Principles and Patterns of JastAdd-Style Reference Attribute Grammars

Niklas Fors
niklas.fors@cs.lth.se
Lund University
Lund, Sweden

Emma Söderberg
emma.soderberg@cs.lth.se
Lund University
Lund, Sweden

Görel Hedin
gorel.hedin@cs.lth.se
Lund University
Lund, Sweden

Abstract

Reference attribute grammars (RAGs) have reached a level of maturity where they are supported by several tools, and have gained traction in both academic and industrial language tool development. However, despite a lot of accumulated knowledge of how to best develop RAGs in practice, there is limited support to guide practitioners.

In this paper, we address this issue by focusing on one RAG tool, JastAdd, and by defining principles and patterns for development of RAGs with this tool. We evaluate the proposed principles and patterns with an exploratory empirical study with 14 practitioners, with a mix of beginners and experienced users from both academia and industry. The results indicate that the principles and patterns capture the practice of developing JastAdd RAGs well, help practitioners to become aware of useful patterns, and provide a common language to more efficiently reason about the practice of developing JastAdd RAGs.

CCS Concepts: • **Software and its engineering → Formal language definitions**; **Translator writing systems and compiler generators**; **Designing software**.

Keywords: reference attribute grammars, semantic specification, patterns

ACM Reference Format:
Niklas Fors, Emma Söderberg, and Görel Hedin. 2020. Principles and Patterns of JastAdd-Style Reference Attribute Grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3426425.3426934

SLE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8176-5/20/11.
https://doi.org/10.1145/3426425.3426934

1 Introduction

Reference attribute grammars (RAGs) [Hedin 2000] is an executable specification formalism that can be used for generating extensible compilers and other language-based tooling. Fundamental differences to classical attribute grammars [Knuth 1968] include that attributes can be reference-valued, which supports superimposing graphs on top of abstract syntax trees, and on-demand evaluation, which supports more flexible attribute dependencies.

RAGs have reached a level of maturity where several systems support them, like JastAdd [Hedin and Magnusson 2003], Silver [Van Wyk et al. 2010], and Kiama [Sloane 2009], and they are used for development of language tools in both academia and industry. For example, there are Java and C compilers implemented using RAGs, like ExtendJ [Ekman and Hedin 2007b], ableJ [Van Wyk et al. 2007], and ableC [Kaminski et al. 2017], as well as industrial products, like Modelon's Optimica Compiler Toolkit [Åkesson et al. 2010a] that implements the Modelica language [Modelica 2017]. RAGs are also used in numerous research projects, and taught in graduate courses.

However, despite a lot of accumulated knowledge of how to best develop RAGs in practice, the support for practitioners is limited. In the early days of a formalism/tool, the knowledge around good practices for it is typically limited, and possibly also less needed since applications are usually small. But as the user population grows, as applications grow in size, and as new users start to use the tool, with little personal connection to the tool creators, the lack of explicit guidelines hampers developers in gaining proficiency with the tool. Consequently, there is a need to find suitable means to document existing knowledge of good practices.

In this paper, we pick one RAG tool, JastAdd, and attempt to describe the main principles and patterns for designing JastAdd RAGs, to guide practitioners. To formulate these principles and patterns, we draw on extensive experience of RAGs from developers of JastAdd, from developers of many applications using the tool, and from courses that use JastAdd. We evaluate the proposed list of principles and patterns empirically via four focus groups with a mix of beginners and experienced JastAdd RAG users, from both academia and industry. We find support for the proposed list in that the principles and patterns appear to capture how JastAdd RAGs are developed in practice, and we further

see that the list helps participants to become aware of new patterns, and provide them with a language to reason about their development of JastAdd RAGs. The contributions of this paper are the following

- a list of principles and patterns for JastAdd-style RAGs.
- an empirical evaluation of the suggested patterns and principles.

We believe that the principles and many of the patterns hold also for other RAG systems, but the exact patterns might vary due to differences in features and the supported AST model. To aid further work in investigating this, we have identified which particular RAG features are used for each pattern.

The rest of this paper is structured as follows; we set the stage with some background about JastAdd RAGs in Section 2, introduce identified principles and patterns in Section 3 and Section 4, empirically evaluate principles and patterns in Section 5, review related work in Section 6, discuss our results in Section 7, and conclude the paper in Section 8.

2 JastAdd RAGs

The JastAdd RAG tool represents programs as abstract syntax trees (ASTs) defined by node types specified in an object-oriented abstract grammar. Attributes are added to node types or node type interfaces in *aspect modules*, using inter-type declarations like in AspectJ [Kiczales et al. 2001]. An **attribute** is a derived node property defined by an equation. The tool generates Java code where the node types and interfaces are translated to Java classes and interfaces, and attributes to methods on the classes.

An **equation** is associated with an AST node type, and defines an attribute (its left-hand-side) as equal to a right-hand side expression over attributes in the AST node and its children. The defined attribute can be either an attribute in the node itself, or an attribute of a child node. In JastAdd, the expression is written as an arbitrary Java expression, but that must be observationally pure [Naumann 2007], i.e., without externally visible side-effects. For practical purposes, it may be written as a Java method body with local side-effects.

An attribute value can be accessed in an equation right-hand side, or from an external tool. To access the value, its corresponding method is called. The attribute evaluator then computes the value of the attribute by evaluating the right-hand side of its defining equation, which may in turn lead to new attribute evaluations. Attribute values are memoized (cached), so that if an attribute value is accessed more than once, the memoized value is returned directly at future accesses [Jourdan 1984]. (If the AST is edited, dependent memoized attributes need to be updated or cleared, e.g. using the algorithm presented in [Söderberg and Hedin 2012].)

2.1 Attribution Mechanisms

A JastAdd attribute can have any Java type, including references to AST nodes. With reference attributes, it is possible to specify graph structures, like name binding graphs, inheritance graphs, control-flow graphs, etc.

JastAdd supports the following attribute mechanisms:

- An **intrinsic** attribute is declared in the abstract grammar and is given a value when the AST is constructed. Typically, it is used for representing a token value, but it can also be a reference to a node in the AST.
- A **synthesized** attribute has a defining equation in the same node as the attribute declaration.
- An **inherited** attribute has a defining equation in some ancestor of the node declaring the attribute. It can be used for propagating information downwards in an AST.
- An equation for an inherited attribute of a child node holds for the whole subtree of that child. The value is said to be ***broadcasted*** to the child subtree. Many AG systems have similar mechanisms for avoiding so called copy rules.
- A **higher-order** attribute (HOA) has the value of a fresh AST, and it can itself have attributes [Vogt et al. 1989]. HOAs are also known as *nonterminal attributes*.
- A **parameterized** attribute (and its corresponding method) takes one or more arguments. The defining equation can make use of the arguments to define the value, so that there is in effect one attribute value per combination of argument values. Each such value is memoized individually. Synthesized, inherited, and higher-order attributes can all be parameterized.
- A node type **interface** can contain both attribute declarations and equations. The interface can be applied to an arbitrary node type, which will then be a subtype of the interface, and inherit (in the object-oriented sense) all the attributes and equations in the interface.
- An equation can be **overridden** in a subtype. This allows **default equations** to be defined in supertypes By placing equations in a most general node type `ASTNode` (supertype of all other node types), defaults can be defined for the whole AST.
- A **collection** attribute is defined by **contributions** that can be anywhere in the AST. The value of a collection is the combination of all its contributions using a commutative operation, for example additions to a set [Boyland 1996].
- A **circular** attribute has a definition that may transitively depend on itself, and is evaluated using fixed-point iteration [Magnusson and Hedin 2007]. In-place attribute-dependent node **rewrites** are also supported and are equivalent to circular higher-order attributes [Söderberg and Hedin 2015].

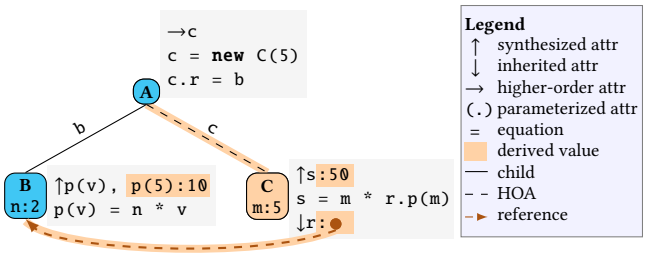

Figure 1. Core visual syntax used in structure diagrams

```
e = A.c.s
    = A[c=new C(5)]
    = {memo A.c=new C(5)}
  = A.c.s
    = A.c[s = m * r.p(m)]
    = A.c.m * A.c.r.p(A.c.m)
    = 5 * A.c.r.p(5)
          = A[c.r = b]
          = {memo A.c.r = A.b}
    = 5 * A.b.p(5)
          = A.b[p(5) = n * 5]
          = A.b.n * 5
          = {memo A.b.p(5) = 10}
    = 5 * 10
    = {memo A.c.s = 50}
  = 50
```

Figure 2. Trace of evaluating the expression `A.c.s`

2.2 Structure Diagrams

We will describe the patterns using structure diagrams annotated with pseudocode. The symbols ↑↓→ are used for denoting synthesized, inherited, and higher-order attributes respectively. An example illustrating core attribution mechanisms is shown in Figure 1. Here, `A` and `B` are nodes in the initial AST, as constructed by, e.g., a parser. `C` is a node constructed as part of the attribute evaluation, using a higher-order attribute →`c`. Furthermore, `n` and `m` are intrinsic attributes, ↑`s` is a synthesized attribute, ↓`r` is an inherited reference attribute, and ↑`p(v)` is a synthesized parameterized attribute. For each node in the diagram, attributes and equations are instantiated from the specification. Derived attribute values are shown in orange, and are the result of attribute evaluation using the equations.

To illustrate how attribute evaluation works, consider the attribute expression `A.c.s`. The trace in Figure 2 shows how the expression is evaluated, by recursively evaluating attributes and memoizing them. For each evaluated attribute, a box shows its subevaluation, ending in a memoization of the attribute ({`memo` *attr* = *value*}). An expression *nodepath*[*equation*] means evaluate the right-hand side of the *equation* in the context of the node denoted by *nodepath*.

3 Principles

To guide JastAdd RAG design, we have identified three overarching design principles: *declarative thinking*, *AST-embedded data structures*, and *structure-shy specification*. These principles are intended to help developers construct reusable and extensible specifications, taking advantage of RAG features and avoid resorting to conventional programming.

3.1 Declarative Thinking

In writing an equation defining an attribute, it is advised to formulate the definition as an expression over other attributes, introducing new helper attributes as needed. This is called the *declarative thinking* principle [Hedin 2009], in contrast to more operational thinking that focuses on how to traverse the AST in order to compute things. Once the new attributes have been introduced, the principle is applied again in order to define their values.

Declarative thinking in RAGs has similarities to stepwise refinement in imperative programming and functional decomposition in functional programming, but requires specific design decisions that newcomers may need some time to learn: deciding *where* to declare a new attribute (in the same node as the equation, in a child node, or in a remote node), deciding what *kind* of attribute it should be (synthesized, inherited, collection, higher-order, ...), and avoid thinking about *order of computation*.

While deciding where to place attributes and what kind to use is often quite easy, it is a learning threshold to understand when to use inherited attributes. The key is to think of inherited attributes as queries delegated to the context. Another learning threshold is to avoid thinking about order of computation. A key to accepting why order of computation is unimportant is understanding that accessed attributes are memoized, so that calling them several times will not lead to inefficiencies.

3.2 AST-Embedded Data Structures

Claiming that the AST is sufficient as a data structure may be counterintuitive for developers that have worked with traditional compilers, and are used to constructing separate symbol tables, call graphs, etc. However, because the AST can be extended in a number of ways, it is possible to embed elaborate structures in the AST: reference attributes allow graphs to be constructed, higher-order attributes allow the AST to be extended with new structure, and abstract interfaces with attributes allow new roles to be added to existing AST nodes. The advantage of using the (attributed) AST this way is that the embedded data structures themselves become extensible, and that computations over them can be formulated declaratively using attributes [Hedin 2000].

3.3 Structure-Shy Specification

Structure-shy software means that computations should use minimal information about the specific data structure they are operating on [Lieberherr 1996] [Lämmel et al. 2003]. In particular, the goal is to make the computation independent on data types that need to be traversed in order to find the interesting parts of the data structure. XPath expressions [Clark et al. 1999] is one example of constructs supporting structure-shy programming.

In RAGs, structure-shy specifications are supported, in particular, by inherited attributes, broadcasting, collection attributes, and equations on the `ASTNode` type. With inherited attributes, equations do not have to explicitly traverse the parent (with `getParent`) to get information from the context, thus making the node structure-shy both to the type of its parent and to its position among the children of its parent. With broadcasting, an inherited attribute of a node is computed by an ancestor node, and is independent on all intermediate nodes in the AST. With collection attributes, the collection value is constructed from contributions anywhere in the AST, independently of other nodes. The use of equations on the most general node type `ASTNode` can be used, e.g., for generic traversal of the tree, without having to take specific node types into account.

All these mechanisms make it easy to write high-level structure-shy specifications where new node types can be added that either reuse or adapt the specification, e.g., by overriding equations. At the same time, a specification can be complex to understand, since equations that may apply to a specific node type may be located in many different places in the specification. Understanding how to extend and adapt a specification can therefore be non-trivial. Tooling that can help in understanding and debugging RAG specifications is a largely unexplored area. An example of a simple tool is DrAST [Lindholm et al. 2016] that can visualize ASTs and their attribute values, allowing a developer to explore a specification interactively.

3.4 Design Trade-Offs

Design involves trade-offs, and just because a principle is possible to apply, it does not mean that it has to be followed slavishly. The goal of the above principles is to make specifications concise, modular, and extensible, but there may also be costs involved, both in performance and in complexity of the specification. Some attributes (rewrites, circular, collection) have higher overhead than others (synthesized, inherited, HOAs), and refactorings to the simpler attributes can lead to more efficient but less elegant and/or less structure-shy code.

Furthermore, it is possible to combine attributes with imperative code. One way is to encapsulate an imperative computation by an attribute, in which case the imperative computation must only have local effects. Another way is to simply let imperative code use attributes. The border between what is computed by attributes and what is computed by imperative code can then be a design trade-off. In JastAdd, external tools can use the AST and access the attributes for any purpose. The external tools can treat the AST as if it is fully evaluated with all attributes according to the specification, although the evaluation of the attributes will actually not happen until they are accessed.

Encapsulating imperative code by an attribute can sometimes be preferable to breaking down the computation into many small attributes. An example of this trade-off is the problem of assigning a unique number to each node in an AST. One design is to use a map attribute in the root node and let the equation right-hand-side do an imperative traversal to fill the map. An alternative design is to define an attribute `pred` that refers to the predecessor in a preorder traversal, and define the number of each AST node as one more than the number of its predecessor [Öqvist 2018, Chapter 3.7]. This latter design may be seen as following the *declarative thinking* and *AST-embedded data structures* principles more closely. But defining the `pred` attribute is fairly complex, and the solution might be less efficient than the map attribute because of the many small attributes that need to be evaluated.

An example of adjusting the border between imperative code and attributes is code generation. In a more declarative design, the code can be a synthesized attribute at the root node that is accessed and printed to a file by the main program. In a more imperative design, code generation can be a side-effectful method that traverses the AST, using attributes to compute code and printing it to a file during the traversal.

4 Patterns

We wanted to identify a number of RAG patterns that solve commonly occurring problems, that are not completely trivial, and that consequently help practitioners take the "next step" in learning about JastAdd RAGs—after reading introductory tutorials and mastering the basics of how different attribute kinds work.

We did not attempt to create a *pattern language* [Alexander 1977], covering the complete domain of language implementation. Rather, a key source of inspiration were the 23 object-oriented design patterns, where a few objects and methods interact in a "microarchitecture" [Gamma et al. 1995]. In a similar way, we describe microarchitectures with a few AST nodes and attributes.

To identify the patterns, we started with a brainstorming session with JastAdd developers to document all patterns and anti-patterns they knew about. Then we systematically checked articles describing RAG mechanisms, and experience papers discussing language tools built with JastAdd. We also looked at course material (in particular the Compilers course, EDAN65, given at Lund University).

Table 1. Patterns, (A)ttribution mechanisms used, and Usage. Mechanisms are ↑ (synthesized), ↓ (inherited), → (higher-order), () (parameterized), *i* (intrinsic reference), *b* (broadcasting), *o* (equation overriding), and *I* (attribute interface).

Pattern	A	Usage
P1: Lookup	↑, ↑(), ↓(), *b*	Name lookup
P2: Local map	↑	Name lookup
P3: Expected type	↑, ↓	Type analysis
P4: Double dispatch subtype comparison	↑(), *o*	Type analysis
P5: Reified structure	→	Reification
P6: Null higher-order attributes (HOAs)	→, ↓, b	Reification
P7: On-demand loading	→()	Reification
P8: Local compile-time instantiation	→, *i*	Expansion
P9: Shared compile-time instantiation	→(), *i*	Expansion
P10: Desugaring via HOA	→	Expansion
P11: Plugin component via interfaces	*I*, *o*	Reuse
P12: Plugin component via HOA	→, *i*	Reuse

Out of around 20 candidate patterns, we decided on 12 that we deemed to be the most interesting ones, and that would be the most useful to practitioners, as shown in Table 1. As seen from the table, we have introduced a few main groups of usage to give some organization of the patterns. For each pattern, we also list the key attribute mechanisms used in the pattern.

We noted that there were some attribute mechanisms that did not turn up in any of the patterns, notably collection attributes, circular attributes, and rewrites. The reason is that we deemed the canonical examples for these mechanisms to be sufficient documentation. However, including these examples as patterns is something that could be considered for future work.

For each pattern, we show a structure diagram with nodes and attributes, illustrating a typical use of the pattern. The node and attribute names correspond to *roles* in the pattern. In applying the pattern, there could be several different node types or attributes that correspond to the same role, similar to how ordinary object-oriented design patterns work.

4.1 Lookup (P1)

Intent: Specify name analysis in a flexible way.
Structure: (see Figure 3)

- Each `Use` node connects to a declaration, by a reference attribute ↑`decl`
- ↑`decl` is implemented using an inherited attribute ↓`lookup(id)`, parameterized with the identifier
- A node with the `Scope` role implements scope rules. It can introduce specific scopes implemented by synthesized attributes, like ↑`localLookup(id)` for a local scope. It can provide equations for ↓`lookup` of its descendents, where it can delegate to other specific scopes, like ↑`localLookup`, and to a context `Scope` by using its own ↓`lookup` attribute.

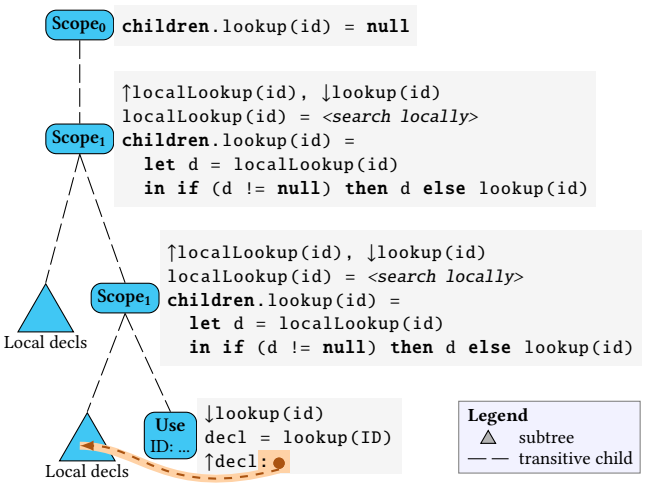

Figure 3. Lookup (P1)

Variants: While the example structure shows only nested scopes, `Scope` nodes can also define attributes by delegating to specific scopes in distant `Scope` nodes referred to by reference attributes. This can be used, for example, to handle inheritance. Multiple overlapping name spaces (for different-kinded names) are typically handled by repeating the pattern for each separate name space. Overloading can be handled by letting ↓`lookup` return a set of declarations that is later disambiguated. Declare-before-use for children in a list is handled by using the child index in the ↓`lookup` equation. If different children have different visibility rules, different ↓`lookup` equations can be given for them.
Discussion: The pattern is a typical example of using an AST-embedded data structure, with both the name bindings and scope rules embedded as attributes in the AST. The pattern is structure-shy in that the ↓`lookup` definitions use broadcasting, allowing nodes between `Uses` and `Scopes` to be ignored. The pattern differs from the "Environment" pattern in traditional attribute grammars, where all visible names for a given node are defined as an inherited attribute ↓`env`. The Environment pattern typically uses an abstract data type with an elaborate implementation to share and update similar environments [Kastens and Waite 1991], and thus does not follow the AST-embedded data structure principle.
Examples: This pattern is used in virtually any JastAdd-based compiler, and is discussed in [Ekman and Hedin 2007b].

4.2 Local Map (P2)

Intent: Replace search by map to improve performance.
Structure: (see Figure 4)

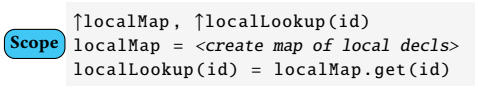

Figure 4. Local Map (P2)

- In the Lookup pattern, the simplest way to implement a ↑`localLookup` attribute is to linearly search a subtree for a particular name. The subtree will then be searched once for each different name that is queried.
- In the Local Map pattern, the ↑`localLookup` queries a ↑`localMap` attribute instead of doing the search. The ↑`localMap` attribute traverses the subtree to compute a map with all local declarations. Because of memoization, this computation is done at most once. This improves complexity from quadratic to linear.

Discussion: This pattern can be ignored for toy languages, but is important for real tools. It trades off a bit of the *AST-embedded data structure* principle to gain performance.
Examples: This pattern is used in production-quality compilers like ExtendJ and JModelica.org. See, e.g., the attribute ↑`TypeDecl.localFieldsMap` in the API documentation of ExtendJ. Another example use is in one of the RAG-based implementations of the train benchmark by Mey et al. [2020b].

4.3 Expected Type (P3)

Intent: Simplify type checks by performing them in expressions rather than in the context.
Structure: (see Figure 5)

- Each expression has an attribute ↑`type` describing the actual type, computed from subexpressions or variables.
- Each expression has an attribute ↓`expectedType` describing the type expected by the context.
- The actual and expected types should be compatible. Incompatibilities can be reported as compile-time errors. Compatible types that are different can be used for generating type-conversion code, e.g., from `int`s to `float`s.
- The checks and conversions can be performed in the expression instead of in the context which simplifies the specification.

Examples: This is a classical pattern in attribute grammars, and is described in [Alblas 1991]. Examples can be found in the Oberon-0 compilers in JastAdd [Fors and Hedin 2015] and Kiama [Sloane and Roberts 2015]. A concrete example is shown below, where the expected type for the right-hand side expression of an assignment depends on the type of the left-hand side variable:

```
// Abstract grammar production:
Assignment ::= Var Expr

// Equation defining expected type of rhs of assignment:
Assignment.Expr.expectedType = Assignment.Var.type
```

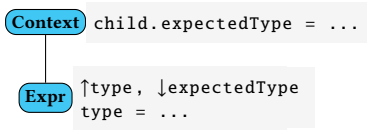

Figure 5. Expected Type (P3)

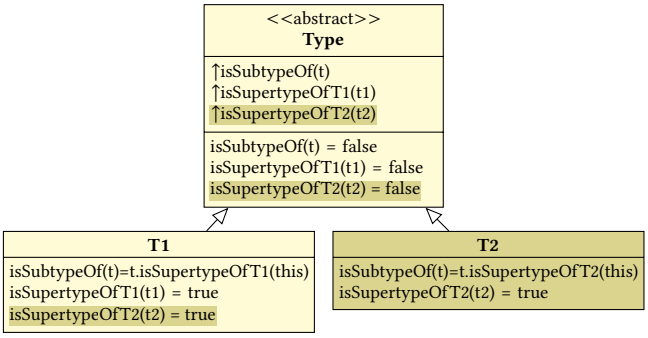

Figure 6. UML diagram for Double-Dispatch Subtype Comparison (P4). This example specifies that `T2` is a subtype of `T1` and that subtyping is reflexive. `T2` can be specified modularly in a separate aspect (highlighted in light brown).

4.4 Double-Dispatch Subtype Comparison (P4)

Intent: Specify subtype comparison in an extensible way.
Structure: (see Figure 6)

- The types in the implemented language are represented by an AST class `Type`, and subclasses thereof, e.g., `IntType`, `FloatType`, etc. (`T1`, `T2` in the diagram).
- To check if one type is a subtype of another, `Type` nodes have an attribute ↑`isSubtypeOf(t)`. The implementation uses double dispatch to disambiguate the two types.
- Each `Type` subclass, say `T1`, introduces an attribute ↑`isSupertypeOfT1(t)` with the default value `false`. Any type that might be a supertype of `T1` provides an equation for this attribute. This way, unrelated types can rely on the default behavior.
- Each class `T` implements ↑`isSubtypeOf(t)` by simply delegating to ↑`t.isSupertypeOfT(this)`, thereby implementing the double dispatch.
- Extending the language with a new type, say `T2`, can be done in a separate aspect, providing the ↑`isSubtypeOf(t)` equation, the ↑`isSupertypeOfT2(t)` attribute and equations for it, as well as equations for ↑`isSupertypeOfX(t)` for any type `X` that `T2` might be a supertype to.

Examples: This pattern is based on the general double dispatch pattern [Ingalls 1986]. It is used for type checking in several JastAdd compilers, including the ExtendJ Java compiler [Ekman and Hedin 2007b]. The pattern avoids having to list all possible type combinations by viewing types as a

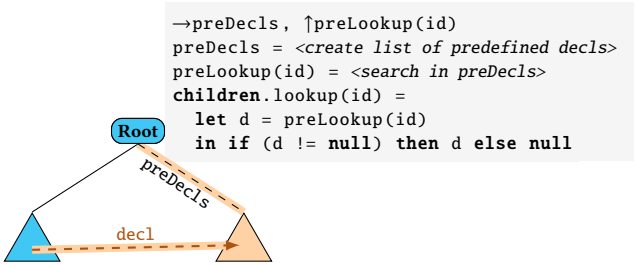

Figure 7. Reified Structure (P5)

lattice and providing default equations. For handling subtype comparison of classes, equations look at the inheritance chain. An example of a modular extension using this pattern is the addition of non-null types for Java in [Ekman and Hedin 2007a].

Discussion: This pattern is a clear example of the AST-embedded data structure principle, where types are represented as AST nodes, and the type comparison implemented as attributes.

4.5 Reified Structure (P5)

Intent: Make implicit elements explicit.
Structure: (see Figure 7)

- Make predefined implicit elements explicit by defining them as higher-order attributes (HOAs), typically in the root of the AST. Examples of predefined elements could be type definitions for primitive types, like `IntType`, `FloatType`, etc. For a Java compiler, another example could be the predefined class `Object`.
- Make the reified elements available throughout the AST using inherited attributes.

Variants: In the diagram, the inherited attribute ↓`lookup` (from the Lookup pattern) is used for making the reified types available throughout the AST. Another variant is to use an inherited reference attribute ↓`root` referring to the root node, and broadcast it throughout the AST, so all nodes can easily access the reified structures via that attribute. In some cases, reified structures are local, and placed in specific nodes instead of in the root.

Examples: This pattern is used in virtually all JastAdd compilers. For example, ExtendJ has a root node, `Program`, with a HOA →`getPrimitiveCompilationUnit` that contains all primitive types. An example of a local reified structure is the extension of Java with Non-Null types. A Non-Null version of each class type is reified and placed as a HOA of the normal class type [Ekman and Hedin 2007b].

4.6 Null Higher-Order Attribute (P6)

Intent: Simplify the specification by reifying missing elements and using the null object pattern.
Structure: (see Figure 8)

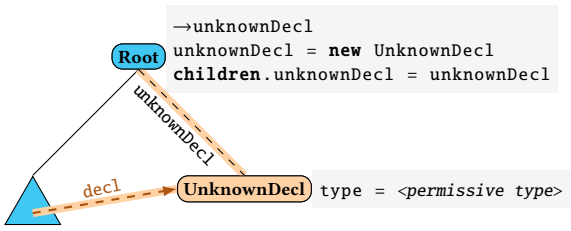

Figure 8. Null Higher-Order Attribute (P6)

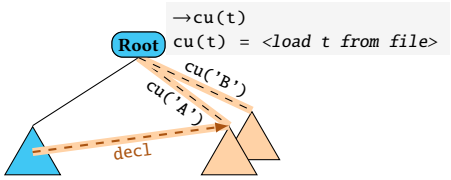

Figure 9. On-Demand Loading (P7) of compilation units.

- Instead of using `null` to represent missing elements, like the missing declaration of an undeclared variable, reify the missing element as a HOA in the root, using the Reified Structure pattern. Typically call it `unknownDecl` or `unknownType`.
- Make the type of the HOA a subtype of the general kind of element, e.g., `Decl` or `Type`.
- Add attributes to the HOA subtype so that it can be treated just like an ordinary element. Typically implement the attributes to be as permissive as possible to avoid that errors are propagated. E.g., for a missing type, make it compatible with all other types.
- This pattern allows client code to be simplified by not having to distinguish between missing and real elements.

Examples: This pattern is based on the Null Object pattern [Woolf 1997]. It is used in most JastAdd compilers, and is discussed in [Hedin and Magnusson 2003]. As an example, the JastAdd implementation of Oberon-0 uses the pattern for missing types (but not for missing declarations) [Fors and Hedin 2015].

4.7 On-Demand Loading (P7)

Intent: Dynamically load files depending on usage.
Structure: (see Figure 9)

- During analysis of a file, dependencies on other files may be discovered, making it necessary to load those files in order to proceed with the analysis. To specify this in a declarative way, use a parameterized HOA.
- The parameterized HOA is typically placed in the root, using the Reified Structure pattern. The parameter specifies which file to load. When the attribute is accessed, the file is read and parsed into an AST subtree.

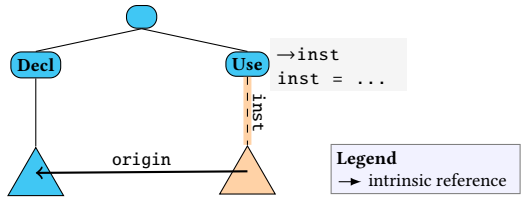

Figure 10. Local Compile-Time Instantiation (P8)

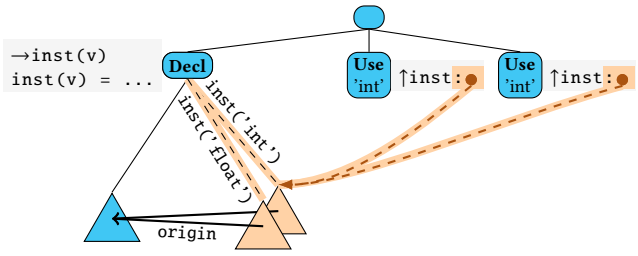

Figure 11. Shared Compile-Time Instantiation (P9)

Due to memoization, multiple accesses of the same file will reuse the same subtree.

Examples: This pattern is used in ExtendJ where both source code files and bytecode files can be loaded dynamically. The attribute is called →`Program.getLibCompilationUnit` and is parameterized with the fully qualified type name. The dynamically loaded source files are also checked for errors. This is implemented using robust iterators [Kofler 1993] over the HOA arguments, since the checks might cause more source files to be loaded.

4.8 Local Compile-Time Instantiation (P8)

Intent: Instantiate and expand definitions at use sites.
Structure: (see Figure 10)

- For uses of macro-like definitions where compile-time instantiation is needed, use a HOA to represent the expanded instance of the definition.
- The HOA subtree might have an intrinsic reference attribute pointing back to the definition, to share information common to all instances.

Examples: The modeling language Modelica has compile-time instantiation of objects, and complex typing rules that allow the structure of objects to be changed at the object allocation site. This is solved using this pattern in the JModelica compiler [Åkesson et al. 2010b]. This pattern is also used for visualization and data-flow analysis for the automation language Bloqqi, which has compile-time instantiation of blocks [Fors 2016].

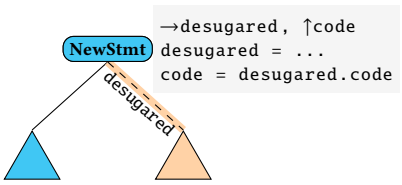

Figure 12. Desugaring via HOA (P10)

4.9 Shared Compile-Time Instantiation (P9)

Intent: Instantiate a specialized variant of a definition and share it between uses of the same variant.
Structure: (see Figure 11)

- For parameterizable definitions, like generic types, use a parameterized HOA to represent the different variants used. Multiple uses of the same variant reuse (point to) the same HOA subtree.
- Similar to P8, a HOA subtree might have an intrinsic reference attribute pointing back to the definition, to share information common to all variants.

Examples: Generics in Java are solved using this pattern in ExtendJ. A generic class has a higher-order attribute that creates a specialized variant of the class with field and method declarations, where the type parameters are substituted with the type arguments. However, the class variants do not include method bodies, since Java uses type erasure for code generation, but reuses these from the original class definition. The attribute is called →`GenericTypeDecl.lookupParTypeDecl`. The technique is described in [Ekman and Hedin 2007b].

4.10 Desugaring via HOA (P10)

Intent: Translate new language construct to core language.
Structure: (see Figure 12)

- For a new language construct, use a HOA (e.g., →`desugared`) to construct a semantically equivalent subtree using the core language.
- The original sugared construct can delegate some computations to the desugared variant, e.g., code generation.
- Other computations, like error reporting, can be done in terms of the original sugared variant, to not confuse the user of the language.

Examples: An example of this pattern is the ExtendJ implementation of *Try With Resources* that was added in Java 7. The pattern is applied to transform the statement into a regular *Try* statement, in order to reuse code generation [Öqvist and Hedin 2013]. Another example is the JastAdd and Silver Oberon-0 compilers [Fors and Hedin 2015; Kaminski and Van Wyk 2015], where `CASE` statements are desugared to `IF`s, and `FOR` to `WHILE` for code generation. For the Silver implementation, the delegating equations can be left out,

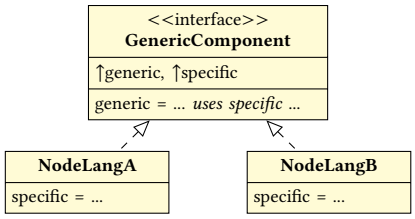

Figure 13. UML diagram for Plugin Component via Interfaces (P11)

because they are implicit due to Silver's support for *attribute forwarding* [Van Wyk et al. 2002] (not supported in JastAdd).

4.11 Plugin Component via Interfaces (P11)

Intent: Create a generic component that can be reused for different languages.
Structure: (see Figure 13)

- A generic component can be defined by specifying interfaces with attribute declarations and equations. To apply the component, add the interfaces to selected classes in the source language.
- The classes can provide equations that define attributes and override equations in the interface, and thereby adapt the behavior to the language.

Examples: An implementation of intra-procedural control- and dataflow uses this pattern [Söderberg et al. 2013]: An interface `CFGNode` captures the general behavior of a node in a control-flow graph, with ↑`succ` and ↑`pred` attributes. Default equations make it easy to apply and adapt the module to a specific language. The dataflow component builds on the control-flow component and defines, e.g., liveness as attributes on the `CFGNode` interface.

Another use of this pattern is scope rule attribution modules as discussed by Kastens and Waite [1994]. They used syntax-independent nonterminal symbols that correspond to interfaces, and supported overriding of equations.
Discussion: A limitation of this pattern is that a component can only be applied once for a language. For example, it might be desirable to construct two control-flow graphs but with different granularity. This is not supported by this pattern since a class can only implement an interface once.

4.12 Plugin Component via Higher-Order Attributes (P12)

Intent: Create a generic component that can be reused for different languages.
Structure: (see Figure 14)

- Define a generic component with an abstract grammar and attributes. The component is reused by creating a mapping from the source language to the abstract grammar in the reusable component. The mapping creates a subtree that is attached to the source AST as a higher-order attribute, with possible intrinsic reference attributes pointing back to the source AST.

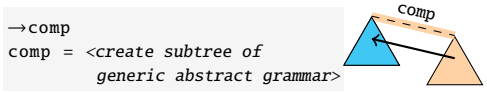

Figure 14. Plugin Component via HOA (P12)

Examples: This core ideas in this pattern were described by [Saraiva 2002], but for grammars without reference attributes. [Mey et al. 2020a] describes the full pattern with back links using reference attributes, and shows how it can be used for developing language independent components for cycle detection analysis and variable shadowing analysis.
Discussion: In comparison to P11, this pattern does not have the same limitations, since a component can be applied multiple times to the same language by defining different mappings. It is, however, a bit more heavyweight than P11 in that defining a grammar mapping may be more work than just applying interfaces.

5 Empirical Study

To better understand how well the proposed principles and patterns capture use of JastAdd RAGs, we performed an exploratory empirical study with 14 practitioners. We sought to answer the following research questions:

- **RQ$_1$** How do practitioners learn to use JastAdd RAGs?
- **RQ$_2$** How do practitioners develop JastAdd RAGs in practice?
- **RQ$_3$** To what extent do practitioners follow the presented principles?
- **RQ$_4$** To what extent do practitioners use the presented patterns?

5.1 Methodology

We ran four semi-structured focus groups [Kontio et al. 2008] with participants from industry and academia. Each focus group was run as a two-hour session with the following structure; a presentation of around an hour about the principles and patterns proposed in this paper[1], a short break, and then a focus group for approximately an hour using the interview protocol included in Appendix A. Our questions covered use of JastAdd RAGs, and in that context principles and patterns. For use of JastAdd RAGs we encouraged participants to help us identify potential improvements of the tool and tell us about issues.

During the focus group session, the first author of the paper gave the presentation, while the second author ran the

[1]Some patterns and principles were renamed after the focus groups; P8 was called 'Compile-time instantiation', P9 was called 'Definition variant', P11 was called 'Attribute interfaces', P12 was called 'Augment AST with reusable subtrees'. The principle of *AST-embedded data structure* was previously called 'The AST is the data structure'.

focus group (with the first author taking notes). One focus group was carried out in person, while three were carried out over video. All focus groups were recorded to complement the taken notes. The resulting material from the focus groups was analyzed by the first and second author and summarized with key observations.

The composition of the focus groups is illustrated in Figure 15, showing the experience of the participants and the division of participants between industry and academia. Participants were identified via past and current collaborations of the authors. The specific composition of each group was dependent on who volunteered to participate at each contacted collaborator, where a contact person would ask around among colleagues. While we strived to get different kinds of groups in terms of industry/academia and beginner/experienced, we did not aim for a specific composition within each focus group.

Threats to Validity In recruiting participants for the focus groups, there may be *selection bias* in who decided to participate and *sampling bias* in our selection of people to contact. For the latter, we aimed to vary the experience of participants and their affiliation (industry vs. academia). We further aimed for focus groups of at least three participants [Kontio et al. 2008] but ended up with one smaller group (AExp). We also planned for in-person focus groups but had to run all but the first over video due to unexpected circumstances (covid-19). The final focus groups may not be representative for all users of JastAdd RAGs, but we tried to get variation to cover different kinds of users (industry / academia, novice / experienced). To compensate for *response bias*, we encouraged participants to give frank feedback about issues and we told them we wanted to identify ways of improving the JastAdd system. Our study further focuses on JastAdd RAGs and may not generalize to other RAG systems.

5.2 Results

In the following subsections, we address each research question and list key observations.

5.2.1 RQ_1: How do Practitioners Learn to Use JastAdd RAGs?

Several participants mentioned that they have learned about JastAdd RAGs in a compiler course (INov, IExp, AMix), and beyond that primarily from looking at examples (INov, AMix), for instance, from the ExtendJ compiler. They further mention that they talk to colleagues, via conversations (AMix) or code review (INov), and they may also inspect the Java code generated by JastAdd (AMix).

Observation 1: Participants learn from examples, collegial knowledge, and by inspecting the generated Java code.

When trying to understand a JastAdd RAG, participants mention a couple of pain-points having to do with documentation and tool support. They mentioned that it can be difficult to locate attributes (AExp) and there is no support for code browsing (INov). As a reaction to the difficulty of reading someone else's JastAdd RAG, one participant said "it is easier if you have written everything yourself" (INov). On the specification level, participants mention a lack of documentation of features, for instance, for how interfaces work (AMix), or interaction between features, for instance, how HOAs are traversed (IExp). They also mention that "rewrites are difficult to understand" (IExp) and that collection attributes may be computed in unexpected ways.

Observation 2: Participants lack support to more easily understand a JastAdd RAG specification, and the hidden details of how the declarative specification is evaluated may hinder adoption of features.

5.2.2 RQ_2: How do Practitioners Develop JastAdd RAGs in Practice?

For this question we analyzed the focus group results with regard to common software development activities; documenting, coding, and testing.

Documenting Participants mention that it can be difficult to locate attributes (AExp), but also express appreciation for available tools helping with documentation of attributes[2] (AExp). One participant mentioned that the company may require models such as sequence diagrams for a system and expressed uncertainty about how to create such diagrams for a JastAdd RAG (INov).

Observation 3: Participants appreciate existing tool support for documentation, but struggle to document and model RAG artifacts to comply with company standards.

Coding Participants in three of the groups develop JastAdd RAGs together (INov, IExp, AExp), with the industry groups (INov, IExp) using software development practices like code review. In the remaining academic group, participants primarily develop JastAdd RAGs separately, but discuss with colleagues to solve issues. In the experienced industry group (IExp), the team has developed practices manifesting as antipatterns. They especially mention two antipatterns; 1) 'don't put cached attributes on nodes that occur a lot in the AST', due to the added memory cost, and 'don't use rewrites', they are nice but tricky to understand.

Observation 4: Practitioners in industry collaborate on joint RAG specifications to a larger extent than in academia, and develop antipatterns.

Participants express appreciation over the possibility to use Java tooling, for instance, for debugging, but they also miss tool support for JastAdd RAGs and mention some issues due to this. For instance, the difficulty of locating attributes

[2]JastAdd RAGs can be documented via tools generating documentation for attributes in a style similar to Javadoc.

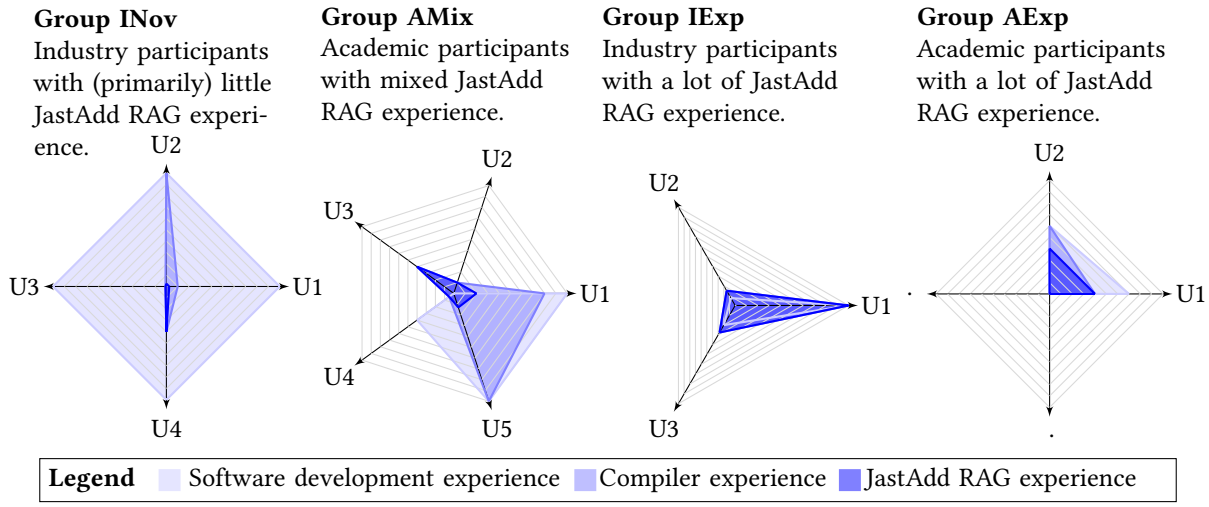

Figure 15. Composition of focus groups, number of JastAdd RAG users (U) in each group and their experience in software development, compilers, and JastAdd RAGs. All axes are in years and experience is shown up until 10 years.

is mentioned as something that may lead to code duplication (AExp). Another participant mentions using the IntelliJ IDE for Java but that, for instance, code completion doesn't work very well because it exposes a lot of internal generated methods (AExp). Participants would also appreciate more support for aspects, for instance, to show dependencies between aspects (AExp).

Observation 5: Participants would like more tool support, especially for locating of attributes, editor services like code completion, and support for seeing dependencies between aspects.

Participants experience a lack of convention for how to manage aspects and find refactorings challenging. One participant mentioned that it can be difficult to structure files with aspects and to know where to put attributes, with the end result being that things are being renamed and moved around a lot (INov). Difficulties to refactor or change a RAG was also mentioned; aspects can be difficult to refactor due to the lack of dependency information (AMix), a grammar change may lead to a lot of equation updates (AExp), and switching between a rewrite or a HOA solution can be difficult when equations need to be added (AMix).

Observation 6: Participants lack coding conventions for JastAdd RAGs, especially on the aspect level, and refactoring a JastAdd RAG can be challenging.

Testing Participants mentioned how they test their JastAdd RAG and related this to other testing that they do in their software development (INov), for instance, while they typically use unit tests for testing of Java classes, they rarely do this for their RAG. Instead, they primarily use acceptance and function tests. They further mention that they have difficulty expressing test coverage for JastAdd code due to uncertainty about to what extent they should test generated internal code.

Observation 7: Participants find it unclear how to best test a RAG artifact and how to comply with company standards for testing.

5.2.3 RQ_3: To What Extent do Practitioners Follow the Presented Principles?

Reviewing the results from the focus groups with each principle in mind they reason about declarative thinking and structure-shy in particular, while they mention no issues with using AST-embedded data structures, which they appear to accept as natural when using JastAdd RAGs.

Observation 8: Practitioners appear to embrace using AST-embedded data structures.

Declarative thinking The extent with which participants reason about problems declaratively varies and appears to correlate with experience. One group expressed their way of reasoning about solutions as "attributes - that's how we solve problems" (IExp), while another group expressed appreciation for gradually being able to adopt declarative concepts (INov). The gradual adoption was also expressed as an active choice to incur technical debt in order to get deliverables ready (INov). Another case where participants decide to not use a declarative specification is when they want to stay close to a documented imperative algorithm (IExp). In addition, the notion of backsliding into imperative Java code was expressed as something that easily happened (AMix).

Table 2. Use of patterns, summarized for each focus group. Y (for Yes) means used and W (for Want) means that they would like to use a pattern but haven't yet.

	Pattern											
Group	1	2	3	4	5	6	7	8	9	10	11	12
INov	Y	Y	W	W	W	W	W					
IExp	Y	Y			Y	Y	Y	Y				
AMix	Y	Y	Y	Y	Y	Y		Y	Y	Y	Y	Y
AExp	Y	Y	Y		Y	Y	Y		Y	Y	W	Y

Observation 9: Practitioners gradually embrace declarative thinking by partly using imperative code, or actively use imperative code to stay closer to a documented imperative algorithm.

Structure-shy specification Participants generally write structure-shy specifications but mention that they may step away from a structure-shy specification for performance reasons (AExp, IExp). Especially circular attributes and collection attributes was mentioned as not providing the needed performance; the fix-point evaluation sometimes used unnecessary iterations for circular attributes (AExp), or because too many nodes were visited when computing collection attributes (IExp). The combination of circular collection attributes was also mentioned as having performance issues (AExp). In addition, one group mentioned using `getParent` more than they should, and expressed a wish to become more proficient in using inherited attributes (INov).

Observation 10: Practitioners step away from using circular attributes and collection attributes to improve performance.

5.2.4 RQ_4: To What Extent do Practitioners Use the Presented Patterns?

The patterns used and the patterns that participants would like to use, as mentioned during the focus groups, are summarized in Table 2. We summarize the content in the table with the following observation:

Observation 11: All groups have used the lookup patterns (P1, P2), and the number of patterns used increase with more experience, but even more so in a research setting.

As listed in Table 2, several participants (INov, AMix, AExp) mentioned patterns that they would like to use. One participant (AExp) summarized their view on the patterns as "this pattern list would be useful to have earlier", referring to when they started to use JastAdd.

Observation 12: The list of patterns appears to aid participants in becoming aware of new patterns, but the applicability of patterns varies - with more applicability at the top of the list than at the bottom.

Participants were generally positive about the listed patterns, but some participants (INov) asked for more examples of when some of them (P8, P9, P10) should be used and mentioned that some did not feel as relevant to them (P11, P12). The latter two patterns, the reuse patterns (P11, P12), were also referred to as being more on a meta level and more abstract by other participants (AExp). As an observation, one participant further pointed out that there are no patterns involving rewrites or circular attributes (INov), and some participants suggested additional possible patterns (AMix); using collection attributes to collect errors, and using inheritance in the type hierarchy to avoid `instanceof` checks by overriding attributes on subtypes.

6 Related Work

Spinellis [2001] describes design patterns for domain-specific languages (DSLs), identifying different main approaches to implementing DSLs, like *Pipeline*, *Language Extension*, and *Source-to-Source* transformation. These patterns are at a much higher level than our patterns, dealing with architectural choices. Mernik et al. [2005] define additional high-level architectural patterns, splitting them into the development phases of *Decision*, *Analysis*, *Design*, and *Implementation*.

Parr [2009] defines a number of patterns for language implementation with Java and ANTLR. These patterns are at a similar level as ours in that they discuss how to effectively use a specific tool and paradigm (in this case Java/ANTLR and imperative OOP) in building language tools. Since the patterns are based on imperative programming, several of them deal with tree walking, like *Tree Walker*, *Tree Visitor*, and *Tree Pattern Matcher*, as well as *Symbol Table* that explains how to traverse the AST in passes to populate a global symbol table.

Cordy [2009] provides a cookbook for how to use the source transformation tool TXL. This work is aimed at approximately the same level as our patterns, namely at how to solve problems once a developer has learned the basics of the specification language. The cookbook does not talk about design patterns, but instead uses the term "coding paradigms", where some are formulated as advice, e.g., *Use sequences, not recursion*, and some as recipes, e.g., *Preserving comment in the output*.

Kats et al. [2009] propose using *attribute decorators* to abstract over common patterns in attribute grammars, and extend the attribute grammar formalism. Examples include defining constructs to avoid copy rules, like the `INCLUDING` mechanism from [Kastens et al. 1982], as well as collection attributes and circular attributes like in JastAdd. They focus on patterns that can be replaced by language constructs. Most of our patterns don't immediately stand out as potential new language constructs. However, some of them might be candidates. For example, it might be useful to express our P2 *Local Map* as a new kind of attribute.

7 Discussion

Reviewing the results of the empirical study, all participants use the presented principles and patterns to some extent. The principles are in some cases used by all (AST-embedded data structures, Observation 8), gradually adopted (declarative thinking, Observation 9), or explicitly not followed to get a specific benefit, like performance or an imperative algorithm (declarative thinking / structure-shy, Observation 10). The patterns are broadly adopted at the top of the list (especially the lookup patterns, Observation 11) and less so towards the bottom (e.g., the reuse patterns, Observation 12). We speculate that the reason may be that the patterns at the end of the list are more abstract and suitable for less common problems, but also more exploratory. This suggests that the patterns towards the top of the list are more mature than those at the bottom.

Further, we saw an interest among participants to try new patterns after becoming aware of them when summarized in the style presented in the paper (Observation 12). We note that participants mention that they primarily learn from examples (Observation 1), but despite having looked at, for instance, the ExtendJ compiler which uses most patterns, they had not been inspired to try these patterns earlier. We see this as an indication that a list of patterns fills a need otherwise not filled by only examples, and we further speculate that summarizing and naming patterns help to clarify whether a pattern is used or not.

Still, more examples are also useful, especially canonical examples of how a certain kind of attribute is to be used, for instance, using collection attributes to gather errors, proposed as a pattern in one focus group. We note that the line between what constitutes as a canonical example and a pattern may not always be clear, for instance, desugaring may be considered to be a canonical example of a HOA but we include this as a pattern (P10). Perhaps there should be a notion of maturity of patterns - we leave exploration of this notion to future research.

Moreover, we note that several observations concern a lack of tool support for activities such as documentation (Observation 3), code search and browsing (Observation 5), and refactoring (Observation 6). We see this as an effect of the often limited resources that can be spared for maintenance of open-source tools in academia, but we also see that available (limited) tool support is appreciated (Observation 3). Beyond a lack of tools, we also see a lack of conventions (especially for aspects, Observation 6) and participants mention developing a team practice to avoid anti-patterns (Observation 4). We observe a lack of guidelines for practitioners (Observation 1), and see the presented principles and patterns as a couple of steps in this direction.

Finally, we see the results from the empirical study as indicators that the proposed principles and patterns capture the practice of developing JastAdd RAGs well. However, we also recognize that our study is limited and could be complemented with a more extensive qualitative evaluation and quantitative study of JastAdd RAG code, which we see as possible directions for future research.

8 Conclusions

To address the need to better document accumulated knowledge of RAGs, we have presented principles and patterns for JastAdd-style RAGs, and evaluated them empirically in 4 focus groups with 14 participants, from academia and industry with varying JastAdd RAG experience. We find indicators that the proposed principles and patterns appear to capture the practice of developing JastAdd RAGs well, help practitioners to become aware of useful patterns, and provide a common language to more efficiently reason about the practice of developing JastAdd RAGs. We hope that JastAdd RAG developers find this work useful, and perhaps we can inspire future research in this direction both for other RAG systems and for other formalisms/tools. An interesting line of future research is to investigate if some of the patterns can be generalized to build a common list for all RAG systems.

Acknowledgements

We thank Jesper Öqvist for contributing to initial discussions of the patterns. We are grateful to the participants in the focus groups for valuable input. We also thank the SLE reviewers and René Schöne for their valuable feedback. This work is partly supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA) in the PiiA project 2017-02371 and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation (KAW).

A Focus Group Protocol

Introduction:

0 Gathered informed consent from participants.

Questions about participants background:

1 How much experience do you have of software engineering? compiler development? development of JastAdd RAGs?

Questions about working with JastAdd RAGs:

2 What have you used JastAdd RAGs for?
3 What type of problems have you solved with JastAdd RAGs?
4 How did you solve these problems? What was simple? What was tricky? How did you proceed when you had a problem?
5 Are there problems that you haven't solved and don't know how to solve?

Questions about principles and patterns:

6 How well do you think the presented principles and patterns describe how you work with JastAdd RAGs?

7 How would you describe your solutions in the form of these patterns?
8 Have you noticed more patterns than the ones we presented?

References

Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. 2010a. Modeling and optimization with Optimica and JModelica.org - Languages and tools for solving large-scale dynamic optimization problems. *Comput. Chem. Eng.* 34, 11 (2010), 1737–1749. https://doi.org/10.1016/j.compchemeng.2009.11.011

Johan Åkesson, Torbjörn Ekman, and Görel Hedin. 2010b. Implementation of a Modelica compiler using JastAdd attribute grammars. *Sci. Comput. Program.* 75, 1-2 (2010), 21–38. https://doi.org/10.1016/j.scico.2009.07.003

Henk Alblas. 1991. Introduction to Attribute Grammars. In *Attribute Grammars, Applications and Systems, International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991, Proceedings (Lecture Notes in Computer Science)*, Henk Alblas and Borivoj Melichar (Eds.), Vol. 545. Springer, 1–15. https://doi.org/10.1007/3-540-54572-7_1

Christopher Alexander. 1977. *A pattern language: towns, buildings, construction.* Oxford university press.

John T Boyland. 1996. *Descriptional composition of compiler components.* Ph.D. Dissertation. University of California, Berkeley.

James Clark, Steve DeRose, et al. 1999. *XML path language (XPath), version 1.0.* Technical Report. W3C.

James R. Cordy. 2009. Excerpts from the TXL Cookbook. In *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers (Lecture Notes in Computer Science)*, João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 6491. Springer, 27–91. https://doi.org/10.1007/978-3-642-18023-1_2

Torbjörn Ekman and Görel Hedin. 2007a. Pluggable checking and inferencing of nonnull types for Java. *J. Object Technol.* 6, 9 (2007), 455–475. https://doi.org/10.5381/jot.2007.6.9.a23

Torbjörn Ekman and Görel Hedin. 2007b. The JastAdd Extensible Java Compiler. *SIGPLAN Not.* 42, 10 (Oct. 2007), 1–18. https://doi.org/10.1145/1297105.1297029

Niklas Fors. 2016. *The Design and Implementation of Bloqqi - A Feature-Based Diagram Programming Language.* Ph.D. Dissertation. Lund University, Sweden.

Niklas Fors and Görel Hedin. 2015. A JastAdd implementation of Oberon-0. *Science of Computer Programming* 114 (2015), 74 – 84. https://doi.org/10.1016/j.scico.2015.02.002 {LDTA} (Language Descriptions, Tools, and Applications) Tool Challenge.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns Elements of reusable object-oriented software.* Addison Wesley.

Görel Hedin. 2000. Reference Attributed Grammars. In *Informatica (Slovenia) (24(3))*. 301–317.

Görel Hedin. 2009. An Introductory Tutorial on JastAdd Attribute Grammars. In *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers (Lecture Notes in Computer Science)*, João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 6491. Springer, 166–200. https://doi.org/10.1007/978-3-642-18023-1_4

Görel Hedin and Eva Magnusson. 2003. JastAdd: An Aspect-oriented Compiler Construction System. *Sci. Comput. Program.* 47, 1 (April 2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0

Daniel H. H. Ingalls. 1986. A Simple Technique for Handling Multiple Polymorphism. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, USA, Proceedings*, Norman K. Meyrowitz (Ed.). ACM, 347–349. https://doi.org/10.1145/28697.28732

Martin Jourdan. 1984. An Optimal-time Recursive Evaluator for Attribute Grammars. In *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings (Lecture Notes in Computer Science)*, Manfred Paul and Bernard Robinet (Eds.), Vol. 167. Springer, 167–178. https://doi.org/10.1007/3-540-12925-1_37

Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and automatic composition of language extensions to C: the ableC extensible language framework. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 98:1–98:29. https://doi.org/10.1145/3138224

Ted Kaminski and Eric Van Wyk. 2015. A modular specification of Oberon0 using the Silver attribute grammar system. *Sci. Comput. Program.* 114 (2015), 33–44. https://doi.org/10.1016/j.scico.2015.10.009

Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. 1982. *GAG: A Practical Compiler Generator.* Lecture Notes in Computer Science, Vol. 141. Springer. https://doi.org/10.1007/BFb0034297

Uwe Kastens and William M. Waite. 1991. An Abstract Data Type for Name Analysis. *Acta Informatica* 28, 6 (1991), 539–558. https://doi.org/10.1007/BF01463944

Uwe Kastens and William M. Waite. 1994. Modularity and Reusability in Attribute Grammars. *Acta Informatica* 31, 7 (1994), 601–627. https://doi.org/10.1007/BF01177548

Lennart C. L. Kats, Anthony M. Sloane, and Eelco Visser. 2009. Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. In *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Oege de Moor and Michael I. Schwartzbach (Eds.), Vol. 5501. Springer, 142–157. https://doi.org/10.1007/978-3-642-00722-4_11

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference (LNCS)*, Vol. 2072. Springer, 327–353. https://doi.org/10.1007/3-540-45337-7_18

Donald E. Knuth. 1968. Semantics of Context-Free Languages. *Math. Syst. Theory* 2, 2 (1968), 127–145. https://doi.org/10.1007/BF01692511

Thomas Kofler. 1993. Robust Iterators in ET++. *Struct. Program.* 14, 2 (1993), 62–85.

Jyrki Kontio, Johanna Bragge, and Laura Lehtola. 2008. The Focus Group Method as an Empirical Tool in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, Sjøberg D.I.K. Shull F., Singer J. (Ed.). Springer, London, 93–116. https://doi.org/10.1007/978-1-84800-044-5_4

Ralf Lämmel, Eelco Visser, and Joost Visser. 2003. Strategic programming meets adaptive programming. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003, Boston, Massachusetts, USA, March 17-21, 2003*, William G. Griswold and Mehmet Aksit (Eds.). ACM, 168–177. https://doi.org/10.1145/643603.643621

Karl Lieberherr. 1996. *Adaptive object-oriented software, the Demeter method.* PWS Boston.

Joel Lindholm, Johan Thorsberg, and Görel Hedin. 2016. DrAST: an inspection tool for attributed syntax trees (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, 176–180. https://doi.org/10.1145/2997364.2997378

Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars – their evaluation and applications. *Science of Computer Programming* 68, 1 (2007), 21–37. https://doi.org/10.1016/j.scico.2005.06.005

Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (2005), 316–344. https://doi.org/10.1145/1118890.1118892

Johannes Mey, Thomas Kühn, René Schöne, and Uwe Assmann. 2020a. Reusing Static Analysis across Different Domain-Specific Languages using Reference Attribute Grammars. *Art Sci. Eng. Program.* 4, 3 (2020), 15. https://doi.org/10.22152/programming-journal.org/2020/4/15

Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. 2020b. Relational reference attribute grammars: Improving continuous model validation. *J. Comput. Lang.* 57 (2020), 100940. https://doi.org/10.1016/j.cola.2019.100940

Modelica 2017. Modelica - A Unified Object-Oriented Language for Systems Modeling, Language Specification Version 3.4, The Modelica Association. https://www.modelica.org

David A. Naumann. 2007. Observational purity and encapsulation. *Theor. Comput. Sci.* 376, 3 (2007), 205–224. https://doi.org/10.1016/j.tcs.2007.02.004

Jesper Öqvist. 2018. *Contributions to Declarative Implementation of Static Program Analysis.* Ph.D. Dissertation. Lund University.

Jesper Öqvist and Görel Hedin. 2013. Extending the JastAdd extensible Java compiler to Java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*, Martin Plümicke and Walter Binder (Eds.). ACM, 147–152. https://doi.org/10.1145/2500828.2500843

Terence Parr. 2009. *Language implementation patterns: create your own domain-specific and general programming languages.* Pragmatic Bookshelf.

João Saraiva. 2002. Component-Based Programming for Higher-Order Attribute Grammars. In *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings (Lecture Notes in Computer Science)*, Don S. Batory, Charles Consel, and Walid Taha (Eds.), Vol. 2487. Springer, 268–282. https://doi.org/10.1007/3-540-45821-2_17

Anthony M. Sloane. 2009. Lightweight Language Processing in Kiama. In *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers (Lecture Notes in Computer Science)*, João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 6491. Springer, 408–425. https://doi.org/10.1007/978-3-642-18023-1_12

Anthony M. Sloane and Matthew Roberts. 2015. Oberon-0 in Kiama. *Sci. Comput. Program.* 114 (2015), 20–32. https://doi.org/10.1016/j.scico.2015.10.010

Emma Söderberg, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. 2013. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Sci. Comput. Program.* 78, 10 (2013), 1809–1827. https://doi.org/10.1016/j.scico.2012.02.002

Emma Söderberg and Görel Hedin. 2012. *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking.* Technical Report 98. Lund University. LU-CS-TR:2012-249, ISSN 1404-1200.

Emma Söderberg and Görel Hedin. 2015. Declarative rewriting through circular nonterminal attributes. *Comput. Lang. Syst. Struct.* 44 (2015), 3–23. https://doi.org/10.1016/j.cl.2015.08.008

Diomidis Spinellis. 2001. Notable design patterns for domain-specific languages. *J. Syst. Softw.* 56, 1 (2001), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Sci. Comput. Program.* 75, 1-2 (2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. 2002. Forwarding in Attribute Grammars for Modular Language Design. In *Compiler Construction*, R.Nigel Horspool (Ed.). LNCS, Vol. 2304. Springer, 128–142. https://doi.org/10.1007/3-540-45937-5_11

Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. 2007. Attribute Grammar-Based Language Extensions for Java. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings (Lecture Notes in Computer Science)*, Erik Ernst (Ed.), Vol. 4609. Springer, 575–599. https://doi.org/10.1007/978-3-540-73589-2_27

Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher-Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*. 131–145. https://doi.org/10.1145/73141.74830

Bobby Woolf. 1997. Null object. In *Pattern languages of program design 3*. Addison-Wesley Longman Publishing Co., Inc., 5–18.