An Approach to Resilient System Design using Railway-Oriented Programming

Structure:

This paper is structured into 6 chapters, as follows:

Chapter 1: Introduction, this chapter serves to describe the motivation behind the paper.

Chapter 2: Metaprogramming, is intended in order to summarily describe the field of Metaprogramming in order to give better context to this paper.

Chapter 3: User Guide is used to shape the design and the capabilities of the presented system, with the purpose of explaining the exposed features of the application, giving the reader the necessary information to practically integrate the application.

Chapter 4: Implementation details contains in-depth information of how the features are implemented, and the motivation behind their implementation.

Chapter 5: The final chapter outlines the limitations of the current system and the extension points for further development.

Abstract

1. Introduction
   1. Motivation and Purpose
   2. Structure
   3. Personal Contributions
2. Metaprogramming
   1. Code Generation
   2. Compiler Generation
3. User Guide
   1. Concepts
   2. Features
   3. Usage
4. Implementation Details
   1. Technologies
   2. Structure
   3. Feature implementations
5. Conclusions, Future Work

References:

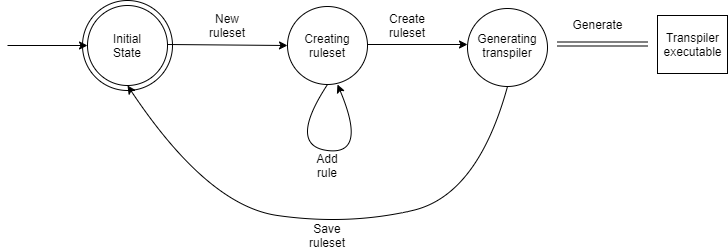
1. <https://en.wikipedia.org/wiki/Compiler-compiler>
2. Metaprogramming in .NET by Kevin Hazzard and Jason Bock - https://www.manning.com/books/metaprogramming-in-dot-net
3. https://levelup.gitconnected.com/four-ways-to-generate-code-in-c-including-source-generators-in-net-5-9e6817db425
4. <https://dotnet.microsoft.com/apps/aspnet/apis>
5. https://github.com/antlr/antlr4/blob/master/doc/index.md
6. <https://fullboarllc.com/using-antlr-4-with-net-core-2-1-and-c-getting-started/>
7. <https://www.willowtreeapps.com/craft/an-introduction-to-language-lexing-and-parsing-with-antlr>
8. <https://channel9.msdn.com/Events/dotnetConf/2020/Create-a-Text-Parser-in-C-with-ANTLR>
9. Source-to-Source Translation and Software Engineering by David A. Plaisted - <https://www.scirp.org/html/5-9301620_30425.htm>
10. SML2Java: A Source to Source Translator - <http://www.litech.org/~vaughan/pdf/dpcool2003.pdf>
11. <https://www.geeksforgeeks.org/source-to-source-compiler/>
12. <https://humbletoolsmith.com/2018/04/05/t4-csharp_s-little-known-code-generator/>
13. Understanding Typescript - <https://www.semanticscholar.org/paper/Understanding-TypeScript-Bierman-Abadi/1469b0cbb109c2a788a346dd0480070de8334dea>
14. Universal-transpiler - <https://jarble.github.io/transpiler/>
15. JetBrains MPS - <https://en.wikipedia.org/wiki/JetBrains_MPS>

* Application lifecycle

The application is composed of two parts:

* A REST API exposing an endpoint for generating a transpiler, referred to as Generator
* A web application which consumes the REST API and exposes an interactive user interface for creating a transpiler, referred to as Frontend
* A runnable application generated on-demand by the Generator which can then be ran from the command line to transpiler an input to an output, referred to as Executable

The life cycle of the applications:

* The Generator is intended to be a functional API, it is stateless, and can be used independently of the Frontend. The Generator has only two easily perceivable states as a web application:
  + Disabled
  + Running
* The Frontend is a web app, partly used to create rulesets used by the Generator, and partly to use those rulesets to consume the Generator. The diagram below depicts the flow of the frontend application. It is stateful, storing sequentially added rules as a ‘ruleset’ used afterwards.
* The executable is designed to be a functional, stateless API, only receiving an input text and producing an output text.
* Feature list
* Generator:
  + REST API endpoint, with input a ruleset and outputting an executable transpiler as a file
  + Consistency check for input Ruleset
  + Error reporting in case of inconsistencies/incompatibility
* Frontend:
  + Web app accessible from the browser
  + Presentation page outlining the purpose of the site
  + Rulesets UX
    - Ruleset entity CRUD
    - Rule subentity CRUD
    - Persistence for rulesets
  + Transpiler Generation
    - Consuming of Generator REST API
    - Web app caching and automatic use of Transpiler for in-browser transpilation
* Executable:
  + Command line execution
  + Functionality to use files for input/output
  + Input test parsing error reporting
* Original contribution

The presented application offers an original contribution through its main feature:

Creating on-demand, customizable text transpilers that are easy to use and integrate with other tools.

This is unique due to the fact that current ‘compiler generation’ frameworks/systems have several limitations, ranging from:

* Lack of customizability
* Can only be used in a certain, constraining context, such as only being usable through the presence of the main codebase of the tool
* Are not designed for integrating with existing programming language development flows
* Very restricting use case, usually by restricting the pool of source/output input
* Cannot be generated on-demand, rather only statically
* Require grasp of advanced concepts to be used efficiently

This application aims to solve or bypass all those limitations by providing a simple to use, simple to use in practice API for on-demand text transpilation.

Lab 5 – 8

1. Technologies
2. Feature subset - Reproductibility conditions
3. Comparison with similar applications – Deployment plan
4. Soure code

https://github.com/Horatzio/LangBuilder