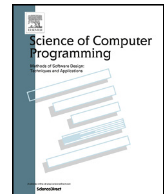
Science of Computer Programming 211 (2021) 102703

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico



PAMOJA: A component framework for grammar-aware engineering

Jackline Ssanyu [a,b,*], Engineer Bainomugisha [a], Benjamin Kanagwa [a]

[a] Department of Computer Science, Makerere University, P.O. Box 7062, Kampala, Uganda
[b] Department of Computer Science, Kyambogo University, P.O. Box 1, Kampala, Uganda

A R T I C L E I N F O

Article history:
Received 14 December 2020
Received in revised form 26 June 2021
Accepted 12 July 2021
Available online 21 July 2021

Keywords:
Component-based software development
Grammarware
Component frameworks
Programming environments
Software architectures

A B S T R A C T

PAMOJA is a Java-based component framework for Grammar-Aware Engineering (GAE) in an Integrated Development Environment (IDE). The PAMOJA system is being developed to explore the possibility of Component-Based Software Development (CBSD) in the grammarware field. Our main goal is to develop a coherent set of small GAE components, where each component is dedicated to a single well-defined task. The components should fit into a general-purpose framework like NetBeans or Eclipse and it should be possible to manipulate them inside the IDE just like any other component. This paper describes the PAMOJA architecture supporting this development style. We illustrate its use with the aid of examples, and present a case of composing new components at a higher level from the existing GAE components. For this case, we use a hybrid text/structure editor application as an example. This case study serves as a proof of concept of our approach.

© 2021 Elsevier B.V. All rights reserved.

1. Introduction

1.1. Grammar-aware software

Many applications do data processing based on grammars. The importance of grammars in the design of software languages and in their implementation is well-known. From a formal grammar, such as (Extended) Context-Free Grammar ((E)CFG), attribute grammar, signature and syntax definition formalism, many language-based tools can be constructed, such as lexical analyzers, parsers, tree walkers, type checkers, pretty-printers or even full compilers/interpreters and language-specific editors. Moreover, grammars and their associated tooling have much wider applicability, such as:

- The lexical analysis technology of regular expressions and finite automata can also be used for search-and-replace algorithms in text editors, spelling correction, smart-phone editors, and pattern matching in genomes, to name just a few applications.
- Parsers can be used for all kinds of structured input, as long as the structure can be described by means of a formal grammar. Some well-known examples are XML-parsing and natural language parsing.
- Other kinds of applications also need some grammar-based tooling, such as:

* Corresponding author at: Department of Computer Science, Kyambogo University, P.O. Box 1, Kampala, Uganda.
E-mail addresses: jssanyu@kyu.ac.ug (J. Ssanyu), ebainomugisha@cis.mak.ac.ug (E. Bainomugisha), bkanagwa@cis.mak.ac.ug (B. Kanagwa).

https://doi.org/10.1016/j.scico.2021.102703
0167-6423/© 2021 Elsevier B.V. All rights reserved.

- *Theorem proving:* The front-end of a theorem prover, inputs logical formulae (i.e., definitions, propositions, and tactics) and checks that they are valid through parsing and construction of an internal representation (usually an Abstract Syntax Tree (AST)). This AST is then forwarded to the back-end for proving.
- *Language-theoretic security:* This is a relatively new research area [1,2] with potentially high impact. It aims to improve security and trustworthiness of software with un-trusted inputs by basing all handling of such inputs on formal language theory. Sets of valid inputs should be considered as a formal language and all input-handling routines as a recognizer for that language. Formal grammars and grammar-related tools like scanners, parsers and matchers are instrumental for such an approach.

The increasing use of grammars in various software development scenarios led to the establishment of an engineering discipline for grammarware [3], which comprises of grammars, grammar-aware techniques and all grammar-dependent software (e.g., lexical analyzers, parsers, compilers, program analyzers, optimizers and translators). The grammarware field is very broad — the work described in this paper has limited scope. It is focused on a subset of grammar-aware techniques — those that exploit CFGs, for example the traditional compiler front-end techniques of lexical scanning, parsing, tree building, pretty-printing and structure editing, and development of software that use these techniques in some way.

1.2. Component-based software and rapid application development

Component-Based Software Development (CBSD) is a major approach for software development. With CBSD a desired software system is assembled by combining and configuring prefabricated software components. Each component is dedicated to a particular task. The components fit into a so-called component framework, an architecture which provides facilities for customization of and cooperation between components [4]. Some well-known component frameworks are NetBeans [5], Eclipse [6], IntelliJ IDEA [7] and Delphi [8]. Such frameworks provide an Integrated Development Environment (IDE) with facilities, such as: (1) building and depositing of components in the *repository* (also called a *palette*); (2) assembling of components retrieved from the repository to build many different software; and (3) composing of components into composite components and store these in the repository as well. This type of environment facilitates Rapid Application Development (RAD) style. A developer selects components from a palette and drops them on a form, and connects them in one way or another. Each component may be customized by editing its *properties* in dedicated *property editors*. Additional custom behavior may be provided to the components by attaching some *event handlers*; thus large parts of an application may be constructed with little or no coding. An assembly of components in the application communicate with each other by sending events when their internal state changes.

The development style just sketched need not be restricted to visual components and GUI design, however. It may be applied as well to non-visual components and other application areas. For instance, the commercial product StatBeans [9] is a JavaBeans component library for statistical data analysis. It contains components for data sources, statistical calculations, tabular software and graphical displays. The components are designed to be embedded in user-written applications or placed on webpages. Development can be done in the development style outlined above. Some other application areas for which component toolkits have been developed are: simulation modeling [10], cryptographic protocols [11], and 3D-modeling [12].

1.3. Grammar-aware components for RAD

In principle the field of grammarware lends itself well to application of component-based methods. It consists of well-defined tasks (e.g., scanning, parsing, tree building, and pretty-printing) with good theory, and a variety of well-studied algorithms for implementing the tasks. Existing tools, however, have not fully utilized the benefits of CBSD, particularly in the construction of language-based tools. Existing research is mostly focused on realizing techniques and tools to simplify composition of a new language from an existing set of reusable language components (e.g., [13–17]). Many language-based tools are packaged as standalone systems [18–21], in practice, they cannot be integrated with ease into general-purpose IDEs in order to be used in the development of GAE-based applications. Several approaches with the aim of making language development tools easier to use and easier to integrate into general-purpose programming environments like Eclipse and IntelliJ IDEA have been explored in systems like Spoofax [22], Rascal [23] and Xtext [24]. Although these systems are very effective in the hands of experts, they usually have a steep learning curve, which can be an obstacle in cases where only a modest amount of grammar-aware techniques is required.

1.4. Introducing PAMOJA

This paper introduces PAMOJA, a Java-based component framework which provides support for GAE in an IDE like NetBeans [5] and Eclipse [6]. We are making the already existing language engineering technology available in a lightweight component toolkit. The main design idea behind PAMOJA is to enable users with less expertise in grammar-aware techniques, such as students and software developers, to incorporate these techniques into their applications with less effort. Rather than designing a large special-purpose framework, our main goal is to develop a coherent set of small grammar-aware components and generators — each dedicated to a single well-defined task — which fit into the RAD style. This yields the following benefits and features for the grammarware area:

Flexibility: The components can be easily customized and combined in different ways to develop grammar-aware applications. For example, a scanner component and a parser component can be combined with a general purpose text editor component to develop a language-specific text editor.

Lightweight: Many kinds of applications do some language processing as a necessary but subordinate task, as discussed in Section 1.1. Usually, a developer has to generate pieces of code for the required grammar-aware tasks and find a way of incorporating them into the main application. In such cases, using a large system for the grammar-aware tasks is an overkill and distracts from the main tasks. Using some lightweight grammar-aware components in an IDE makes language engineering technology more accessible to main stream software developers with little effort.

Ease of use: Many software developers use an IDE which supports RAD of applications from components; thus when a developer has a set of compliant grammar-aware components available, he can use language engineering technology without the problems of learning to use a large system and relating it to his application.

Variety: Having at one's disposal a variety of components for a certain grammar-aware task makes it easier to experiment with different techniques and to pick the component most suited for the task at hand. This can be a great asset in a language prototyping lab and educational environment. For example, in a compiler lab, having several parser components parsing according to different strategies, working on a common grammar representation and producing a common type of parse trees, makes it easy to experiment with the different parsing strategies.

Our contribution in this paper is two-fold. First, we present a lightweight component architecture, named PAMOJA, that brings the benefits of CBSD to the construction of GAE applications. Second, we present a case study demonstrating how to compose new components from existing PAMOJA components. We use a hybrid text/structure editor application as an example. Additionally, we have implemented our architecture on the NetBeans environment.

1.5. Organization of the paper

Section 2 briefly discusses related work. In Section 3 we present the PAMOJA architecture. Section 4 discusses error handling in PAMOJA. In Section 5 we illustrate how to use PAMOJA to construct grammar-aware software, whereas in Section 6 we illustrate how to compose new components at higher levels from the existing grammar-aware components. We use a hybrid text/structure editor application as an example. In Section 7 we discuss the outcomes and limits of PAMOJA with respect to CBSD aspects, and Section 8 presents concluding remarks and future work.

Throughout this paper, our examples are based on the Oberon0 [25] language. Oberon0 has been used in the LDTA [26] tool challenge to implement a compiler for Oberon0, with the goal of being able to compare different language implementation tools. It is a small language, yet it illustrates many important aspects of compilers, making it a good choice for the purpose of illustrating the capabilities of PAMOJA.

2. Related work

The idea of composing a language processing system from subsystems dedicated to specific tasks like scanning and parsing spans decades of research work. A lot of research has gone in generating such subsystems from language specifications. Various tools contain subsystems which were designed to cooperate with each other and with other subsystems. Moreover, the subsystems can be used stand alone in a variety of applications. Some examples are Lex and Yacc [27], JavaCC [28], ANTLR [18], and LISA [29]. Their subsystems could be considered "components".

An early example of a component-based language engineering environment is the ASF+SDF Meta-Environment [30] (a predecessor of Rascal [23]). It is a stand-alone IDE which integrates a number of cooperating tools connected via a coordination architecture, the ToolBus [31]. Examples of such tools are: a user interface, scannerless GLR parser generator, unparser generator, text editors, structure editors, compiler, interpreter, and parse tree repository. The ToolBus controls the interactions between tools by using T-Scripts, that model the possible interactions between tools. Tools can be written in different languages and exchange data using ATerms via the ToolBus. The ToolBus is based on strict separation between behavior which is done inside the tools and tool interaction which is performed in the ToolBus itself. This facilitates the design of flexible tools, reusable in a variety of applications [32]. PAMOJA shares some similar goals with the ASF+SDF Meta-Environment, for instance, flexibility of combining components in new ways, reuse and extensibility. We note these differences. First, in terms of scale, PAMOJA has small components dedicated to one task, all components work on the same datatypes. Second, PAMOJA supports the *observer-observable* design pattern for component interactions and the composite pattern for compositions [33]. This enables basic level components to be composed into a new component. The external behavior of this composition can be abstracted away in an interface to facilitate composition at higher levels, and so on. Third, PAMOJA is based on object-oriented principles which encourage reuse by means of interface inheritance. Forth, PAMOJA is integrated into a general purpose framework, NetBeans, and facilitates the integration of grammar-aware techniques into software applications using the RAD style.

Kiama [34] is a Scala library for language processing where formalisms, such as grammars, parsers, rewriters and analyzers, are embedded into a general-purpose language, Scala, with the aim of simplifying their integration into a programmers development process. The Kiama language processing library consists of components each addressing a single processing

task, like packrat parser based on combinators, semantic analyzer, transformer, translator, pretty printer and code generator. The components can be compiled and executed by the standard Scala implementation. Kiama uses trait and mixin features of Scala to keep components separate from each other and to compose them into language tools (e.g., compilers, interpreters, and static analysis tools) in flexible ways [35]. Kiama and PAMOJA share the idea of simplifying the integration of language processing into the software development process. However PAMOJA's approach emphasizes component-based design concepts and visual component assembly, while with Kiama glue-code has to be written to achieve collaboration among "components".

JastAdd [19] system supports extensible implementation of front-ends, compilers and related tools like source code analyzers, transformation tools and Eclipse-based language-sensitive editors [36]. Generation of tools is based on extensible object-oriented attribute grammar specifications which "allows the tools to be implemented as composable extensible modules" [37]. JastAdd has no support for scanning and parsing, it depends on external Java-based scanner generators (such as JFLex and JLex) and parser generators (such as CUP, Beaver, JavaCC, and ANTLR) that support user-defined semantic actions. The Java code generated by the external scanner and parser generators is incorporated with that of the Java AST classes generated by JastAdd from the abstract grammar specifications [38].

There exist several other systems with the aim of making language development tools easier to use and easier to integrate in general-purpose programming environments. A well-known example is Eclipse IMP (IDE Meta-Tooling Platform) a project "to support the development of richly-featured, language-specific IDEs in Eclipse" [39,40]. Some examples of language processing tools based on Eclipse IMP are Spoofax [41] and Rascal [42]. Another notable tool on the Eclipse platform is Xtext [43].

Spoofax [22] is a textual language workbench. From high-level declarative definitions of a programming language several tools can be derived, such as a scannerless GLR parser, formatter, typechecker, interpreter, compiler and full-featured IDE for Eclipse and IntelliJ IDEA [7]. To facilitate reuse, Spoofax provides a Java-based API which may be used to programmatically compose the generated tools, for instance, to produce a language frontend, or to embed languages and their generated tools directly into users application code [41]. Spoofax is a modular architecture that can be easily extended with new tools such as parser generators.

Rascal [23] is a meta-programming language and IDE Eclipse plugin for source-code analysis and transformation. Rascal supports modular generation of language tools, similar to those generated by Spoofax, however the parser is based on scannerless GLL parsing. To facilitate reuse, Rascal libraries can be used to compose the generated modules, for instance to produce a language frontend.

Xtext [24,43] is another language workbench in Eclipse that supports development of languages and their respective tools, for instance, an ANTLR-based parser, type-safe AST, formatter, typechecker, compiler and full-featured Eclipse-based IDE. Apart from Eclipse, Xtext languages and tools can be integrated into other IDEs such as IntelliJ IDEA, Visual studio Code, and all editors that support the Language Server Protocol.

JastAdd, Spoofax, Rascal and Xtext are language workbenches geared towards developing of software languages and their tooling in general. They are much more complex than PAMOJA, but provide powerful tooling infrastructure for language engineering.

The goal of PAMOJA is not a highly sophisticated language engineering platform but a collection of small grammar-aware components conforming to a standard component model, the JavaBeans component model, that can be integrated into the RAD style of applications that need grammar-based language processing of some kind. To the best of our knowledge, such a component library does not exist yet.

3. PAMOJA architecture

In this section we describe the structure and underlying principles of PAMOJA. We start by presenting the conceptual framework of PAMOJA in Section 3.1 and derive from there the kinds of components which should be present in PAMOJA and how they relate. In Section 3.2 we present the main design principles which guided the design of the PAMOJA component set, before we give an overview of the kind of components currently available in Section 3.3. Then in Sections 3.4 to 3.8 we describe component interfaces, data flow, composition, PAMOJA component model, and the intended use of our PAMOJA component set.

3.1. Conceptual framework

In this subsection we present an overview of the various formal language concepts on which the design of PAMOJA is based. Fig. 1 shows those concepts and their relations graphically. We explain this figure step by step. The elements of PAMOJA have much in common with those occurring in the front-end of a compiler. In compiler construction the analysis phase is usually modeled as a mapping from a textual representation of a program to an AST. It is common to decompose this mapping into a few simpler ones, such as *scanning* (from character strings to symbols), *screening* (filtering out reserved words etc.), *parsing* (from symbol strings to parse trees), *abstraction* (from parse trees to ASTs), and *identification* (linking use of names to their declaration) (see e.g., [44–46]). In PAMOJA some of these mappings are present as well. They occur in the blue part of Fig. 1, more specifically as the upward arrows: a `scanner` maps a 1-dimensional character string (labeled as `Text(1D)`) to a 1-dimensional string of symbols `SymbolStream(1D)`, which a `parser` subsequently maps

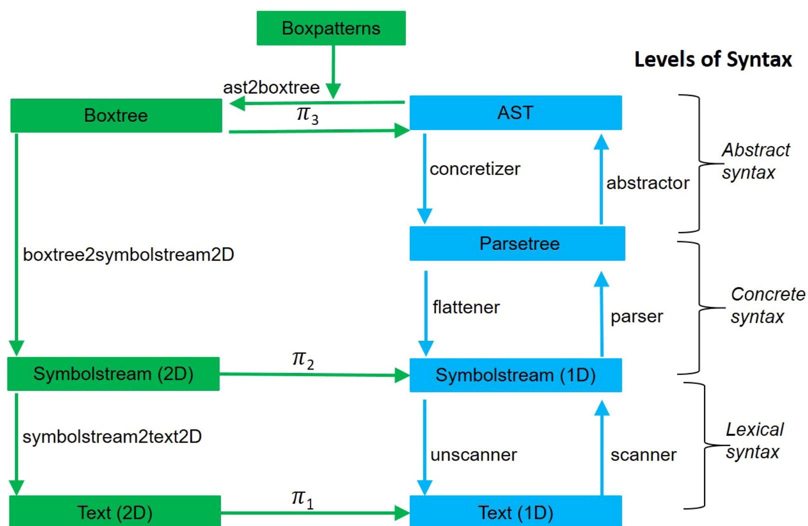


Fig. 1. Conceptual overview of the PAMOJA Component Framework. Boxes denote data, and arrows denote mappings. 1D and 2D stand for one-dimensional and two-dimensional respectively. The kind of syntax that plays a role at each level of transformation is also indicated. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

to a `Parsetree`, which an `abstractor` subsequently maps to an `AST`. Each of these phases is controlled by a particular kind of syntax, indicated in the right margin of Fig. 1.

PAMOJA also deals with mappings in the opposite direction from `AST` to `Text(1D)`, decomposed into `concretizer`, `flattener` and `unscanner` (not be confused with pretty-printers; We explain the connection with those below). These functions each have a simple structure: for instance `flattener` just yields the sequence of leaves of a parsetree, and `unscanner` just appends the text strings corresponding to the individual symbols, with some whitespace in-between. Each of the rectangles in the blue diagram has to satisfy some important mathematical requirements. The downward arrow should be a total injective mapping (otherwise there would be ambiguities), and the corresponding upward arrow should be the inverse of the downward arrow. In simple words: for each data x of the top of a rectangle, if we first apply the downward arrow to it, and then the upward arrow to the result, we should get x again. These requirements are useful for specifying and testing in a modular way.

For practical reasons — readability, and the limited size of paper and screens — text is usually represented 2-dimensionally with line breaks and indentations. To achieve this, layout information has to be taken into account. This leads to the green part of the diagram in Fig. 1, which can be seen as a *pretty-printer*. Layout can be described by means of nested horizontal or vertical arrangements of boxes [47,48]. Given a set of `Boxpatterns`, `ast2boxtree` maps an `AST` to a `Boxtree`, which is subsequently mapped to a 2-dimensional list of lists of symbols (`Symbolstream(2D)`), and then to a list of strings (`Text(2D)`). At each level in this chain, layout information can be projected away using simple projection functions π_1, π_2, π_3 to obtain the counterpart in the blue diagram. Consistency requires that the green/blue rectangles commute.

Fig. 1 should be seen as a *conceptual* diagram. In a concrete application not all elements need be present, and some may be combined. For instance, at the bottom level the `Text(2D)` and `Text(1D)` interpretations may be obtained by different access properties of the same *StringList* class. Also, `scanner` and `parser` mappings may often be merged (for instance in scannerless parsing), thus avoiding an explicit `Symbolstream(1D)` representation. In other cases an explicit symbolstream would be useful to have, for instance when using a backtracking parser. Fig. 1 has been very useful in identifying what kinds of components should be present in PAMOJA and how they relate. This is the subject of the following subsections.

3.2. Guiding design principles

In this subsection we present the general design principles for PAMOJA. The structure of the PAMOJA component toolkit is derived from both Fig. 1 and from the following design principles:

1. **Separation of concerns:** In general for each kind of data, such as text, symbolstream, parsetree, AST, and boxpatterns, and for the notions representing language syntax, shown in Fig. 1, we provide:
   (a) A component which holds the data, maintains its well-formedness, and communicates with other components. Each data has both a structural representation and a textual representation, with a bijective mapping between them;
   (b) One or more views on the data to facilitate inspection of its properties;
   (c) A special-purpose (structure) editor for the data, ensuring consistency thereof.

We give an example of how this principle works. For a grammar we have: (1) a component holding an ECFG data structure and its corresponding textual representation. The component also ensures that the data structure is well-formed, i.e., it correctly represents the formal notion of ECFG. For example, the set of non-terminal symbols and the set of terminal symbols should be disjoint and every non-terminal has a definition; (2) a grammar view for inspecting a grammar and analysis information, such as `nullable`, `first` and `follow` entities; and (3) an editor for editing the grammar in a well-formedness preserving way.

2. **Decoupling of interface from implementation:** PAMOJA components have simple well-defined interfaces. Each interface can be implemented by one or more components. For example, for each of the main mappings in Fig. 1, such as scanning, parsing and abstraction, there exists an interface with possibly several components implementing that interface according to different algorithms and possibly with very different characteristics. This facilitates: easier experimentation with different algorithms for the same task and to get an understanding of them, easier selection of an algorithm that suits a particular requirement, and easier accommodation of new implementations. In Section 3.4 we return to this design principle and present sample interfaces for the scanner and parser mappings.
3. **Integration of generators into a development environment:** Generators play a prominent role in the development of grammar-based language processing tools. Given a suitable form of a language specification, some parts for processing that language may be generated automatically, either as data or source-code, using well-defined algorithms. Therefore, in the PAMOJA framework, rather than using generators as external tools, we prefer to integrate them inside the development environment and make them available via the facilities that are already available there.
   (a) *Generators of language-specific data* are implemented as components observing a change in a language component to adapt their state by generating new language specific data. For instance, an `SLRTables` component contains a "hidden" SLR parse-table generator. When the `SLRTables` component is connected to a `Grammar` component and observes a change in the latter, it will invoke its generator to generate new parse tables, thus adapting its parse tables to the modified grammar. This mechanism is discussed in more detail in Section 3.5.
   (b) *Generators of language-specific source code* are implemented as program wizards. This enables developers to rapidly generate and integrate pieces of source-code into their applications.

   Integration of generators into the development environment improves developer productivity since construction of grammar-aware software is done within a single environment with a favorite IDE.

In addition to the above design principles, PAMOJA components conform to certain design patterns which determine division of tasks and ways of cooperation. The main design patterns applied in the design of the components are: *strategy*, *observer-observable* and *composite*, as defined in [33]. We describe these in Sections 3.4 to 3.6.

3.3. PAMOJA ToolKit

This subsection gives an overview of the kind of components, and wizards of the PAMOJA toolkit that are currently available. See Table A.3 in Appendix A for a full list. This enumeration of components was guided by the conceptual framework in Fig. 1 and the design principles presented in Section 3.2. The current version of PAMOJA offers components for:

- **Language syntax:** These components hold the specifications of a given language. Following design principle 1: (1) there is a grammar component which holds the specification of lexical and concrete syntax, and a signature component which holds abstract syntax; and (2) Views, and editors in form of component customizers, for both the grammar and signature components. A description of PAMOJA language specifications is given in Appendix B.
- **Data structures:** These components hold the main data structures (blue/green boxes in Fig. 1) which represent a language in various forms. Following design principle 1, PAMOJA offers: (1) components which hold the text, symbolstream, parsetree, AST, boxpatterns, and boxtree; and (2) views and editors for these data structures.
- **Mappings between these data structures:** This is a set of cooperating components which deal with the transformation of the various data structures from a concrete textual representation of a program to its corresponding AST and the reverse transformation from AST to concrete text. In accordance with design principle 2, for each mapping there is a public interface which several components may implement.
  - *From* `Text(1D)` *to* `AST`*:* For *lexical scanning* currently there is one scanner component — a Lex-like DFA scanner [45] which is parameterized with a scan table. For *parsing* there are currently three types of parser components which closely cooperate with a grammar and a parsetree builder component. These are: a parser which recursively interprets an ELL(1) grammar, a backtracking parser which recursively interprets an arbitrary non left-recursive grammar, and an SLR(1) parser which is parameterized with parse tables. These three parser types make use of the same representations for grammars and for parse trees. For *abstraction* there is an abstractor component parameterized with a node factory object which creates different kinds of AST nodes. For *abstraction* there is an abstractor component parameterized with a node factory object which creates different kinds of nodes used in the construction of ASTs.
  - *From* `AST` *to* `Text(2D)`*:* There are three components: (1) a format tree (box tree) producer, *AST2Boxtree*, parameterized with box patterns (see Appendix B.3); (2) a format tree consumer, *Boxtree2Symbolstream*, which generates a two-dimensional symbolstream; and (3) a two-dimensional text producer, *Symbolstream2Text*.

```
interface IScannerBase{
    void setText();
    void nextSym();
    Symbol getSym();
    boolean finished();
    void reset();
}

interface IBacktrack extends IscannerBase{
    int mark();
    void recall(int p);
}
```

Listing 1: The scanner interfaces. The upper code listing shows the main interface for all scanners, and the lower code shows the interface implemented by scanners which have to deal with backtracking parsers.

```
interface IParserBase{
    CParseResult parse();
    CParseResult parseNonTerminal(CNonTerminal nt);
    void parseNext();
}
```

Listing 2: The parser interface.

- **Generators of language-specific data/source-code:** Following design principle 3(a), PAMOJA currently offers components which cooperate with a grammar to generate scan tables and parse tables which are used to control DFA-based scanners and SLR(1) parsers. There are also views to facilitate inspection of scan tables and parse tables. In accordance with design principle 3(b), PAMOJA offers program wizards which allow a user to rapidly generate language-specific source-code files in Java, from a suitable language specification. Currently the available program wizards are:
  - **ELL(1)ScannerGenerator:** takes a lexical grammar and generates source-code for an ELL(1) scanner.
  - **RecursiveDescentParserGenerator:** takes an ELL(1) grammar given in ECFG and generates Java source-code for a deterministic recursive-descent parser. The recursive descent parser generator wizard will be described in detail in Section 5.2.
  - **SignatureAPIGenerator:** takes a signature of a language and generates a hierarchy of AST classes. We explain how a PAMOJA signature is specified and how the "SignatureAPIGenerator" works in Appendix B.2.
- **Syntax highlighting:** There is a symbol style customizer component which maps grammar symbols to symbol categories and a component editor which facilitates editing of symbol categories, font and color attributes.

The set of components and program wizards which PAMOJA currently offers at the lexical, concrete and abstract level, along with their descriptions, are summarized in Appendix A. Although the current collection of components offered by PAMOJA is rather limited, it is an open system that can be extended with new components implementing the already defined interfaces, or with new components for other functionality, such as type checking, tree rewriting, and tree matching.

3.4. Component interfaces

Following the strategy pattern [33], PAMOJA components have simple well-defined interfaces. As described in Section 3.2, for each of the mappings (e.g., scanning, parsing, and abstraction) in Fig. 1, there exists small public interface(s) to capture the essentials of that task. In this section we present the interfaces for lexical scanning and parsing as examples.

Listing 1 shows two interfaces for lexical scanners. The main interface is `IScannerBase`. It is minimal, and intentionally so, as it should be implemented by all scanners. It provides methods for initiating the input string, recognizing the next symbol, inspecting the current symbol and its attributes, detecting end of input, and resetting the scanner to the beginning of input. In some cases, support for backtracking is needed as well, by means of methods for marking certain positions in the input and returning to them. Such extra methods can easily be incorporated by defining a new interface `IBacktrack` which extends `IScannerBase` and which specifies these methods. For each of the two interfaces, there may be several scanners implementing the interface according to different techniques. For example, a handwritten scanner, an ELL(1) scanner generated from a lexical grammar, or one that builds a finite automaton.

Similarly, Listing 2 shows the interface `IParserBase`, which is the main interface for parsers. It is also minimal, as it should be implemented by all parsers. It provides methods for parsing an input string starting from either the start expression or with a given non-terminal, and obtaining the next symbol either from a scanner, symbolstream structure, or input string. The parsing methods return an object `CParseResult` which represents the result of the parsing. Via `CParseResult` the parse tree and syntax errors can be retrieved. For some parsing methods it may be necessary to provide an interface with more functionality. For instance, the GLR [49] and GLL [50] parsing methods may return a collection of parse results, rather than just 0 or 1 result. Just as we did with the scanner interfaces, we can easily incorporate this extra functionality by defining a new interface which extends `IParserBase`.

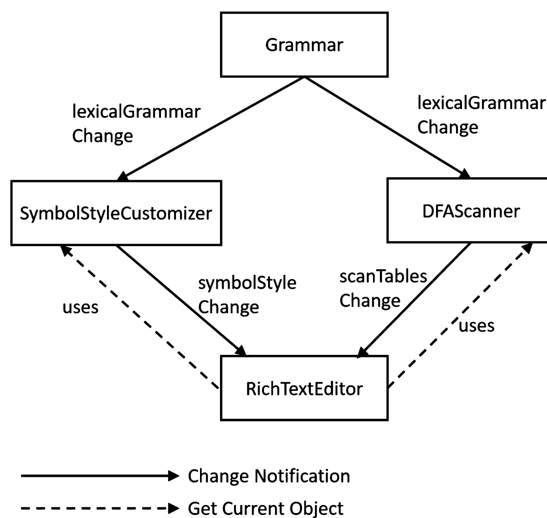


Fig. 2. Data flow among `SyntaxHighlighter` subcomponents.

3.5. Data flow between PAMOJA components

PAMOJA framework implements the observer-observable design pattern [33] to maintain consistency and to aid the flow of information between a variety of cooperating components. Typically, when the state of a PAMOJA component changes, its observers are notified about the state change. For small change notifications, the observers may adjust their own state directly ("push" model); for more involved changes, the observers may react to the notification by consulting the observable directly for additional information ("pull" model). As observers can in turn be observed by others, a spreadsheet-like flow of control between components can be realized. This flow of control becomes particularly interesting when a language specification (e.g., consisting of a lexical and a context-free grammar) is also held in a component and scanner and parser components have their own "hidden" generators and are observers of the language specification. Then, when the language specification changes, scanners and parsers can adapt themselves immediately. In the following example we illustrate how data flows between a set of cooperating components which achieve syntax highlighting. In Section 5.1 we will return to the example and use it as a vehicle to explain how to construct a syntax highlighter application, in the concrete setting of the NetBeans IDE.

Consider the following configuration of components, which together achieve syntax highlighting:

`Grammar:` An observable component which holds a lexical and context-free grammar and which ensures that the grammar is well-formed. In accordance with design principle 1(a) (Section 3.2), there are two properties to access and modify the value of a grammar, namely, `GrammarStructure` which represents the internal structure of a grammar, and `GrammarText` which represents the grammar in text form. Whenever the grammar is changed this component checks the well-formedness of the new grammar before accepting it. In addition to this check, this component invokes a grammar analyzer which analyzes the grammar and annotates each grammar element with some general information, such as the sets `First`, `Last`, `Follow`, and the predicates `reachable`, `nullable`, `left recursive` and `ELL(1)`. This information is useful for most parsers and signals potential causes of problems, as will be described later in Section 4.

`DFAScanner:` This component is a lexical scanner, based on a DFA. The component has a hidden scanner generator, based on the Lex algorithms [45], and it observes the `Grammar` component for changes in the lexical grammar.

`SymbolStyleCustomizer:` This component maps grammar symbols to symbol categories and symbol categories to font and color attributes. It observes the `Grammar` component and maintains consistency between its own valid symbol domain and the symbols defined in the grammar. It has a component editor (see Fig. 5) which can be used to edit the list of symbol categories (e.g., identifier, keyword and number), set their desired color and font attributes, and classify symbols into appropriate categories. The editor can be invoked at design time, but also at runtime, if so desired.

`RichTextEditor:` A text editor with text coloring and font style capabilities. It uses the `DFAScanner` and `SymbolStyleCustomizer` components to scan its text and to render the recognized symbols respectively, and it observes their changes.

Fig. 2 depicts the flow of information between the various cooperating components of a syntax highlighter. The architecture depicted in Fig. 2 has a close resemblance with the classical Model-View-Controller (MVC). When the lexical part of

the Grammar component changes, a property change is observed by both the DFAScanner and the SymbolStyleCustomizer components. The following two scenarios take place:

- The SymbolStyleCustomizer reacts to the observed property change of the Grammar by adjusting its domain of valid symbols to that of the new grammar. The SymbolStyleCustomizer in turn sends a property change notification to its observers. In this case RichTextEditor observes the change and reacts by using the DFAScanner to scan its text and display the symbols recognized according to the new mapping in the SymbolStyleCustomizer.
- The DFAScanner reacts to the change in the Grammar component by invoking its hidden scanner generator to regenerate its scan tables for the new language. The change in the DFAScanner is in turn observed by the RichTextEditor, which uses the adjusted DFAScanner and the SymbolStyleCustomizer to highlight its text according to the new grammar.

3.6. Composition of PAMOJA components

PAMOJA components are hierarchically composable. Therefore, a group of cooperating components may be turned into a new PAMOJA component, which subsequently can be used like any other component. Components are composed by connecting their appropriate interfaces. The visual development environment aids in the process of linking the different components together through their interfaces. We give three examples of possible PAMOJA component compositions:

- **Syntax Highlighting:** Reconsider the syntax highlighting example of Section 3.5. As this task is a feature of many text editors, a composite component SyntaxHighlighter might be constructed for it, containing an assembly of the DFAScanner, SymbolStyleCustomizer and RichTextEditor as subcomponents. The SyntaxHighlighter composite may be an observer of a Grammar component, whereas it internalizes the collaboration between DFAScanner, SymbolStyleCustomizer and RichTextEditor. In Section 5.1 we return to this example in the concrete setting of the NetBeans IDE.
- **Going from text to AST:** Since the three sub mappings: scanner, parser, and abstractor, which map text to an AST often occur together in front-ends that involve parsing, one can assemble variants of Text2AST components, according to different strategies, in order to satisfy different kinds of user requirements. For example, using the existing PAMOJA components a Text2AST composite may contain an assembly of:
  - a DFAScanner, table-driven or interpreting parser (e.g., SLRParser, LimitedBackTrackerParser), TreeBuilder, and Abstractor sub components; or
  - a DFAScanner, generated recursive-descent parser and Abstractor sub components.

  To facilitate reuse, again Text2AST may be an observer of a Grammar and make use of AST class implementations which can be generated from a signature using a "SignatureAPIGenerator" wizard (see Appendix B.2), using similar techniques like those employed in systems such as ApiGen [51] and JastAdd [52].
- **Structure editor:** A *structure editor* (also known as a *projectional editor*) maps an AST representation of a program to on-screen artifacts that can be edited directly. Pure structure editing can be described in isolation, and there are many systems of this kind (e.g., see [53]). With PAMOJA, we can construct a language-independent StructureEditor composite (e.g., see Fig. 3) containing an assembly of a PanelTreeView, TreeEditor, and a GenericTreeView. The StructureEditor composite makes use of AST classes, and may be an observer of a Language component (containing lexical, concrete and abstract syntax) and a Presentation component (containing symbol style and format specifications).

3.7. PAMOJA component model

The PAMOJA component model has been based on the JavaBeans component model [54] and on the Swing component framework. JavaBeans provides an attractive component model for realizing most of the requirements of our PAMOJA components. In particular, the most interesting aspects are:

- The JavaBeans event model nicely accommodates the implementation of observer/observable behavior of our PAMOJA components.
- Persistence of data elements is offered through automatic Java serialization mechanism.
- The Java interfaces give a nice implementation concept to realize the idea of interface definitions of our PAMOJA components.
- Java's introspection mechanism enables a component interface to be exposed to developers and component integration environments at both application runtime and design time. No extra specification files are required to be maintained independently from the component's code.

Although PAMOJA has been based on the JavaBeans component model, its requirements can be realized in other environments.

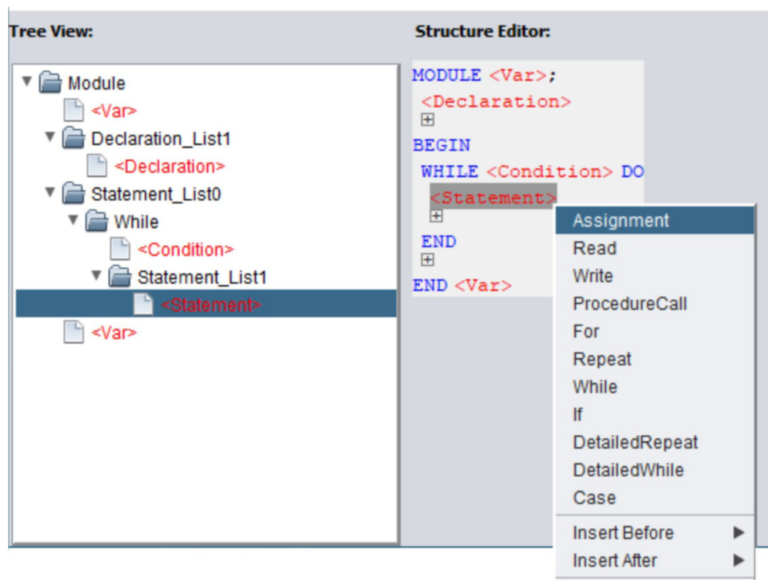


Fig. 3. Sample Structure Editor assembled from existing PAMOJA components. On the Left: a `GenericTreeView` component displaying an AST of Oberon0 program. On the right: a `PanelTreeView` component, supporting structure editing by invoking a popup menu to replace a statement hole with an assignment.

3.8. Use cases

The PAMOJA component framework supports development of grammar-aware software in ways commonly found in modern IDEs like NetBeans, Eclipse and Delphi:

- It allows a **drag-and-drop** development style for quick and easy construction of large parts of a language front-end with little or no coding. Components are dragged from a palette and dropped on a form; their properties are set by means of property editors and/or customizers. See Section 5.1.
- It provides **wizards** that allow a user to rapidly generate language-specific pieces of code. The user is guided through a multi-step dialog to set his preferences about the project, language, and the language-specific service, which are then used to customize some existing PAMOJA components and/or generate some specialized source code-based grammar-aware components. See Section 5.2.

Additionally, the grammar-aware components may also be used to build more conventional stand-alone grammar-aware applications, such as scanner generators, parser generators or a "grammar workbench".

4. Error handling in PAMOJA

In software systems there are potentially many places where something might go wrong due to erroneous input or other unexpected situations. In this section we describe a few common categories of errors in language processing and how PAMOJA currently prevents or handles them.

4.1. Error handling for language specifications

We focus here on context-free syntax. For lexical syntax and abstract syntax the approach is similar.

In PAMOJA context-free syntax is specified by means of an ECFG, described later in Appendix B.1. As discussed already in Sections 3.2 and 3.5, the `grammar` component is responsible for maintaining a well-formed ECFG and for computing ECFG analysis information which is useful for most parsers. All PAMOJA's parser components closely cooperate with a grammar component; thus, they operate only on well-formed and analyzed grammars.

Parsers which operate directly on the grammar may require additional grammar properties to operate correctly. For instance, a backtracking recursive descent parser requires that the grammar is not left-recursive, and a deterministic recursive descent parser requires the grammar to be ELL(1). This information can directly be obtained from the grammar analysis produced by the grammar component. Other parser components will invoke a hidden parser generator, which may perform additional checking. For instance, the SLRParser component will invoke the hidden SLR parser generator, which will attempt to generate SLR(1) parse tables and signals shift-reduce or reduce-reduce conflicts.

4.2. Error handling by the parsers

Most parsers will have to deal with input which is not syntactically correct. PAMOJA includes some basic form of error handling. The backtracking parser systematically tries alternatives and returns the first complete parse it finds, or else returns failure. The deterministic recursive and SLR(1) parsers return either a complete parse or else failure at the first symbol that is not a valid continuation of the input processed thus far. Currently no attempt is made to continue parsing or to "repair". A more general solution for error handling is needed. This is a subject for future work.

4.3. Consistency between components

A characteristic feature of PAMOJA is the way it preserves consistency between components by means of the observer-observable pattern, as described in Subsection 3.5. This is especially useful for maintaining consistency between a language specification and the components depending on it, such as the lexical grammar and the scanner in Fig. 2. That example also shows a potential risk, however. Consider the case where the lexical grammar has been extended with the definition of a new symbol, but the SymbolStyleCustomizer has not yet been extended with style attributes for the new symbol. In that case the RichTextEditor will ask for style attributes which have not been specified. In this particular case the solution is that the SymbolStyleCustomizer returns some default style values. The general approach is that components should react in a stable and predictable manner to unexpected situations.

4.4. Data structure integrity for trees

PAMOJA consists of tree data structures of various kinds, for instance parse trees, ASTs and box trees. All node classes ultimately descend from base classes, which provide general operations for inspecting and modifying nodes, like returning the number (N) of subtrees, and getting and setting the ith subtree respectively, with precondition $0 \leq i < N$. These general operations are used by all PAMOJA components that provide tree related services, like the tree builder, the structure editor and the tree views. More specific kinds of tree nodes may have suitable access methods that allow type safe access to subtrees. A detailed description of an example implementing an AST node for a `While` statement is given in Appendix B.2.

There is another risk of compromising data structure integrity for trees, however. Although in PAMOJA each tree is represented by means of objects with pointers to subtrees, not every structure with objects and pointers is a tree. There might be shared subtrees, cycles, and even dangling references. This will never occur inside PAMOJA itself — parsers and tree builders will construct "pure" trees, and the structure editor will preserve the purity of trees.

But purity is not guaranteed for data structures created outside PAMOJA and passed to a PAMOJA component. Checking absence of shared subtrees and of cycles is possible, but not until after checking absence of dangling pointers, which is hard. At the moment we do not have a satisfactory solution for this problem but we feel that a more general solution is needed.

This problem is not unique to PAMOJA, however. It is a well-known problem with component software that the component developer may have full control over the consistency of the components and their interactions, but not over the ways clients use or abuse the components. See for example the Section "Units of dispute" in [4].

5. Example use cases

This section illustrates how to use the PAMOJA component toolkit to develop grammar-aware applications, in the concrete setting of the NetBeans IDE. We show two examples. One using the drag-and-drop development style to construct a simple syntax highlighter application using the existing components presented in Section 3.5, and one using the wizard development style to generate source-code of a deterministic recursive-descent parser. See Appendix B.1 for a guide to an example of a PAMOJA-grammar which describes Oberon0 language.

5.1. Constructing a syntax highlighter

When we open the NetBeans IDE (see Fig. 4), we see on the component palette among others the following four components that suit our purpose: `Grammar`, `DFAScanner`, `SymbolStyleCustomizer` and `RichTextEditor`. To construct a syntax highlighter from these components, we follow these steps:

1. Drop a `Grammar` component on the `JFrame` and use its property editor to set the lexical grammar. The property editor shows the following two properties among others: `grammarStructure` is used to open a component editor (see Fig. B.11) for editing parts of the grammar and `grammarText` is used to set the grammar in text form. Use any of the two properties to set the grammar. When finished, the `Grammar` component checks that lexical grammar is well-formed, updates its contents and sends a change notification to its observers, if any.
2. Drop a `DFAScanner` component on the `JFrame` and connect it to the `Grammar` component by setting its `grammar` property to the instance of `Grammar` placed on the frame in step 1. The following actions take place:
   - `DFAScanner` is added to the list of observers of the `Grammar` component.

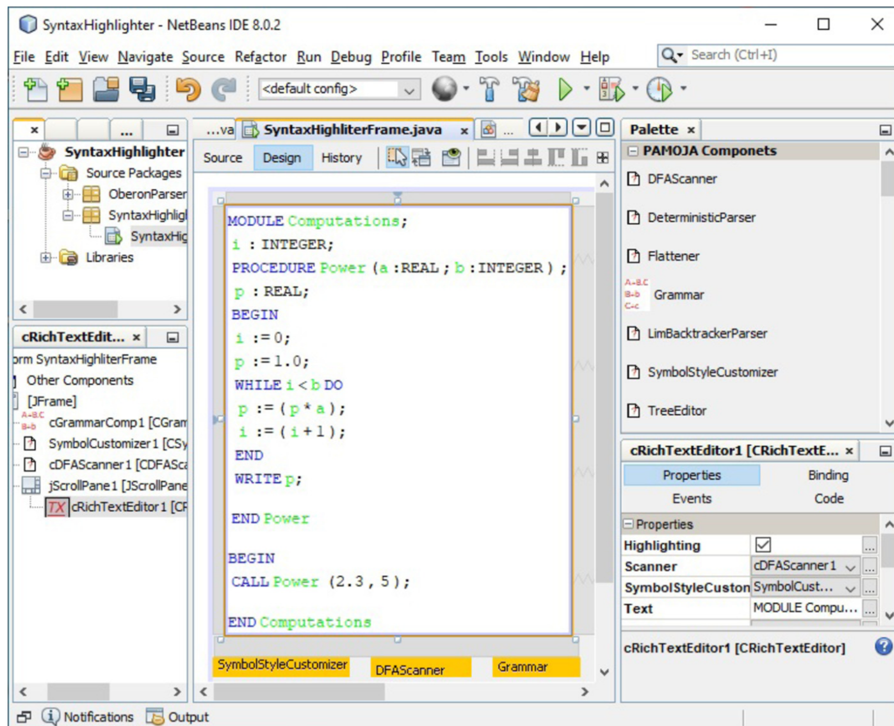


Fig. 4. The NetBeans IDE in design mode shows the component palette in the right-upper window. The middle window is a design form containing four PAMOJA components. When a component is selected, its properties show in the property sheet (bottom right). For instance, the RichTextEditor is currently selected, its properties are shown in a property sheet and can be edited. The value (here: some Oberon0 text) of its Text property is also shown in the component itself. The property Scanner with a value cDFAScanner1 indicates that the RichTextEditor is connected to the DFAScanner component on the form.

- `DFAScanner` invokes its hidden scanner generator to construct its scan tables and sends a property change to its observer components, if any.

3. Drop a `SymbolStyleCustomizer` component on the `JFrame` and connect it to the `Grammar` component by setting its `grammar` property to the instance of `Grammar` placed on the form in step 1. Similarly, The following actions take place:
   - `SymbolStyleCustomizer` is added to the list of observers of the `Grammar` component.
   - `SymbolStyleCustomizer` adjusts its domain of valid symbols to that of the grammar.

   Open a component editor for a `SymbolStyleCustomizer` and set the values of symbol categories, their desired color and font and classify symbols into their appropriate categories (see Fig. 5).
   When the `SymbolStyleCustomizer` closes a property change is sent to its observer components, if any.
4. Finally, drop a `RichTextEditor` component on the `JFrame`.
   Enter text in its `text` property. Connect it to the `DFAScanner` and `SymbolStyleCustomizer` by setting its `scanner` and
   `symbolStyleCustomizer` properties respectively. `RichTextEditor` component reacts in the following way:
   - it starts to observe changes in the scan tables of the `DFAScanner` and symbol properties of the `SymbolStyleCustomizer`.
   - it uses both the `DFAScanner` and `SymbolStyleCustomizer` components to highlight the text according to the lexical syntax of the grammar.

Your finished interface should now look like the middle window of the screen shot in Fig. 4.

Note that this example ends without having written a single line of code. However, if we want the syntax highlighter to have a facility for editing the grammar and the symbol styles at runtime, we can for instance add buttons or menu items and event handlers to activate the customizers for the `Grammar` and `SymbolStyleCustomizer` components.

5.2. Generating a recursive descent parser

To give an impression of how to use the wizard development style to generate language-specific source code-based PAMOJA components and add them into a developers project, we explain in general terms how we generate a deterministic recursive-descent parser which is based on a ELL(1) grammar.

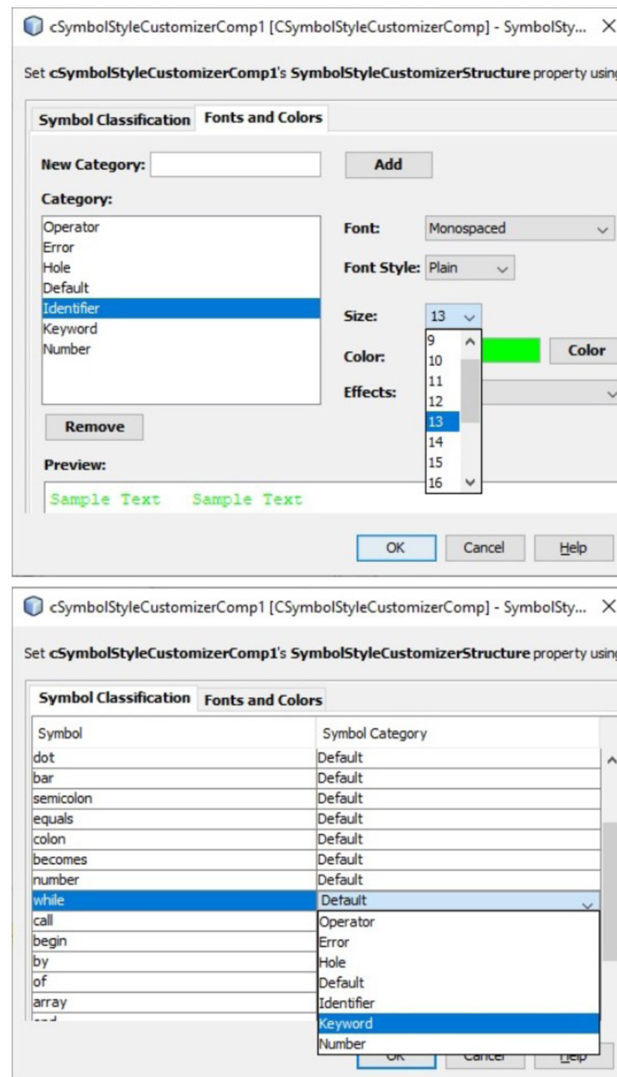


Fig. 5. An example of a component editor for the SymbolStyleCustomizer. On top: Tab, Fonts and Colors lists the current symbol categories with their font and color properties. On bottom: Tab, Symbol Classification lists Oberon0 symbols and their corresponding categories. For instance, `while` symbol is being set to the category keyword.

To begin construction of the parser, the developer must supply basic information about the project and the language. This is done through a "Open ParserGen Wizard" (Fig. 6) that collects the name of the package and the project in which the parser will reside, and so on. The next step in the process is the provision of the grammar. Some customization about the grammar and the parser to be generated is provided through parameters set in the wizard (e.g., augmenting the grammar, and including tree construction constructs in the parser implementation). Finishing the wizard initiates the generation of a deterministic recursive-descent parser component comprising of source files, which are integrated into the developers project and compiled.

The PAMOJA wizards for ELL(1) scanner generator and signature API generator, work in a similar style as described above.

6. Case study: hybrid text/structure editor

This section demonstrates how to assemble new components from the existing grammar-aware components. We use a hybrid text/structure editor application as an example, and show how to assemble two versions of components for hybrid editing at different levels of composition. The first version is a small and simple component, `CoreHybridEditor`, which provides the essential functionality for hybrid editing, and which can be adapted to specific user needs by connecting it to other PAMOJA components for the editors, support tools and specifications. The second version is a more specialized component, `BasicHybridEditor`, with fixed editors and support tools which has fewer connections. A user can just connect it to the formal specifications of the desired language.

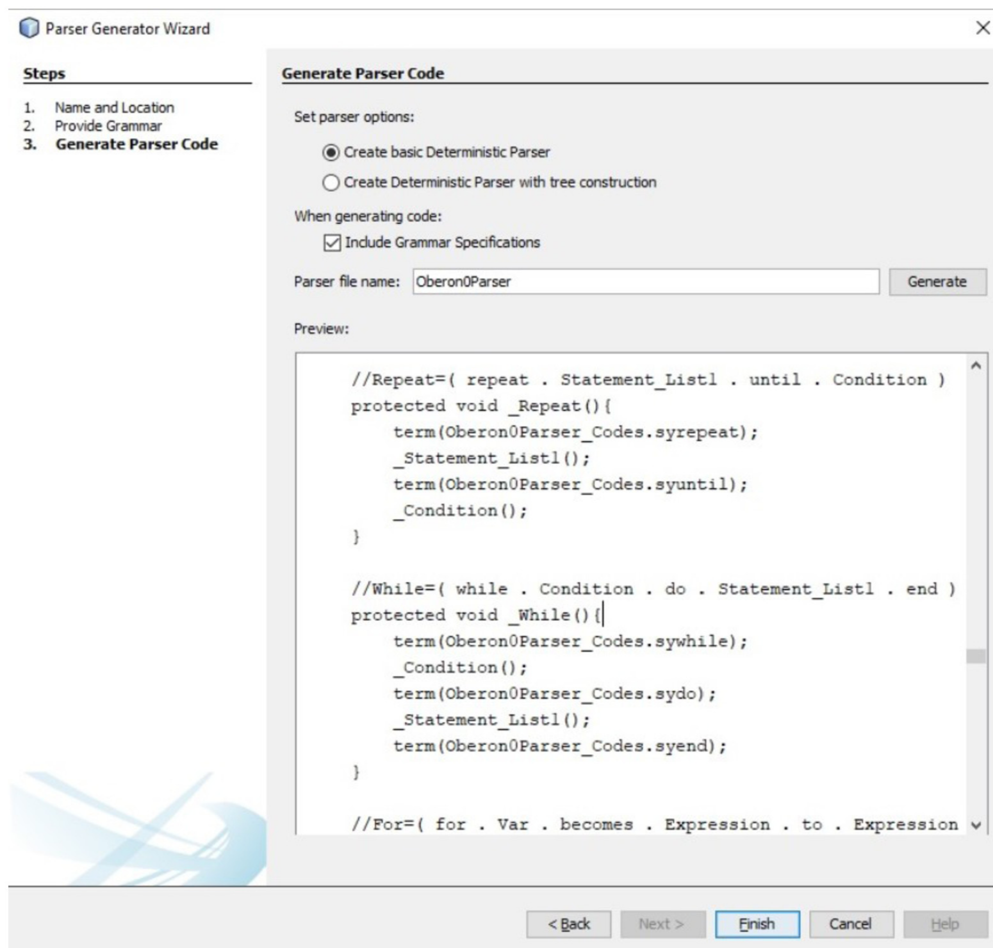


Fig. 6. Parser generator wizard, at step 3, with a preview window showing sample code generated for the *Repeat* and *while* production rules of the Oberon0 grammar.

6.1. Scenario description

In this subsection, we give an informal description of our hybrid text/structure editor.

A hybrid text/structure editor is a combination of a *text editor* and a *structure editor*, which enables a user to switch back and forth between typing text and manipulating the AST of a program.

A *text editor* enables development of well-formed program text using other support tools, such as: lexical scanner, parser, tree builder and type checker. For well-formed text an AST is produced, else an appropriate error message is generated. A *structure editor* maps an AST of a program to on-screen artifacts that can be edited directly, using *templates* and *placeholders* (also called *holes*). This editor maintains a well-formed but generally incomplete AST with placeholders where other subtrees still have to be supplied. In addition there is a *focus*, a variable pointing to a particular node in the AST. The subtree with that node as root is the *focused subtree*. Most editing commands (e.g., replace, delete, cut, copy and paste) apply to the focused subtree.

A *hybrid editor* allows a user to switch back and forth between text editing and structure editing. When the hybrid editor is in structural mode, the text editor is empty. The AST in the structure editor can be navigated using a mouse to select a *focused subtree* which can be edited using the tree editing commands. On switching from structural mode to textual mode the *focused subtree* is mapped to its corresponding textual representation, using the support of a pretty printer (also called a formatter). This textual representation is then displayed in the text editor for text editing, and the structure editor is frozen. While in textual mode and text editing is finished, the text is parsed and when successful a subtree is constructed to replace the focus in the AST. The editor switches back to structural mode. When parsing is not successful the editor stays in textual mode. A text editor, structure editor and their support tools, need to have knowledge of the grammar for the language in question.

To achieve such a hybrid editing functionality, we identify the following minimal requirements:

Table 1
List of components used in the construction of a hybrid editor. Currently PAMOJA does not have a full text editor component - we use a syntaxhighlighter.

HybridEditor element	GAE composite	Sub-components
Text editor	`SyntaxHighlighter`	`RichTextEditor` `DFAScanner`
Structure editor	`PanelTreeView`	`AST2BoxTree` `BoxTree2PanelTree`

Text $\mapsto$ AST	`Text2AST`	`DFAScanner`
		`DeterministicParser`
		`TreeBuilder`
		`Abstractor`

AST $\mapsto$ Text	`AST2Text`	`AST2BoxTree`
		`BoxTree2Stream`
		`Stream2Text`

Languages	`Language`	`Grammar` `Signature`
Presentations	`Presentation`	`SymbolStyleCustomizer` `Patterns`

1. **Editors:** a *text editor* for manipulating text, and a *structure editor* for manipulating ASTs.
2. **Support tools:** for the *bidirectional mapping* between text and AST.
3. **Specifications:** *language* specifications (lexical, concrete, and abstract) serving as a basis for the mappings and for editor operations, and *presentation* specifications, i.e., color and font styles for the text editor, and formatting rules for the mapping from AST to text.

The following sections discuss how the hybrid editor components are constructed and incorporated into the PAMOJA component toolkit.

6.2. Solution approach

Rather than using the basic grammar-aware components to construct a hybrid editor as a monolithic entity, we identified six composite components that each address a single processing element of hybrid editing. The components and their compositions are summarized in Table 1. These components are common and can be used many times in the construction of other applications; hence we constructed them and they are part of the PAMOJA component toolkit.

To realize the hybrid editing functionality we discussed in Section 6.1, we decided to have a flexible component, `CoreHybridEditor`, which can be adapted to specific user needs by connecting to different kinds of components for the editors, support tools and specifications; hence its design was based on the following design decisions.

A `CoreHybridEditor` component:

1. Should conform to the rules and conventions of the PAMOJA framework so that it may be combined with other components of the toolkit.
2. Encapsulates a data structure with the following elements:
   - a partial AST representing a (partial) program.
   - a non-visual tree editor, working on a focused subtree and AST, with tree editing commands.
   - program text corresponding to the focused subtree.
   - control function(s) to maintain consistency between program text and the focused subtree, and to allow smooth transition between text editing and structure editing.
3. Should cooperate with the following collaborators:
   - a component which holds language specifications. In general, a hybrid editor component should be able to work on different languages.
   - a component which holds a mapping $text \mapsto AST$.
   - a component which holds a mapping $AST \mapsto text$.
     Similarly, the hybrid editor component should be able to work with different versions of algorithms for $text \mapsto AST$ and $AST \mapsto text$.
   - A node factory object which is used to create instances of AST nodes.
   - a component which holds the specifications for presentation, serving as a basis to customize the presentation style (i.e color and font, text layout) of text to user preferences.
   - one or more views/editors to visually display/edit the AST in a structure preserving way.
   - A view/editor to display/edit program text.

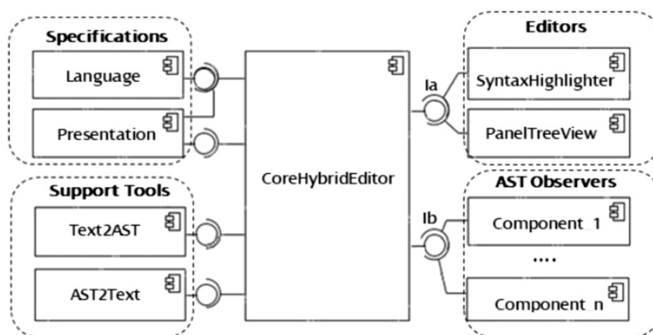


Fig. 7. A component-based architecture of a `CoreHybridEditor`. Ia and Ib denote the provided interfaces: ICoreHybridEditorComp and INodeObject respectively.

```
1   interface ICoreHybridEditorComp extends IPAMOJAComp{
2   //connections to specifications and support components
3       void setLanguage(ILanguageComp language);
4       void setPresentation(IPresentationComp presentation);
5       void setText2Ast(IText2ASTComp text2ast);
6       void setAST2Text(IAST2TextComp ast2text);
7       void setNodeFactory(CNodeFactory nodeFactory);
8   //methods for interacting with the editors
9       void setAST(CNode AST);
10      CNode getAST();
11      void setFocus(CNode focus);
12      CNode getFocus();
13      CTreeEditorComp getTreeEditor();
14      void setText(String text);
15      String getText();
16  //control functions between text and structure mode
17      void toText();
18      void toStructure();
19      void abort();
20  }
```

Listing 3: Interface for the `CoreHybridEditor` component.

An important consequence of these design decisions is that the resulting hybrid editor is a small and flexible component providing just the core functionality. Additional functionality is obtained by connecting it to collaborators suitable for specific needs.

6.3. Organization of a `CoreHybridEditor` component

Fig. 7 illustrates the component-based architecture of a `CoreHybridEditor` component. This component encapsulates: (1) an AST representation of a program (usually with holes), (2) a focused subtree, (3) a textual representation of the focused subtree, (4) a non-visual `TreeEditor` component, and (5) controls for switching between a text editor and a structure editor. To achieve hybrid editing, collaborator components for the *specifications*, *editors*, and *support tools*, have to be connected to a `CoreHybridEditor`. For this reason, this core component provides an interface, `ICoreHybridEditorComp` (see Listing 3) with corresponding methods for connecting and interacting with its collaborators. The interface extends `IPAMOJAComp`, the main interface for all components in PAMOJA, which provides some essential functionality, such as the observer and the persistency mechanisms.

A `CoreHybridEditor` observes the specification components. When it receives property change events from these components, it updates itself and also forwards the events to the targeted collaborators. For instance, a property change event from a `Language` component is forwarded to its `TreeEditor` subcomponent and the support components. Note that a `Presentation` component also observes a `Language` component in order to maintain consistency between its own valid symbol domain and the symbols of a language.

As presented in Section 6.2, a `CoreHybridEditor` encapsulates how the editor components interact; hence the `SyntaxHighlighter` and `PanelTreeView` components do not communicate to each other directly. The `CoreHybridEditor` provides methods via the `ICoreHybridEditorComp` interface (Lines 9 - 19), which facilitate their interaction using the observer mechanism.

Switching between textual and structural mode of editing proceeds as follows. From structural to textual mode, a `CoreHybridEditor` uses an `AST2Text` component to derive a textual representation of a focused subtree and sends a text property change via `setText` method (Line 14). The `SyntaxHighlighter` reacts to the change and updates its view for

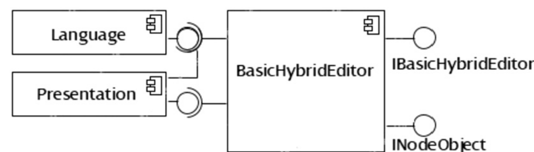


Fig. 8. A component-based architecture of a `BasicHybridEditor`.

Table 2
Sizes of the handwritten code during the implementation of the hybrid editor composites. Size is measured in non-blank, non-commented, and non-automatically generated logical SLOC of Java.

Component	Logical SLOC
CoreHybridEditor	168
BasicHybridEditor	75

editing. Switching from textual to structural mode, the `CoreHybridEditor` uses a `Text2AST` component and a node factory object to derive a subtree representation of the edited text. When successful, it replaces the focus in the AST with the produced subtree, and sends an AST property change via the `setAST` method (Line 9). The `PanelTreeView` reacts to the change and updates its view. Then, in coordination with a `TreeEditor` component, the `PanelTreeView` provides support for visual editing of the AST representation of a program. On error, an appropriate error message is generated and the editor stays in textual mode.

Through the `INodeObject` interface, a `CoreHybridEditor` component can be extended by connecting it to other components which perform operations on the AST representation of the edited program, such as tree visualizers, tree matchers, and tree rewriters. Such components can receive AST property change events triggered by a `CoreHybridEditor` and adapt their state. `INodeObject` contains a single method, `getNode()`, which is used to return a tree. It may be implemented by all components that hold observable trees, such as parse trees, ASTs, and box trees.

The architecture presented in Fig. 7 offers a flexible approach to compose hybrid editors in many different ways, since it provides freedom to attach collaborator components that suit specific user needs.

6.4. *Reusing a* `CoreHybridEditor`

In this subsection we show how to reuse a `CoreHybridEditor` in the construction of a higher-level and more specialized hybrid editor component, which can simply be adapted to the given specifications.

A `CoreHybridEditor` component gives a lot of flexibility since it can be adapted to specific user needs by connecting it to different components for the *specifications*, *editors*, and *support tools*. However, we could construct a more specialized hybrid editor component consisting of standard components for the *editors*, and *support tools* which are sufficiently simple and general to handle most kinds of languages. The hybrid editor component can be connected to a `Language` and a `Presentation` component, yielding a hybrid editor for a particular language.

Fig. 8 gives an overview of how a `BasicHybridEditor` composite component is organized. In terms of Fig. 7, this component consists of an assembly of a `CoreHybridEditor` component, and fixed components for the editors and support tools. It receives property change events triggered by `Language` and `Presentation` components and forwards them to the targeted sub components. It realizes two interfaces, `INodeObject` in a similar way like a `CoreHybridEditor` component and `IBasicHybridEditor` which provides services for communicating with `Language` and `Presentation` components, and the AST node factory object.

6.5. *Realizing the hybrid editor components*

Both the `CoreHybridEditor` and the `BasicHybridEditor` components have been implemented and are part of the PAMOJA component toolkit. The sizes of the handwritten lines of code which had to be added during implementation are summarized in Table 2.

A `CoreHybridEditor` component (Section 6.3) is non-visual therefore, we used ordinary programming, mainly implementing: (1) the public properties and their corresponding getters and setters presented in Listing 3; and (2) the methods which control switching between textual mode and structural mode. A `BasicHybridEditor` component (Section 6.4) is a visual component containing an assembly of components. We used the drag-and-drop development style to assemble its subcomponents, and added source-code for its functionality. This included source-code for: (1) properties for the AST representation of a program, the language, presentations and node factory, along with their getters and setters; and (2) event handling for the explicit mode switch between a text editor and a structure editor.

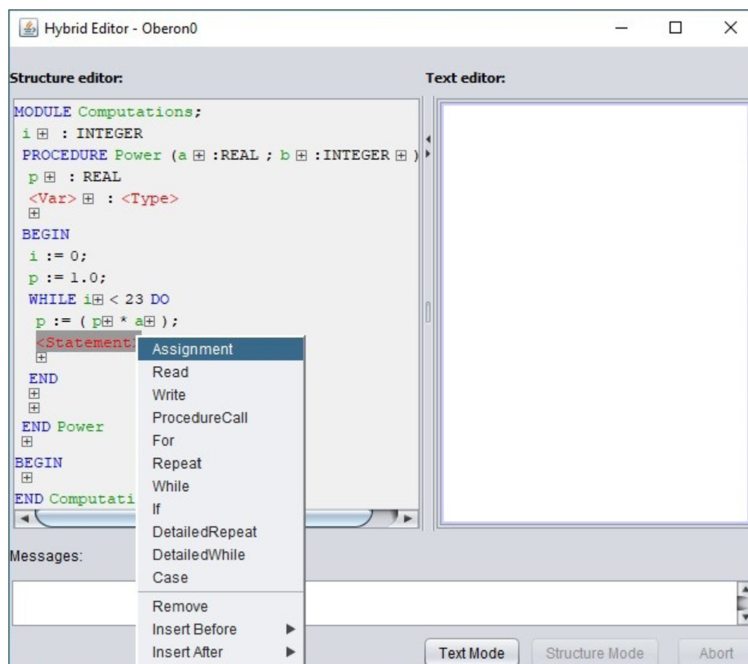


Fig. 9. A hybrid editor in structural mode showing: (1) a structure editor with a partial program for Oberon0 in the upper-left window. (2) The upper-right window is the text editor. (3) Buttons in the lower right-part, used for switching between structure and text mode. (4) The middle-lower box used to display messages produced by the editor.

We Implemented two hybrid editor applications, one for Oberon0 and another for GCLSharp (a simple language based on Dijkstra's guarded commands [55]), by connecting a `BasicHybridEditor` component to the *specification* components. For this implementation not a single line of code had to be written. Figs. 9 and 10 give an impression of how the hybrid text/structure editor application for Oberon0 looks like. In Fig. 9 the hybrid editor is in structural mode, activating a structure editing operation by (1) structurally selecting a `<Statement>` hole and (2) interacting with a popup menu showing permitted language constructs among other things and choosing an `assignment` construct to replace a `<Statement>` hole. In Fig. 10 is an example of a hybrid editor in textual mode. The textual representation of a focused subtree for a `while` construct is displayed in the text editor for textual editing.

7. Discussion

In this paper, we have presented the use of CBSD principles to assist in the design and development of grammar-aware systems. The PAMOJA component framework, is aimed at making GAE technology easily accessible to users with less expertise in compiler construction, such as students and software developers. This section evaluates PAMOJA with respect to CBSD aspects, in particular *flexibility*, *lightweight* and *ease of use*.

PAMOJA's flexibility is illustrated in two ways. First, the components can be customized, assembled into composites in different ways, and (re)use the different components to construct grammar-aware systems. For example, both the `BasicHybridEditor` component (Section 6.4) and its sub composites are available for reuse. For example, the `Text2AST` may be reused in the construction of other software systems based on parsing technology. In addition, the `BasicHybridEditor` component may be reused in several ways. For example: (1) to construct hybrid editor systems for different languages by combining it with `Language` and `Presentation` components. (2) by connecting it to tree visualizers, tree matchers and semantic tools like type checker, evaluator or compiler. Second, for some of the language processing tasks, such as scanning and parsing, more than one component is available — each employing a different technique and the toolkit can be extended with other components.

The PAMOJA component toolkit is particularly lightweight compared to stand-alone language processing systems, such as ANTLR [18], JastAdd [19], Coco [20] and Silver [21]. It consists of simple grammar-aware components dedicated to well-defined tasks, which take the form of a component library in a general-purpose IDE. This lightweight nature makes grammar-aware techniques easily accessible to users who would otherwise not be exposed to standalone language processing systems. Moreover, since the components are pure JavaBeans, PAMOJA automatically gains advantage from JavaBean tools, such as, IDE support for assembling the components into complex components and incorporating them into applications, further simplifying the adoption process.

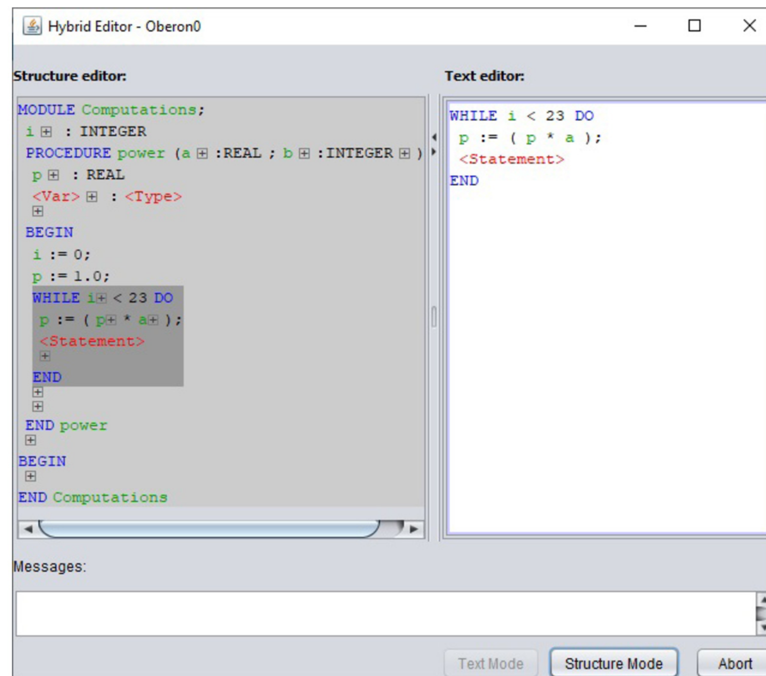


Fig. 10. A hybrid editor in textual mode. The structure editor is frozen. The currently focused subtree (a while statement) is displayed in textual form in a text editor and is available for text editing.

PAMOJA enables users to focus on language processing tasks (e.g., experimenting with various algorithms, developing front-ends, component integration, etc.) rather than to spend time learning the technicalities of how to use a tool. Ease of use is there because of the several decisions made in the design of the PAMOJA architecture. Examples include:

- PAMOJA components feature a clear component concept with well-defined communication interfaces.
- For all relevant grammar-aware data there is a corresponding view, or more views for inspecting the different aspects of the data.
- Component editors are provided to present complex data structures like grammars, signatures and trees, in a form that is meaningful to the user, so that a user can modify data structures and parts of data structures easily. Additionally, the editors ensure that the modified data structures are well-formed (i.e., have no errors). For example the `Grammar` component editor (see Fig. B.11) provides a simple interface through which a user can edit grammars and parts of grammars, and view analysis information which might be needed to keep track of a grammar during its development.
- The RAD style makes it possible to construct grammar-aware applications with little or no glue-code. For instance, only a few lines of code had to be written to construct the hybrid editor components of Section 6 (e.g., see Table 2). In addition not a single line of code was written during the construction of an application for syntax highlighting (Section 5.1) and the applications for Oberon0 and GCLSharp hybrid editors (Section 6.5).
- The observer/observable mechanism replaces glue-code which has to be written to achieve collaboration among components and to cause cascading of updates. PAMOJA components maintain consistency among themselves, so that if there is a change in one component, the other components get updated accordingly.

However, PAMOJA has some potential limitations. First, the current collection of components is restricted. There is need to extend the toolkit with more components. For example for parsers it would be good to include parser components employing more general methods like the GLR [49] and GLL [50]. Second, error handling is currently done in a ad hoc way — there is need to develop a general solution for systematic error handling. Third, the current component editors for grammars and signatures provide basic support for language design. There is still room for improvement in several ways. For instance, for the grammar component editor, to support grammar transformations and grammar debugging. In addition, there is need to develop a component editor which facilitates creation of box patterns in a structure preserving way.

8. Conclusions and future work

We have presented a CBSD approach for designing a coherent set of small grammar-aware components that fit into a general-purpose framework. Our contributions in this paper are: (1) a lightweight component architecture, named PAMOJA,

that brings the benefits of CBSD to grammar-aware software. (2) a case study demonstrating how to compose new components from existing components, using a hybrid text/structure editor application as an example. Additionally, the architecture is realized on the NetBeans environment. PAMOJA currently provides components for the tasks of scanning, parsing, tree building and formatting. It supports rapid development of grammar-aware software in ways commonly found in modern IDEs, i.e., drag-and-drop and wizard approach. Our CBSD approach makes language processing technology available in a general-purpose framework to non-experts and for a large variety of applications - not necessarily compilers.

For future work, we intend to continue and extend application of PAMOJA in education to teach compiler-like courses, also as a continuous source of feedback for improvement. Other areas of future work include to design a general solution for error handing, to extend our approach to the design and implementation of other grammar-aware components (e.g., generalized parsers, tree matchers and tree rewriters), and to extend the possibilities of our grammar and signature component editors to support advanced features like transformations and debugging.

PAMOJA source codes, associated documentation, and supplementary data associated with this paper can be found in the online version [56].

CRediT authorship contribution statement

Jackline Ssanyu: Conceptualization, Data curation, Investigation, Methodology, Software, Writing – original draft. **Engineer Bainomugisha:** Conceptualization, Supervision, Validation, Writing – review & editing. **Benjamin Kanagwa:** Conceptualization, Supervision, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We are grateful to the anonymous reviewers for their detailed and knowledgeable comments and for their many constructive and helpful suggestions to improve this paper.

Appendix A. PAMOJA components and Wizards

Table A.3
List of components and wizards currently available in the PAMOJA component toolkit at lexical, concrete and abstract syntax.

Level	Components/Wizards	Description
Lexical analysis	RichTextEditor	A text editor with facilities for different font and color attributes.
	ScanTables	Generates NFA and DFA from a lexical grammar.
	ScanTableView	A tabular view for NFA and DFA tables.
	DFAScanner	A mapping from plain text to a symbolstream.
	Symbolstream	Holds a stream of symbols.
	SymbolstreamView	A tabular view for symbol properties.
	SymbolStyleCustomizer	Holds a mapping from lexical symbols to symbol categories and symbol categories to font and color attributes.
	Stream2Text	Maps 2D-symbolstream to 2D-text.
	ELL(1)ScannerGenerator	Generates ELL(1) scanner source-code from a lexical grammar.
Conrete syntax	Grammar	Holds an ECFG, maintains its consistency and computes grammar properties (e.g., First, Last, Follow and lookahead sets, and predicates such as Null, Empty and Reachable).
	GrammarView	Provides facilities for inspecting grammar definitions and properties.
	Flattener	Maps a parse tree to a 1D-symbolstream.
	SLRTables	Generates SLR(1) parse tables from a grammar.
	SLRTableView	A tabular view for SLR(1) parse tables.
	SLRParser	Holds SLR(1) algorithm which maps a symbolstream to a parse tree.
	DeterministicParser (interpreter)	A recursive descent parser which interprets ELL(1) grammars.
	LimitedBackTrackerParser (interpreter)	A recursive descent, limited backtracking parser which interprets an arbitrary non left-recursive grammar.
	TreeBuilder	Provides facilities for constructing different kinds of parse tree nodes.
	ParseTree	Holds a structural representation of a parse tree.
	ParserGenerator	Generates recursive-descent parser source-code from an ELL(1) grammar.

Table A.3 (*continued*)

Level	Components/Wizards	Description
Abstract Syntax	Signature	Holds abstract syntax and maintains its consistency.
	AST	Holds a structural representation of an AST.
	AST2BoxTree	Maps an AST to a box tree.
	BoxTree2Stream	Maps a box tree to a 2D-symbolstream.
	Abstractor	Maps a parse tree to an AST.
	GenericTreeView	A JTree-like view which displays PAMOJA-trees (e.g. parse trees, ASTs, and box trees) and allows inspection of their nodes.
	PanelTreeView	Displays an AST as a hierarchy of nested panels and allows its structural editing.
	Patterns	Holds box layout specifications for producing 2D-text and maintains their consistency.
	TreeEditor	Contains basic operations for editing different kinds of AST nodes.
	TreeGraph	Provides visual presentation of different structures (e.g. regular expressions, NFA, parse trees and ASTs).
	SignatureAPIGenerator	Generates AST classes for abstract syntax.

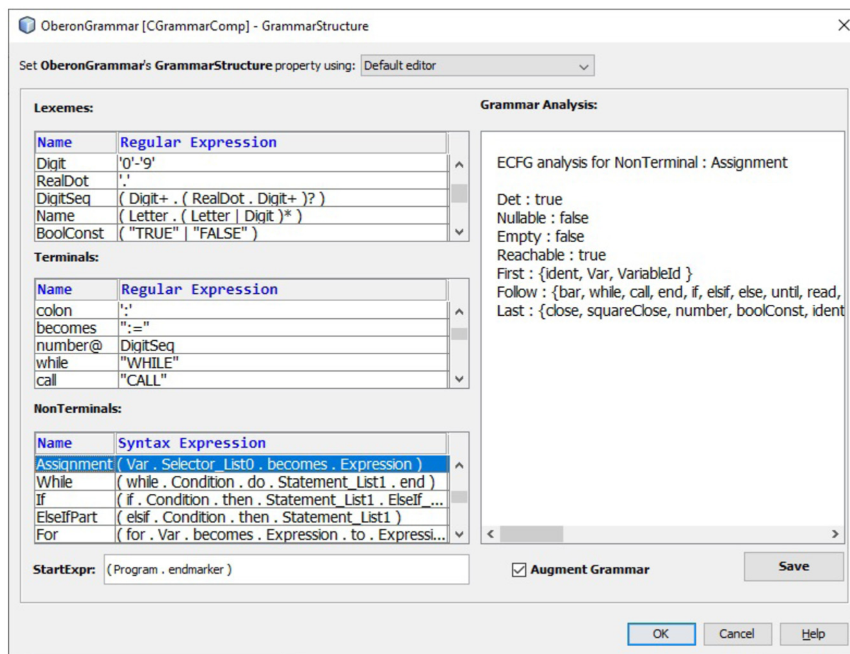


Fig. B.11. An editor for the Grammar component showing parts of the Oberon0 grammar on left. On the right is the analysis information for the selected Assignment non-terminal.

Appendix B. PAMOJA's specifications

In this section, we give an impression of how language and presentation specifications are defined in PAMOJA, making use of parts of the Oberon0 language as examples. The complete Oberon0 specifications can be found online [56].

B.1. Lexical and concrete syntax

PAMOJA's lexical syntax and concrete syntax specifications have a familiar structure; they are defined by means of regular expressions and ECFG respectively. Part of the PAMOJA-grammar specification for Oberon0 is shown in the grammar's component editor, in Fig. B.11.

Lexical syntax specifications consist of lexemes and terminal symbol declarations, which is the name of a lexeme/terminal and the corresponding regular expression. Terminals which contain data, such as numbers and identifiers, are identified using their terminal names which are appended with the '@' character.

Concrete syntax specifications also have a similar structure; they are declared using the name of a non-terminal and the corresponding production rule written in ECFG style. In the non-terminals section, lists are represented as `N_List0` (`N*`) and `N_List1` (`N+`) for any non-terminal `N`.

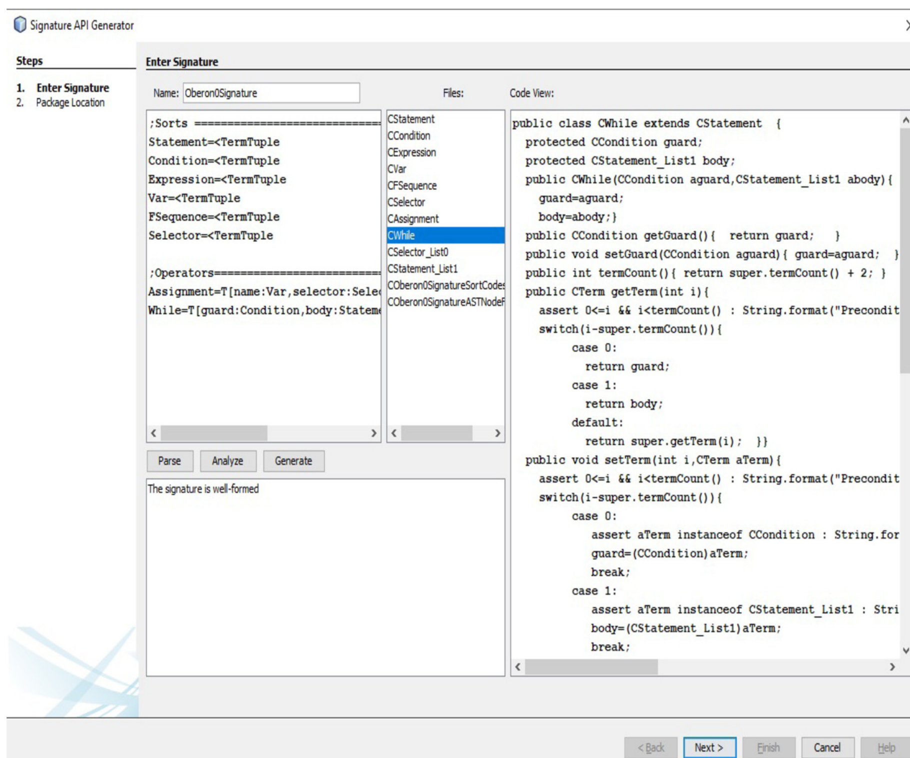


Fig. B.12. Signature API generator wizard at step 1, showing abstract syntax for `Assignment` and `While` statements of Oberon0, a list of class files generated, and sample code for a `While` operator.

B.2. Abstract syntax

PAMOJA's abstract syntax is defined using the usual notion of a term algebra over a signature. A signature consists of a finite set of sorts and a finite set of operators, each of which has a sequence of argument sorts and a result sort. For example the operators for `Assignment` and `While` statements of Oberon0 are defined as follows:

```
Assignment=T[name:Var,selector:Selector*,expr:Expression] <Statement
While=T[guard:Condition,body:Statement+]<Statement
```

`T` is the list of sub-terms of an operator. `Var`, `Selector`, `Expression`, `Condition`, and `Statement` are sorts. Lists are represented by regular operators: `N*` and `N+` for any non-terminal `N`.

Given a signature definition, PAMOJA's signature API generator wizard automatically generates implementations of AST classes in Java. Fig. B.12 is a sample of a signature API generator wizard at step one, showing the abstract syntax definition for `Assignment` and `While` statements, the list of generated class files and a sample source code for the `While` operator. On clicking "Next" button the wizard moves to step two, which requires a user to set the developers project and location in which the generated AST classes will reside. Finishing the wizard initiates the integration of the generated AST classes into the developers project and compiled.

We present an AST as an object-oriented class hierarchy using an approach similar to ApiGen [51] and JastAdd [52]. All sorts are mapped to general base classes, and all operators are mapped to specialized concrete subclasses. The subclasses have members with names derived from the sub-terms and which are references to objects of base classes. There is a super class for all AST classes which provides methods for general operations on terms which have to be overridden in suitable subclasses. For example, `termCount()` returning the number of subterms in a term, `t`, `getTerm(i)` and `setTerm(i,t)` for getting and setting the ith subterm, with a precondition $0 \leq i <$ `termCount()`.

The subclasses may implement suitable access methods that allow type safe access to subterms. For instance, the AST node representing a `While` sort has a member `guard` of a specific type `Condition`, together with access methods `getGuard()` and `setGuard(CCondition aguard)`. In order to maintain a well-typed tree, `setTerm(i,t)` should always check that its term argument is of the appropriate subtype. This check is included in the code generated for the AST classes. See for instance part of the code generated for a `While` sort in Fig. B.12.

B.3. Presentations

Presentation of a text program in PAMOJA is considered in two aspects: syntax highlighting and formatting. Syntax highlighting is achieved via a

`SymbolStyleCustomizer` component which contains a visual editor for setting text appearance in different colors and fonts according to the category of symbols. See Fig. 5.

Formatting consists of a mapping — parameterized with box patterns — from an AST to a box tree, which is subsequently mapped to nicely printed text [47,48]. The resulting box tree describes how elements should be laid out, e.g., horizontally, vertically or indented, and how adjacent elements should be spaced relative to one another.

As an example consider the following box pattern for formatting a `While` statement of Oberon0:

```
While=Sel(Ver([
             Hor([Term(while), Node(0), Term(do)],1),
               Ind(Node(1)),
             Term(end)
            ],0))
```

`while` and `do` are both terminals which correspond to terminal names in the lexical syntax. The `Condition` is formatted using the function `Node(0)` (0 is the index of the first node, i.e., `Condition`, in the AST of a `While`). All three sub-boxes are laid out horizontally (`Hor`) and the horizontal spacing between the elements should be 1 as indicated. The body of the `While` statement should be indented (`Ind`) and it should be formatted using the function `Node(1)` (1 is the index of the second node, i.e., the body, in the AST of a `While`). Finally, the three sub-boxes are wrapped in a vertical box (`Ver`) so that the while header, statement-body and `end` are placed vertically with zero vertical spacing.

References

[1] E. Poll, Formal methods for security?, https://www.cs.ru.nl/~erikpoll/papers/researchagenda_fm.pdf, 2018. (Accessed 26 April 2021).
[2] LANGSEC: language-theoretic security, http://langsec.org/. (Accessed 26 April 2021).
[3] P. Klint, R. Lämmel, C. Verhoef, Toward an engineering discipline for grammarware, ACM Trans. Softw. Eng. Methodol. 14 (3) (2005) 331–380, https://doi.org/10.1145/1072997.1073000.
[4] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, USA, 2002.
[5] Apache Software Foundation, Apache Net Beans, https://netbeans.org/. (Accessed 12 October 2020).
[6] Eclipse Foundation, Eclipse: the platform for open innovation and collaboration, https://www.eclipse.org/. (Accessed 12 October 2020).
[7] Jet Brains, IntelliJ IDEA: capable and ergonomic IDE for JVM, https://www.jetbrains.com/idea/. (Accessed 12 October 2020).
[8] Embarcadero Technologies, Delphi IDE, https://www.embarcadero.com/products/delphi. (Accessed 22 October 2020).
[9] StatBeans, StatGraphics: JavaBean components for statistical analysis, https://www.statgraphics.com/statbeans. (Accessed 24 April 2021).
[10] H. Praehofer, J. Sametinger, A. Stritzinger, Concepts and architecture of a simulation framework based on the JavaBeans component model, Future Gener. Comput. Syst. 17 (5) (2001) 539–559, https://doi.org/10.1016/S0167-739X(00)00038-8.
[11] P. Nikander, A. Karila, A JavaBeans component architecture for cryptographic protocols, in: Proceedings of the 7th USENIX Security Symposium, USENIX Association, San Antonio, Texas, 1998, pp. 107–121.
[12] E. Grinkrug, 3D modeling by means of JavaBeans, in: Proceedings of the 12th International Workshop on Computer Science and Information Technologies, CSIT'2010, Moscow – Saint-Petersburg, Russia, 2010.
[13] X. Wu, B.R. Bryant, J. Gray, M. Mernik, Component-based LR parsing, Comput. Lang. Syst. Struct. 36 (1) (2010) 16–33, https://doi.org/10.1016/j.cl.2009.01.002.
[14] E. Vacchi, W. Cazzola, Neverlang: a framework for feature-oriented language development, Comput. Lang. Syst. Struct. 43 (2015) 1–40, https://doi.org/10.1016/j.cl.2015.02.001.
[15] T. Cleenewerck, Component-based DSL development, in: Generative Programming and Component Engineering, Springer, Berlin, Heidelberg, 2003, pp. 245–264.
[16] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann, Systematic composition of independent language features, J. Syst. Softw. 152 (2019) 50–69, https://doi.org/10.1016/j.jss.2019.02.026.
[17] L.T. Van Binsbergen, P.D. Mosses, N. Sculthorpe, Executable component-based semantics, J. Log. Algebraic Methods Program. 103 (2019) 184–212, https://doi.org/10.1016/j.jlamp.2018.12.004.
[18] ANTLR: ANother Tool for Language Recognition, https://www.antlr.org/. (Accessed 2 November 2020).
[19] JastAdd, jastadd.org. (Accessed 2 November 2020).
[20] M. Löberbauer, H. Mössenböck, The compiler generator Coco/R, http://www.ssw.uni-linz.ac.at/Coco/ (Last update 2018).
[21] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan 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.
[22] L.C. Kats, E. Visser, The Spoofax Language Workbench: rules for declarative specification of languages and IDEs, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, in: OOPSLA '10, vol. 45, Association for Computing Machinery, New York, NY, USA, 2010, pp. 444–463.
[23] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm, J. Vinju, Modular language implementation in Rascal — experience report, Sci. Comput. Program. 114 (2015) 7–19, https://doi.org/10.1016/j.scico.2015.11.003.
[24] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edition, Packt Publishing Ltd., 2016.
[25] N. Wirth, Compiler construction, https://people.inf.ethz.ch/wirth/CompilerConstruction/CompilerConstruction1.pdf, 2017. (Accessed 2 November 2020).
[26] M. van den Brand, Introduction to the LDTA tool challenge, Sci. Comput. Program. 114 (2015) 1–6, https://doi.org/10.1016/j.scico.2015.10.015, LDTA (Language Descriptions, Tools, and Applications) Tool Challenge.
[27] J.R. Levine, T. Mason, D. Brown, Lex & Yacc, 2nd edition, O'Reilly & Associates, Inc., USA, 1992.
[28] JavaCC: a parser generator for use with Java applications, https://javacc.github.io/javacc/. (Accessed 2 November 2020).
[29] P.R. Henriques, M.J.V. Pereira, M. Mernik, M. Lenič, J. Gray, H. Wu, Automatic generation of language-based tools using the LISA system, IEE Proc., Softw. 152 (2) (2005) 54–69, https://doi.org/10.1049/ip-sen:20041317.

[30] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, J. Visser, The ASF+SDF meta-environment: a component-based language development environment, Electron. Notes Theor. Comput. Sci. 44 (2) (2001) 3–8, https://doi.org/10.1016/S1571-0661(04)80917-4, LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).
[31] J.A. Bergstra, P. Klint, The ToolBus coordination architecture, in: P. Ciancarini, C. Hankin (Eds.), Coordination Languages and Models, Springer, Berlin, Heidelberg, 1996, pp. 75–88.
[32] P. Klint, A meta-environment for generating programming environments, ACM Trans. Softw. Eng. Methodol. 2 (2) (1993) 176–201, https://doi.org/10.1145/151257.151260.
[33] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., USA, 1995.
[34] A.M. Sloane, Lightweight language processing in Kiama, in: J.M. Fernandes, R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Springer, Berlin, Heidelberg, 2011, pp. 408–425.
[35] A.M. Sloane, M. Roberts, Oberon-0 in Kiama, Sci. Comput. Program. 114 (2015) 20–32, https://doi.org/10.1016/j.scico.2015.10.010.
[36] E. Söderberg, G. Hedin, Building semantic editors using JastAdd: tool demonstration, in: Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications, LDTA'11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 1–6.
[37] G. Hedin, An introductory tutorial on JastAdd Attribute Grammars, in: J.M. Fernandes, R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Springer, Berlin, Heidelberg, 2011, pp. 166–200.
[38] N. Fors, G. Hedin, A JastAdd implementation of Oberon-0, Sci. Comput. Program. 114 (2015) 74–84, https://doi.org/10.1016/j.scico.2015.02.002.
[39] P. Charles, R.M. Fuhrer, S.M. Sutton Jr, IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 485–488.
[40] P. Charles, R.M. Fuhrer, S.M. Sutton Jr, E. Duesterwald, J. Vinju, Accelerating the creation of customized, language-specific IDEs in Eclipse, ACM SIGPLAN Not. 44 (10) (2009) 191–206, https://doi.org/10.1145/1639949.1640104.
[41] MetaBorg Revision, The Spoofax language workbench, http://www.metaborg.org/en/latest/source/overview/publications.html. (Accessed 17 October 2020).
[42] Centrum Wiskunde and Informatica (CWI), Rascal metaprogramming language and IDE, https://github.com/usethesource/rascal/blob/master/CITATION.md. (Accessed 23 October 2020).
[43] Eclipse Foundation, Xtext: language engineering for everyone, https://www.eclipse.org/Xtext/. (Accessed 12 October 2020).
[44] F. Bauer, F. DeRemer, M. Griffiths, U. Hill, J. Hornig, C. Koster, W. McKeeman, P. Poole, W. Waite, Compiler Construction: An Advanced Course, 2nd edition, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 1977.
[45] A.V. Aho, M. Lam, R. Sethi, J. Ullman, Compilers: Principles, Techniques and Tools, 2nd edition, Pearson Education India, 2006.
[46] R. Wilhelm, H. Seidl, S. Hack, Compiler Design, Springer, Heidelberg, 2013.
[47] M. van den Brand, E. Visser, Generation of formatters for context-free languages, ACM Trans. Softw. Eng. Methodol. 5 (1) (1996) 1–41, https://doi.org/10.1145/226155.226156.
[48] M. de Jonge, A pretty-printer for every occasion, in: Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools, CoSET2000, vol. 5, 2000, pp. 68–77.
[49] M. Tomita, Generalized LR Parsing, 1st edition, Springer, Boston, MA, 1991.
[50] L.T. van Binsbergen, E. Scott, A. Johnstone, Purely functional GLL parsing, J. Comput. Lang. (2020), https://doi.org/10.1016/j.cola.2020.100945.
[51] M. Van den Brand, P.-E. Moreau, J. Vinju, A generator of efficient strongly typed abstract syntax trees in Java, IEE Proc., Softw. 152 (2) (2005) 70–78, https://doi.org/10.1049/ip-sen:20041181.
[52] G. Hedin, E. Magnusson, JastAdd: a Java-based system for implementing front-ends, Electron. Notes Theor. Comput. Sci. 44 (2) (2001) 59–78, https://doi.org/10.1016/S1571-0661(04)80920-4, LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).
[53] A. Gomolka, B. Humm, Structure editors: old hat or future vision?, in: International Conference on Evaluation of Novel Approaches to Software Engineering, Springer, 2011, pp. 82–97.
[54] Microsystems Sun, JavaBeans API specification, https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/, 1997. (Accessed 2 November 2020).
[55] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Commun. ACM 18 (8) (1975) 453–457, https://doi.org/10.1145/360933.360975.
[56] J. Ssanyu, PAMOJA grammar-aware component toolkit, v1.0. Software, Jun. 2021.