each vertex is a method which is associated with a list of direct and indirect side effects. Any indirect side effect connects the vertex to another vertex with the `from` reference. We will call this graph the *side effect graph*.

To give an example of a side effect graph we give a small code example.

```
public class A {
    int i = 0;
    void f() {
        g();
    }
    void g() {
        h();
```

```
    }
    void h() {
        i++;
    }
}
```

The corresponding side effect graph for the above code sample is shown in Figure 3.8. We have omitted any indirect side effect from the associated list of side effects for

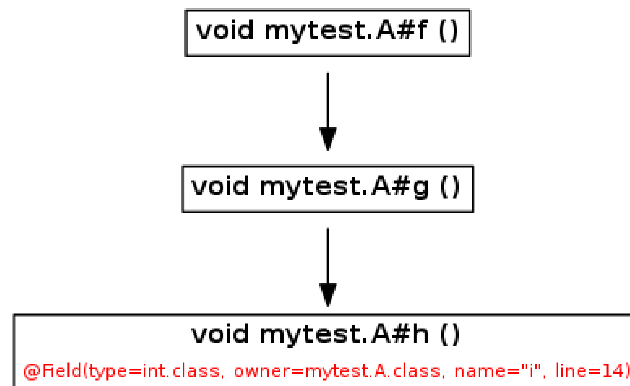

Figure 3.8: A graph representation of the analysis result produced by Purano for our code example.

the vertices since any indirect side effect will correspond to an edge in the graph. An example summary for the method `f` that is output by Purano is shown in Figure 3.9. The function `f` in the code above is impure since it modifies a preexisting object.

void mytest.A#f ()

FieldModifier
@Field(type=int.class, owner=mytest.A.class, name="i", from = "void mytest.A#g ()")

```
   L0
    LINENUMBER 6 L0
    ALOAD 0
    INVOKEVIRTUAL mytest/A.g ()V
   L1
    LINENUMBER 7 L1
    RETURN
   L2
    LOCALVARIABLE this Lmytest/A; L0 L2 0
    MAXSTACK = 1
    MAXLOCALS = 1
<<<<<<<<<<<<<<
    6: ALOAD
    6: INVOKEVIRTUAL
@Field(type=int.class, owner=mytest.A.class, name="i", from = "void mytest.A#g ()")
    7: RETURN
```

Figure 3.9: The method summary for `f` generated by Purano for our code example.

When the method `f` is invoked the `this` object already exists and during its execution `f` indirectly modifies that object. The `this` object is modified because when `f` invokes `g` it will in turn invoke `h` and this method directly modifies the field `this.i`.

Suppose now that we would want to analyze `f` for concurrent purity. We know that `f` is concurrently pure if it is neither directly or indirectly impure. Therefore, if the associated list of side effects for the method `f` is empty we know that `f` must be concurrently pure. If on the other hand we find that `f` is impure it would be interesting to know which program statements that cause `f` to be impure.

For example, if we would like to know why the method `f` in our example is impure we could use the analysis result shown in Figure 3.9. We would then find that `f` is indirectly impure because it invokes the method `g`. To proceed we would then look at the analysis result of `g` and conclude that `g` is indirectly impure because it calls `h`. Finally, we would find in the analysis result that `h` is directly impure and that it is not indirectly impure. Suppose now instead we analyzed a function that indirectly calls a large amount of functions that cause it to become indirectly impure. Doing the same type of analysis would be very time consuming. In the next section we will talk about our extensions to Purano that tries to solve these problems.

3.2.6 Extensions to Purano

To identify all the sources of indirect impurity for a method in a time efficient and accurate way we extend Purano to produce its result on a new format. We translated the internal graph representation of Purano, i.e., the side effect graph, to the graph description language DOT. This could then be used to automatically output an image of the graph. An example of such an image is the one shown in Figure 3.8 which we have generated with our extension. The extension is useful for visualizing the sources of indirect impurity in a method. However, for some examples the graphs become so large that converting the DOT file to an image takes too long time and produces impractically large images.

For very large graphs, we instead only report the direct side effects reachable from the method. For example, the only reachable direct side effect from `f` in Figure 3.8 is the one in the node for the method `h`. We also associate each direct side effect with one of the possible paths that leads from the entry methods to that direct side effect. For example, our extension would for the method `f` find the direct side effect in `h` and associate that side effect with the path `A#f()` → `A#g()` → `A#h()`. The paths to a direct side effect are found by using a depth-first traversal over the side effect graph, following the pseudo code in Algorithm 1.

To illustrate the format of the reported warnings we give a real world example for JModelica.org. We have simplified the example for clarity by omitting package names. The reported warning is shown in figure 3.10. The first line shows the method (`geSrcStoredDefinition`) that is analyzed which is then followed by a call chain leading to a reported side effect. The figure shows that the attribute `SrcLibNode` (a HOA) indirectly calls the `setChild` method. When using our extended version of Purano on JModelica.org it can in some cases produce a lot of output. One of the causes of this is methods with side effects on static fields. Any other method that calls such a method will also be considered indirectly impure regardless of whether or not the function is called on a local object. For example, we found that methods were identified to be indirectly impure anytime that they called the `Iterator#next` method despite it being a local object. The cause for this was that there was a class

Algorithm 1 Reachable Direct Side Effects

1: **procedure** ENTRY_METHOD(method)
2: methodStack ← new-stack
3: VISIT_METHOD(method, methodStack, ϕ)
4: **procedure** VISIT_METHOD(method, methodStack, visitedMethods)
5: PUSH(method, methodStack)
6: visitedMethods ← UNION(visitedMethods, method)
7: **for** each direct side effect s in method **do**
8: REPORT_WARNING(s, methodStack)
9: **for** each indirect side effect s in method **do**
10: **if** s.from $\notin$ visitedMethods **then**
11: VISIT_METHOD(s.from, methodStack, visitedMethods)
12: POP(methodStack)

```
void A#g () --->
void A#f ():
        @FieldEffect(
                type=int,
                owner=A,
                name="i",
                line=7)
```

```
void A#g ():
        @StaticEffect(
                type=A,
                owner=A,
                name="a",
                from = "A#f ()",
                line=10)
```

Figure 3.10: An example of a side effect warning reported by JPA. The warning is for a HOA defined in JModelica.org.

that extended the Iterator class and the overridden next method had a side effect on a static field.

We found a possible solution to this problem by making the analysis run in different modes. For example, in one mode that we call "no static" mode, side effects on static variables are ignored. This solves the aforementioned problem with the `Iterator#next` method since any side effects on static fields are ignored.

Another mode that we added is a "static" mode in which the analysis only outputs side effects on static fields. In this mode, any indirect side effect where the `from` pointer points to a method with no static effects is truncated. An example of such a case is shown below.

```
public class A {
    public static A a = new A();
    public int i = 0;
    public void f() {
        i++;
    }
    public void g() {
        a.f();
    }
}
```

If the function `g` is analyzed normally our analysis will output the direct side effect in `f` and the call chain `A#g()` → `A#f()`. However, with only this information it is

not clear that a static field is affected. In the scope of `f` the direct side effect that occurs is on the non-static field `i`. In our static analysis mode the call analysis will instead output the indirect side effect `a.f()` and the associated call chain which in this case is only `A#g`. The statement `a.f()` is the first place that it can be detected that a static field is modified. See Figure 3.11 for a comparison of the two analysis modes using the text format output of the tool. Finally, we also added a mode

```
void A#g () --->
void A#f ():
        @FieldEffect(
                type=int,
                owner=A,
                name="i",
                line=7)
```

```
void A#g ():
        @StaticEffect(
                type=A,
                owner=A,
                name="a",
                from = "A#f ()",
                line=10)
```

Figure 3.11: A comparison of the output of JPA when analyzing the code sample in 3.2.6 in default mode and static mode. The output from the default mode is shown to the left and the output from the static mode is shown to the right.

"native mode" that only identifies calls to native methods.

Another problem that we discovered was how to deal with calls to the Java standard library. Our algorithm presented above will traverse to the Java standard library and find direct side effects. This can causes a number of redundant side effects to be reported, for example when calling the Java standard library `Hashtable#put` the information that is of interest is that the `Hashtable` is modified, however when analyzing this method our analysis finds 16 different direct side effects. This tends to cloud the analysis with too much information. A possible solution would be to not propagate to any standard impure library function and just showcase that a call to an impure standard library function is made. We tried this approach and discovered that it can truncate propagation to functions that are not part of the standard library that are of interest to analyze. For example, suppose that we have implemented a class that extends the `Iterator` class part of the Java standard library and its `hasNext` writes to a static field. Any function that invokes `Iterator#hasNext` will then be considered indirectly impure. But the analysis will only show that a call to an impure standard library function is made, not that it is caused by our new iterator class.

Our solution is, instead, to maintain a so called "truncation list" which manually specifies which parts of the standard library that should be truncated. This can reduce the number of identified direct side effects for our previous example of the `Hashtable` from 16 to 1. Moreover, if it is found that a method calls an impure standard library function which is truncated and it is unclear why the method is impure the method can simply be removed from the list. We also maintain a white list which makes Purano assume that the listed methods are pure.

3.3 Maintaining Separate ASTs

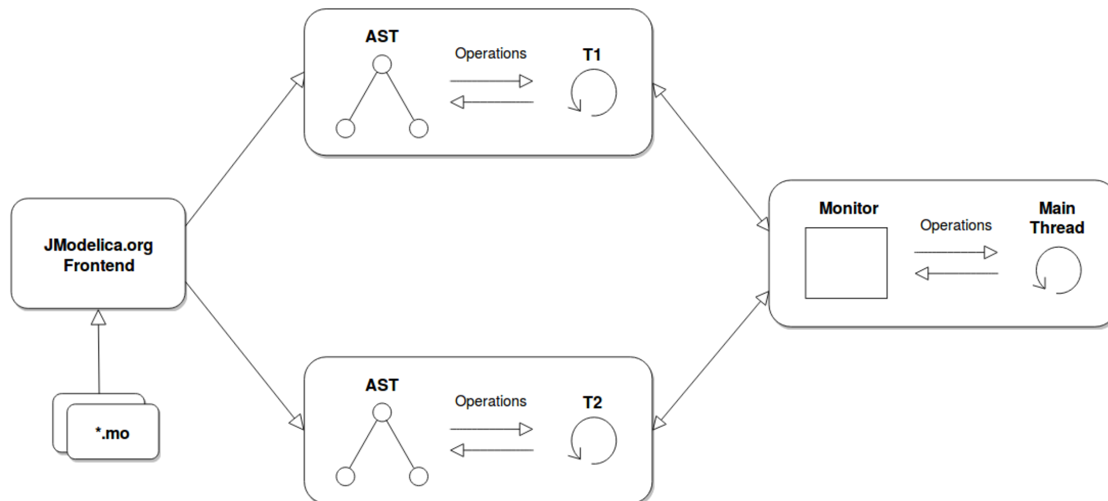

Figure 3.12: A high level overview of our approach of maintaining separate ASTs for JModelica.org.

The JModelica.org compiler builds the AST in a lazy manner. Initially, the AST is minimal in both size and in the number of evaluated attributes. This is achieved using a combination of rewritable nodes and lazy attributes. We will call this minimal syntax tree the *base AST*. When an attribute is evaluated the AST is expanded on demand when parts of the AST that need to be rewritten are accessed. Furthermore, any attribute that a evaluating attribute depends on will as a consequence also be evaluated.

Our new approach of maintaining separate ASTs uses these facts, each thread is given a separate base AST. An overview of the approach can be seen in Figure 3.12. The JModelica.org front-end parses the source code (*.mo files) and builds a base AST, which is then assigned to a *worker thread*. The worker threads are instructed by the main thread to perform operations on their AST and report back the result. In Figure 3.12 `T1` and `T2` are the worker threads. When a worker thread is finished with its computation it stores the value in a cache structure that is owned by a monitor object and then signals to the main thread. The cache is a concurrent hash map with `<String, Object>` key value pairs. The cache can be used both for worker threads to store the result of the operation they have been assigned to do, but also to store computationally expensive attributes that can be shared among the worker threads.

The idea is to make concurrent attribute evaluation safe by evaluating on differ-

ent ASTs. If the threads do no share any data then no race conditions can occur, i.e. concurrent attribute evaluation is safe. The threads do not share any data as long as they do not perform I/O write operations or operate on any static fields. However, to allow the worker threads to perform I/O we have added support for this in the monitor object. If a thread needs to do I/O it is granted exclusive access to do I/O via the monitor object, blocking any other thread until finished. Similarly, reading or writing to static fields can be done via synchronized methods in the monitor object. However, if it is known that a static field is only ever read, i.e., never modified by the threads then there is no need to synchronize the access.

To detect writes to static fields one could use our tool JPA to analyze a given attribute in the "static" analysis mode described in Section 3.2.6. Furthermore, JPA can also be used to identify usage of I/O. This can be done by running the analysis in "native" mode. I/O is always performed by the operating system and is as such always native. The identified native methods can then be inspected to find if any of them are related to I/O.

Being able to share data between threads is often desirable because it can potentially speedup execution and reduce memory usage. This is why we have also added support for controlled data sharing between the threads by using the cache in the monitor object. The result of costly evaluations can be stored in the monitor cache which can then be reused by other worker threads. Suppose that the monitor object is available for all `ASTNodes` and we have a computationally expensive attribute `expensiveAttr` on the program root. We could declare `expensiveAttr` in the following way.

```
syn int Program.expensiveAttr() {
    if (!monitor.cache.containsKey("expensiveAttr")) {
        int val = calcExpensiveAttr();
        monitor.cache.putIfAbsent("expensiveAttr", val);
    }
    return monitor.cache.get("expensiveAttr");
}
```

This is similar to the algorithm described by Öqvist and Hedin [26] for memoized non circular attributes. The difference here is that it is done manually and that the attribute is cached in a monitor cache.

The threads can safely put and get values from the monitor cache concurrently. However, even though the monitor cache is thread safe the objects in the cache might not be. As such, there are limitations of what they do can with the values. If a non thread safe object is cached the worker threads should not modify it after it has been shared, since this can cause race conditions.

An advantage of maintaining separate ASTs as we do in this approach is that legacy rewrites, i.e., non circular HOAs are supported. This is not true for standard concurrent attribute evaluation [26] where only circular HOAs are compatible. Since the threads do not share the AST it is safe to modify it, as is done with legacy rewrites. Furthermore, the requirements on attribute equations are less strict. Take for example the following attribute.

```
aspect F {
    public int i = 0;
```

```
    syn int x() {
        return i++;
    }
}
```

The attribute `x` is not concurrently pure. However, each AST has its own version of `i` and so do each worker thread as a consequence. Therefore, each worker thread can safely evaluate `x` concurrently despite the attribute being concurrently impure.

Chapter 4

Evaluation

In this section we present the answer to research question 1-4 introduced in Section 1. To address RQ1 we have measured the responsiveness of the JModelica.org compiler for a computationally expensive operation when using our approach of maintaining separate ASTs. In Section 4.3 we introduce the operation that was tested along with our experimental data. To answer RQ2 we have documented impure attribute in JModelica.org and categorized these according to their use pattern. Furthermore, to address RQ3 we propose ways to refactor these attributes to be thread safe. In Section 4.1 we present the different categories of impure attributes in JModelica.org along with suggestions how they can be refactored. To answer RQ4 we developed our new tool JastAdd-Purity-Analyzer (JPA) for purity analyzing JastAdd grammars. In Section 4.2 we present measurements that compare JPA to purity analysis with standard JastAdd generated code and Purano.

4.1 JModelica.org Purity Survey

Below we present some example use patterns of impure attributes in JModelica.org. Our method for finding impure attributes in the compiler is, both to collect information about known cases from the developers, and to apply JPA to the compiler. Furthermore, some of the impure attributes have been found when working with the source code. However, all of the attributes in the survey can be verified to be impure with JPA since the tool generates true positives when analyzing them. We will for each case explain the pattern and also propose how to either remove or deal with the side effects of the particular case to establish thread safety.

4.1.1 Memoization

We have found examples of attributes in JModelica.org that are impure because they indirectly call methods that memoize a static field the first time it is needed. That is, the field is null until the method is called for the first time and then the field is assigned a value. All subsequent invocations of the method returns the memoized value. An example of where this occurs in JModelica.org is in the `getSCCContributors` method of the `AbstractBiPGraph` class.

Memoization is in a multi threaded environment a potential cause of race conditions. For example, multiple versions of the memoized object can escape to the rest of the program. One of the ways memoization can be supported in a multi threaded environment is the strategy that is used for lazy attributes in concurrent RAGs. Here a special operation called compare-and-swap is used to write to the `_value` field of the lazy attribute. Multiple threads may start evaluating the attribute but only one thread will succeed in writing to the `_value` field. Since the returned value from the attribute evaluation is the value stored in the `_value` field all threads that evaluate the attribute are guaranteed to return the same value. Instead, one could also protect the memoization block or the method where the memoization occurs with mutual exclusion.

4.1.2 Aborting Fixed Point Iteration

JastAdd uses fixed point iteration to compute circular attributes. Circular attributes are evaluated multiple times, at least twice to ensure that the fixed point is found.

In JModelica.org there are examples of attributes with circular dependencies where it is known that if the attribute evaluation is circular then it is not necessary to evaluate the attribute a second time. If the attribute is declared as circular then it will inevitably be evaluated twice since it is computed using fixed point iteration. In JModelica.org these attributes are not declared as circular and have instead been implemented using a custom evaluation technique. The node that owns the attribute is given a boolean flag that is set to true when the attribute evaluation starts. When the attribute is evaluated it is checked if this flag is true, if it is then the evaluation terminates prematurely preventing evaluation a second time. This type of side effect occurs in JModelica.org in the `calcVariability` method in the `InstAssignable` class.

This is an example of using a side effect in an attribute to perform of an optimization technique on circular attributes. These attributes could be transformed to be pure by instead declaring the them as circular. An alternative to making the attribute pure is is to declare the boolean flag that is used to abort the fixed point iteration prematurely as a thread local object. The attribute would still be impure but the writes to the flag would only be visible to the thread that performed the write.

4.1.3 Accessing Inherited Attributes

In order to access inherited equations in JastAdd a node has to be part of the AST. Therefore, any freshly created node will not be able access to inherited equations unless it is added to the AST. Since any expansion of the AST costs memory, a potential drawback of using inherited attributes is memory usage. In JModelica.org a special way of accessing inherited equations without permanently expanding the AST has been developed in order to optimize memory usage. To access the inherited equations a node is temporarily put into the AST, when the inherited equations are no longer needed the node is removed from the AST. An example of where this is used in JModelica.org is the `dynamicFExp` method in the `FExp` node class. The method accepts an `FExp` parameter and uses `setChild` to put the paramater `FExp` in the AST.

To allow a node to temporarily access inherited attributes one can set the parent reference of the node and not link it as a child of the AST. This is desirable because if the node accessing inherited attributes is fresh this operation is thread safe. Nodes in the AST are connected using a combination of references. The children of a node are stored in an array and the parent is stored in a single reference. With our proposed solution the AST is not modified: it is only the parent reference of the node accessing the inherited attribute that is modified. This works because the algorithm for finding inherited attribute equations uses the parent reference to traverse upwards in the tree. This approach will work correctly for inherited attributes that are broadcast, i.e., attribute available in an entire sub tree. The solution will, however, not work for basic inherited attributes (non-broadcasted). The attribute evaluation algorithm will for basic inherited attributes verify that the caller node is the child that the attribute is defined for.

4.1.4 Impure Rewrite Equations

We have observed several instances of rewritable nodes in JModelica.org that modify an existing boolean flag when they are rewritten. The flag is modified in the corresponding JastAdd generated `rewriteRule` method. One example of such a rewrite is the
`InstShortClassDecl` node that sets a boolean `simpleRewriteDone` to true when the rewrite happens. This is used to prevent the rewrite from happening additional times since the rewrite is only allowed to happen when the flag is false. Rewrites can typically not happen again once they have been fully rewritten. However, JModelica.org modifies the `is$Final` flag which indicates that a node can not be rewritten further.

A possible strategy for dealing with the impure rewrite equations that we observed in JModelica.org could be to change the boolean that the equations modify to be thread local. However, since concurrent RAGs require that rewrites are implemented as circular HOAs the compiler would first have to changed to implement rewrites this way. It is possible that the desired behaviour for the impure rewrites are not compatible with circular HOA implementation.

4.1.5 Rewriting Final Nodes

In JModelica.org there are examples of nodes setting the `is$Final` flag to false after they have been fully rewritten to allow the node the be rewritten again. An example node in JModelica.org that is rewritten this way is the `FAssignStmt` node.

Since the rewrite equation for these nodes modify the `is$Final` flag they are not compatible with circular HOAs. This is becuase, in the circular HOA implementation, this flag has been removed. We have not identified any way of achieving the same behaviour using normal rewrites. It is our belief that these rewrites need to fully revised to make JModelica.org compatible with circular HOAs and concurrent attribute evaluation.

4.1.6 Logging

To support logging of, for example, the amount of primary memory used during compilation JModelica.org has a system for adding logging information dynamically. There exists a global logger object which is accessible through a static field in the `ASTNode` class. Thus, the logger object is reachable from any place in the entire program, and in particular from any attribute. We have found examples of attributes that during their evaluation write logging information to the global logger object. These attributes are, as consequence, impure because they modify the global logger object. An example of an attribute that modifies the global logger object is a HOA called `SrcStoredDefinition` which is defined on nodes called `SrcLibNode`. The HOA will during its creation for example log that a file has been read or log debug information when an exception is thrown.

The logging mechanism in JModelica.org could possibly be re-implemented with collection attributes: collection attributes are often used to solve these kinds of problems and they are supported in concurrent attribute evaluation. Alternatively, the logging mechanism could be changed so that any read or write operation to the logger object was protected using mutual exclusion. The advantage of this approach is that a large part of the source code could remain the same whereas the collection attribute solution requires substantially more changes.

4.1.7 Error Checking

During error checking in JModelica.org, static errors are reported to a global error handler which resides in the AST root node. When errors are reported the global error handler will add the error to an internal list. The error reporting method is statically declared, making the handler is available in the entire program. The handler is not only modified by attributes related to error checking but also to seemingly unrelated attributes. For example, the previously mentioned HOA `SrcStoredDefinition` reports an error if parsing the library fails. A possible reason for implementing the error checking in this way instead of the idiomatic approach which is to use collection attributes could be related to the survey phase of collection attributes. Typically, the survey phase will not visit the child nodes of HOAs and these nodes cannot directly contribute to the collection.

The error checking system used in JModelica.org is not safe for concurrent evaluation due to the global error handler that is used. To transform the attributes part of the error checking to become pure the global error handler could be replaced with a collection attribute rooted in the AST root node. JastAdd has recently added support for contributing to collections from HOA child nodes. To make the error checking thread safe rather than pure, the methods that modify the global error handler could be changed to be synchronized.

4.1.8 Constant Propagation

JModelica.org uses a modified version of the constant propagation optimization technique [14]. The algorithm searches for equations that can be solved at compile time. These equations are removed from the final equation system and any reference to the equations are replaced by the found solutions. The optimization technique was shown to both reduce compile and simulation times. We have seen multiple places where the implementation performs impure operations on the AST. An example is the cache system that the algorithm uses. Some `ASTNodes` have a cache field named `evaluationValue` that is modified during the evaluation. For example, this field is modified on `InstComponentDecl` nodes by using the `setLocalCachedEvaluationValue` method.

We have not been able to find a way to remove the side effects in the constant propagation algorithm in JModelica.org. The algorithm is complex and would require a comprehensive investigation to find if side effects could be removed from the implementation. An alternative approach could be to protect any read or write operation with mutual exclusion. For example, the cache field that the algorithm uses that we previously described could be protected using mutual exclusion: anytime a thread wanted to read or write to the cache field it would have to acquire a lock.

4.2 Purity Analyzis of JastAdd Attributes

With our new tool JastAdd-Purity-Analyzer (JPA), JastAdd attributes can be analyzed for direct and indirect side effects, direct and indirect calls to native methods. We have not performed any analysis for evaluating if any true positives are removed by the tool. It would, however, be desirable to do so. A possibility could be to, first, identify every place in JastAdd syntax in which it is possible to add side effects. Next, a JastAdd project with a direct side effect in each of these places could be analyzed with the tool. Finally, the result of this analysis could be used to verify that none of those direct side effects are removed. This would verify that none of the direct side effect are removed. However, it would not verify that the tool preserves all indirect side effects, which is much harder to verify. It is harder because each direct side effect can cause multiple methods to be indirectly impure. That is, for each direct side effect it would be required to verify that all methods that are indirectly impure due to that side effect are still indirectly impure using JPA code generation.

Table 4.1: The result of our experiments where the impact of our side effect filtered code generation is tested. The compilers ExtendJ and JModelica.org are written in short as ExtJ and JMod, respectively. The left column specifies the type of code generation, either normal code generation with JastAdd or our side effect filtered code generation with JPA.

	Code size (NCLOC)		Nr of Impurities		Analyzis Time (s)	
	ExtJ	JMod	ExtJ	JMod	ExtJ	JMod
JastAdd	120 602	419 498	50 316	432 651	49.7	802
JPA	102 282	393 908	15 415	181 130	42.3	157

We have, instead, focused on measuring the impact of JPA code generation. Our experiments were run on an Intel Core i5-4300 at 1.9 GHz running ubuntu 16.04 LTS, Java version 1.8.0_212 (Oracle JDK) and a Java heap size of 7 Gb. We evaluated the number of found side effects for JModelica.org and ExtendJ when the code is built with JastAdd 2.3.0 and JPA. To count side effects we traverse the side effect graph and visit all nodes that are not part of the Java standard library. For each node we then count the number of side effects that Purano has found for the method. The result of our measurements is shown in table 4.1. It should be noted that the analysis finds more side effects than lines of code for JModelica.org. This is because one line of code can have multiple side effects. An example would be invoking a method that modifies two arguments and a static field, this single line of code would have 2 + 1 side effects.

We have also compared the code size of the generated code for JModelica.org and ExtendJ using JastAdd 2.3.0 and JPA. The code size is measured in non comment lines of code (NCLOC) using the tool `cloc`. To account for non-generated Java code, we first removed all the generated code and ran `cloc`. This number was then subtracted from the number we obtained when running `cloc` after generating the code. The result of our measurements are presented in Table 4.1. As can be seen in the table the number of identified side effects when generating with JPA instead of JastAdd v.2.3.0 are reduced by 69 % and 58 % for ExtendJ and JModelica.org, respectively. This can be compared to the code size reductions which are 15 % and 6 % for ExtendJ and JModelica.org, respectively.

We have also measured the time it takes for Purano to construct the side effect graph when analyzing JModelica.org and ExtendJ when the Java code is generated by JPA and JastAdd v2.3.0. We constructed the side effect graph 10 times for each case. The result of these experiments can be seen in table 4.1. Our experiments found that JPA reduces the analysis time by 15% and 80% for ExtendJ and JModelica.org, respectively.

4.3 Compile Time in JModelica.org

We tested our approach with maintaining separate ASTs for JModelica.org for a known case where long compilation time is a problem. The case is the startup of an interactive Modelica editor. The editor allows the user to edit Modelica models interactively using graphical editing features like drag-and-drop. On startup the editor queries the compiler to compute an *inheritance graph*. The inheritance graph (IG) is a data structure that incorporates the inheritance relationships of all loaded Modelica code (developer source code and loaded external libraries). Here inheritance refers to inheritance as in object oriented programming. Constructing the IG is in general costly: the operation has to account for the developer source code as well as any external libraries that the developer has loaded.

We tried to improve the responsiveness of the compiler to the editor by using two separate ASTs. One is used to construct the IG, we will call this thread 1, and the other handles other queries, we will call it thread 2. In sequential evaluation the compiler is responsive once the base AST and the IG have been built. With separate ASTs the compiler will become responsive as soon as the base AST of thread 2 has been built.

As input for our experiments we used the Modelica Standard Library (MSL). We compiled MSL and built an IG both with sequential evaluation and parallel evaluation using two separate ASTs. We measured the response time for each case 10 times. The experiments were run on an Intel Core i7-5600 at 2.6 GHz running Windows 10 Pro, Java version 1.8.0_202 (Oracle JDK) and a Java heap size of 6 GBb. We found that using sequential evaluation the compiler became responsive after 12.4 s. When using parallel evaluation thread 2 became responsive after 2.7 s, i.e., a speedup of about 4.6X.

Chapter 5
Conclusion and Future Work

5.1 Conclusion

The goal of this thesis was to examine the possibility of using parallelism in JastAdd compilers, such as JModelica.org, in order to improve compilation speed. We have investigated different approaches for this endeavour. One approach was to try to use the concurrent attribute algorithms for RAGs [26]. Our study of impure attributes in JModelica.org has shown that this approach is difficult because as we found there are many examples of impure attributes in JModelica.org. The study has also shown that JModelica.org does not fulfill the requirement of circular HOAs. This is because there are several examples of rewritable nodes that are incompatible with circular HOAs.

We have found that it is sometimes very difficult to verify that an attribute is concurrently pure. This is because its purity depends on the attributes that are part of its evaluation. For example, by tracing JastAdd attribute evaluation we found that several hundred attributes are involved when constructing the inheritance graph mentioned in Section 4.3. Furthermore, the tracing does not cover any non attribute Java code, which is commonly used in JModelica.org. We concluded that doing this analysis manually by inspecting the code is not viable. It would be both time consuming and prone to errors.

Due to the non determinism of concurrent evaluation it is harder to test concurrent RAGs. In sequential evaluation the attributes in a RAG can be tested using standard software testing techniques such as unit testing. With this it is possible to conclude that an attribute will always evaluate to an expected value for a specific input because the evaluation is deterministic. This is not true when evaluating attributes concurrently. Impure attributes and any attribute affected by the side effects of those impure attributes will only evaluate to their correct value with "lucky" timing. Purity analysis is a highly valuable tool when applying the concurrent at-

tribute evaluation extensions to RAGs [26]. This is why we focused on developing a way of purity analyzing RAGs for concurrent purity.

These efforts lead to the development of our new tool JastAdd-Purity-Analyzer (JPA) that analyzes RAGs for purity. We introduce a new purity definition, concurrent purity, which has advantages for analyzing thread safety compared to other purity definitions. We believe that this definition is better suited for ensuring the correctness of concurrent RAGs than the observational purity definition which has been used in previous work [26].

The tool combines our simplified code generation for JastAdd with the existing purity analysis tool Purano. We have shown that the impact of our code generation is significant, reducing both the number of false positives and analysis times. Furthermore, we extended Purano (see Section 3.2.6) with different analysis modes and an algorithm for finding all the side effects of an attribute.

Due to the difficulty in verifying and transforming the JModelica.org grammar to fully support concurrent attribute evaluation [26], we developed a new approach for evaluating attributes concurrently. Our new approach maintains separate versions of the AST for each thread which minimizes the amount of shared data between threads. It is easier to establish safe concurrent attribute evaluation with this approach. This is because if it is established that the threads do not share any data then safe concurrency is guaranteed. Moreover, our new method also supports legacy rewrites, i.e., the concurrent attribute evaluation does not require that rewrites are implemented as circular HOAs. A limitation of this approach is that it can significantly increase memory usage. If the AST is the dominating factor in memory cost having more than one AST can be very high.

With the results of our work, pure attributes can be identified for RAGs. Moreover, if several pure attributes are identified these can safely be evaluated with concurrent attribute evaluation [26]. Our tool JPA also gives information about the causes for an attribute not being pure which can be used as a guide in transforming the attribute to be pure. Our new approach for concurrent attribute evaluation using separate ASTs can be used to support concurrent evaluation for any grammar that is correct in sequential evaluation, does not perform I/O write operations or modify any static fields.

5.2 Future Work

We have identified additional ways that the JastAdd code generation could be modified to improve the accuracy of Purano. The purity analysis by Purano will for a function call consider the function being called as well as all the methods that override it. This can in some cases lead to false positives. Take for example the following abstract grammar:

```
// Abstract Grammar
A ::= B;
B;
C;
D;
aspect F {
```

```
    public static int A.i;
    rewrite B {
        to C {
            A.i++;
            return new C();
        }
    }
    rewrite C {
        to D {
            A.i++;
            return new D();
        }
    }
}
```

Since the `A` node has a child `B` JastAdd will generate a Java method `getB`, as shown below.

```
public B getB() {
    return (B) getChild(0);
}
```

A part of the call graph for `getB` is shown in Figure 5.1. The nodes that are not directly impure are colored blue and have no label. Methods that are directly impure are colored red and have the label "Developer" indicating that the method has a side effect added by a developer. As we can see in the figure, the `getB` method

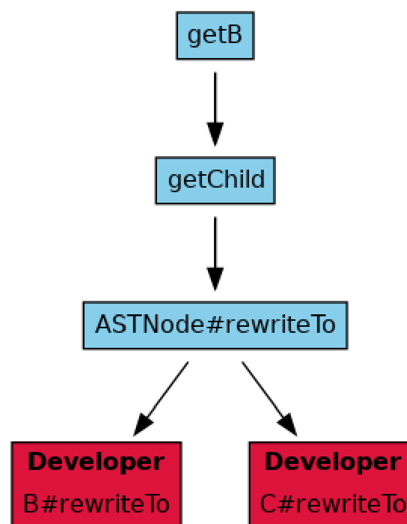

Figure 5.1: A part of the the call graph for the `getB` method generated by JastAdd which gets a `B` node from the AST.

is connected to the rewrite method for the `C` node. Therefore, JPA will generate a warning for `getB` due to the side effect in the `C#rewriteTo` method. This is, however, a false positive: calling the `B` method can never cause `C#rewriteTo` to run since the child of the `A` node is a `B` node. The cause for the false positive is that when `getB` calls `ASTNode#getChild` type information is lost.

For methods like getB that returns a node from the AST it would be possible to improve the accuracy of the analysis. These methods will only ever invoke side effects that are caused by calling rewrite methods. Furthermore, we know from the method signature which rewrite method could be called (only one is possible). Therefore, these methods could be changed to only call the possible rewrite method instead of getChild. Applying this change to getB would yield the call graph shown in Figure 5.2. The usage of colors and labels are equivalent to those in figure 5.1. As can be seen in the figure, getB is no longer connected to C#rewriteTo.

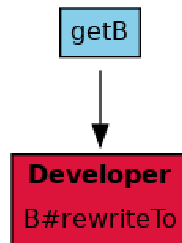

Figure 5.2: A part of the call graph for the getB method using the hypothetical JPA code generation.

Consequently, the analysis will not generate a warning for the side effect in the C#rewriteTo method. There are likely more cases like this one which could be taken advantage of to further improve the accuracy of Purano.

In this work we have used functional purity analysis to serve as an indication of thread safety. Any set of concurrently pure attributes are safe to evaluate in parallel, i.e., will not suffer from race conditions. However, concurrently impure attributes can also, in some cases, be safe to evaluate in parallel. A race condition can only occur if more than one thread accesses a shared memory location and at least one of those accesses is a write. Therefore, an attribute with a side effect is safe to evaluate concurrently if no other thread is reading or writing to the memory location affected by the side effect. For example, consider the following attribute grammars:

```
// Abstract Grammar
A;
aspect F {
    public int i = 0;
    syn int x() {
        i++;
        return 0;
    }
    syn int y() = i;
    syn int z() = 0;
}
```

As we can see, x has a side effect that affects the variable i. It would not be safe to evaluate x concurrently with y because x and y read/write to a shared memory location without any synchronization. However, it would be safe to evaluate x with z since z never reads to the field i.

It would be interesting to apply analysis of thread-shared data to parallel attribute evaluation. For example, TSA (Thread Sharing Analysis) [10], a tool for

identifying thread-shared data in multi-threaded programs in Java, could be applicable for parallel attribute evaluation. TSA implements algorithms for both static and dynamic analysis. It would be interesting to see how useful these algorithms are when applied to RAGs. The static algorithm has been verified to have reasonable analysis times for large scale projects with sizes similar to JModelica.org. Therefore, the static algorithm seems to be promising for real world RAG compilers.

Bibliography

[1] asm.ow2.io. Atomic access. https://asm.ow2.io/.

[2] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege De Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible aspectj compiler. In *Transactions on aspect-oriented software development I*, pages 293–334. Springer, 2006.

[3] Mihai Budiu and Seth Copen Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999.

[4] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In *European Conference on Object-Oriented Programming*, pages 147–171. Springer, 2004.

[5] Torbjörn Ekman and Görel Hedin. The jastadd system—modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[6] Brian Goetz. *Java Concurrency In Practice.* Addison-Wesley, 2006.

[7] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[8] Görel Hedin. An introductory tutorial on jastadd attribute grammars. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 166–200. Springer, 2009.

[9] Görel Hedin and Eva Magnusson. Jastadd—a java-based system for implementing front ends. *Electronic Notes in Theoretical Computer Science*, 44(2):59–78, 2001.

[10] Jeff Huang. Scalable thread sharing analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1097–1108. ACM, 2016.

[11] jastadd.org. Reference manual for jastadd. http://jastadd.org/web/documentation/reference-manual.php.

[12] JModelica.org. jmodelica.org.

[13] Mikael Johnsson. *Purity checking for reference attribute grammars*. Department of Computer Science, Faculty of Engineering, LTH, Lund University, 2017. Master Thesis.

[14] Jonathan Kämpe. *Applying Constant Propagation in a Modelica compiler*. Department of Computer Science, Faculty of Engineering, LTH, Lund University, 2013. Master Thesis.

[15] Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[16] Sergiy Kolesnikov, Ing Sven Apel, and Christian Lengauer. An extensible compiler for feature-oriented programming in java. *Masterarbeit. Universität Passau*, pages 12–13, 2011.

[17] A. Krall. Efficient javavm just-in-time compilation. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, pages 205–212, Oct 1998.

[18] Lijesh Krishnan and Eric Van Wyk. Termination analysis for higher-order attribute grammars. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, pages 44–63, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[19] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[20] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin. Using hard macros to reduce fpga compilation time. In *2010 International Conference on Field Programmable Logic and Applications*, pages 438–441, Aug 2010.

[21] Nir Shavit Maurice Herlihy. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[22] modelica.org. The modelica language. https://www.modelica.org/modelicalanguage.html.

[23] David A Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376(3):205–224, 2007.

[24] Robert HB Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1992.

[25] Jesper Öqvist. Extendj: extensible java compiler. In *Programming*, pages 234–235, 2018.

[26] Jesper Öqvist and Görel Hedin. Concurrent circular reference attribute grammars. In *10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017)*, pages 151–162. ACM, 2017.

[27] David J Pearce. Jpure: a modular purity system for java. In *International Conference on Compiler Construction*, pages 104–123. Springer, 2011.

[28] Michael Pradel and Thomas R Gross. Fully automatic and precise detection of thread safety violations. In *Acm Sigplan Notices*, volume 47, pages 521–530. ACM, 2012.

[29] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for java programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 199–215. Springer, 2005.

[30] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[31] Emma Söderberg and Görel Hedin. Declarative rewriting through circular nonterminal attributes. *Computer Languages, Systems & Structures*, 44:3–23, 2015.

[32] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000.*, pages 281–293, 2000.

[33] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher-order attribute grammars. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, pages 131–145, 1989.

[34] Haiying Xu, Christopher JF Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 75–82. ACM, 2007.

[35] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Revealing purity and side effects on functions for reusing java libraries. In *International Conference on Software Reuse*, pages 314–329. Springer, 2015.

[36] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 221–234. ACM, 2005.

Appendices

EXAMENSARBETE Contributions to Parallelizing a Modelica Compiler
STUDENT Joachim Wedin
HANDLEDARE Jesper Öqvist (LTH), Christoph Reichenbach (LTH), Axel Mårtensson (Modelon)
EXAMINATOR Görel Hedin (LTH)

Kan flerkärniga processorer göra kompilatorer snabbare?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Joachim Wedin**

Snabbhet är en viktig egenskap för kompilatorer. Ett sätt att göra kompilatorer snabbare kan vara med parallellism. Detta arbete har undersökt olika tillvägagångssätt för att parallellisera JModelica.org i syfte att göra kompilatorn snabbare.

Dagens datorer har ofta processorer med flera kärnor. Dessa processorer kan utföra flera instruktioner samtidigt, något som brukar kallas för parallellism. Genom att fördela instruktioner över flera kärnor kan man göra datorprogram snabbare. För att göra detta i praktiken så krävs det att datorprogrammen är speciellt anpassade för det syftet. Man måste, bland annat, se till att de olika kärnorna inte förstör för varandra. Ett vanligt problem är exempelvis att två kärnor skriver till samma plats i minnet samtidigt, vilket kan leda till problem.

Jag har i mitt arbete undersökt olika sätt att parallellisera JModelica.org i syfte att göra kompilatorn snabbare. JModelica.org är en kompilator för språket Modelica som används för att modellera och simulera komplexa fysikaliska system. JModelica.org är utvecklad med JastAdd vilket är ett verktyg speciellt gjort för att bygga kompilatorer. För att bygga en kompilator med JastAdd uttrycker man hur den ska fungera med en speciell formalism kallad referens-attribut-grammatik.

JastAdd genererar utifrån detta Javakod, vilket är ett vanligt programmeringsspråk. Den genererade koden har tidigare inte varit anpassad för att köras parallellt. Nya tillägg till JastAdd har dock ändrat på detta. De nya tilläggen kräver att koden som ska köras parallellt är fri från sidoeffekter. Ett exempel på kod med sidoeffekter är kod som ändrar på globala variabler.

Ett av sätten att parallellisera JModelica.org som jag undersökt är ifall dessa tillägg är applicerbara på kompilatorn. För att ta reda på detta har jag undersökt ifall kravet om frihet från sidoeffekter uppfylls. Jag presenterar i mitt arbete en studie över sidoeffekter som finns i JModelica.org. I studien ger jag även lösningar hur man kan ta bort dessa sidoeffekter.

Arbetet har även resulterat i ett nytt verktyg som analyserar kompilatorer utvecklade med JastAdd för sidoeffekter. En av svårigheterna med en sån analys är att den genererad koden innehåller många ointressanta sidoeffekter. De intressanta sidoeffekterna drunknar ibland dessa. Verktyget löser detta genom att filtrera den genererade koden ifrån de ointressanta sidoeffekterna. Koden undersöks sedan med ett existerande verktyg som analyserar sidoeffekter. För att lättare identifiera var i koden som sidoeffekter uppstår har jag utökat det existerande verktyget ytterligare. Mitt verktyg kan användas för att identifiera kod som är fri från sidoeffekter och som därför är säker att köra parallellt. För den koden som inte är det ger verktyget information om var det sker i koden. Verktyget kan alltså även användas som en guide för att ta bort sidoeffekter.