A disintegrated development environment

Eine reintegrierte Entwicklungsumgebung
Master thesis
Author: Sven Keidel

Day of submission: April 9th, 2015
Examiner: Prof. Dr.-Ing. Mira Mezini
Supervisor: Dr. rer. nat. Sebastian Erdweg

TECHNISCHE UNIVERSITÄT DARMSTADT

Department Computer Science
Software Technologie Group

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 9. April 2015

(Sven Keidel)

Abstract

Nowadays developers use feature-rich tools to help them during their work day. An Integrated Development Environment (IDE) is a collection of tools integrated under a coherent graphical user interface, designed to minimize friction during the work on different tasks. There are plenty of IDEs a developer can choose from, but not all IDEs have the same set of features. Hence developers either find themselves switching constantly between different IDEs to get the best possible support for the task they want to archive, or staying in one particular IDE making compromises not having first-class support for the task. Even worse is the situation for those, who want to extend the functionality of IDEs. Efforts made to extend one IDE does not transfer automatically to other IDEs.

The Monto project has set the goal to change this deficiency, both for the users and for those who extend IDEs. The project decouples services from IDEs to a level where they can be reused in other IDEs. This process is called *disintegration*. The contributions of this thesis are to explore the field of services and to establish standards for the communication between services and the IDE. Moreover contributions include the extending the functionality of Monto, but to retain the simplicity of developing new services.

Contents

1 Introduction

The term IDEs describes a collection of tools for software development. The word “integrated” in this context means, that the tools are geared to each other to give the developer the smallest amount of friction possible during development.

Todays IDEs combine tools like:

- Source code editor
- Build automation tool
- Compiler or Interpreter
- Debugger
- Project Management
- Version control system
- Class Hierarchy Browser
- Terminal
- GUI Designer

The editor contained in the IDE has features like

- Syntax Highlighting
- Intelligent code completion
- Marking of errors in the code
- Auto Indentation
- Guided Refactoring

Not all these features are directly build into the core of an IDE, but are provided as plug-ins. Many IDEs provide plug-in mechanisms for extension. Such a mechanism allows independent developers or third parties to extend the IDE to support new programming languages, the latest technology and services. While this mechanism is very beneficial and useful for the ecosystem of an IDE, it induces a problem for plug-in developers. The mechanism is not standardized between different IDEs, thus efforts made to develop a plug-in for one IDE do not carry over to other IDEs. In other words a plug-in is closely coupled to the plug-in Application Programmable Interface (API) of the IDE (hence the “integrated” in IDE).

Another problem that plug-in developers face, is the huge effort that some projects in this field require. See for instance the Scala support for Eclipse. As of February 2015 the Scala Eclipse plug-in project has been developed for 5 years, has 3924 commits, 52 contributers and consists

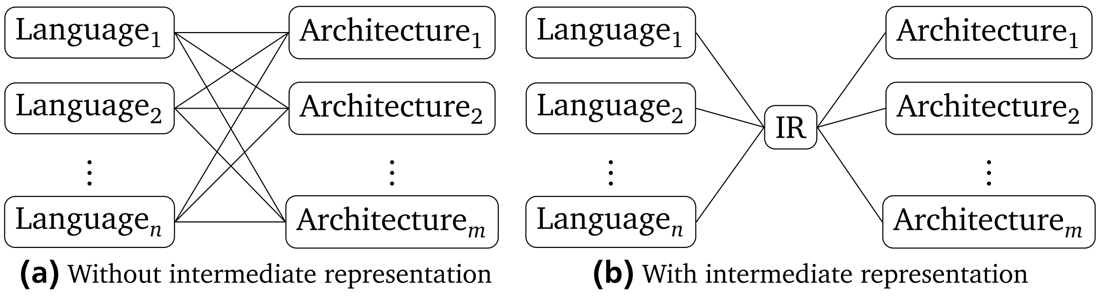


Figure 1.1: Compiling code for multiple CPU architectures

of ca. 100.000 lines of code. Porting this code to another IDE requires a huge amount of work and seems unpractical. This situation forces developers to switch often between different IDEs and editors that support the languages or features that they need.

The broad fragmentation of developers that use different IDEs is amplifying this problem. As of February 2015, Wikipedia lists around 100 different IDEs [1] and 135 Text Editors [10]. If a language designer wants to support even a small fraction of other developers with his work, he has to port the functionality to many different IDEs. As seen by the example of the Scala plug-in the effort to develop such a plug-in is not feasible for one developer.

Summarizing the current situation of IDE development imposes two problems on to different parties. One the one hand language designers cannot develop a plug-in for every IDE and editor that exists. On the other hand this forces developers, who want to benefit from support for this language to switch between IDEs and editors constantly.

Portable Compilers

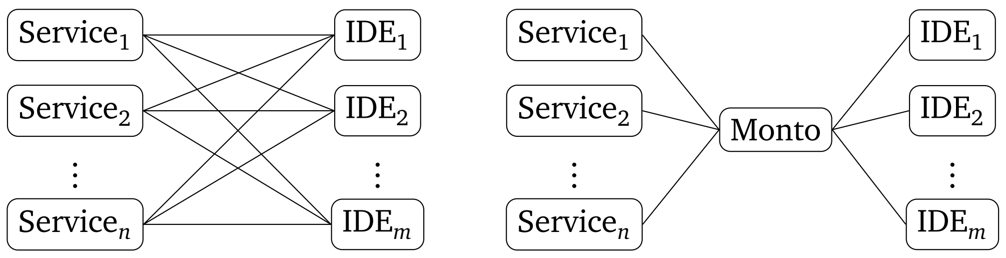
Luckily a similar problem has occurred and been successfully solved in the past. In the 1970s and 80s the growing number of CPU architectures imposed the same problem on developers, who tried to write code that runs on multiple architectures. Figure 1.1a shows the situation before portable compilers were invented. Each of the edges represents an implementation of a compiler that compiles from a particular language to an CPU architecture. The designer of a programming language, who wanted to support m different CPU architectures, had to write m different compilers.

Martin Richards describes in [20], [19] and [18] how he introduced OCODE, a machine independent code for an abstract stack based machine, to solve this problem in the BCPL compiler. This intermediate representation is used in the compiler chain at the second last position, right before the code generator produces machine dependent code. This enables every language that compiles to OCODE to run on every architecture that is supported by the BCPL code generator. Figure 1.1b shows the situation after the invention of portable compilers and the introduction of an intermediate representation.

Monto: A Disintegrated Development Environment

Today plug-in developers are in a similar situation like compiler writers in the 70s as shown in Figure 1.2a. Each edge represents an integration of a particular service into an IDE. For n services and m IDEs the total amount of implementations that have to be written is $n * m$.

Similar to how the BCPL compiler archives portability, the Monto project tries to ease the work of developer to write plug-ins and at the same time making these plug-ins available under



(a) Current Situation **(b)** Monto as mediator between services and IDEs

Figure 1.2: IDE plugin situation in the past and with Monto

multiple IDEs. Monto [23] is a framework created by Tony Sloane *et al.*, first presented at the International Conference on Software Language Engineering 2014.

In an ideal situation where all services would use Monto and Monto would be supported in any IDE, the amount of implementations would be reduced to just $n+m$. Figure 1.2b represents this situation with Monto as a mediator between services and IDEs.

The project tries to archive this goal by limiting the coupling of services to a particular IDE to a minimum while still allowing features to be integrated. The authors of Monto coined the term Disintigrated Development Environment (DDE) to refer to this approach. Minimal requirements and dependencies of Monto simplify the development of services and even allow the use of a different programming language as the IDE they are used in.

Figure 1.3 shows an overview of the architecture of Monto (the diagram has been derived from [23] page 214). The Monto architecture is structured in four components:

- *Sources* that publish changes whenever the user interacts with the IDE. One can think of *sources* as source code files, but the concept is more general than that,
- *Servers* that receive updates from sources and produce products based on that,
- *Sinks* that consume products from servers.
- *Broker* that coordinates the communication of *sources*, *servers* and *sinks* and decouples these components.

The system involves two kinds of messages, *version* messages that represent changes to sources and *product* messages that are produced by servers. Once the user changes a source (step 1), a new version of the source gets published to the broker (step 2). The broker then broadcasts the new version to all servers (step 3). Servers that can respond to a particular version derive a product and send it back to the broker (step 4). Finally, the sink that is responsible for showing a particular product receives the new product (step 5) and the result can be displayed to the user (step 6).

This architecture is intentionally designed to be very simple. New services can be added easily and Monto be integrated with just a small effort into new IDEs. However, the architecture does not support services that have dependencies to one another. One example where this might be useful would be a server that type checks source files. Type checking is very often based on the Abstract Syntax Tree (AST) of a source file. If there would be two separate servers, one that produce ASTs and one that does the type checking, the work of producing an AST does not

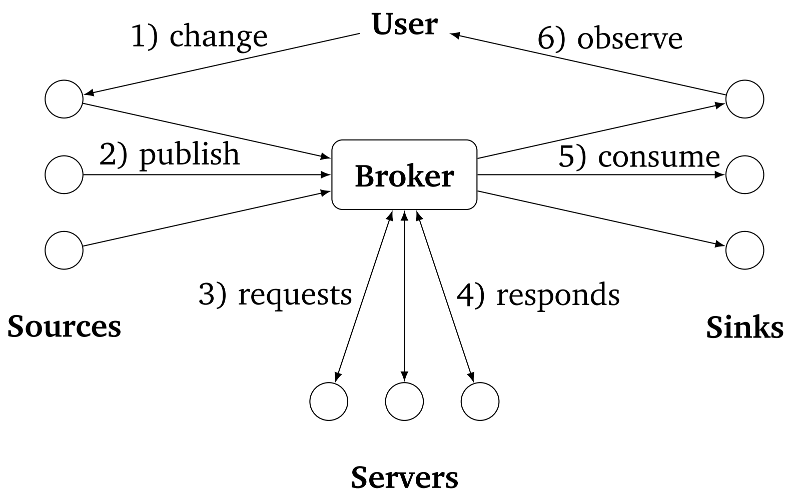


Figure 1.3: Monto architecture

have to be repeated by all services that need an AST. This kind of dependency is called *Service Dependency* because a serveice depends on products of another service.

Another shortcoming of the original architecture is that a product cannot depend on multiple sources. The type checking for instance depends on all sources that are imported into the source for which the type checking is requested. The base architecture does not allow servers to communicate back to the broker that they need other products to complete their work. We call this kind of dependency *File Dependency* because the product of one source file depends on the product of other source files.

Contributions

The Monto project decouples and “disintegrates” plug-ins from IDEs and makes them more reusable. This simplifies the work of plug-in developers and benifits developers that use IDEs with Monto support. The contributions of this thesis are, to use the main idea of Monto, but to address the two shortcomings described above with the current design. This thesis adds supports for service and file dependencies by developing a new broker. Exemplary services like syntax highlighting, an outline and code completion, we try to preserve the simplicity of building services while extending the functionality of Monto. The Monto project since has adopted some of these products message formats as standards and promotes them on their Bitbucket account [22].

Roadmap for the rest of the thesis

Chapter 2 starts with a more in-depth description of the architecture and the used technologies of Monto. Chapter 3 explores the field of services, what data formats need to be established to successfully communicate between services and IDE. Table 1.1 describes which chapter discusses which kind of dependency. Chapter 8 draws a conclusion of the work in this thesis. Chapter 6 contains references to related work in this field. Chapter 7 describes which work has not been realised in the as part of this thesis and opens an opportunity for future work.

		Service Dependencies	
		no	yes
File Dependencies	no	chapter 2	chapter 4
	yes	-	chapter 5

Table 1.1: Different dependency kinds are described and discussed in individual chapters

2 Background

This chapter gives an overview of the architecture of Monto and the used technologies. The chapter discusses the advantages and drawbacks of the existing architecture of Monto especially in contrast of the support of the different kinds of dependencies.

2.1 Architecture of Monto

As shown in figure 1.3, Monto consists of four different kinds of components. Sources that represent source documents that can change and post updates, servers that wait on source updates and produce products, sinks that consume products and the broker that mediates between the three parties. The current architecture uses broadcasting for the communication to the servers to submit new source updates and broadcasting for the communication to the sinks. The role of the broker consists just of forwarding messages from one socket to another. It does not maintain a list of sources, servers or sinks. One advantage of the original architecture of Monto is that the implementation of the broker is stateless and thus very simple.

The broadcasting of source updates to servers implies, that each server gets every source update, even if the server cannot handle the language of the source. This introduces unnecessary overhead because each server has to copy the contents of the message into memory and has to decide if it can handle the message or not. The same argumentation holds for the broadcasting of products to sinks.

The current architecture of Monto does not explicitly support dependencies between servers. As a workaround servers can be declared as sinks such that they can receive the products that they need. But this introduces many problems for these servers. The servers have to store the products on that they depend in memory and have to lookup their memory if a new source update arrives. But what if the server does not have the product if the new source update arrives? The server has to retroactively send out their product if all their dependencies are satisfied. Another consequence is that these servers are no longer stateless. This property affects the scalability of servers and introduces even more complexity. One goal of Monto is to make it very easy to develop servers. This goal cannot be reached by the original architecture for servers that have dependencies to other servers.

The problem becomes even worse if products for one source file can depend on products of other source files. Servers that store products have to know on which products of other source files a particular product depends. If a server receives a new source update, it has to purge products that depend on products that depend on this source. Otherwise the servers could depend on outdated information and their products could potentially confuse the IDE. This kind of dependencies within the original Monto architecture not only complicates the implementation of servers, it makes it impossible to maintain consistency between products.

```
1 {
2   "source": "project/Hello.java",
3   "language": "java",
4   "contents": "public class Hello {\n  public void sayHello() {\n System.out.println(\"hel
  }\n}",
5   "selection": [{"begin":6, "end":10}]
6 }
```

Listing 2.1: Version message for the source `Hello.java`

2.2 Technologies in Monto

Monto aims for very easy integrability and compatibility with different programming languages. The choice of technologies reflects this goal. The main two technologies that languages have to interface with in order to communicate with Monto, are the messaging library ZeroMQ and the data format JSON.

2.2.1 JSON

JSON is a compact open source data format that uses a notation originally derived from JavaScript. The format is used for the messages that are sent between the different components of Monto. Monto defines two kinds of messages. Version messages that are sent from sources to the servers to signal that a new version of a source is available and product messages that contain a product derived from a particular version.

The base version of Monto defines four fields for a version message. The field `source` that contains a unique identifier for a given source, `language` that describes the language that the content of the message is encoded in, the content of the source document itself and `selection`, the text selection of the source document. Listing 2.1 presents an example of a version message of a Java file. The source attribute in the example is a project relative path. An IDE has to manage the values of the source and has to associate sources to sinks. The language field allows services that are specialized for a particular programming language to select these messages that they can process. The contents field contains a properly escaped contents of the Java file. The selection in line 5 is optional. In the example the user selected the class name.

Listing 2.2 contains a product derived from the version message in listing 2.1. The source field has to be the same as from the version message the product is derived from. This allows the IDE to identify which product refers to which source. It is important to standardize the names of products, such that the IDE can recognize special products such as syntax highlighting and can integrate these products better. The actual language in which the product is written is arbitrary, however JSON is a good choice since other servers have a dependency on JSON anyway because of the format of version and product messages.

2.2.2 ZeroMQ

ZeroMQ is a small messaging library that transports messages via different protocols like IPC,TCP, TIPC, or multicast over the network. The library also supports messaging patterns like publish-subscribe, push-pull or router-dealer. The library has been ported to a variety of

```
{
  "source": "project/Hello.java",
  "product": "length"
  "language": "number",
  "contents": "92"
}
```

Listing 2.2: Product message for the source `Hello.java`

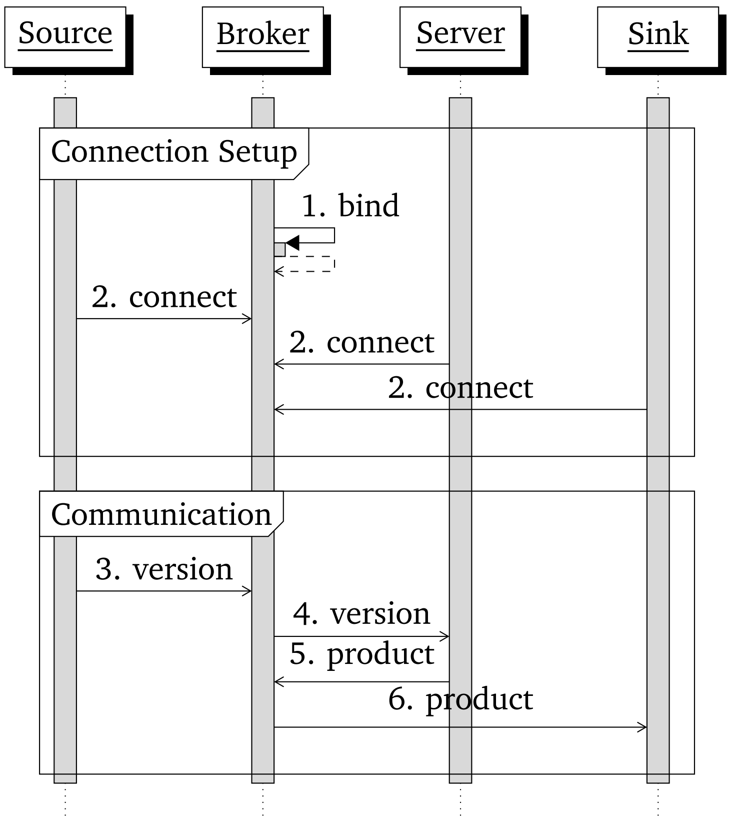


Figure 2.1: Messages send between the broker, sources, servers and sinks

different platforms and programming languages (43 as of February 2015). The library is very essential for the communication between components of Monto. Source, servers, and sinks have to use this library to successfully communicate with Monto.

Figure 2.1 shows a sequence diagram that describes an examplary communication between the broker, a source, a server and a sink. The following pages will explain how one can setup a connection to the broker, what socket types are used and how the system reacts to an updated source.

In step one, the broker has to setup ZeroMQ sockets for sources, servers, and sinks, to which these components can connect in step two. After the connection setup phase, sources can start to submit updates to the broker (step three). The broker then broadcasts the new version to all servers simultaneously in step four. When the servers are ready, they send their product back to the broker (step five) and the broker forwards the product to the sinks (step six). Step three to six then repeat for every new update to a source.

```
1 import montolib
2
3 def reverse(str):
4   str[::-1]
5
6 def handle_reverse(message):
7   return [('reverse', 'text', reverse(message['contents']), True)]
8
9 montolib.server(handle_reverse, 'text')
```

Listing 2.3: Monto Server written in Python that reverse the message content

2.3 Monto Source

A Monto source is often implemented by the IDE plugin for Monto, but could theoretically also be implemented in a stand-alone process. The main task of a source in the IDE plugin is to track changes to physical source code documents. Whenever the underling file of a source changes, the source has to notify the Monto broker of that change. The implementation of the source usually looks very different depending on the IDE, but in general IDEs allow to register callbacks that get called when a file changes. The callback for a source should gather the full contents of the file (not just the change) and then submit a new version message with the name of the file and the new contents to the broker. For ordering purposes, Monto requires a strict consecutive numbering of version message for one source file. This should also be handled by the callback.

2.4 Monto Server

A Monto server is just a method callback that gets called if a new version message arrives. Listing 2.3 shows an example for a Monto server written in python. The server in this particular example reverses the contents of the incoming message. Line 6-7 define a callback that handles new version messages. `message['contents']` extracts the contents of the version message and the method `reverse` actually reverses the string. `handle_reverse` then returns a list of product messages (in this case just one product). A product in this example is a 4-Tuple consisting of the product name, language, contents of the product, and a continuation flag. With the continuation flag, the server can signal if it wants to accept new version messages or to be terminated. Line 9 registers the server and subscribes it to version messages consisting of plain text.

This particular example is written in the Python language, but servers can be written in any programming language that supports ZeroMQ and JSON.

2.5 Monto Sink

A Monto sink consumes products of a particular kind. Like the source, the sink is often implemented by the Monto IDE plugin, but could be a stand-alone process. In the case of the IDE plugin, the sink is usually coupled with a particular source. The idea is that the source publishes updates to the source file and the sink waits on products for this file and supplies the IDE with features like syntax highlighting, etc. The implementation of the sink also depends very much on the IDE, but in general the sink listens on the appropriate ZeroMQ socket for new product messages for the correct source document. If a new relevant product message arrives, the sink parses the product and calls the appropriate methods in the IDE for the content of the product. One could say, a sink represents a demultiplexer that mediates between different products and the corresponding features in the IDE.

Sometimes the IDE enforces a particular control flow whenever a source document changes. The IDE might register callbacks that produce for example the syntax highlighting. In this case, the sink has to use time-outs inside these callbacks to not freeze up the user interface. If a product exceeded the time-out, the sink has to return a acceptable default value. This could be the previous product or a value like `null` depending on the feature.

3 Product Conventions

The IDE plugin for Monto requires that certain products conform to a predefined format. These are products used for syntax highlighting, code completion, an outline, etc. In other words, the "contents" field of product needs to have a certain form such that the sink can correctly parse the product and call the appropriate method in the IDE. Certainly not all products need this kind of specification. For example products that are displayed to the user without any modification can have an arbitrary form and not necessarily need to be standardized.

This thesis presents four products and their formats, that provide basic functionality for an IDE. To lower the burden to integrate these products in other IDEs, the product formats all use JSON; a terse and readable data format that has support in many programming languages. Much like what XML-Schema is to XML, JSON Schema is to JSON. To specify the exact format of products, each section also contains the JSON Schema for the product.

3.1 Syntax Highlighting

The highlighting and coloring of source code is a very essential feature of IDEs and editors. Source code suits very well for this markup because it has a very strict structure. The code can be partitioned into small units called tokens. Tokens are the smallest non dividable unit of code. Each token is part of a category. Example token categories are numbers, identifiers, operators or keywords. The syntax highlighting presented here is solely based on tokens. There are more sophisticated syntax highlighting methods that involve the AST or even type information, but these methods are out of scope of this thesis.

The product format described here is basically just a list of tokens, their positions in the source document and the category. Invisible tokens that are uninteresting for syntax highlighting like whitespace do not need to be included in the product message.

Listing 3.1 shows the JSON format for the beginning of the java document `public class Foo { ...`. The field *offset* represents the start position of the token in the source document. The number of characters that the token contains is encoded in *length* and the category the token belongs to in *category*. This information is enough to highlight the tokens in the IDE.

The possible values for the field *category* have to be fixed to allow the IDE to present the user a configuration dialog where he can adjust the color settings for the different token categories. For this purpose the set of token categories have been derived from the set that the text editor

```
[
  { "offset": 0,  "length": 6, "category": "modifier"   },
  { "offset": 7,  "length": 4, "category": "structure"  },
  { "offset": 12, "length": 3, "category": "identifier" },
  ...
]
```

Listing 3.1: Example product for syntax highighting

```
{
  "title": "Syntax Highlighting",
  "type": "array",
  "items": {
    "type" : "object",
    "properties": {
      "offset": {
        "type": "integer",
        "minimum": 0
      },
      "length": {
        "type": "integer",
        "minimum": 1
      },
      "category": {
        "type": "string",
        "enum": [
          "comment", "constant", "string",
          "character", "number", "boolean",
          "float", "identifier", "statement",
          "conditional", "repeat", "label",
          "operator", "keyword", "exception",
          "type", "modifier", "structure",
          "punctuation", "parenthesis", "delimiter",
          "meta", "whitespace", "unknown"
        ]
      }
    },
    "required": ["offset","length","category"]
  },
  "minItems": 0,
  "uniqueItems": true
}
```

Listing 3.2: JSON Schema for syntax highighting

Vi Improved (Vim) uses [15]. Table 3.1 shows the possible predefined categories for the use in the syntax highlighting format.

It would be certainly possible to replace the field *category* with a field *color*, but that would remove this flexibility from the user.

Listing 3.2 contains the JSON Schema [7]

3.2 Abstract Syntax Tree (AST)

The syntax of programming languages is often based on a grammar. The grammar contains information about possible shapes the expression of this language can have. The structure of the grammar allows to represent each expression of the language in tree form also called AST. The AST of an expression is usually generated by a parser. Nodes of the AST contain the head of the grammar rule that generated the expression or subexpression. The leafs of the AST contain tokens. The leaf nodes are also terminal symbols in the grammar.

Category	Description
comment	any comment
constant	any constant
string	a string constant, "this is a string"
character	a character constant, ’c’, ’\n’
number	a number constant, 234, 0xff
boolean	a boolean constant, TRUE, false
float	a floating point constant, 2.3e10
identifier	any variable name
statement	any statement
conditional	if, then, else, endif, switch, etc.
repeat	for, do, while, continue, break, etc.
label	case, default, etc.
operator	sizeof, +, *, etc.
keyword	any other keyword
exception	try, catch, throw
type	int, long, char, etc.
modifier	public, private, static, etc.
structure	struct, union, enum, class etc.
punctuation	any non alphanumeric tokens that are no operators.
parenthesis	() [] {}, <> etc.
delimiter	, . ; etc.
meta	C preprocessor macros, Java Annotations, etc
whitespace	Non visible character
unknown	Unknown token, can occur during text insertion

Table 3.1: Possible Token Categories for Syntax Highlighting

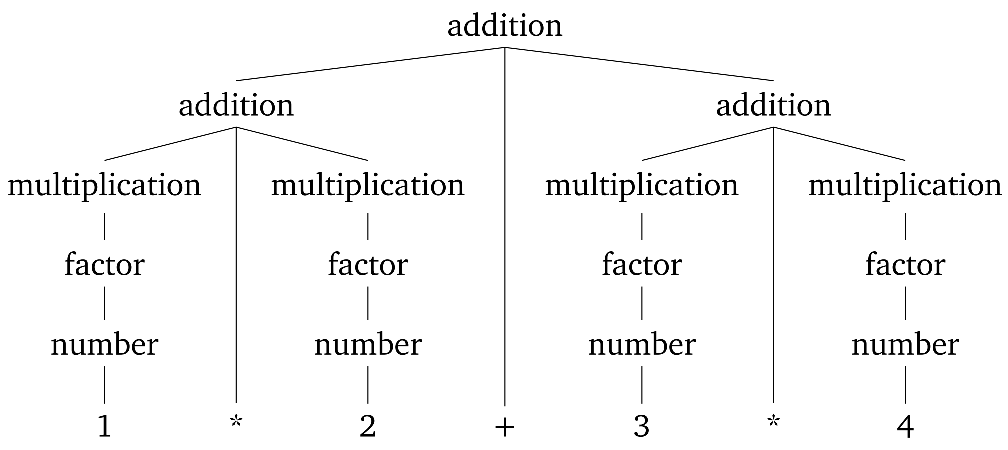


Figure 3.1: AST for the arithmetic expression `1 * 2 + 3 * 4`

A JSON format that represents an AST has to convey the structure of the tree, the label of each node and the position of each token in the source document. The only way to preserve the order of the branches is to encode the tree as nested JSON lists. Figure 3.1 shows the AST of the expression `1 * 2 + 3 * 4`. The grammar of the language is not important for this example, because it is not necessary for the JSON encoding. Table 3.4 shows the corresponding encoding of the expression above. The grammar rule head is stored as the first element of the list. All children of this node in the AST follow as the next elements in the list. The terminal symbols of the AST are encoded as a JSON object with an offset and length that represents the position of the token in the source document. This particular encoding of tokens does not contain the text of the tokens. Services that need to know the text of the token can extract that information from the source document. Including this information in the tokens is redundant and excluding it reduces the AST message size.

Listing 3.3 contains the JSON Schema for the AST product.

3.3 Outline

An outline is a popular Graphical User Interface (GUI) element of an IDE. It summarizes information and the structure of the source document currently edited. In the case of Java an outline gives the user a quick overview of the edited class. It lists the name of the class, attributes, methods, and nested classes. The outline also contains references into the source document. A click on an particular element of the outline navigates the source document to the position were the element is defined. Figure 3.2 shows the outline of a Java class in the Eclipse IDE. Since Java classes have a hierarchical structure, the outline of the class contains elements that are children of other elements, represented by indentation.

Considering all these requirements, the JSON encoding of an outline has to include the text that is displayed on every item of the outline, the reference into the source document and a file path to the icon, that is displayed at the beginning of each item. Listing 3.5 contains the generated JSON for the outline of the Java class shown in figure 3.2.

The field *description* contains a description of the type of item. In the case of Java, this could be *package*, *class*, *field*, or *method*. The field *label* contains the reference inside the source document such that the user can click on the item and gets redirected to the correct position. The field

```
[ "addition",
  [ "addition",
    [ "multiplication",
      [ "multiplication",
        [ "factor",
          [ "number",
            { "offset": 0, "length": 1 }
          ]
        ]
      ],
      { "offset": 2, "length": 1 },
      [ "multiplication",
        [ "factor",
          [ "number",
            { "offset": 4, "length": 1 },
          ]
        ]
      ]
    ]
  ],

  { "offset": 6, "length": 1 },

  [ "addition",
    [ "multiplication",
      [ "multiplication",
        [ "factor",
          [ "number",
            { "offset": 8, "length": 1 }
          ]
        ]
      ],
      { "offset": 10, "length": 1 }
      [ "multiplication",
        [ "factor",
          [ "number",
            { "offset": 12, "length": 1 }
          ]
        ]
      ]
    ]
  ]
]
```

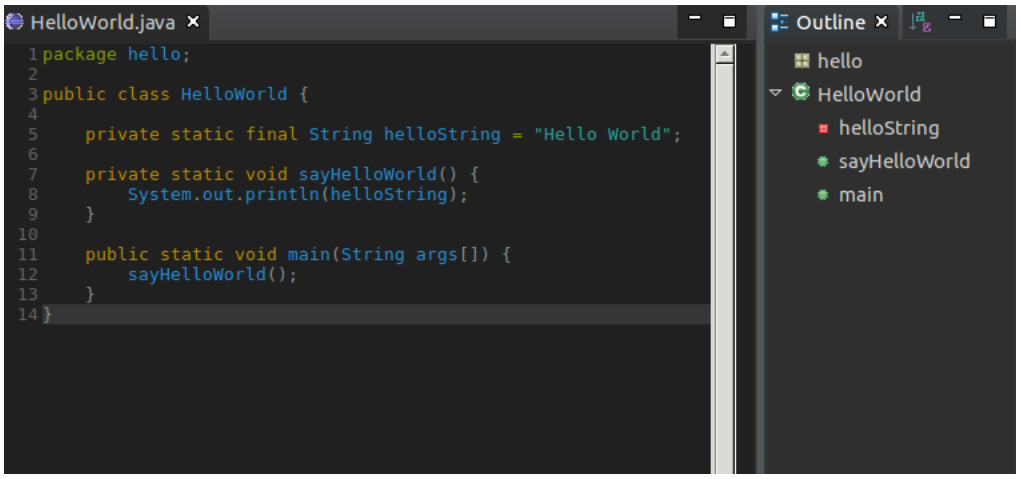
Listing 3.3: Example product for an ASTs

```
{
  "title": "Abstract Syntax Tree",

  "definitions": {
    "leaf": {
      "type": "object",
      "properties": {
        "offset": {
          "type": "integer",
          "minimum": 0
        },
        "length": {
          "type": "integer",
          "minimum": 1
        }
      },
      "required": [ "offset", "length" ]
    },
    "node": {
      "type": "array",
      "items": [
        { "type": "string" },
        {
          "oneOf": [
            { "$ref": "#/definitions/node" },
            { "$ref": "#/definitions/leaf" }
          ]
        }
      ]
    }
  },

  "$ref": "#/definitions/node"
}
```

Listing 3.4: JSON Schema for the ASTs



```
package hello;

public class HelloWorld {

    private static final String helloString = "Hello World";

    private static void sayHelloWorld() {
        System.out.println(helloString);
    }

    public static void main(String args[]) {
        sayHelloWorld();
    }
}
```

Figure 3.2: Outline in the Eclipse IDE

```
{
  "description": "compilationUnit",
  "label": { "offset": 0, "length": 245},
  "children": [
    {
      "description": "package",
      "label": { "offset": 8, "length": 5},
      "icon": "/fullpathto/packd_obj.gif"
    },
    {
      "description": "class",
      "label": {"offset": 29, "length": 10},
      "icon": "/fullpathto/class_obj.gif",
      "children": [
        {
          "description": "field",
          "label": { "offset": 72, "length": 11},
          "icon": "/fullpathto/methpri_obj.gif"
        },
        {
          "description": "method",
          "label": { "offset": 123, "length": 13},
          "icon": "/fullpathto/methpub_obj.gif"
        },
        {
          "description": "method",
          "label": { "offset": 200, "length": 4 },
          "icon": "/fullpathto/methpub_obj.gif"
        }
      ]
    }
  ]
}
```

Listing 3.5: Example product for the outline in figure 3.2

```
{
  "title": "Outline",
  "definitions": {
    "label": {
      "type": "object",
      "properties": {
        "offset": {
          "type": "integer",
          "minimum": 0
        },
        "length": {
          "type": "integer",
          "minimum": 1
        }
      },
      "required": [ "offset", "length" ]
    },
    "outline_item": {
      "type": "object",
      "properties": {
        "description": { "type": "string" },
        "label": { "$ref": "#/definitions/label" },
        "icon": { "type": "string" },
        "children": {
          "type": "array",
          "items": { "$ref": "#/definitions/outline_item" }
        }
      },
      "required": [ "description", "label" ]
    }
  },

  "$ref": "#/definitions/outline_item"
}
```

Listing 3.6: JSON Schema for the outline

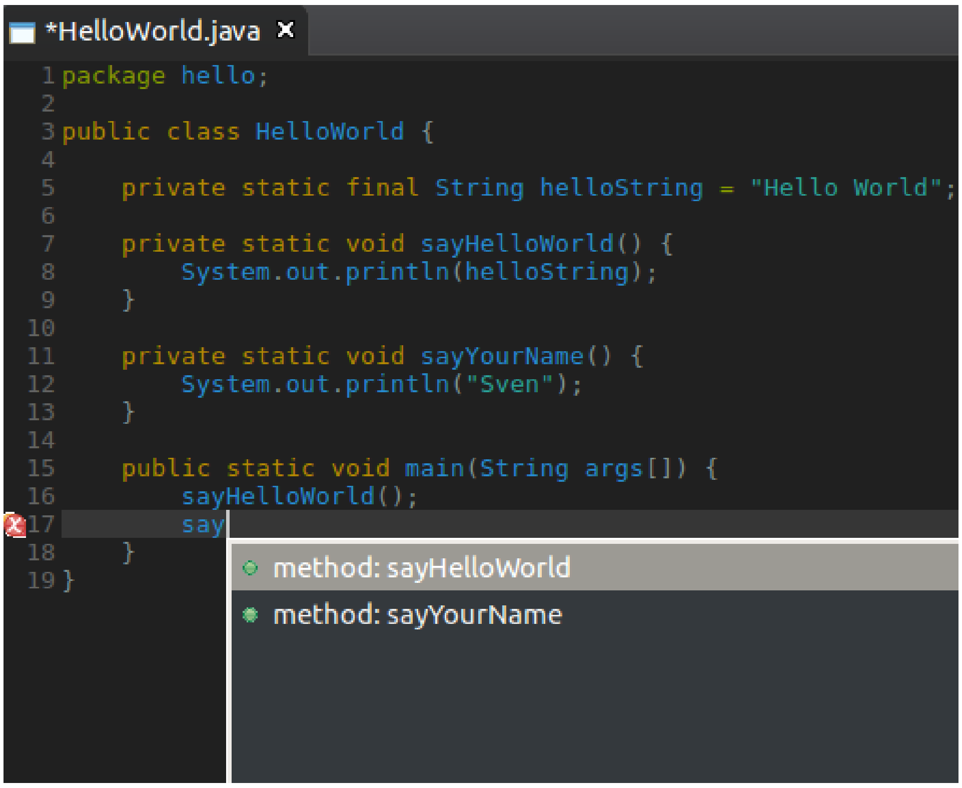
icon is used to transmit the file path of the image that should be displayed at the beginning of each outline item.

Listing 3.6 contains the JSON Schema for the outline product.

3.4 Code Completion

Code completion is another essential feature of modern IDEs. Especially statically typed languages allow IDEs to give the developer very good hints which next code snippet makes sense in a particular context. Figure 3.3 shows the Eclipse IDE during the suggestions of code snippets.

Code completion suggestions are usually presented in a window that overlays the source code document. In the case of Eclipse the cursor marks the position where the completion is inserted. As additional information code completion entries contain the completed text, a description of the kind of completion, and an icon.


*HelloWorld.java

```java
package hello;

public class HelloWorld {

    private static final String helloString = "Hello World";

    private static void sayHelloWorld() {
        System.out.println(helloString);
    }

    private static void sayYourName() {
        System.out.println("Sven");
    }

    public static void main(String args[]) {
        sayHelloWorld();
        say
    }
}
```

Figure 3.3: Code completion in the Eclipse IDE

Listing 3.7 shows the code completion product used in figure 3.3. The JSON format contains a list of all code completion entries relevant for this particular section. Each completion entry contains an *insertionOffset* which marks the position in the source document at which the completion gets inserted. Moreover it contains a description of the completion that is displayed to the user and the replacement which then gets inserted into the document. Finally each entry contains the file path to an icon that gets displayed at the beginning of each code completion entry.

3.5 Discussion

In this chapter product conventions for four very essential features of IDEs have been presented. This should give the reader a good understanding on how to integrate these services into other IDEs and editors or to develop their own services. It has turned out to be very beneficial to develop new product conventions in the presence of a real IDE. We developed the four conventions and at the same time integrated the services in Eclipse. This approach allowed us to figure out the attributes of the JSON format that are really needed and those that do not matter.

Furthermore the JSON used in the product messages seems sometimes very verbose and repetitive. For instance the list of tokens in the syntax highlighting section contains very often the keywords `offset` and `length`. A more efficient encoding could potentially reduce the size of the message significantly. Besides these concerns we did not encounter any performance issues caused by large messages. Therefore we choose the more descriptive JSON format. If one nonetheless encounters performance issues due to this problem, a fast lossless compression algorithm (like LZ4 [11]) used on the JSON messages could solve this. The problem with this

```
[
   {
      "insertionOffset": 319,
      "icon": "fullpathto/methpub_obj.gif",
      "description": "method: sayHelloWorld",
      "replacement": "HelloWorld"
   },
   {
      "insertionOffset": 319,
      "icon": "fullpathto/methpub_obj.gif",
      "description": "method: sayYourName",
      "replacement": "YourName"
   }
]
```

Listing 3.7: Example product for the code completion in figure 3.3

```
{
  "title": "Code Completion",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "insertionOffset": {
        "type": "integer",
        "minimum": 0
      },
      "icon": { "type": "string" },
      "description": {
        "type": "string"
        "minLength": 1
      },
      "replacement": {
        "type": "string",
        "minLength": 1
      }
    },
    "required": [ "insertionOffset", "description", "replacement" ]
  }
}
```

Listing 3.8: JSON Schema for code completion

approach is that both sides, the services and the IDE have to support the compression algorithm. This could potentially limit the portability of these services.

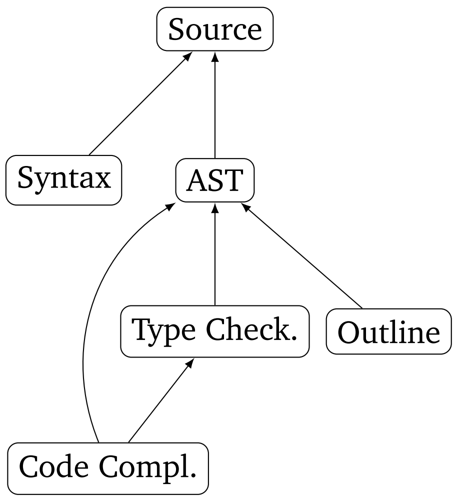
4 Service Dependencies

As discussed in previous chapter, services can have dependencies on other services. For instance a type checking services may have a dependency on the parsing services that generates ASTs. All dependencies of all services are the edges of a Directed Acyclic Graph (DAG) like the example in figure 4.1. Some services even may have multiple dependencies like the code completion service in the figure that depends on the AST and the type checking of a file. The original version of Monto does not support dependencies between services, although there is a very good reasons for having these. The sharing of products gives on the one hand a performance improvement, because the same work does not have to be done multiple times. The sharing of the AST product of figure 4.1 is a good example for that. The type checker, outline, and code completion service can all use the same AST produced by the parser and do not have to compute their own. On the other hand this approach allows services to become more minimal and concentrate on one particular topic. For instance writing a parser for a general purpose programming language like Java is a lot of work. If this work has been done once, developers of other services that depend on java ASTs can benefit by not having to write their own parsers.

4.1 Stateless Broker, Stateful Servers

The current implementation of Monto does not explicitly support service dependencies. The broker sends new versions to all servers, the servers respond with products and the broker sends these to the sinks. The only way to allow a service to receive products is to make it a server and sink at the same time. Figure 4.2 shows this approach.

The problem with this approach is, that it forces services that have more than one dependency to become stateful. For example the code completion server from figure 4.1 has to wait for the AST and the type checking before it can start with its own work. The only way to do that



$$\begin{tikzcd}
 & \text{Source} & \\
\text{Syntax} \arrow[ur] & \text{AST} \arrow[u] & \\
 & \text{Type Check.} \arrow[u] & \text{Outline} \arrow[ul] \\
\text{Code Compl.} \arrow[uur, bend left] \arrow[ur] & &
\end{tikzcd}$$

Figure 4.1: DAG of dependencies

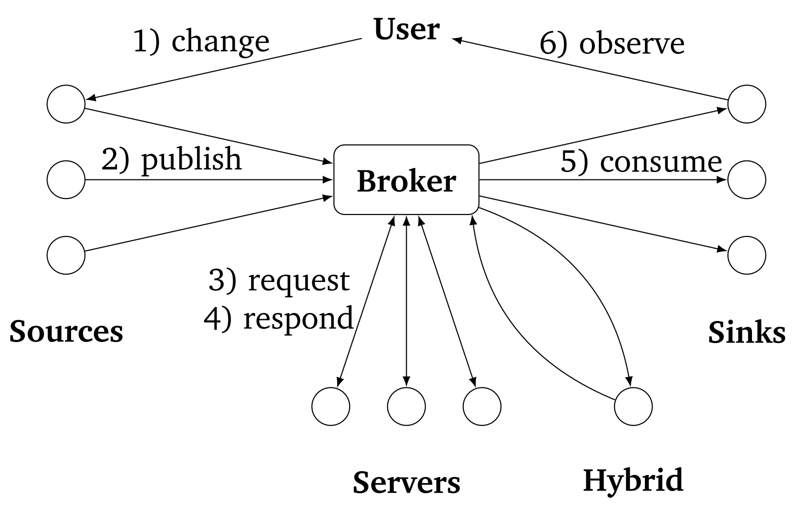


Figure 4.2: Monto architecture with service dependencies

is to wait until the AST product has arrived, store it in a map, wait until the type checking product has arrived, lookup the AST product from the map and do its work. This approach not only introduces an unnecessary level of complexity on services, it also increases the memory footprint of the whole system. Each service needs to store all products that it potentially will need in the future. The only upside to this approach is that the broker can be left stateless. Its role is literally forwarding messages from sources to servers and from servers to sinks.

4.2 Stateful Broker, Stateless Servers

As described in the previous section, the broadcasting of the broker keeps the broker logic simple and increases complexity on servers that have dependencies. But one of the goals of the Monto project is to simplify the development of services as much as possible. This goal is not achievable with the broadcasting mechanism.

This thesis presents an alternative implementation for the Monto broker that is almost dual to the existing implementation. It delegates more logic and complexity to the broker to simplify the development of services. Along these lines, the broadcasting communication is changed to unicast and the broker is designed to manage all the state and servers can be stateless.

In the original implementation, described in section 4.1, dependencies were handled implicit. The information of which server depends on which products were just known by the servers themselves. The alternative implementation however requires that the broker maintains a graph of server dependencies. This allows the broker to coordinate the communication between the servers. The implementation of the broker can be summarized in the following steps:

1. If the broker receives a new version message from the sources
    a) The broker sends the new version to each server that depends just on the version message
    b) The broker deletes all stored products that depend on this version
2. If the broker receives a new product
    a) It forwards the product to the sinks

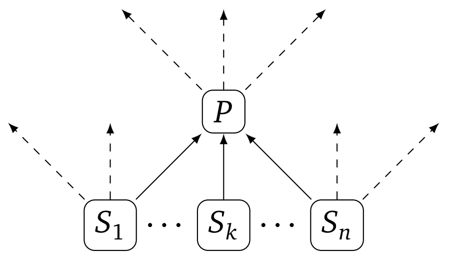


Figure 4.3: Direct Dependency Graph Lookup

b) It stores the product

c) It looks up in the dependency graph, which servers have satisfied dependencies and sends a compound message with all dependencies to that servers.

All the servers have to do, is to wait on the request that bundles everything the servers need to complete their job. This property allows the servers to be stateless.

One important issue that has to be addressed in this implementation is, how does the broker know which servers have satisfied dependencies? This issue can be addressed in two different ways, a direct dependency graph lookup or an Mealy machine. This thesis describes each approach separately and discusses their advantages and disadvantages over one another.

4.2.1 Direct Dependency Graph Lookup

This approach uses only the dependency graph and the information which messages are currently stored in the broker. When a new product P arrives at the broker, the broker looks up in the dependency graph, which servers depend $(S_1, \dots, S_n)$ on this product and then looks up which of these servers have now satisfied dependencies. Figure 4.3 visualizes the for this example interesting part of the dependency graph. One problem with this approach is its high runtime complexity. In a dependency graph with n nodes, a product can be a dependency for at most $n-1$ other servers. One of these servers can than depend on at most $n-1$ other products. Another one of these servers can depend on at most $n-2$ other products (because the graph is acyclic), and so on. With that reasoning, we get $\sum_1^{n-1}(n-1) = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)n}{2}$ lookups which gives us a worse complexity of $O(n^2)$.

4.2.2 Mealy Machine

One interesting property of server dependencies is that they never change during program execution. This property can be exploited to build a Mealy machine that knows in every step, which products are available and which servers have satisfied dependencies and have to be notified next. This reduces the complexity to $O(1)$ assuming the Mealy machine can change its state and lookup the outputs in constant time.

The algorithm for building the machine presented in this thesis requires the dependency graph to be a complete lattice. The partial order in this case is the dependency relation of two elements. The graph also needs to contain a node that has no dependencies (the source document) and a node that is no dependency from any other node (we call this node top). Further every two nodes of the graph need two have a lowest element that they both transitive depend on.

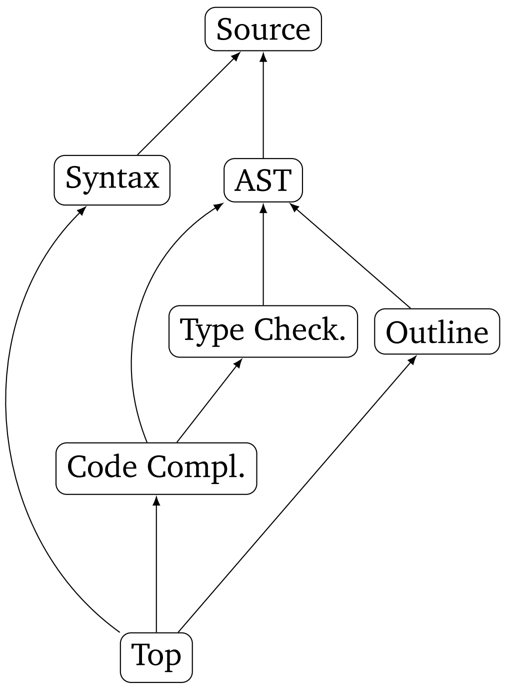


Figure 4.4: Dependency graph that is a complete lattice

Finally every two nodes of the graph need a greatest node that they are both a transitive dependency of. The reason for the requirement that the graph is a complete lattice will become clearer later.

Figure 4.5 shows the Mealy machine for the dependency graph of figure 4.4. The dependency graph contains a top node, that is connected to every element that is no dependency of any other product. This ensures that the graph is a complete lattice. Each state contains a description of which products are currently available. The symbol $\perp$ denotes that the product has not yet arrived and the symbol $\top$ denotes that the product has arrived. Each transition contains a label that describes which product arrives and optionally which servers have satisfied dependencies and need to be notified next. For instance the label "AST,[Out,Typ]" denotes that the product AST has arrived and that the outline and type checking servers have satisfied dependencies. In the initial state all products are not available. The machine models the non-determinism of the system. For instance it is not clear if the syntax highlighting or the AST product arrives first. Both products depend only on the source document. After a new version of the source document arrived, the machine contains two succeeding states depending on which product arrives earlier. Another important property of the machine is, that every two states have transitions to a common succeeding state. This property ensures that the broker initiates the production of all products and converges in the state where all products have arrived. Indeed each trace from the initial state to the final state contains each product of the graph exactly once.

The easiest way to create the machine for a given dependency graph is to build a machine for just one edge of the graph and then to merge the machines with a special union on machines. Figure 4.6 shows how a single product p of the dependency graph and all its immediate dependencies q_1, q_2 and q_3 are translated into a Mealy machine. The set of states in this example are the permutations of the set of products and $\{\top, \perp\}$. The initial branching and rejoining of the machine models the non-determinism of the creation of dependencies for the end-product p.

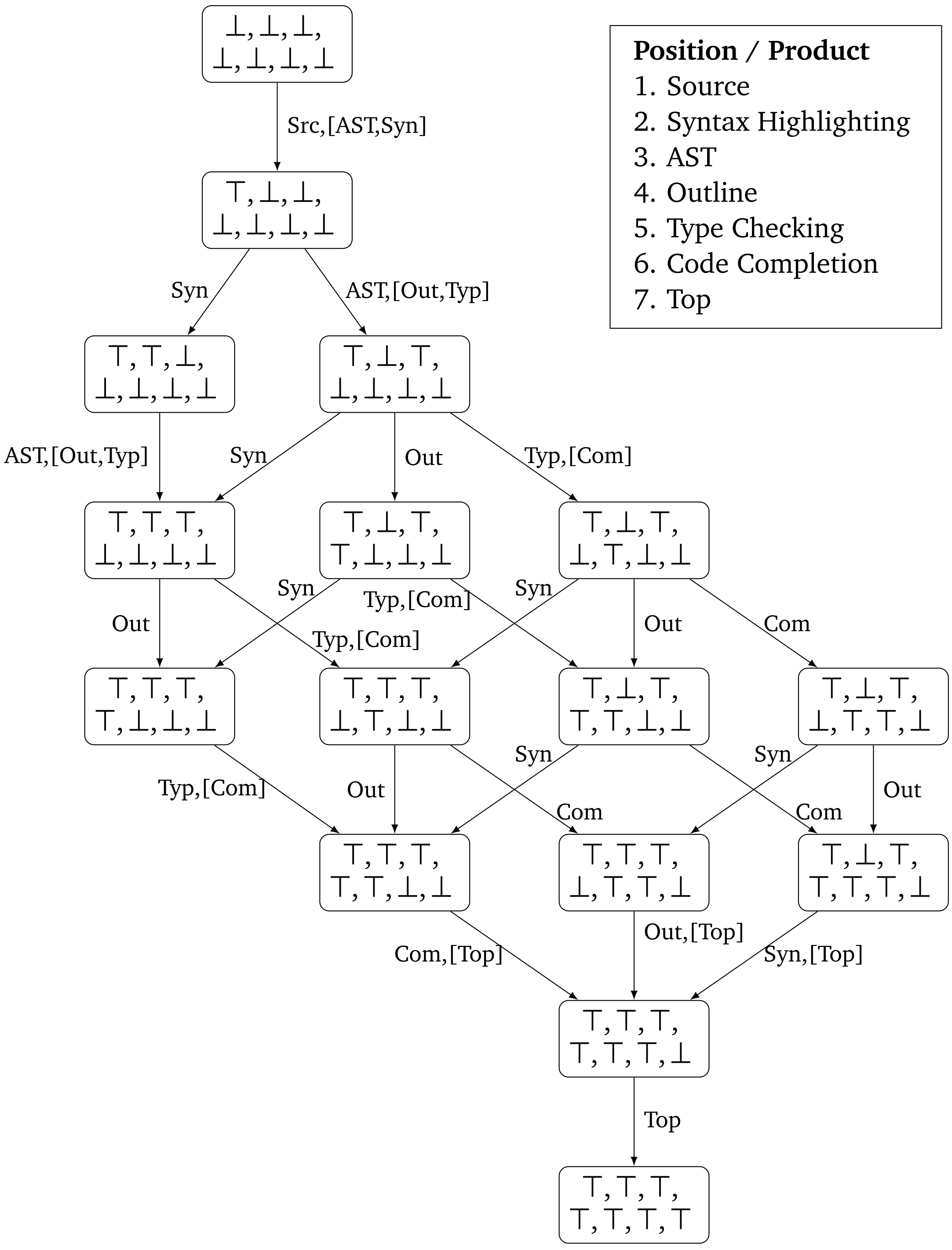


Figure 4.5: Mealy machine for dependency graph of figure 4.4

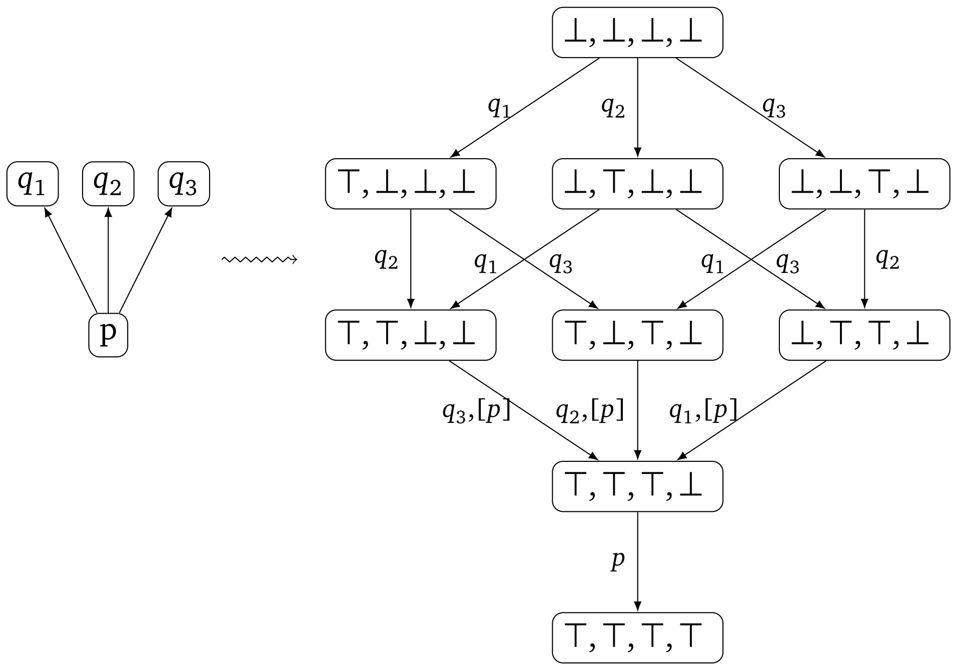


Figure 4.6: Mealy machine for a small part of the dependency graph

Once all dependencies are available in state $(\top,\top,\top,\bot)$, the end-product can be build and the machine can transition into its final state $(\top,\top,\top,\top)$.

The Mealy machine for the whole dependency graph can now be aggregated by merging these smaller machines for single products with a union operation. The union operation uses a equivalence relation that checks if two states have products in common that these products have the same availability. More precise, a state is a partial mapping from products to availability $\{\top,\bot\}$. Two states s_1, s_2 are equivalent iff $\forall p : p \in s_1 \land p \in s_2 : s_1(p) = s_2(p)$. We also write $s_1 \equiv s_2$. The transitions are included into the resulting machine if the starting and final states of a transition of one machine match with the states of the transition of the other machine. For the transitions of the first machine $T_1 : S_1 \times \Sigma \longmapsto S_1 \times \Gamma$ and the transitions of the second machine $T_2 : S_2 \times \Sigma \longmapsto S_2 \times \Gamma$ the decision what transitions are included in the resulting machine can be divided into three **non-exclusive** cases:

for all $s_1, i_1, s_1', o_1, s_2, i_2, s_2', o_2 : T_1(s_1, i_1) = (s_1', o_1) \land T_2(s_2, i_2) = (s_2', o_2)$

- if $s_1 \equiv s_2$ and $s_1' \equiv s_2'$ then let $T'(s, i_1) := (s[i_1 \mapsto \top], o_1)$ and $T'(s, i_2) := (s[i_2 \mapsto \top], o_2)$ where $s = s_1 \cup s_2$
- if $s_1' \equiv s_2$ then let $T'(s[i_1 \mapsto \bot], i_1) := (s, o_1)$ and $T'(s, i_2) = (s[i_2 \mapsto \top], o_2)$ where $s = s_1' \cup s_2$
- if $s_2' \equiv s_1$ then let $T'(s[i_2 \mapsto \bot], i_2) := (s, o_2)$ and $T'(s, i_1) = (s[i_1 \mapsto \top], o_1)$ where $s = s_2' \cup s_1$

The union of two states $s_1 \cup s_2$ is defined as the smallest set s that satisfies $\forall p : (p \in s_1 \land p \notin s_2 \Rightarrow s[p \mapsto s_1(p)]) \land (p \notin s_1 \land p \in s_2 \Rightarrow s[p \mapsto s_2(p)]) \land (p \in s_1 \land p \in s_2 \Rightarrow s[p \mapsto s_1(p) \sqcap s_2(p)]) \land (p \notin s_1 \land p \notin s_2)$

The least upper bound $a \sqcap b$ of two avialability statuses a and b is defined as $\top$ if $a = \top$ or $b = \top$ else $\bot$.

		Original	Alternative
Servers	State	Stateful	Stateless
	Code	Complex	Simple
Broker	State	Stateless	Stateful
	Code	Simple	Complex
Dependencies		Implicit	Explicit
Communication		Broadcast	Unicast
Messaging Overhead		High	Low
Memory Footprint		High	Low

Table 4.1: Comparison of Implementations

4.3 Discussion

As described in this chapter, dependencies between services require either complex logic in the services or in the broker. Both implementations have their own advantages and disadvantages. This section discusses and summarizes the different implementations.

The original architecture designated the state and with that the complexity in the implementation to the servers. The responsibility of the broker in this case is just forwarding messages from the sources to the servers and from the servers the sinks.

The alternative architecture designates all state to the broker. This allows the servers to be state less. In contrast to the original architecture this eases the implementation of servers, because they do not have to manage any synchronization. The alternative architecture handles dependencies more explicit because it uses a dependency graph that has to be provided at the start of the broker. On the other hand, the dependencies in the original implementation were handled more implicit. The servers had to know which products they have to cache in order to have access to them later. The handling of state only in the broker also has the benefit that the overall memory footprint of the system is lower. Different products are stored only in the broker and not multiple times in different servers. The trade-off in the alternative implementation is that the complexity lies within the broker. We presented two different solutions for dealing with the state in the broker. One involves only the dependency graph between different services, the other builds a Mealy machine that encodes which servers have to be notified next. Although the broker has become more complicated, the messaging overhead in the alternative architecture is less than in the original, because the alternative uses multi-cast instead of broadcast in the original. Table 4.1 summarizes the different attributes of the original and the alternative design.

5 File Dependencies

Beside dependencies between servers, as discussed in the last chapter, files can have dependencies between each other. An example of this would be a Java file that has a number of import statements at the beginning of the file. Each of this imports represents a dependency from the file with the import statement to the imported file. This kind of dependencies are not important for all Monto services. A syntax highlighting service for example can operate on the file alone without considering import statements. In contrast products of other services like the type checking service depend on the type checking information of the imported files. The base version of Monto does not support this kind of dependencies.

There are a set of problems that have to be solved to support file dependencies. This chapter describes these problems and possible solutions for them. Moreover, often there is no single true solution for these problems, so different solutions are compared to each other to give an overview over the design spectrum of this field.

5.1 Consistency

It may happen that a service bases its product on a product of another source file that has recently changed. In some cases this may not be an issue, in other cases the depending product becomes wrong and violates the contract between Monto servers and the IDE. To give an concrete example: the type checking of a Java file A depends on the type checking of the Java file B. If file B now changes its interface in a way that file A would not type check anymore, the product for file A has to be regenerated to regain consistency. In other words, if file B changes, all products that depend on this file have to be invalidated to maintain consistency.

In section 4.1 different approaches have been discussed, which components of Monto should maintain the state of the system. The two approaches basically where that the broker or alternatively the servers maintain all the state. In the later approach, especially in contrast to file dependencies, maintaining consistency imposes a non trivial problem. The problem can be compared to maintain consistent state in a distributed system. We made some efforts in this space to archive at least eventual consistency between different products, but the complexity of the implementation of servers became unnecessary high.

Managing the state within the broker, like in the case of service dependencies, provides mutual benefit. The issue with consistency can be dealt easily and the implementation of servers becomes much easier.

5.2 Direct Graph Lookup versus Mealy Machine

Like in chapter 4, the dependencies between files can be represented as a directed acyclic graph (cyclic dependencies are future work and are discussed in chapter 7). Except the graph of file dependencies constantly changes and the graph of service dependencies stays the same during the execution of the program. This new aspect requires to rethink the solutions of the last chapter.

In this context servers need a way to communicate back to the broker, which products of other files they need to produce their product for the given file. This can easily be solved by adding an additional communication channel to the broker that receives these requests. The role of the broker then becomes to manage the graph of file dependencies. When a new product request arrives at the broker, the broker adds the appropriate edges to the dependency graph and when a source document is updated, products that depend transitively on the source need to be removed from the graph.

The critical question that needs to be answered is, which servers need to be notified when a new product arrives at the broker. We presented two approaches in the previous chapter. The direct graph lookup that looks up if all predecessors of the arriving product have satisfied dependencies or the Mealy machine that tracks with its state which products are already there and which servers have satisfied dependencies. For a static non-changing dependency graph the mealy machine is the better choice because, once the machine is constructed, the runtime complexity of dealing with products is lower than the direct graph lookup. But in the case of a constantly changing dependency graph of file dependencies, this approach is not the best solution. A problem is that the graph for file dependencies easily becomes much larger than the graph for server dependencies. The Mealy machine for a dependency graph with n nodes has in the worst case $O(2^n)$ different states. Rebuilding the machine every time the dependency graph changes is a costly computation. In comparison the runtime cost for the direct graph lookup are $O(n^2)$. In the context of an often changing graph the direct graph lookup is the cheaper approach in terms of runtime costs.

5.3 The interplay of service and file dependencies

The broker now maintains two dependency graphs, a Mealy machine for each source, and a cache of products that have arrived. This leaves the question, how does the broker manage both kinds of dependencies?

Figure 5.1 shows an example that illustrates problems that occur during the generation of products with cross-file dependencies. The figure shows a mix of service and file dependencies. In this example the broker tracks the dependencies for two services, the parsing service that creates ASTs and the type checking service. The type checking product for file S_3 depends on the type checking product of file S_2 and this depends on the type checking product of file S_3. The figure shows different states of the broker during the execution. In the initial state a, the parsing product of file S_3 has just arrived. This state exposes the first problem that the broker has to deal with: The type checking product $Type_3$ has satisfied service dependencies, but needs still file dependencies. In this case, the broker needs to notify the IDE that it needs the content of other source files of all products that $Type_3$ depends on, namely S_1 and S_2 (see state b). State c occurs after the parsing product and the type checking product for S_1 have arrived. This state exposes the second problem that can occur: The type checking product $Type_2$ has satisfied file dependencies, but has a missing service dependency AST_2. In this case, the broker does not need to initiate any action because the process for creating AST_2 has already been started. Once the AST_2 product arrives as in state d, the broker uses the direct graph lookup method to see which products that directly depend on AST_2 have satisfied dependencies. $Type_2$ has now satisfied dependencies and can be created (see state e). In state e, the broker also detects with the direct graph lookup method that $Type_3$ has satisfied dependencies and can be build. Consequently, state f represents the arrival of $Type_3$ that could finally be build.

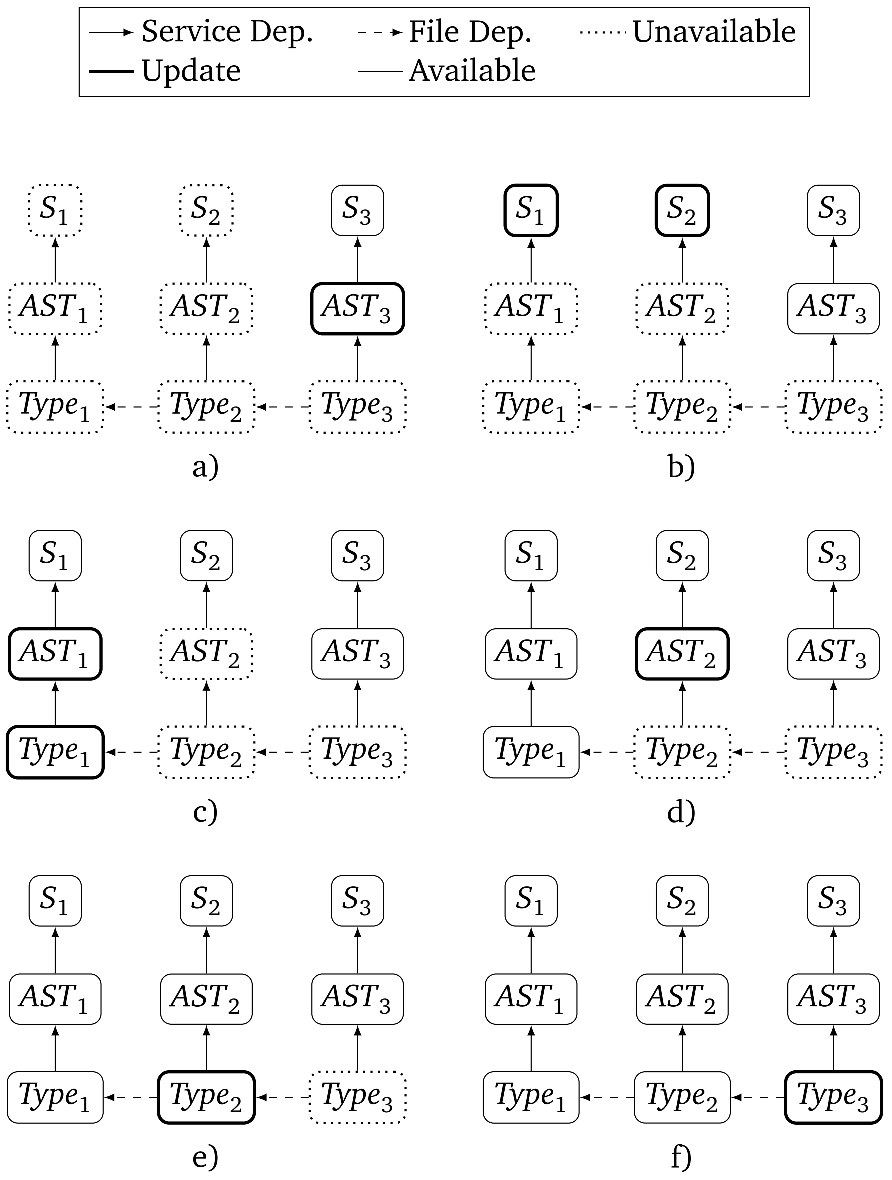


Figure 5.1: Example for the interplay of service and file dependencies

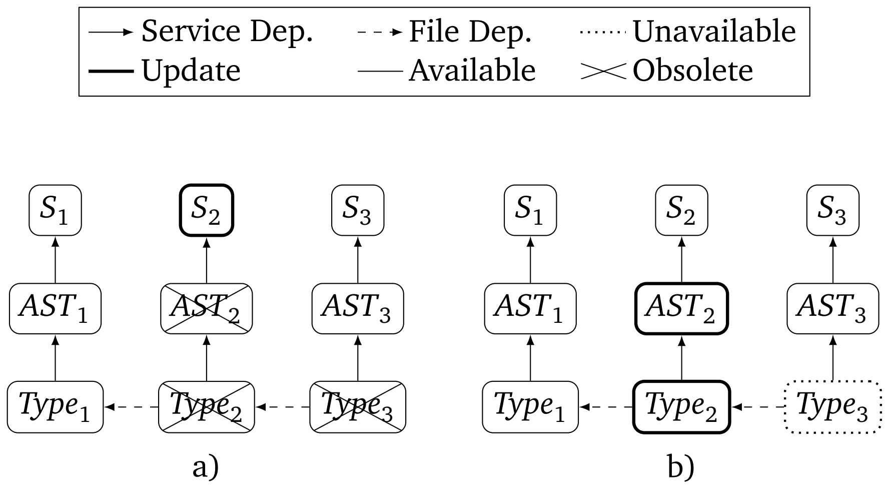


Figure 5.2: Example of updates in the interplay of dependencies

Figure 5.2 shows another problem that occurs if one of the source files is updated. In this case S_2 is updated and the broker has to delete products that depend on this source file (state a). The broker than continues building products for S_2. After the AST_2 and $Type_2$ product arrived in state b, the question arrases if $Type_3$ needs to be recreated or not. The user has recently edited source S_2, which triggered the deletion of product $Type_3$. While editing source S_2, the user is probably not interested in the type checking result for S_3. If the user then edits source file S_3, the type checking product $Type_3$ is then recreated as usual. This is certainly a compromise and is further discussed in the next section.

Generalizing solutions for the problems that figure 5.1 revealed:

- *The broker needs to build a product P that has unsatisfied file dependencies (state a).* All source files that are dependencies of products which are transitive file dependencies of P and are not already available in the broker need to be requested from the IDE. In other words, all sources that are reachable by a depth first search over the file dependencies graph and the broker does not have a version message for these sources available.
- *The broker needs to build a product Q that has satisfied file dependencies, but unsatisfied service dependencies (state c).* If the user needs a product that depends on Q then the sources for the generation of Q have already been requested from the IDE and nothing needs to be done. If the user does not need a product that depends on Q then Q does not need be built in the first place and the broker can ignore Q.
- *The broker recieves a new version of a source file.* All products that depend transitively on this source file need to be deleted from the message cache of the broker to maintain consistency.

5.4 Eagerness and Laziness

This leads to the core design principles behind the broker that was developed in this thesis. Products along services dependencies are produced eagerly and products along file dependencies are produced lazily. In other words, the product for a particular source file are produced eagerly and the products that are a file dependency of another product are produced lazily.

These design principles have been chosen to give good performance while the user is interacting with Monto, without wasting resources. The principles support the following use case very well: The user starts editing a new source file in the IDE and then consecutively performs editing operations on this file. During the editing, the user is not interested in products of other source files that depend on the edited file. This is certainly a compromise since these information could be useful. Especially products like type checking give useful feedback if the user stops editing and sees that the project does not type check anymore.

6 Related Work

Language workbenches

In 2005 Martin Fowler coined the term *language workbench* [9]:

> *"Essentially the promise of language workbenches is that they provide the flexibility of external [domain specific languages] DSLs without a semantic barrier. Furthermore they make it easy to build tools that match the best of modern IDEs. The result makes language oriented programming much easier to build and support, lowering the barriers that have made language oriented programming so awkward for so many."*

Language oriented programming in this context describes a programming style that involves the usage of domain specific languages. In a sense language workbenches have similar goals compared to the Monto project. Language workbenches try to simplify the work of language developers. [6] gives an overview of the State of the Art in language workbenches. There are several projects that fall into this category. The IDE Meta-tooling Platform (IMP) [3] project provides a layer on top of the Eclipse plugin architecture to address this problem. The Monto support for eclipse has been build with the help of IMP. The Spoofax/IMP project [8] provides a language workbench that integrates the term-rewriting language Stratego and SDF for formulating grammars. The mentioned projects all integrate solely with Eclipse.

The Sugarclise project [5] tries to improve the extensibility of IDEs by organizing services in *editor libraries*. An advantage of this particular organization is that editor libraries for multiple languages can be composed within a file. For the extension of language the project uses SugarJ [4] and for the extension of the IDE the language workbench Spoofax/IMP. Although the project provides no guarantees for the modularity of editor services, the composition of editor services usually resolves without conflicts. If, nevertheless, there are conflicts, when for example two editor services provide code completion for the same piece of code, these conflicts can be resolved by simple combining the two sets of completions into one. Editor libraries can be selected on a file-by-file basis with import statements. This allows fine granular control over e.g. which code completion the user wants to see for a given file. In Monto this use-case can be realized by swapping the servers for a code completion. This does not give the same level of control as editor libraries, but allow some flexibility for the user. The composition of services is still an open problem in Monto and the use of SugarJ and similar technologies could be a benefit in these cases.

NeoVim: A fork of Vim with an extensible plugin API

The project NeoVim [16], a fork of the Vim project, tries to refactor the legacy code of the project and rethink the plugin architecture. Plugins for the old plugin architecture had to be written in Vim-Script, thus these plugins have a tight coupling to the Vim API and cannot be used anywhere else. The new plugin architecture allows plugins to communicate via TCP, named pipes, or with standard input and standard output with NeoVim. The data exchange format is MessagePack, a compressed version of JSON for good interoperability. With this approach, plugins can be written in any programming language as long as they support these technologies.

```
1 @neovim.plugin
2 class ExamplePlugin(object):
3
4   @neovim.autocmd('BufEnter',
5                   pattern='*.py',
6                   eval='expand("<afile>")',
7                   sync=True)
8   def autocmd_handler(self, filename):
9     # handle 'BufEnter' event
```

Listing 6.1: Example for a NeoVim plugin

The NeoVim editor acts as a Monto broker. It sends events that happen in the editor (like saving a buffer) to plugins. Plugins can subscribe to certain events with a pattern matching syntax. Listing 6.1 shows an example for a NeoVim plugin described in the documentation on GitHub [17]. Line 4 following show an example of the subscription to an event. In this case the plugin subscribes to a “BufEnter” event, that is produced if a user changes the focus to a buffer and starts editing. With the pattern `*.py` the plugin subscribes to to “BufEnter” events for Python files. Line 6 describes a NeoVim command that is executed before the plugin is notified. In this case the name of the file is expanded and sent as an argument to the plugin. The sync option lets NeoVim block until the plugin has responded.

srclib: A polyglot code analysis library

The srclib project [21] is a project that tries to archive the same goals as Monto. The project consists of so called language analysis toolchains with a common output format. Toolchains are similar to Monto services. They are started on the command line and communicate with JSON over standard input and output. Multiple toolchains are coordinated with a tool called *src* written in Go. Editors and IDEs have to interface only with *src* and not with the toolchains directly. Each language specific toolchain is composed of five parts:

1. A *scanner*, that traverses the project file system looking for source documents,
2. A *grapher*, that uses static analysis to build a code graph,
3. A *dependency resolver*, that follows URLs to other source code repositories,
4. A *formater*, that transforms language-specific data into a Go data structure.

Toolchains can run in two modes, as a normal program that is installed as a system package or inside a Docker [2] container that includes all the toolchain’s dependencies. srclib currently handles package detection, global dependency resolution, type inference, definition-use graphs, version control systems, etc. It supports the Emacs, Sublime, and Atom text editors. The project supports a standard set of toolchains for Python, Go, Ruby, JavaScript, Java, Haskell, and PHP.

In contrast to the Monto project, srclib does not allow the development of arbitrary plug-ins. The plug-in has to fit in one of the five components of toolchain described above. Furthermore the control flow between the components is fixed and cannot be freely chosen like in Monto.

Monto related projects

The Monto project has been integrated into two platforms, the Sublime Text 3 editor [14] and Eclipse [13]. These projects can be used as a reference to integrate Monto into other IDEs and text editors. The additions for the Monto broker have been submitted to its own repository [12].

7 Future Work

The product conventions presented in chapter 3 cover just a small portion of features modern IDEs have to offer. These conventions play an important role making sure services and IDEs can communicate on the same basis. More services and product conventions have to be developed to establish a vital ecosystem around Monto. To name just a few services that have not been covered in this thesis and would be great showcases for Monto: type checking, debugging, and a Read Eval Print Loop (REPL).

As discussed in chapter 3 one could explore the effect of a lossless compression on the JSON messages. Since many JSON messages contain similar parts, compression could bring a huge benefit in a reduced size of the message. One should choose a compression algorithm that is rather fast than giving a good compression ration because if the compression takes longer than the transmission over the messaging system, no speedup is gained. It is important to keep in mind the availability of this compression algorithm in different programming languages. A more exotic compression algorithm, that has not very wide spread support may reduce portability and raises the burden of integration.

Another feature that needs more investigation are cyclic dependencies between files. Not many programming languages support this kind of dependencies. Programming languages that include files within a preprocessing step like C and C++ do not support this kind of dependency. An exception to this is Java as its compiler allows files to have mutual dependencies between each other. The broker developed in this thesis assumes that the dependencies of all files form a Directed Acyclic Graph. A cycle in this graph would probably cause the broker to run in an infinite loop. The proper support for cyclic dependencies is out of scope of this thesis and is part of future work.

Section 5.4 discusses the design trade off this thesis accepted for the implementation of the broker. It would certainly be interesting to allow products to define if they should be recreated eagerly or lazily. For example the type checking product could be recreated eagerly throughout a project. The user would get fast feedback if recent editing operations affect the type checking of the rest of the project. This addition requires massive changes to the broker presented in this thesis and might not be easy to implement.

8 Conclusion

In this thesis, we extended the work of Tony Sloane *et al.* on Monto. A tool that *disintegrates* IDEs and their extensions into individual reusable components. One goal of Monto is to keep it simple to implement new servers. We tried to verify this property during the development of new features for Monto by building exemplary servers for syntax highlighting, parsing, an outline, and code completion. The developed product formats in this thesis have since been contributed to the Monto project and are officially supported.

We extended the base version of Monto to support two new kinds of dependencies. Dependencies between Monto services that allow to reuse common products for a particular source document and dependencies between files, that allow servers to request products of other source files. Both kinds of dependencies have different requirements and cannot be solved in the same way. Service dependencies are known from the start of the Monto system and do not change during the execution. The most efficient way to handle these static dependencies is to build a mealy machine that keeps track of which products are available and which servers need to be notified next. File dependencies on the other hand change very often. If a server detects that it cannot compute his product for a given source file, it communicates back to the broker that it needs products of other source files. For this constantly changing dependency graph it is more efficient to not maintain a mealy machine but to lookup in the dependency graph directly if all dependencies for a server are satisfied.

In this thesis an Eclipse plugin for Monto has been developed. The original broker and the extended broker are "mostly" interchangeable in conjunction with the plugin. The only changes that have to be made to support the extended broker are to integrate a new socket on which the broker can ask for other source files that have not been opened yet. Moreover, the rest of the plugin could stay the same to support both broker versions. The Monto servers on the other hand need more adjustment to work with one or the other version of the broker.

The Monto project is still very young and needs adoption, in IDEs, text editors, as well as the development of Monto servers. The success of the project depends on the degree of adoption within the developer community.

Acronyms

API Application Programmable Interface. 5

AST Abstract Syntax Tree. 7, 8, 15, 16, 18–20, 26, 27, 29, 34

DAG Directed Acyclic Graph. 26, 41

DDE Disintigrated Development Environment. 7

GUI Graphical User Interface. 18

IDE Integrated Development Environment. 2, 5–8, 10, 11, 13–15, 18, 20, 22, 23, 25, 33, 34, 36–39, 41, 42

IMP IDE Meta-tooling Platform. 38

JSON JavaScript Object Notation. 3, 11, 13, 15, 18, 23, 38, 39, 41

REPL Read Eval Print Loop. 41

Vim Vi Improved. 16, 38

Glossary

Eclipse Open Source IDE for multiple languages. 5, 18, 20, 22, 23, 38, 39

Java Popular object-oriented programming language. 11, 18, 26, 33, 41

JavaScript Dynamic Programming language, most commonly used as part of web browsers. 11

JSON Schema A validation schema for JSON. 15, 16, 18, 20, 22, 24

MessagePack A compressed version of JSON. 38

Monto A Disintiraged Development Environment. 2, 3, 6–8, 10–13, 15, 26, 27, 33, 37–42

NeoVim Fork of the Vim project. 38, 39

plug-in piece of code that can be added to a system. 5, 6, 8, 39

Scala Functional / Object-Oriented language for the JVM. 5, 6

SDF Domain-Specific language for formulating grammars. 38

Spoofax Platform for developing textual domain-specific languages with full-featured Eclipse editor plugins. 38

srclib A polyglot code analysis library. 39

Stratego Domain-Specific language for program transformation. 38

Vim-Script Scripting language to write Vim plugins. 38

ZeroMQ Open Source messaging library. 3, 11–14

References

[1] *Comparison of integrated development environments*. URL: http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments.

[2] *Docker, an open platform for distributed applications*. URL: https://www.docker.com/.

[3] *Eclipse IDE Meta-tooling Platform*. URL: http://eclipse.org/proposals/imp/.

[4] Sebastian Erdweg and Felix Rieger. “A framework for extensible languages”. In: *ACM SIGPLAN Notices* 49.3 (2014), pp. 3–12.

[5] Sebastian Erdweg et al. “Growing a language environment with editor libraries”. In: *ACM SIGPLAN Notices*. Vol. 47. 3. ACM. 2011, pp. 167–176.

[6] Sebastian Erdweg et al. “The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge”. In: *Software Language Engineering - 6th International Conference, SLE 2013*. 2013, pp. 197–217.

[7] *JSON Schema*. URL: http://json-schema.org/.

[8] Lennart CL Kats and Eelco Visser. “The spoofax language workbench: rules for declarative specification of languages and IDEs”. In: *ACM Sigplan Notices*. Vol. 45. 10. ACM. 2010, pp. 444–463.

[9] *Language workbenches: The killer-app for domain specific languages?* 2005. URL: http://martinfowler.com/articles/languageWorkbench.html.

[10] *List of text editors*. URL: http://en.wikipedia.org/wiki/List_of_text_editors.

[11] *LZ4, a fast lossless compression algorithm*. URL: https://code.google.com/p/lz4/.

[12] *Monto Broker Additions*. URL: https://github.com/svenkeidel/monto-broker.

[13] *Monto front-end as a plugin for Eclipse*. URL: https://github.com/svenkeidel/eclipse-monto.

[14] *Monto front-end as a plugin for Sublime Text 3 Editor*. URL: https://bitbucket.org/inkytonik/sublimemonto.

[15] Bram Moolenaar. *Vim, token categories*. URL: http://vimdoc.sourceforge.net/htmldoc/syntax.html#group-name.

[16] *NeoVim, a fork of the Vim project*. URL: https://github.com/neovim/neovim.

[17] *NeoVim plugin example*. URL: https://github.com/neovim/neovim/blob/master/runtime/doc/remote_plugin.

[18] Martin Richards. “The BCPL Cintsys and Cintpos User Guide”. In: (2015).

[19] Martin Richards. “The portability of the BCPL compiler”. In: *Software: Practice and Experience* (1971).

[20] Martin Richards and Colin Whitby-Strevens. *BCPL: The language and its compiler*. Cambridge University Press, 1984, p. 124.

[21] *srclib, a polyglot code analysis library*. URL: `https://srclib.org/`.

[22] Scott Buckley Tony Sloane Matt Roberts and Shaun Muscat. *Monto*. URL: `https://bitbucket.org/inkytonik/monto/`.

[23] Scott Buckley Tony Sloane Matt Roberts and Shaun Muscat. “Monto: A Disintegrated Development Environment”. In: *Software Language Engineering* (2014), pp. 211–220.