Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

An Extendable Multi Programming Paradigm Code Generator for B

Masterarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von
Dominik Brandt

Beginn der Arbeit:	10. Januar 2023
Abgabe der Arbeit:	10. Juli 2023
Erstgutachter:	Prof. Dr. Michael Leuschel
Zweitgutachter:	Dr. Jens Bendisposto

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 10. Juli 2023

[illegible]

Dominik Brandt

Abstract

This thesis introduces `MPPCG`, a code generator currently capable of generating code of the B Method to Java and Prolog code. It is implemented with extendability in mind, providing the foundation to support multiple different input languages, as well as output languages of different programming paradigms.

MPPCG was inspired by B2Program, a code generator for the B Method targeting various different languages. However, B2Program faces restrictions when targeting programming languages of other, non-imperative programming paradigms, especially when targeting Prolog.

MPPCG can successfully generate Java and Prolog code by using a combination of a template-based code generation approach and intermediate code generation. This approach enables the separation of the output languages, provides improved extendability and maintainability, but requires a certain level of redundancy across the output languages. For Prolog, MPPCG implements optimizations during the generation, so that fewer lines of code will be generated. It does currently not generate model checking code, but supports the `XTL` interface for Prolog. Hence, `ProB` can execute and model check the generated Prolog code. The presented code generator currently supports less target languages than B2Program, and does not implement features of B2Program like model checking or animation. However, it gives some ideas of improvement for B2Program, to support even more output languages.

This thesis discusses the implementation and the components of MPPCG and how to use and extend it. It also compares benchmarks of the model checking performance of ProB executing machine files, ProB executing the generated Prolog code, and the generated Java model checking code of B2Program.

Acknowledgements

I would like to extends my thanks to Fabian Vu, for the valuable support and guidance provided during both my project work and Master's thesis. I am grateful for his helpful tips whenever I encountered difficulties. With his expertise and input, I was able to enhance the quality of my work. Thank you.

Contents

1 Introduction and Motivation

Code generation is an important procedure in software development. Its purpose is the generation of source code, or even executable code, from a given model, or given input code. There are many existing translating code generators, translating code of one language/model, to code of another language. One such code generator is `B2Program`[46, 44], which translates models of the `B-Method`[2] to various other (target) programming languages. So far, B2Program's supported target languages are Java, C++, Python, JavaScript/Typescript, and Rust. However, these target languages are mostly imperative programming languages[47].

Programming languages differ in their syntax and their programming paradigms. Each programming paradigm has its own characteristics, purposes, advantages, and disadvantages, depending on the problem to solve. Thus, the same program across different programming languages and paradigms can vary much.

B2Program shows, that code can be generated efficiently to target languages of the same programming paradigm, without many changes to the generator when adding new target languages. When B2Program aimed at support for Prolog, which is a logic and declarative programming language, some design choices contradicted with Prolog's programming paradigm. Code generation for Prolog is in general not as straight forward as for imperative programming languages, due to its remarkable different program structure. However, the problems during implementation led to the assumption that, when adding a target language of another programming paradigm, B2Program might not provide the best approach.

In this thesis, I implemented a code generator which generates Java and Prolog code from given B models. The question that arose was, if it is possible to design a code generator in a way that different programming paradigms can be handled. One reason for B2Program to support Prolog was to compare the performance of the generated Prolog code with `ProB`[33, 34]. ProB is model checker for B, which uses a Prolog interpreter for the B models. Hence, in this thesis I focused mainly on Prolog support, to make the desired comparison. Meanwhile, this code generator needs to be as simple as possible, but as powerful as needed, in order to keep the level of complexity regardless of the amount of implemented languages. While B2Program is restricted to the B-Method as input language, it is also interesting to see if the code generator can not only swap the output languages, but also the input languages. However, this is not examined in this thesis, as only one input language is supported, but the design choices I made prepared the code generator for this procedure in the future.

This multi programming paradigm code generator, referenced to as `MPPCG`[1], is required to fulfill three main aspects:

1. Extendability
2. Maintainability / Readability
3. Versatility

MPPCG should grant the best possible **extendability** when adding or modifying input languages, parser generators, and output languages. **Maintainability** has also a high priority in terms of readability and complexity. MPPCG is required to be as powerful as possible, while being easy to **maintain** because of simple data structures and simple program structures. Lastly, **versatility** is one of the core functionalities, as one objective is to enable support for many different input language as well as many different output languages.

When developing the code generator, it needs to be designed in a way that the main generation part is independent of the code generator's common code. This means, that each language is responsible for its output and the languages are independent of each other. If this is possible, it needs to be discussed whether this approach of code generation is simpler in terms of extendability and maintainability, compared to other approaches, like in B2Program. The results can not only be used as an alternative to B2Program, but also as an inspiration for improvements of B2Program, to support more target languages. Another problem to discuss is the potential redundancy between the generator code of output languages.

If it is possible to design such a code generator, with regard to the mentioned goals, then it could develop over time to a versatile translator, usable for translating code between various languages.

Chapter 2 gives a brief overview over the B-Method, programming paradigms, model checking, and more background knowledge. In chapter 3, B2Program is described in further detail, and the problems with other programming paradigms than the implemented ones are discussed. Design decisions are addressed in chapter 4, and in chapter 5, the implementation and components of MPPCG are described. Afterward, we will have a look at the implemented languages in chapter 6, as well as problems during the implementation, and a comparison with B2Program. In chapter 7, we compare the model checking performance of the generated Prolog code, executed with ProB, to the model checking performance of ProB and the model checking performance of the generated Java code of B2Program. Chapter 8 presents other code generators and related work, and finally, in chapter 9 the foregoing questions, as well as future work, are discussed.

[1]Available at: https://github.com/dobra101/mppcg

2 Background

This chapter gives an overview and basic understanding of the main topics of the implemented code generator. Since MPPCG aims to support different programming paradigms, programming paradigms in general are explained in section 2.1. Problems during the work with B2Program [46] have been the main motivation for a multi programming paradigm code generator. Since B2Program's input language is the B Method, MPPCG implements it as well and it is described in section 2.2. This is also necessary, as the B Method is implemented as a first input language of MPPCG, to grant a better starting point for the comparison with B2Program. B2Program supports model checking. Model checking and the model checker `ProB`[34, 33] are mentioned in section 2.3, as MPPCG aims to generate model checkable output code, which can be executed by ProB. Further, a general understanding of the concepts of code generation is required to fully understand the design of MPPCG and B2Program. These required information are given in section 2.4. Then, as most of the general concepts required to understand B2Program are given, B2Program is introduced in section 2.5 to prepare the foundation for the comparison of B2Program and MPPCG later in this thesis. Many programming languages are based on a type system, but when generating type sensitive code and no explicit types are provided, these types need to be inferred. Section 2.6 introduces the type inference algorithm by Hindley and Milner [24, 11], which is used in the implemented code generator. Partial evaluation can be used to optimize the execution of Prolog programs. Since the generated Prolog code is related to partial evaluation, it is introduced in section 2.7.

2.1 Programming Paradigms

Programming paradigms are principles, concepts, and approaches defining the programs structure and design. Each programming paradigm has a different use case, advantages, and disadvantages. The most common programming paradigms are listed in this section.

Object-Oriented programming. Object-oriented programming [37], or short OOP, encapsulates the code into classes representing objects, hence the name. These classes can contain data and determine how to interact with it. It includes concepts like inheritance to inherit data and behavior from super classes. Many popular programming languages are object-oriented, for example Java and Python.

Functional Programming. In functional programming languages [27], computations are treated as the evaluation of mathematical functions. Mutable data and states are avoided, and higher-order functions, i.e. functions can be passed as parameters, and immutability are emphasized. Popular functional programming languages are Haskell and Lisp.

Declarative Programming. Declarative programming means, that the problems are described focusing on *what* to achieve, instead of *how* to achieve it [47]. Query languages like SQL and logic programming languages like Prolog are declarative programming languages.

Imperative Programming. In contrast to declarative programming, imperative programming focuses on *how* to solve problems [47]. Therefore, step-by-step instructions are provided. Variables, assignments, loops, and conditionals are some of the concepts of imperative programming. Imperative programming languages are for example C and Java.

Next to the presented paradigms, there are even more programming paradigms, not further stated here. As we have already seen with imperative and object-oriented programming, a language is not restricted to one singular paradigm, but instead, multiple paradigms can be combined. However, not all paradigms are easy to combine. For example, the concepts of imperative programming contradict the concepts of declarative programming. Hence, such contradictions have to be considered when designing a multi programming paradigm code generator.

2.2 B Method

The B Method[2] is a formal method focusing on the development of correct and reliable systems. Therefore, it uses the formal specification language, called the B language, where concepts from set theory and first-order predicate logic are combined. In the B language, systems are represented by models (or machines), which describe their behavior using mathematical constructs.

The base structure of a machine consists of variables, constants, sets, invariants, and operations. The models are initially described in a generalized way, with constraints on its properties, and invariants which must hold true. Invariants are predicates which are evaluated in each state, i.e. after each operation, and their outcome determines the correctness of the machine and its behavior after executing each operation.

Once the base model is designed, it is refined in several refinement steps. In a refinement step, the machine gets progressively more details and constraints, specifying the exact behavior. Thus, the initial and more general machine becomes a precise, correct, and reliable machine during the refinement.

Once the machine is fully refined, code generation can be performed. There are multiple code generators capable of generating B models to code of various programming languages, like C and Java. One such code generator is `AtelierB`[10], which supports implementations written in `B0` (a subset of B). It also provides tools for specification and verification, ensuring a system's correctness. Like `ProB`, `AtelierB` supports the development of complex, safety-critical systems and helps the developer with modular development for an increased maintainability. Another code generator is, as mentioned, ProB[33, 34], which also supports other languages like TLA+, Z, and CSP-M.

The B Method is being used in various use cases, like railways, critical infrastructure, and transportation systems, and is an important tool for developing safety-critical systems. Example use cases are the automatic Paris Métro lines 1 and 14[18, 6].

Each machine also contains an initialization, assigning the initial values for the variables and creating the initial state. A machine changes between states with transitions defined by operations. An operation can be parameterized and can contain guards specifying whether the machine is in a valid state, such that the given operation can be executed.

Listing 1: Example B Machine of a Lift Controller

```
MACHINE Lift
VARIABLES level
INVARIANT level >= 0 & level <= 100
INITIALISATION level := 0
OPERATIONS
    inc = PRE level < 100 THEN level := level + 1 END;
    dec = PRE level >   0 THEN level := level - 1 END
END
```

An example state machine is shown in listing 1. It contains a variable `level`, which holds the value of the current level. The invariants determine, that the state is only valid and safe, as long as the `level` is between 0 and 100, including the borders. It is initialized at `level = 0` and two operations are defined: `inc` and `dec`, which increase the `level` if the guard `level < 100` holds, and decrease the `level` respectively.

In a refinement step, for example a call button for each level could be introduced, to call the lift. This enables, that the level does not change arbitrarily, but approaches the level of the activated call button, as a real lift would do. Then, in a next refinement step, buttons inside the lift could be introduced, as well as the behavior, that the lift tries to approach the level of the activated button inside (we call this level `L1` for now). Furthermore, the lift is required to stop only at levels between the current level and `L1`, if a call button is activated. We can see, that the refinement steps lead to a lift behaving as we know it from the real world, by procedurally adding more information to the machine.

2.3 Model Checking

Model checking is a verification technique to ensure a system's correctness. This is done by an analysis of the entire state space, which is a set of all possible states a system can have. In general, a system's state is represented by a model containing the system's assigned variable values. When analyzing the state space, each state is checked against a given set of properties (or invariants in the B Method) and is said to be valid, if all properties hold true and no errors or other violations occur. It is also checked for deadlocks, where a state has no enabled transitions. Models of the B Method define the possible transitions, guards, invariants, and other constraints or properties representing a system's state. When model checking, the state space is generated first by exploring all possible states, i.e. by exploring all possible transitions in every state. During the exploration, model checkers can evaluate the invariants and deadlock freedom.

Model checking is popular in development of safety-critical systems and verification of hardware. Due to the exhaustive analysis of all possible states a system can have, the system can be proven correct within the defined properties and invariants. However, model checking also has some disadvantages, especially when facing the state space explosion problem[9]. This is, that as for every state successor states are calculated, the amount of states might grow exponentially. Consequently, it is challenging to maintain the storage space, while simultaneously facing an exponential increase in analysis time. For such problems, various optimization strategies have been developed to reduce the state space. One strategy is *Symbolic Representation*, where instead of explicitly enumerating every possible state, sets of states are represented compactly using a symbolic representation. Another optimization strategy is *Partial Order Reduction*[20], where independencies between actions are exploited by pruning equivalent or redundant states. Hence, only relevant interleavings of actions are considered.

Model Checking Example. When model checking the machine of listing 1, each state is traversed. There are 101 states, as there are exactly 101 possible variable assignments, and each value of `level` between 0 and 100 can be reached through the operations. Since the operation's guards prevent `level` from exceeding the bounds of 0 and 100, the invariants hold true in each state. Hence, the machine is correct.

ProB[33, 34] is a sophisticated model checker used for system analysis and verification. In ProB systems can be modeled, animated, and verified using the B Method[2], but also using other high-level specification formalisms like Z, CSP, Promela, and more. Through its graphical interface, ProB offers an intuitive environment where B models can be executed interactively, allowing users to observe and analyze the behavior of the system. This graphical interface greatly enhances the understanding and validation of the modeled systems and supports the user during development.

One feature of ProB is its support of temporal logic formulas. *Temporal logic formulas* are time related propositions. That is, that terms require a condition for example to be *always* true, or to be *eventually* satisfied. These formulas enable the specification of safety and liveness properties, to check properties which are required to hold true at all times (in every state). By checking these properties against the B model, ProB helps to ensure the correctness and reliability of the system. This is particularly valuable in safety-critical systems.

ProB has also a consistency checking mechanism that performs static analysis of the model's specifications. This helps to identify potential errors, such as incomplete definitions, inconsistencies, or conflicting constraints, and provides guidance during the refinement of the model. By this guidance, ProB increases the efficiency of the development, and decreases the likelihood of introducing new critical errors.

Another aspect of ProB is its ability to generate executable code for target languages like Java, C, and Ada. With that feature, ProB facilitates the process of integration of formal methods with software development.

Next to the model development features of ProB, it can model check these models and implements various model checking optimizations like *Partial Order Reduction*. However, models written in Prolog can also be model checked if they provide predicates for the XTL interface. The XTL, short term for executable temporal logic, interface is utilized by providing the relevant information for model checking for ProB. This is done by providing three components in the Prolog code: *start*, *trans*, and *prop*.

start. This predicate is used to calculate and provide the initial state of the system.

trans. This component represents the transitions of the system. It describes the guards and conditions to enable a transition, and the following operations on the system to progress from one state to another. In general, this component represents the operations of a model of the B Method.

prop. The prop component provides the temporal properties that need to hold true during execution. Typically, temporal logic is used to express these properties. The contents of this component are among other things the invariants of the model. ProB checks, if *prop* returns *unsafe* in order to determine if a system is in a safe state.

2.4 Code Generation

Code generation has a significant role in software development. It is the process of producing source code (or executable code) from a different representation [4]. Therefore, a higher-level representation is transformed to code in several sequential steps, also seen in figure 1 and [44]. Starting with the *lexical analysis* step, the relevant information about the structure of the code is collected and extracted. In this step, a lexer splits the code into token and passes them to the parser.

In the *syntactical analysis*, the parser takes these token and checks, if they appear in the correct order. Hence, the parser checks the syntax of the program and generates an abstract syntax tree (AST), using the received token. The *lexical* and the *syntactical* analysis are typically connected, as the parser expects the exact token of the lexer.

The next step is the *semantic analysis*, where the AST of the parser is processed to verify the input's semantic correctness. This is, for example, where type checking is applied and scoping of variables is evaluated and checked.

After the analysis steps, the AST is passed to the code generator which transforms the input code into

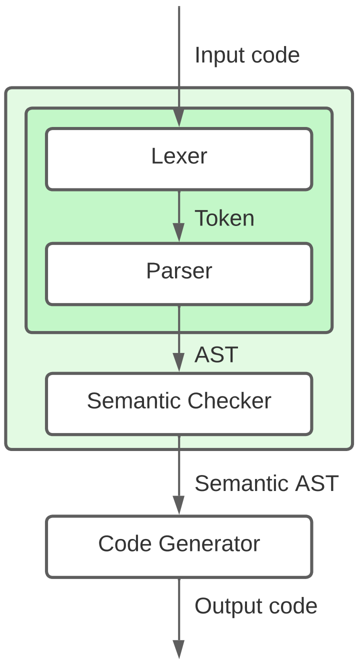

Figure 1: Input Code to Output Code [4]

the output code's representation. The code generator can also apply optimizations, like dead code elimination as well as optimizations in terms of performance. Finally, the output code's representation is rendered as source code or executable code. Section 5.3, provides a more in-depth explanation of the mentioned steps and how they are implemented in MPPCG.

While the code generator can preprocess the AST, the input code can also be preprocessed before it is passed to the lexer. MPPCG generates code with input code of the B Method. As stated in section 2.2 and section 2.3, the B models are typically model checked and verified before code generation. This guarantees, that if the steps of code generation are implemented correctly, the output code is verified and working as expected. However, generated code needs to be tested and verified carefully, as many bugs can be introduced unknowingly to the generated program. For example, B2Program and MPPCG implement BTypes and their methods. While the generated code might be correct, these implementation might contain errors, such that the program contains unexpected behavior. To validate the output's correctness, common practice in industry is to use multiple different code generators, developed by different teams with different techniques. These different code generators can then be used to validate the *main* code generator.

Code generation is used in various areas of software development. Compilers generate bytecode or machine code from source code of their high-level input programming language. Other use cases are software development tools which generate repetitive code patterns to reduce the development times for common tasks. Lastly, it is used for example in template-based code generation, where predefined code snippets with placeholders are filled with values, like B2Program[46] uses it. However, generated code has also some limitations, for example in terms of readability, where not meaningful variable names might be generated.

2.5 B2Program

B2Program[46] is a code generator able to generate code from to B Method to various output languages. It is capable of generating model checking code[44] supporting explicit-state model checking, verification of invariants, and deadlock checks. However, as of now, LTL and symbolic model checking are not supported. The model checking performance has already been evaluated and compared with ProB and `TLC` in [44]. It was concluded, that B2Program generates model checking code which can model check faster than ProB and `TLC` in many cases. Nevertheless, ProB performs much better on other machines, due to its constraint solving capabilities and operation caching techniques[31, 44]. B2Program does also provide visualizations, where it generates an interactive HTML validation document. In this document, operations, states, scenarios, and the history of operations can be observed.

Currently, its supported target languages are Java, C++, Python, JavaScript/Typescript, and Rust. However, those output languages are mainly imperative programming languages. The design is carefully chosen and works well with the supported languages, but implementing support for additional output languages with other programming paradigms comes in hand with drawbacks and limitations.

After a B machine was refined, an *implementable* subset of B, often referred to as *B0*, is reached. There are multiple code generators targeting programming languages such as C or Ada, which use *B0* as input language. Examples are the code generators in `AtelierB`[10], `B2LLVM`[7], which emits the LLVM intermediate representation, or `jBTools`[43] which generates low-level Java code. B2Program, however, targets drawbacks and limitations of such existing code generators. It can generate B code to multiple different output languages, and supports higher-level constructs, resulting in a larger supported subset of B, compared to *B0*. To achieve this, B2Program uses a templated-based approach for the generation steps to achieve also some software engineering principles such as *generic programming* [5] and *don't repeat yourself* [42].

This templated-based approach works by providing templates for each output language, i.e. templates for the various code constructs for each output language. Each template file for each target language consists of multiple templates, with mainly uniform names and parameters across the different target languages. Then, after the B machine was parsed, the parsed syntax tree is traversed and for each node the parameters for the templates are evaluated and rendered recursively. For this recursive rendering, the visitor pattern[17] is used. This design pattern enables efficient processing of data structures like abstract syntax trees by separating the data structure and the operations. The visitors are classes containing operations which can be implemented for each class of the data structure. B2Program simplifies the implementation of additional output languages. If no additional information are required, it is sufficient to implement the new templates, and since the template- and parameter names are identical, the visitor classes can handle the code generation. When B2Program produces the correct output code, the B types have to be implemented in the new target language. B2Program stated for such implementation cases an example (see [46] section 3.3) where additional type information are required in the new target language. This example is again evaluated in section 6.5 and compared to MPPCG.

2.6 Hindley-Milner Type Inference

This chapter describes the type inference by J. Roger Hindley and Robin Milner[24, 11] as MPPCG implements this algorithm. Functional programming languages and languages with functional features, like Java, rely on the use of type systems to ensure type safety and correctness. Types are assigned for example to variables and expressions and define the behavior and possible operations. Hindley-Milner's type inference algorithm, also known as Damas-Milner[2], is widely used for inferring types in functional programming languages. It is a unification based type inference algorithm, where a set of equations is created, associating variables and expressions with variable types. Finally, the equations are solved to determine the correct types.

[2]Luis Damas proved the correctness and completeness of the algorithm in [11]

2.6.1 Example Language of the Algorithm

In the proof of the algorithm (see [11]), a simple language *Exp* of expressions `e` is created, assuming a set *Id* of identifiers `x`. This language's syntax can be seen in listing 2.

Listing 2: Language *Exp* in [11]

```
e ::= x | e e’ | λ x.e | let x = e in e’
```

2.6.2 Type Inference

The core type inference algorithm is `Algorithm W`, which is used to infer principal types for expressions. Principal types are the most general types for given expressions, and the goal of the algorithm is, to assign these principal types to each expression. Pseudocode for `Algorithm W` is listed in listing 3.

Listing 3: Pseudocode for `Algorithm W` in [11] Rewritten in Kotlin

```
fun inferType(expression: Expression): Type {
    when (expression) {
        is Variable -> return newTypeVariable()
        is Application -> {
            val t1: Type = inferType(expression.e)
            val t2: Type = inferType(expression.e1)
            val t3: Type = newTypeVariable()
            unify(t1, FunctionType(t2, t3))
            return t3
        }
        is LambdaAbstraction -> {
            val t1: Type = newTypeVariable()
            environment[expression.x] = t1
            val t2: Type = inferType(expression.e)
            return FunctionType(t1, t2)
        }
        is LetExpression -> {
            val t1: Type = inferType(expression.e)
            environment[expression.x] = t1
            return inferType(expression.e1)
        }
        else -> {
            // other language constructs
        }
    }
}
```

In the algorithm, the syntax tree of an expression is recursively traversed. First, the algorithm creates relations between variables and expressions of the program. This means, type variables are created and stored in an environment. The environment is basically a map containing the types for expressions, which might depend on other types, which are not evaluated yet. This can be seen for example in line 12 and line 13 of listing 3, where first a new type variable is created, and stored in the environment for `x`. In this case, `x` is the lambda variable. It can be seen that for untyped lambda abstractions (e.g. `x -> someFunction(x)`) the type of their variables depend on their usage on the right side of the abstraction. Hence, the environment stores a new type variable for expression `x`. Continuing with the example of the lambda abstraction, the algorithm is called recursively on it's remaining expressions (in this case, this is the right side of the abstraction: expression `e`). As the environment contains the type variable of `x`, the recursive call on `e` can access this type variable, and unifies it with a value.

2.6.3 Unification

During unification the algorithm tries to assign concrete values to type variables. Therefore, the algorithm defines the syntax of types τ and type-schemes σ which are listed in listing 4. Type-schemes are mainly used to introduce polymorphism in a programming language. Hence, type-schemes are for the foregoing example not relevant. Type variables are represented by α and primitive types by ι.

Listing 4: Syntax of Types and Type-Schemes in [24, 11]

$$\tau \ ::= \ \alpha \ | \ \iota \ | \ \tau \rightarrow \tau$$
$$\sigma \ ::= \ \tau \ | \ \forall \alpha \sigma$$

The unification is done by replacing type variables with concrete values. In listing 3 unification is also used for *Applications* (line 8). In this case, *Applications* are function calls.

Example. Let us assume, that the current expression is `functionA(3)`. In line 4–10, type `t3` of the expression `functionA(3)` is returned. `functionA(3)` consists of the components `functionA (expression.e)` and `3 (expression.e1)`. Since `functionA` is a function, it takes in this case one parameter of type τ_a, and returns a value of type τ_b. Hence, the type of `functionA` is `FunctionType(`τ_a`, `τ_b`)`. τ_a and τ_b are known after line 5, `t2` is known after line 6. So, line 8 would equal `unify(FunctionType(`τ_a`, `τ_b`), FunctionType(t2, t3))`. This means, that τ_a would be unified with `t2` and τ_b with `t3` respectively. If we assume now, that τ_a and `t2` are still type variables (α_1) and τ_b is a primitive type ι_1, the type variable `t3` of line 7 would get the primitive value ι_1 assigned. Without knowing the primitive type of α_1, the method returns a primitive value, which can then be used for further unifications.

2.7 Partial Evaluation

In Prolog programming, partial evaluation is a technique to optimize the execution of the program. Therefore, parts of the program are evaluated at compile time, rather than at runtime. This results in significant performance improvements. The optimization is done by producing new programs specialized to specific inputs or known values. Executing the specialized programs, redundant computations can be eliminated and branches can be resolved based on known information. Further, partial evaluation can reduce the search space, as more specific information to the Prolog engine enable faster resolution of queries.

Partial evaluation is not a built-in feature of Prolog, but can be implemented using specialized tools. However, generating Prolog code is related to partial evaluation, as certain optimizations can be generated. For example, the generated Prolog code can be specialized regarding the XTL interface mentioned in section 2.3.

3 Extending B2Program vs. Developing a new Generator

This chapter gives a more detailed view on B2Program[46], and we discuss the potential of extending B2Program to support other programming paradigms. Arising problems implementing programming languages with other programming paradigms, i.e. with non-imperative programming languages, are explained and the main motivation of creating a multi programming paradigm code generator is shown in the following.

During the implementation of support for Prolog in B2Program, we encountered that certain design choices of B2Program contradicted the programming paradigm of Prolog. This is, because Prolog is a declarative and logic programming language. Although B2Program is a powerful code generator supporting multiple languages and features, it does have some drawbacks. Specifically, the extendability, or more precise, the limited extendability of non-imperative and non-object-oriented programming languages. In Java, expressions and predicates can be passed as method parameters. Furthermore, you can chain and nest expressions and use language constructs like if-else, while- or for-loops. These constructs and nested expressions are well-supported in B2Program, such that one has only to provide the correct `StringTemplate`[3] templates to support other languages which contain the same constructs. But having a look at Prolog, a declarative and logic programming language, these constructs are not supported. Due to the syntax and structure of Prolog programs, it is required to evaluate expressions and predicates one after another.

Listing 5: Nested Expressions in B

```
max(dom({0 |-> 1}))
```

Listing 5 shows nested expressions in B, where the maximum value of the domain of a

[3]https://www.stringtemplate.org/

relation is calculated. The components of the expressions are colored, as this will indicate the problem B2Program has with generating Prolog code. In the following we will see how B2Program handles such expressions, and what problems occur when trying to handle such expressions for Prolog. Note, that the same expressions could be represented in a different way in Java, but B2Program utilizes its BTypes.

B2Program evaluates such expressions by starting from the innermost expression, and inserting the rendered code snippet into the next (outer) expression. Hence, B2Program would generate the Java code shown in listing 6.

Listing 6: Listing 5 Rendered to Java Code by B2Program

```
new BRelation<BInteger, BInteger>(
    new BTuple<>(new BInteger(0), new BInteger(1))
).domain().max()
```

This evaluation order is determined by B2Program, depends on the underlying abstract syntax tree, and is the same for every target language. In this example, to support another target language it is sufficient to add the templates for `tuple`, `relation`, and `unary expression`.

The same nested expression calls rewritten in Prolog can be seen in listing 7.

Listing 7: Listing 5 Rewritten in Prolog

```
1: % max(dom([0/1], Expr_0), Expr_1)
2:
3: domain([0/1], Expr_0),
4: max(Expr_0, Expr_1)
```

In line 1 of listing 7, we can see what B2Program would generate if the templates for Prolog are provided. However, this is not a valid statement in Prolog. The correct Prolog code is given in line 3 and line 4.

The evaluation order is, again, from the inside to the outside, and even if it looks straightforward, this is not possible in B2Program. We can observe the problem B2Program currently has, just by looking at the colors of the code components: B2Program returns a colored component as a plain string for a given input code, which is valid for Java and other imperative programming languages. The color of an outer node / expression can wrap around the color of an inner node. However, the color of an inner node can not occur multiple times inside the color of an outer node.

In the previous example, the outer node is the maximum-call, and the first inner node is the domain-call. We can see in listing 7, that the color of the domain-call occurs before the maximum-call (line 3), and inside the maximum-call (line 4). Currently, B2Program can not handle this problem, as rendering a node returns just a string. To enable the same color of a component to occur multiple times in the component of its outer node, the color, and hence the returned component has to be split. However, splitting the returned string to receive the relevant information is not trivial. Thus, a rendered (colored) component should not be returned as a whole string, but as a class that can contain more information than just the string.

For a better understanding, the responsible parts of B2Program's implementation for generating unary expressions are shown in listing 8 (template) and listing 9 (implementation).

Listing 8: B2Program's Java Template for Unary Expressions

```
unary(operator, obj, args) ::= <<
<obj>.<operator>(<args; separator=", ">)
>>
```

The Prolog template for unary expressions would need an additional *before*-parameter. This parameter needs to contain the code of line 3 in listing 7. Hence, this parameter contains the code to execute *before* the unary expression and which returns a variable. The returned variable holds the result of the previous executed expression and can then be used in the outer expression (the maximum-call).

Listing 9: B2Program's Implementation to Generate Unary Expressions

```
 1: private String generateUnaryExpression(
 2:     ExpressionOperatorNode.ExpressionOperator operator,
 3:     List<String> expressionList
 4: ) {
 5:     ST expression = generateUnary(operator);
 6:     TemplateHandler.add(expression, "obj", expressionList.get(0));
 7:     TemplateHandler.add(
 8:         expression,
 9:         "args",
10:         expressionList.subList(1, expressionList.size())
11:     );
12:     return expression.render();
13: }
```

However, B2Program fills the templates as shown in listing 9. As already mentioned in section 2.5, the template files for the different target languages have mostly the same template parameters. This is, because the same generator code passes the parameters to templates of different target languages. When targeting languages of the same programming paradigm, this is as simple as it can be, and can be seen in line 6 and lines 7–11 in listing 9. Hence, it is sufficient to change the template files in order to generate code for multiple target languages with the same generator code implementation. Meanwhile, this is the main problem when targeting languages of other programming paradigms. The resulting rendered string of `generateUnaryExpression` is passed directly into the next, outer rendering method. Having a look at line 3 and line 4 of listing 7 shows, that Prolog needs additional information. This information is on the one hand the variable name of the variable which contains the result of the previous expression. In this example, this is

`Expr_0`. And on the other hand Prolog needs the next expression count (this could be a 1 in this case), to identify the variable which stores the next result. In this example, this variable is `Expr_1`.

At this point, there is no possibility to extract only the relevant information of line 3 in listing 7 (`Expr_0`), since line 3 is just a string. Hence, without modifying the rendering methods, no valid Prolog code can be rendered.

Solving this problem is not trivial, since major refactoring would be required. Additional information need to be passed through multiple rendering methods, and each rendering method gets more complex with each additional parameter. This might be possible, however, over time B2Program would get hard to maintain. This limitation was the motivation to create a code generator with the ability to support different programming paradigms.

4 Design Decisions

This chapter describes some of the pre-development design decisions. B2Program is implemented in Java, however, Kotlin has some interesting features in terms of readability and maintainability. Thus, section 4.1 compares Java and Kotlin and explains why Kotlin was chosen for the implementation. Another decision was to select a template engine, which fulfills the needs and goals of MPPCG. This decision is described in section 4.2.

4.1 Kotlin vs. Java

Java. Java is an object-oriented programming language developed by `Sun Microsystems` (now owned by `Oracle` [4]). Java programs run on a Java virtual machine, or short JVM, and execute Java bytecode. It is one of the most popular programming languages and aims at goals like simplicity, security and robustness, and independence of the architecture [21]. Java follows an object-oriented programming paradigm (see section 2.1), meaning that programs contain objects interacting with each other. Therefore, the four main principles of object-oriented programming are supported: encapsulation, inheritance, polymorphism, and abstraction.

Kotlin. Kotlin is a statically-typed programming language developed by `JetBrains` [5] and was introduced in 2011. It runs on the JVM and also supports multiplatform projects[29], enabling code to be compiled to JavaScript or native code. Combining object-oriented and functional programming concepts, Kotlin became a versatile language and is for example used in Android development.

Kotlin was introduced as an alternative to Java, using the advantages of Java and fixing

[4]https://www.oracle.com/
[5]https://www.jetbrains.com/

some of Java's disadvantages and issues, and increasing developers productivity with an enhanced syntax. Further, Kotlin is interoperable with Java, such that Kotlin can interact with Java code and libraries. Thus, projects can be migrated step-by-step from Java to Kotlin. Due to its enhanced syntax and features like its Domain-specific language (DSL) support, code can be written compact and readable.

Kotlin has several additional language building blocks that Java does not have. One of these building blocks is crucial for MPPCG: extension functions. Extension functions enable the programmer to add new methods to existing classes. The code snippet in listing 10 adds the method `powerSet` to the class `Set<T>` and is, if available in the calling scope, present for every `Set` instance. In the same way, also operator functions can be added to existing classes, enabling operations using for example the `'+'`-operator on custom classes.

Listing 10: Extension Function for Sets

```kotlin
public fun Set<T>.powerSet() : Set<Set<T>> {
    // ...
}
```

MPPCG uses extension functions to add methods to nodes. These methods are only accessible inside each output language, enabling the separation of the output languages. Thus, every output language defines a `renderSelf` method for each node, such that all nodes can access only the `renderSelf` methods of the calling output language. Therefore, all output languages are separated from each other. More detailed information are described in section 5.5.

Kotlin addresses also some issues of Java [28]. Since Kotlin's type system has nullable types and non-null types, `NullPointerExceptions` only occur when explicitly trying to get a non-nullable value from a nullable type via the `!!`-operator. Null-safety does not only help avoiding unexpected behavior due to null-values, it also reduces the code complexity. A Java example is given in listing 11 and a Kotlin example respectively in listing 12.

Listing 11: Null-check in Java

```java
Set<Integer> canBeNull = getNullableSet();
if (canBeNull != null) {
    canBeNull.doSomething();
}
```

Listing 12: Null-check in Kotlin

```kotlin
val canBeNull: Set<Int>? = getNullableSet()
canBeNull?.doSomething()
```

In Java, one would need to check if `canBeNull` is not `null` and then call `doSomething`. Kotlin calls `doSomething` only if `canBeNull` is not `null`. This reduces not only the lines of code needed, but increases also the programs readability.

Most of MPPCG's nodes are `data classes`. These simplify the code again, an example is shown in listing 13.

Listing 13: Example Kotlin Data Class

```
data class Person(val name: String, var age: Int)
```

An instance of class `Person` is initialized for example by calling `Person("MyName", 30)`, and contains already methods like `toString` for pretty-printing the instance, `equals` and `hashCode`, as well as getter methods for `name` and `age`, and a setter method for `age`, as the value of `age` is not final.

Because of the listed features and simplifications of Kotlin over Java, and since a goal of MPPCG is the maintainability and readability, MPPCG was implemented in Kotlin.

Further differences and enhancements are listed at [28].

4.2 Template Engine

One important aspect of a code generator is the template engine. While there are several template engines for Java and Kotlin, only a few were shortlisted for MPPCG. Reasons for that are for example, that some template engines are designed especially for webpages written in HTML. Others are almost as complex as an own programming language, providing too much power and overcomplicating the use case. However, according to MPPCG's goals, a simple template engine without much logic is required and sufficient. The final choice for MPPCG was one of `Apache FreeMarker`[6], `StringTemplate`[7] or Kotlin's default templating structures.

Listing 14: Example String Template in Kotlin

```
val body = "System.out.println(x);"
val method = """
public String myMethod() {
    $body
}
"""

/* method would result in:
public String myMethod() {
    System.out.println(x);
    }
*/
```

The latter one was promising at first, but became rather unusable as different indentation levels would have to be maintained manually. On multiline strings containing variables, the

[6]https://freemarker.apache.org/
[7]https://www.stringtemplate.org/

indentation changes are applied at runtime and not at compile time. An example is given in listing 14, where the indentation of a multiline string (of the right brace in line 11) is not as expected. Thus, keeping track of the different indentation levels for each variable and each output language was against the goal of an easy to extend and easy to maintain code generator. Further, MPPCG would have needed functions building and returning these string templates, which in general is not as readable, as for example `StringTemplate`'s templates.

`Apache FreeMarker` is a Java template engine for generating text output, for example for web pages, e-mail templates, or source code. The templates itself are written in a simple language called FTL (FreeMarker Template Language). Programming languages like Java are used to fill the placeholders in the templates, and `Apache FreeMarker` then renders these templates.

`Apache FreeMarker` and `StringTemplate` both focus on model-view separation. This means, that the template engine focuses on the presentation of the data, while another language which fills the placeholders, focuses on *what* to present. This is also called Model View Controller (MVC) pattern[13] and is used when separation leads to an improved outcome. For example, when developing web pages, page designers can design the templates while developers create the program itself. Thus, model and view can be developed from different teams without the need of knowledge in both areas.

`StringTemplate` is also a Java template engine for generating any kind of formatted text. The developer states, that it is "particularly good at code generators" [8]. It benefits from separating the implementation and model logic from the template rendering[19]. For MPPCG, the model-view separation enables retargetability, since the *model* can stay the same, while the *view* changes between the output languages. However, `StringTemplate` is powerful enough, that simple expressions like conditionals can be evaluated inside the templates. It was designed with maintenance in mind, since most other template engines come with more power than is needed for simple templating[19]. In other template engines, when maintenance is not important, the model gets part of the engine and logic changes can break the model. Another useful feature is template inheritance, such that the same templates do not have to be rewritten for every output language. This can also be used when supporting multiple versions of the same language, where only some parts change.

Compared to `StringTemplate`, `FreeMarker` is a bit more complex, as classes representing the data-model have to be implemented. This is indirectly done by MPPCG, as each node is rendered. However, the rendering logic is not meant to be implemented inside the nodes itself, but in the output environments (see section 5.5). Thus, due to its simplicity, `StringTemplate` fits best for the implemented code generator. As mentioned in section 4.1, Kotlin is interoperable with Java code. Hence, a Java template engine can be used from Kotlin code.

[8]https://www.stringtemplate.org/

5 Multi Programming Paradigm Code Generator

This chapter describes the main components of the implemented multi programming paradigm code generator (MPPCG). Section 5.1 gives an overview of how MPPCG's components interact with each other. Then, in section 5.2 we will see in more detail, what MPPCG's first parser generator is, and how it works. This parser generator parses the input code and emits an abstract syntax tree. Subsequently, MPPCG translates the generated AST into an intermediate code representation (section 5.3) which builds an internal syntax tree with custom nodes. MPPCG makes no assumption about the correctness of the input code, but translates the code as provided by the AST into code of the output language. The only part where the input code's correctness is being verified, is during type checking. However, not every parser generator provides type information. Thus, section 5.4 describes how MPPCG implements a type inference module, which calculates missing types, and verifies given types one-the-fly. Some input languages have custom operators. For example, B has relations and methods for these relations. Output language specific implementations of these methods are required to execute the generated code. These implementations are provided in the *input language module*, which is described in section 5.5, along with the *output language module*. This module is responsible for the main rendering step in MPPCG. The implemented code generator implements a simple testing framework for generating executable test files. This framework enables the execution of generated code while testing for specified expected results and is described in section 5.6. Lastly, section 5.7 gives insights on how to use and extend MPPCG.

5.1 Generator

The generator module itself combines the other modules and is the start and the endpoint of MPPCG's code generation. It does not only start a recursive rendering process and writes the results to an appropriate file, it can also perform pre and (theoretically) postprocessing of the input code.

Postprocessing is in general possible, but not implemented as it is not required for the implemented languages. Preprocessing, however, is implemented as the type inference (mentioned in section 5.4) is performed, and a keyword handler processes the intermediate code.

The keyword handler traverses all identifier nodes of the intermediate syntax tree via reflection and takes the keywords of the output language into account. Whenever a node contains an identifier which resembles a keyword, all occurrences of this identifier are replaced by a modified identifier of the form `keyword_someNumber`. To achieve a correct keyword handling, each output language has to provide a StringTemplate file containing all keywords of the language.

Before having an in-depth look at the base components of MPPCG, we will now have an overview about how they interact with each other. This can also be seen in figure 2.

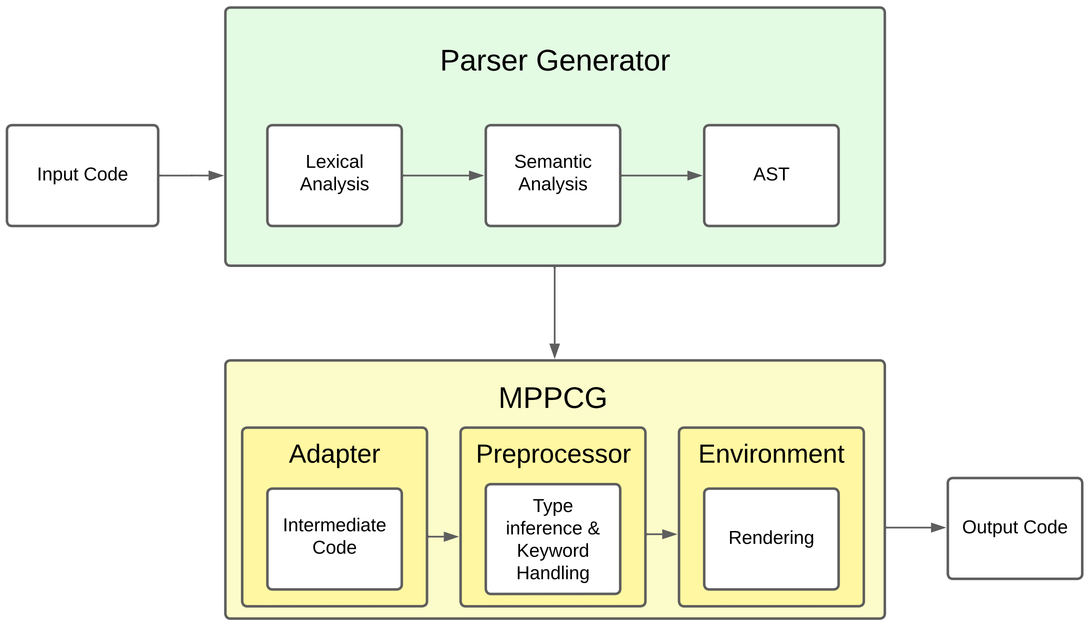

Figure 2: MPPCG

Parser Generator. First, the parser generator parses the input code and creates an abstract syntax tree from it. Hence, it executes the lexical and the semantic analysis and returns the resulting AST. This is described in section 5.2 and is represented by the upper, green box of figure 2. The generated abstract syntax tree is then passed to MPPCG's adapter.

Adapter. Generation starts by converting the parser generator's abstract syntax tree into MPPCG's custom nodes. Therefore, starting from the root node, the implemented adapter converts recursively the entire abstract syntax tree to the intermediate code representation. The tree is traversed by visitors, described in section 5.3. Each visitor returns a created MPPCG node, and the resulting intermediate code representation is again a tree. On this tree, preprocessing can be performed.

Preprocessor. The preprocessing can be extended if required. For now, MPPCG calls the type inference module (see section 5.4) to verify and complement the nodes types. After this step, the keyword handler is called to prepare the nodes for rendering and to avoid for example compile time errors of the generated code. When preprocessing is done, the intermediate code tree is passed to the environment.

Environment. For each output language, MPPCG implements an environment. The environment is the component, where the main rendering logic of the output language is implemented. It implements the rendering method for each MPPCG node, as well as output language specific information. For example for Prolog, part of these information can be a counter, keeping track of the variable storing the current result. The rendering is done recursively by visiting every (for the output code required) child node. MPPCG's code generation is complete when the environment has rendered the entire intermediate code.

5.2 Lexer and Parser

An abstract syntax tree (AST) is the representation of the abstract syntactic structure of code, in the form of a tree[4]. The AST consists of nodes, which can have one or more child nodes, sibling nodes, and one parent node. It is called abstract, since it stores no detailed information about the nodes, but represents only the syntactic structure. An example of an AST is given in figure 3, which consists of only three nodes: A Binary Expression node with two Expression nodes as child nodes. In this example, each node contains also the token returned by the lexer. In this case, the `BinaryExpr` node contains the operator,

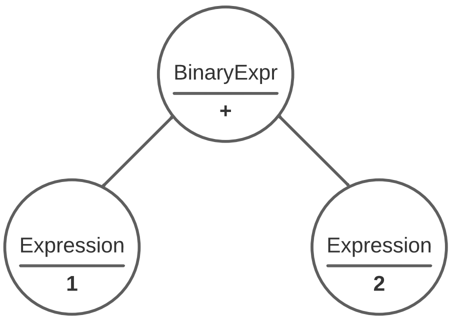

Figure 3: AST with Three Nodes

such that other operators might also generate `BinaryExpr` nodes. A different approach to design such an AST is to implement distinct nodes for each operation, for example with `AddNodes`, `MultiplyNodes`, and `SubtractionNodes`. The parser generator MPPCG uses, provides distinct nodes for each operation. A core component for MPPCG is the intermediate code representation, which stores these information in data class fields. This is similar to the nodes in the example AST. When an input file is parsed, it is first analysed lexically, then syntactically, and finally semantically[4]. The following steps describe, how the input `1 + 2` is analysed, and how the AST of figure 3 is being created.

Lexical analysis. The input for the lexical analysis is a stream of characters and letters, the output is a stream of token. Hence, the content and words of the file are broken into token by a lexer. This is done, by reading one character after another until for example a regular expression matches the input. Depending on the matching regular expression, the lexer returns a token for the processed characters. In our example, the lexer reads the `1`, and returns a `NumberToken`. The `1` is followed by a whitespace, such that a `WhiteSpaceToken` is returned. The next character is the '+', for which an `AddToken` is returned. This token is followed by another `WhiteSpaceToken` and another `NumberToken`. Note: In our example, whitespaces can be neglected, and will be *ignored* by our example parser.

Syntactical analysis. The syntactical analysis is done by a *parser* which receives the stream of token of the lexer. The parser consumes the token, until a grammar rule (syntactic rule) matches the sequence of token. If such a rule matches, the parser returns an AST node for the consumed token. In our example, the parser ignores the `WhiteSpaceToken` such that it reads `NumberToken AddToken NumberToken`. Each `NumberToken` is recognized as an `Expression` so that the parser returns `Expression` nodes for the `NumberToken`. Then, the parser reads `Expression AddToken Expression` and recognizes this sequence as a binary expression. Hence, the parser returns a `BinaryExpr` node with two child nodes.

Semantic analysis. In the semantic analysis, the AST is validated for its semantic correctness. A syntactically correct AST does not imply its semantic correctness. As an example, the input `1 + true` would be syntactically correct, as a `BinaryExpr` with two `Expression` nodes could be returned. However, this abstract syntax tree is semantically not correct, as most languages would not support the addition of integers and booleans. Further checks of the semantic analysis are for example scope errors or type checks.

These steps can be executed by parser generators. Two popular parser generators are `SableCC`[16] and `ANTLR`[39]. B2Program uses an existing ANTLR parser[9] for B in its implementation. MPPCG uses the SableCC version[10] which is also being used in ProB. Both versions have advantages and disadvantages over another. As an example, the ANTLR parser supports only a subset of B, but provides type information. Thus, the ANTLR parser yields a typed AST. The SableCC version however, supports B in total, but does not provide type information. I decided to use the SableCC implementation, as it enables the chance to generate a larger subset of B, compared to the ANTLR parser. However, to generate Java code, or code for other type sensitive languages, a type inference algorithm implementation is additionally required.

SableCC supports also multiple tree traversing features, like the DepthFirstAdapter[16] which traverses the abstract syntax tree in depth first order. To achieve the goal of extendability, such that the input and output languages are exchangeable, this tree should always consist of the same node types / classes. Thus, the parser generated nodes have to be converted to MPPCG's custom nodes, such that the entire AST is transformed to MPPCG's intermediate code representation. With SableCC this transformation can be done due to the tree traversing features. Crucial to this transformation is the structure of the parser's AST. Complex and badly designed parser generators require more complex adapters. The more similar the parser's node classes are to the nodes of MPPCG, the more trivial the adapter is. However, providing a lexer and parser for each input language is not part of MPPCG's responsibility.

[9]https://github.com/hhu-stups/antlr-parser
[10]https://github.com/hhu-stups/probparsers/tree/develop/bparser

5.3 Intermediate Code

As mentioned in section 5.2, the parser generator's AST has to be converted to a custom intermediate code representation in order to keep the languages exchangeable. The intermediate code's nodes are designed to have as few different node types as possible. This is, to keep the output environments as simple as possible, as fewer rendering methods have to be implemented. The conversion is done by an adapter and the main mapping mechanism can be seen in figure 4. An adapter maps nodes of one representation, to nodes of MPPCG's representation. As different parser generators have different node representations, one adapter is required for every parser generator.

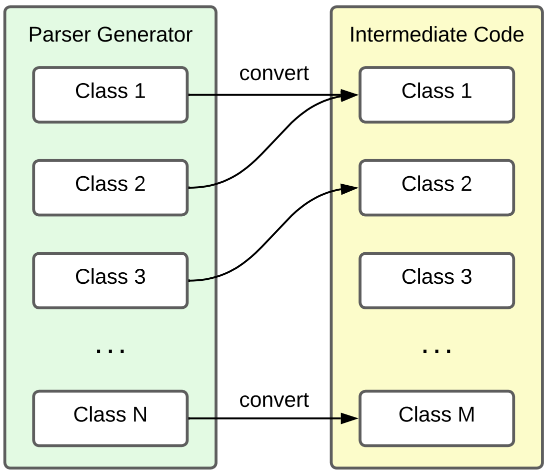

Figure 4: MPPCG's Parser Generator Adapter

Most of MPPCG's nodes have one of the following node types: `Collection`, `Expression`, `Predicate`, or `Substitution`. Input languages might need additional node types, which can not be represented by the common types. For B, these are for example nodes to represent transitions. Hence, the nodes itself inherit from one of the base node types. For example, the `BinaryExpression` inherits from `Expression`, and contains two child nodes of type `Expression`, as well as an operator. The base node class every other node inherits from is the `MPPCGNode`. This class ensures, that every node has a `render`-Method called on the current `Environment` and provides the name of the responsible template. MPPCG's nodes are data classes without any method implementation. As different input languages might contain unique language constructs, like B has quantifier constructs, custom nodes have to be created. For the implemented input language, the B Method, the adapter converts the SableCC nodes to MPPCG nodes. Therefore, the adapter utilizes the DepthFirstAdapter of SableCC, and implements visitors, which in turn implement operations depending on the visited node. Listing 15 presents the code which converts an `AModuloExpression` and an `APowerOfExpression` of SableCC to MPPCG nodes. Both nodes are `BinaryExpressions` and contain a left and a right expression each, but differ in their operator.

Listing 15: Part of MPPCG's ExpressionVisitor using SableCC's Visitor Pattern

```
override fun caseAModuloExpression(node: AModuloExpression) {
    result = BinaryExpression(
                    node.left.convert()!!,
                    node.right.convert()!!,
                    BinaryExpressionOperator.MOD
                )
}

override fun caseAPowerOfExpression(node: APowerOfExpression) {
    result = BinaryExpression(
                    node.left.convert()!!,
                    node.right.convert()!!,
                    BinaryExpressionOperator.POW
                )
}
```

Listing 15 shows also, that MPPCG contains much less intermediate code nodes compared to the used SableCC implementation, as multiple SableCC nodes, which represent binary expressions, can be represented by a single MPPCG class. This code is part of the `ExpressionVisitor`, which returns only `ExpressionNodes`. `convert` is an extension function MPPCG implements for each of the super classes of SableCC's nodes and is, for example for expressions, implemented as seen in listing 16. These extension functions apply the `ExpressionVisitor` on expressions (and for example the `SubstitutionVisitor` on substitutions respectively) and return the visitors result. Thus, in listing 15, the `BinaryExpressions` receive as first two arguments the results of the visitors of the left and the right nodes.

Listing 16: Example `convert`-Method for Expressions

```
fun PExpression?.convert(): Expression? {
    if (this == null) return null

    val visitor = ExpressionVisitor()
    this.apply(visitor)
    return visitor.result
}
```

The most careful design decisions took place in the design of the custom nodes. On the one hand, the goals of MPPCG had to be considered, such that fewer MPPCG nodes result in less rendering-implementation logic. On the other hand, it was required to decide which nodes might be common in most languages and which nodes are only relevant for B. MPPCG encapsulates this whole transformation process in its adapters, which are required for each input language (and each parser generator).

All method extensions MPPCG implements rely on the intermediate code representation.

The output environments implement these extension methods to add rendering methods to the MPPCG nodes. Thus, the adapters are used to transform the input AST to the intermediate code, which then can be rendered. This does not only reduce the amount of different nodes and prepares these nodes for simpler rendering implementations. It does also enable the change of the input language.

5.4 Type Inference Module

The provision of node types by parsers varies depending on their implementation. When working with a type-providing parser, the type inference can be skipped or rather be used as a type verification step. However, when working with a parser like the SableCC B parser[11] which MPPCG used, no types are given. My first approach was calculating types when creating nodes inside the adapter (see 5.3). Calculations were mainly based on the operator (if a node contained one) but this procedure was unstable, error-prone, and would have to be recreated in each parser adapter. It might be a workaround for some input languages with a simpler type system, such that most types could be inferred just from operators. For example a '+' could always indicate that both types are integers. But as B also contains relations, deferred sets, functions, and more unique structures, this approach was too unstable and became quickly complex and hard to maintain.

This complexity contradicted the goal of an easy extendable code generator and was finally replaced by a type inference module. This module implements the Hindley-Milner type inference algorithm[24, 11], as described in section 2.6. However, the algorithm requires an environment, also called `context`, which contains some base types as starting points. As an example, consider the expression `x+2`. If this expression is encountered, the environment needs the information that the `ValueNode` containing `2` is of type integer (or at least a subtype of number). With this information, the type of `x` could be inferred. But these rules change from language to language. Therefore, nodes like `ValueExpression`, which contain primitive types, have in MPPCG a type assigned during their creation in the adapter. Hence, the `context` of the algorithm is created on-the-fly by each adapter indirectly. The algorithm itself is implemented as described in section 2.6 and as seen in listing 3, but needs to provide a *when*-branch for each MPPCG node. Additionally, some rules (branches) might change between languages. The different branch implementations for the same node can be switched, as the type inference module has knowledge about the current input language.

5.5 Code Rendering

Rendering code for an output language requires the implementation of two modules for this output language. The *input language module* is an interface between the actual input language and the generated output code. In this module, input language specific components

[11]https://github.com/hhu-stups/probparsers/tree/develop/bparser

are implemented as libraries which can then be used by the output code. Currently, as MPPCG implements only the B Method as input language, this is the module where the BTypes are implemented for Java and Prolog. Hence, the *input language module* is responsible for providing libraries to *execute* the generated code.

The *output language module* is the module where the rendering is implemented. MPPCG provides an `OutputLanguageEnvironment` containing the abstract `renderSelf` methods for each common node. *Common nodes* are nodes which are present in many input languages and programming paradigms, such as binary expressions or assignments. Additionally, it provides the routing of a node's `render` method to the correct `renderSelf` implementation in the specific output language (as described in section 5.1). Furthermore, this module acts like a template engine interface, as it loads the template files, fills the parameters of a template, and renders the template itself. The `OutputLanguageEnvironment` is designed such that it is as simple as possible to implement a new output language. For better readability, it also provides `render` extension functions for lists, such that a whole list can be rendered at once, and a list of results is returned.

As shown in the code example in listing 17, it is designed in a way that you can call `render` whenever needed, without taking care of the exact field type. This increases the readability and reduces the complexity.

Listing 17: An Example Mapping of Template Variable Names to its Values

```
val map = mapOf(
    "expression" to expr.render(),
    "predicate" to pred.render(),
    "listOfParameters" to params.render()
)
```

Rendering-methods for operators are also implemented, as operators might be called differently in different output languages. For example, for a boolean conjunction the `JavaOutputEnvironment` would render the operator '`&&`' while the `PrologOutputEnvironment` would just render a single '`,`'.

As mentioned, input languages might contain operations or methods which are not covered in the output language's default libraries. These are the operations implemented in the *input language module*'s libraries. For these operations, which are represented as custom nodes, abstract rendering methods have to be provided. Since the `OutputLanguageEnvironment` provides only abstract rendering methods for the *common nodes* (which are common in most languages), MPPCG implements a `BEnvironment`, which provides the remaining abstract methods for the B Method's custom MPPCG nodes. This `BEnvironment` complements the `OutputLanguageEnvironment` and can be exchanged with other input language specific environments.

To add an output language, a subclass of `OutputLanguageEnvironment` for this output language is required. This subclass implements the `renderSelf` methods and is therefore able to handle the entire translation from intermediate code to output language code.

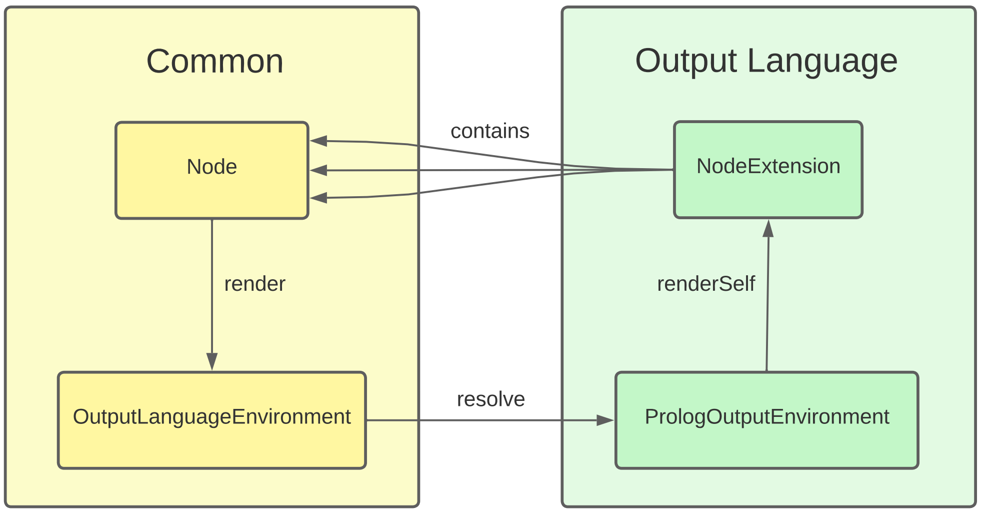

Figure 5: MPPCG's Rendering Mechanism

In figure 5, the rendering process of the output environment can be seen. MPPCG starts the rendering by calling a recursive, tree traversing rendering mechanism on the root node. This is done by calling `render()` on the root node. However, to get the environment involved, the `render`-method calls the environment, which in turn delegates the call to the associated abstract `renderSelf`-method. Note, that up to this point, the delegation is not part of any specific output language. The specific output language environment provides just the `renderSelf()` implementation, to which the call is resolved. In figure 5, this is the `PrologOutputEnvironment` that implements `renderSelf()`, which in turn is an extension method of the called node. This node might contain child nodes, which call again their `render()` method. For example, if a `BinaryExpression`-node is being rendered, `render()` is also called on both of its child nodes. At this point, the cycle is complete and starts with each child node as a new starting point.

Note, that it is not possible to call `renderSelf()` directly on the child nodes inside the `PrologOutputEnvironment`. This is, because the child nodes are types of generalized superclasses, for example `Expression` or `Substitution`, and the `renderSelf()` implementations are extensions of the subclasses, for example `BinaryExpression`. Hence, the environment does not only delegate the call to the correct output language, but also resolves the node to the correct subclass.

The return value of `render()` and `renderSelf()` is a `RenderResult`. A `RenderResult` contains the rendered String, as well as an `info`-map for additional information required by the environment. Depending on the output language, the `info` might not be necessary. Thus, `info` is an optional map, providing flexibility in the amount and types of the returned values. In MPPCG's Prolog implementation, most rendering methods of expression nodes provide a *before*-String in this info map. This is, to address the *coloring*-problem mentioned in chapter 3, where information of a previous rendered node need to be extracted. The `renderSelf`-methods create a map, mapping the template engine's parameter names to the `RenderResults` (or other primitive values for the template engine), and passes this map to a `renderTemplate`-method. This is the map shown in listing 17.

The `renderTemplate`-method is also an extension function for each node. It gathers the correct template for each node (provided in the `templateName` field of each node), and fills the template according to the passed mapping. The resulting rendered String is then returned and encapsulated inside the `RenderResult` of the calling `renderSelf`-method. After the whole intermediate code tree is rendered, the last returned `RenderResult` contains the entire rendered output code.

5.6 Testing Framework

MPPCG provides several default tests for the B Method as input language. The idea of the testing structure was inspired by B2Program and contains execution tests and generation tests. Generation tests are used for testing an output language's capability of handling different types of nodes. Hence, generation tests check if every possible node can be rendered. Execution tests consist of many different machines and execution paths and test whether the generated code yields the desired behavior.

Instead of execution paths, B2Program uses for each machine and each output language a file written in the output language itself. This file contains a simple program executing the generated and translated B Methods. For example, a Java file contains a class with a main method. Inside the main method, an instance of the machine is instantiated which then executes operations. Further, B2Program has for each machine a *.out file which contains only the expected result after executing the program.

This concept enables the ability to execute each machine in a custom way, however, many additional executable files have to be written by hand. In numbers, MPPCG contains 168 machine files for execution tests. In B2Program, this would require additional 168 *.out files and for each output language again 168 execution files. Supporting two output languages, this would end up in a total of 672 files, increasing by 168 for every new output language. Thus, MPPCG simplified this procedure, but grants the same results. Instead of *.out files and executable files, MPPCG has for each machine exactly one corresponding file: a *.execPath file. Each file contains the operations to execute, as well as the expected result. Listing 18 shows an example for the content of such a file.

Listing 18: Example execPath File

```
plus(5)
addOne
*** Expected Result:
getResult=7
```

Line 3 is parsed like a breakpoint: For the previous lines, the framework generates a file for each output language, which instantiates the machine and executes the operations in these lines. For example for Java, the file with the main method described before is being generated. In this case, `plus` is a method (or predicate, depending on the output language) with `5` as parameter and `addOne` is a method (or predicate) without any parameter.

After generating the executable operations before the breakpoint, the results (line 4 in listing 18) are parsed. Again multiple results can be tested. In this case, a method/predicate `getResult` is called without any parameter and returns a value. This value is expected to be equal to `7`. The generated file calls next to the operations `getResult`, and prints the result. This output stream is being read by the test framework and compared to the expected value.

To translate the content of *.execPath files, for each output language a short StringTemplate setup file is required. In case of Java for example, this StringTemplate returns a template for a class with variable name, a main function executing operations, and a template for calling methods. If more machines need to be tested, only the machine file and the *.execPath are required.

The tests are executed by first generating the output code for the machine file. Then, the framework generates the execution file, which in turn executes the prior generated output code. Finally, the results are compared.

5.7 Working with MPPCG

This section explains how to work, modify, and extend MPPCG. In MPPCG it is possible to change the input language, the input language's parser generator, and the output language.

Exchanging the Parser Generator. To support a new parser generator, the parser generator's implementation itself is required. Note, that this implementation is independent of MPPCG. MPPCG implements a `Launcher` which is the starting point of the code generation. Here, the input and output languages (and therefore the corresponding output environment), and the parser generator is defined. The launcher must be modified to be able to execute the new parser generator. The next step is to provide an adapter which converts the parser generator's AST into MPPCG's intermediate code representation. This adapter was described in section 5.3. When the parser generator's AST is converted into the intermediate code representation, the new parser generator can be used. In general, when changing the parser generator, the only crucial part is the mapping from AST nodes to MPPCG's nodes.

Adding a new Input Language. First of all, the input language needs to have a corresponding parser generator. The input language might also have for example some control structures, which are not covered by the common nodes of the intermediate code. For B, this might be for example a comprehension set, which needs to be represented in an own new node. If new nodes are required, these nodes have to be integrated into MPPCG.

First, the nodes have to be implemented in the `TypeInference module`, otherwise the algorithm throws an exception when discovering unrecognized nodes. Secondly, a custom `LanguageEnvironment`-interface has to be implemented, containing the abstract `renderSelf`-methods for these new nodes. For example for the B Method, this is the implemented `BEnvironment`. Additionally, if needed, an environment configuration can be implemented. In B's `BEnvironmentConfig`, for example, this holds the values for the smallest and the largest implementable integer. Note, that if a new input language is added, the existing output languages can not handle the input language, unless each output environment implements the `renderSelf`-methods for each new node. The last step when adding a new input language is to provide libraries for each output language. These libraries are written in each output language and contain implementations for input language specific operators. Examples are the BTypes[46] implemented in Prolog and Java.

Adding a new Output Language. While implementing MPPCG, it was one goal to keep this step as simple as possible. To add a new output language, an `OutputEnvironment` subclassing the `OutputLanguageEnvironment` (see section 5.5) has to be created. This `OutputEnvironment` needs to implement the `renderSelf`-method for each MPPCG node. Additionally, the StringTemplate templates have to be implemented. Lastly, to successfully execute the generated code, the already mentioned input language libraries have to be implemented in this new output language.

6 Language Implementations

This chapter describes the implementation of the first two supported output languages Java (section 6.1) and Prolog (section 6.2). It shows also what code MPPCG generates when targeting Java and Prolog with the first implemented input language, the B Method. Previously, the code generator implemented an *optimizer module* for generating both optimized, and non-optimized code. However, this module got merged into the *output language module*, such that one output language implementation can not generate different versions of the same code. In section 6.3, we will have a look at the currently implemented optimizations, especially regarding Prolog, as well as the decision to remove this module. The difficulties and problems during the implementation of these first languages are described in section 6.4. Finally, section 6.5 compares MPPCG's implementation with the implementation and design of B2Program.

Currently, MPPCG supports only a subset of B. Not supported machine sections are `includes`, `extends`, `sees`, `uses`, `promotes`, and `refines`. Thus, only single machine files are supported. Enumerated sets and set operations are implemented, except for finite subsets and generalized expressions on sets. Most operations on relations are supported, except for closures and translations between relations and relations / functions (`fnc`, `rel`). Sequences and functions are fully supported, however, records and strings, and their operations are not. MPPCG supports also `if-then-else`, `while`, and `quantifier` constructs, as well as `select` substitutions. Not supported substitutions are `any`, `let`, `var`, `assert`, `choice`, and `case`. To support a larger subset, the conversion methods in the adapter, and probably additional MPPCG nodes have to be implemented. Further implementations are required in the type inference module, where the new nodes have to be considered, as well as the rendering methods in the output environments, and the templates itself.

Through this chapter, we will compare parts of the generated code of the CAN Bus machine file[12]. The relevant excerpts of this file are shown in listing 19. The generated Java and Prolog code will be shown in this chapter, as well as the generated Java code of B2Program.

Listing 19: Excerpt of the CAN Bus Machine File

```
SETS
    % more sets
    T3state = {T3_READY,T3_WRITE,T3_RELEASE,T3_READ,T3_PROC,T3_WAIT}

INVARIANT
    % more invariants
    T3_state : T3state & BUSwrite : 0..5 +-> INTEGER

% initialization, other operations, ...
T3writebus(ppriority,pv) =
    PRE
        ppriority = 4 & pv = 0
        & T3_state = T3_WRITE
    THEN
        BUSwrite := BUSwrite <+ {ppriority |-> pv}
        || T3_state := T3_WAIT
    END
```

6.1 Java Implementation

The implementation of the Java support was not the main focus of this thesis. However, the generated Java code is executable and implements BTypes similar to the BTypes of B2Program[46]. Java is an object-oriented and imperative programming language which is also supported by B2Program and was mainly implemented to demonstrate MPPCG's ability to generate code for multiple programming paradigms.

In Java, the BTypes are more complex, as MPPCG implements different Java classes for different B data structures, like B2Program does [46]. Thus, the input language module for Java emits a jar-file which can be used by the generated Java code.

Listing 20 shows the Java code generated by MPPCG of the mentioned CAN Bus excerpt. The implementation was inspired by B2Program's implementation, such that the guard and the substitution of the B operation are seperated. The guard, prefixed by *tr_* takes the operation's parameters as arguments and returns a boolean indicating whether the

[12]In all listings, unneccessary whitespaces have been removed.

machine's state is valid to execute the substitution. This guard implementation can be optimized, for example by collapsing negated boolean predicates to an *unequal*-operator. However, as Java was not the main focus, this optimization has not been implemented yet.

Listing 20: Excerpt of the CAN Bus Java Code

```
 1: // rest of class
 2:
 3: // Guard
 4: public boolean tr_T3writebus(Integer ppriority, Integer pv) {
 5:     if (!(ppriority == 4)) {
 6:         return false;
 7:     }
 8:     if (!(pv == 0)) {
 9:         return false;
10:     }
11:     if (!(T3_state == T3state.T3_WRITE)) {
12:         return false;
13:     }
14:     return true;
15: }
16:
17: // Substitution
18: public void T3writebus(Integer ppriority, Integer pv) {
19:     BUSwrite = BUSwrite.override(new BRelation<Integer, Integer>(
20:                     new BCouple<>(ppriority, pv)
21:                 ));
22:     T3_state = T3state.T3_WAIT;
23: }
```

The substitution method is the place where the state is being modified. MPPCG does not generate code for primitive Java values, but generates their nullable types instead. For example, instead of generating `int`, MPPCG generated `Integer` in order to allow nullability. Nullable values are required for generic BTypes and thus preferred also for every other use case where primitive values would be sufficient. B sets, like `T3state` in this example, are represented in Java as enumerations. Hence, in line 22 of listing 20, `T3_state` gets the value `T3_WAIT` of the enum `T3state` assigned.

6.2 Prolog Implementation

As B2Program has particularly problems generating Prolog code, this section gives some insights in MPPCG's generated Prolog code and its `PrologOutputEnvironment`.

MPPCG generates Prolog code which implements the XTL interface mentioned in section 2.3, to enable ProB to execute the generated code. Figure 6 shows the workflow of model checking a B machine with Prolog code. First, the B machine is passed to a parser generator. The parser generator creates an abstract syntax tree of the machine file, as already seen

Listing 21: Part of a B Machine and its Prolog Representation

```
1: % ...
2: VARIABLES        x
3: INVARIANT        x : NATURAL +-> NATURAL
4: INITIALISATION   x := {1 |-> 2, 3 |-> 4, 2 |-> 3}
5: % ...
6:
7: % State in Prolog after initialisation:
8: % [x-[1-2, 2-3, 3-4]]
```

in section 5.2. MPPCG's adapter transform this AST to MPPCG's intermediate code representation, which is the starting point for the main code generation step. The generated Prolog code implements the XTL interface and can be executed and model checked by ProB.

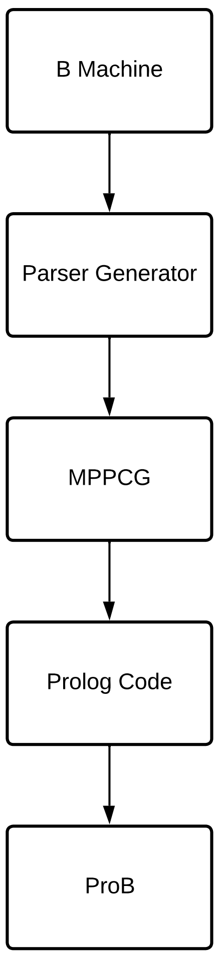

Figure 6: From B Machine to ProB Execution

The machine states are represented as ordered lists with `name - value` entries. B-Relations are again stored as ordered lists. Listing 21 shows a simple B machine and its initialisation, as well as how the generated Prolog code represents the machine state after initialisation (line 8).

Unlike for example in Java, a variable in Prolog cannot be reassigned. Thus, machine states cannot be modified directly, but have to be stored in a separate variable after modification. To access the correct, newest state representation, MPPCG's generated Prolog code enumerates the state and increases a state counter after each modification. For example, after `State_0` was updated, `State_1` contains the updated values, and `State_0` contains the same values as before.

Updating values of a state is done via an `update/4` predicate of the form `update(<variableName>, <value>, <StateBefore>, <StateAfter>)`. For the prior example, the update call would be `update(x, [1-2,2-3,3-4], State_0, State_1)`.

To receive values of a state, the Prolog code implements a `get/3` predicate of the form `get(<State>, <variableName>, <value>)`. Prolog variables containing machine values or other intermediate values are represented by `Expr_`, suffixed with the current expression count. For example, `Expr_0` contains the value of the first evaluated variable, `Expr_1` contains the value of the second one respectively. Nodes (and so their corresponding subtrees) are stored in a map, with their expression count as a value. Hence, when a node is encountered again in the same scope, only the `Expr_<exprCount>` will be rendered to reduce redundant code. However, the expression count, the state count, and the node map are reset between certain code sections. Thus, every transition and invariant has its own scope for these values.

Sets, (concrete) constants, and concrete variables, whose values are constant are represented differently. The generated Prolog code stores their values in runtime predicates with names equal to the name of the set / constant, prefixed by `s_` for sets and `c_` for constants. Listing 22 shows an example of a B Set *EXAMPLE* and its generated Prolog code.

Listing 22: Example B Set and its Generated Prolog Code

```
SETS EXAMPLE={A,B,C,D,E}

% MPPCG generates:
% s_EXAMPLE(['A', 'B', 'C', 'D', 'E']).
```

To access the values of the set, the generated Prolog code calls `s_EXAMPLE(Expr_0)` (or with a different expression count, if already incremented).

MPPCG renders constants in a similar way, however, their values might be the result of a more complex expression / calculation. To avoid the same calculation of the same constant on every access, MPPCG renders an initialisation step, where the values are calculated and the predicates are asserted[13] at runtime.

MPPCG implements the B operators for Prolog in a single Prolog file (`btypes.pl`) inside its *input language module*. To avoid the redefinition of existing SICStus Prolog predicates, the predicates are named as `mppcg_<operatorName>`.

Listing 23: Excerpt of the CAN Bus Prolog Code

```
% initialization, other predicates, ...

trans('T3writebus'(ppriority=Expr_ppriority, pv=Expr_pv), State_0,
        ↪State_2) :-
    % Guard
    mppcg_equal(Expr_ppriority, 4),
    mppcg_equal(Expr_pv, 0),
    get(State_0, 'T3_state', 'T3_WRITE'),

    % Substitution
    get(State_0, 'BUSwrite', Expr_0),
    mppcg_override(Expr_0, [Expr_ppriority-Expr_pv], Expr_1),
    update('BUSwrite', Expr_1, State_0, State_1),
    update('T3_state', 'T3_WAIT', State_1, State_2).
```

Listing 23 shows excerpts of the Prolog code MPPCG generates for the CAN Bus model.

[13]Assertions in Prolog enable the creation of predicates at runtime.

While in Java the guard and the substitution are being separated, MPPCG renders both of them inside the `trans` predicate of the XTL interface. The `mppcg_equal` predicates in line 6 and 7 are required, as the `trans`-predicate can also be called with variables for `ppriority` and `pv`. The implementation of `mppcg_equal` unifies such variables with the second passed value, while a comparison via the equal operator '==' would fail. Such a failing equality check would suggest, that the transition is not valid.

The use of the state count and expression count of the `PrologOutputEnvironment` is also shown throughout listing 23, as `State_2` contains the final state representation after every update and is *returned* from the transition as a third argument.

Without passing concrete values for the operation parameters `ppriority` and `pv`, Prolog is able to calculate every possible parameter combination and every resulting state via unification. However, the current state (`State_0`) needs to have a valid value.

6.3 Optimizer

In earlier stages, MPPCG included an optimizer module for each output language. That module enabled MPPCG to generate optimized code, as well as the per default non-optimized code. Current optimizations of the generated Prolog code are optimized comparisons and unifications, tracked variables to avoid redundant state accesses, tracked expressions to avoid redundant calculations, and precalculated constants. These optimizations lead to fewer lines of generated code.

In the following, the different types of optimizations are explained with examples. Without any optimization, MPPCG would generate for a comparison like `x == y` the Prolog code shown in listing 24.

Listing 24: Default Equality Check in Prolog

```
...
get(State_0, 'x', Expr_0),
get(State_0, 'y', Expr_1),
Expr_0 == Expr_1,
...
```

In this version the values of `x` and `y` are retrieved from the current state via the `get/3` predicate, and afterward these values are checked for equality. However, this check can take place already as soon as the value of `x` is known, as shown in listing 25.

Listing 25: Optimized equality check in Prolog

```
...
get(State_0, 'x', Expr_0),
get(State_0, 'y', Expr_0),
...
```

Note, that the variable `Expr_0`, containing the value of `x`, is inserted in the `get/3` predicate in line 3 to attempt the unification of the value of `y` with the value of `x`. In larger files, that alone can save multiple lines of code. As another example, Prolog's optimizer keeps track of which values are stored in variables. Thus, without optimization, the pseudocode `z = x + y; z1 = x + y` would be rendered to the code seen in listing 26.

Listing 26: Untracked Variables in Prolog

```
...
get(State_0, 'x', Expr_0),
get(State_0, 'y', Expr_1),
Expr_2 is Expr_0 + Expr_1,
update('z', Expr_2, State_0, State_1),
get(State_1, 'x', Expr_3),
get(State_1, 'y', Expr_4),
Expr_5 is Expr_3 + Expr_4,
update('z1', Expr_5, State_1, State_2),
...
```

Keeping track of the variables means, that `x`'s and `y`'s values do not need to be retrieved from the current state again, as they did not change between `State_0` and `State_1`. The optimized version is shown in listing 27, where `Expr_0` and `Expr_1`, which contain the values of `x` and `y` are reused to calculate `Expr_3`.

Listing 27: Tracked Variables in Prolog

```
...
get(State_0, 'x', Expr_0),
get(State_0, 'y', Expr_1),
Expr_2 is Expr_0 + Expr_1,
update('z', Expr_2, State_0, State_1),
Expr_3 is Expr_0 + Expr_1,
update('z1', Expr_3, State_1, State_2),
...
```

Furthermore, MPPCG's Prolog environment implements optimizations in terms of tracking evaluated expressions, such that the previous example results in the code of listing 28.

Listing 28: Tracked Expressions in Prolog

```
...
get(State_0, 'x', Expr_0),
get(State_0, 'y', Expr_1),
Expr_2 is Expr_0 + Expr_1,
update('z', Expr_2, State_0, State_1),
update('z1', Expr_2, State_1, State_2),
...
```

Tracking expressions has potentially the largest impact on the lines of generated code. Large nested reoccurring expressions will not be generated again, if a variable contains the result already. This is done by comparing subtrees. An evaluated subtree (this subtree has to have an expression node as its root node) is mapped to the variable storing its result. When the same subtree is encountered again in the same scope, then MPPCG does not recursively visit the subtree a second time. Instead, just the name of the variable which holds the result of the evaluation is being rendered.

These three optimization features reduce the generated code from 8 lines to 5, for only two statements of the pseudocode. It can easily be seen, that large files can be reduced much more. This results in faster execution times of the Prolog code, as arbitrarily large, reoccurring subtrees can be neglected while executing the generated code, by just reusing the result of the previous evaluation. While implementing optimization, the `OutputEnvironment`'s code for each output language became more complex and harder to maintain over time. Thus, a design decision was required.

The possibility to generate non-optimized code as well as optimized code was a feature which might not be used much, as optimized code would most likely be used if available, making non-optimized code unnecessary. However, even if not used much, this feature is a major increase of the output environment's complexity, which contradicts the goal of having an easy to extend and maintain code generator. Thus, the separated optimizer module was removed and the optimized node rendering was implemented as the default rendering for the implemented output languages. However, if in some cases both an optimized and a non-optimized version of the same code is required, this can still be achieved. Either by having two separate output environments for the same output language, or by creating for example an optimized output environment. This optimized output environment can be subclassed by a non-optimized environment, which overrides only the required rendering methods.

Listing 29: Code Extract of a Modified Version of the `Train` Model

```
SETS
  BLOCKS={A,B,C,D,E,F,G,H,I,J,K,L,M,N};
  ROUTES={R1,R2,R3,R4,R5,R6,R7,R8,R9,R10}

CONCRETE_CONSTANTS
  nxt,
  rtbl

PROPERTIES
    nxt : ROUTES --> (BLOCKS >+> BLOCKS)
  & rtbl = {b,r|b : BLOCKS & r : ROUTES &
             (r : dom(nxt) & (b : dom(nxt(r)) or b : ran(nxt(r))))}
```

MPPCG implements also another optimization, especially for Prolog: Models of the B-Method might contain concrete variables or concrete constants, which are variables whose values are known. My first approach was to generate predicates for such variables, so that the values can be obtained by calling the corresponding predicate. For more complex values, which are computed inside the predicate, this approach led to performance issues. Consider the code snipped in listing 29 of a modified version of [32] of the Train interlocking from [1]. The generated Prolog code for `rtbl` is shown in listing 30, where `Expr_10` contains the resulting value for `rtbl`.

Listing 30: Comprehension Set in Prolog

```
findall(
    (Expr_b - Expr_r),
    (
        s_BLOCKS(Expr_0),
        mppcg_member(Expr_b, Expr_0),
        s_ROUTES(Expr_1),
        mppcg_member(Expr_r, Expr_1),
        c_nxt(Expr_2),
        mppcg_domain(Expr_2, Expr_3),
        mppcg_member(Expr_r, Expr_3),
        (c_nxt(Expr_4),
        mppcg_callFunction(Expr_4, Expr_r, Expr_5),
        mppcg_domain(Expr_5, Expr_6),
        mppcg_member(Expr_b, Expr_6);
        c_nxt(Expr_7),
        mppcg_callFunction(Expr_7, Expr_r, Expr_8),
        mppcg_range(Expr_8, Expr_9),
        mppcg_member(Expr_b, Expr_9))
    ),
    Expr_10
)
```

When executing this code every time the value of `rtbl` is required, which may be several thousands of times per model checking execution, this same code returns always the same value, and takes always some time to execute. To avoid this redundant execution, MPPCG contains for Prolog an initialisation step, where such expressions are evaluated. After evaluation, the resulting expressions are asserted, such that the corresponding predicate does no code execution, but instead returns the previous calculated value. Assertions in Prolog are a way of creating predicates during runtime. This optimization provides fast access times even for complex concrete variables and constants, which are not stored in the machine's state.

6.4 Problems and Improvements

This chapter covers some difficulties and problems during the implementation of MPPCG, as well as how the current implementation solves them.

While implementing MPPCG, my first approach was to have templates for the output language specific implementation of the B operators and functions. The generator had then tracked per generation process which custom methods were required for the generated machine. Finally, only the templates for the required B operations have been rendered in the resulting file. On the one hand, this approach generated a single runnable file, such that no additional files were needed for executing the code. On the other hand, the same code for the same B operators was generated again for each machine, increasing code redundancy across the output files. This redundancy can be neglected as the generation time and generated file size might not be important. However, keeping track of the required methods, which in turn might depend on some other custom methods, can get very complex. Thus, to avoid this complexity, every input language has now one module for every output language, where these operators are implemented. Hence, MPPCG introduced the *input language module*.

Also, some prior design decisions of the code generator's implementation itself led to problems. Since not every parser generator provides types, these types might need to be inferred by the generator. First, this was implemented in the node's data classes and partially in the AST adapter itself. This led to hardly maintainable code and was not in accordance with the goal of supporting multiple input languages, because these implementations would have been necessary in each AST adapter of untyped ASTs[14]. Thus, MPPCG implemented the type inference module (see section 5.4). This module, however, might get complex with more input languages containing custom nodes. To reduce this complexity, MPPCG faces the same design choices as with the other components: A careful decision of which nodes are *common* across different programming languages is required, such that common parts can be extracted and do not need to be implemented for every language again.

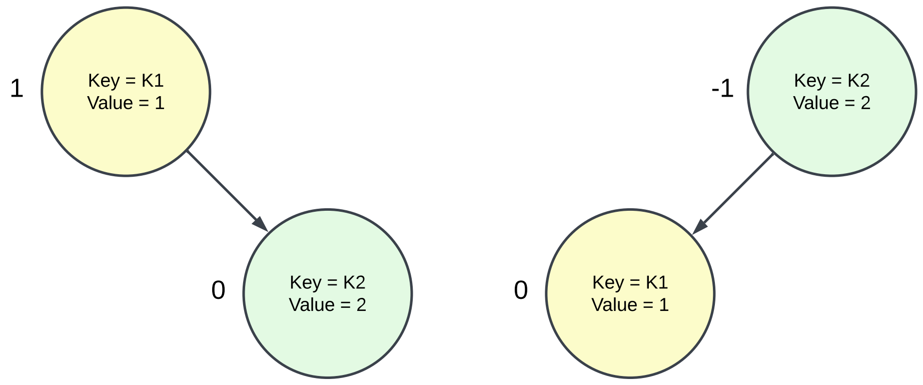

Figure 7: Two Different AVL Trees

During the implementation of the `PrologOutputEnvironment`, one of the problems was the state representation. The first approach of the state representation was using AVL trees[3]. AVL trees are balanced, binary trees and provide fast access times in terms of searching (O(log n)) and insertion and deletion of nodes (both with the worst case complexity of

[14]As long as the output language requires types.

O(log n)). This would have been a speed-up in various operations, however, a crucial part to the structure of AVL trees is the insertion/deletion order of the nodes.

In figure 7, two AVL trees with the same content are presented. Both trees have a node with *K2* as key and *2* as value (green), and a node with *K1* as key and *1* as value (yellow). The weights, also called *Balance Factor*, of the nodes are displayed on the left side of each node. This balance factor is the result of subtracting the height of the left node from the height of the right node. Listing 31 shows, how these AVL trees are represented by SICStus Prolog's avl library. The syntactic structure of such an AVL representation is shown in line 1. Line 2 contains the representation of the left tree of figure 7, line 3 of the right tree respectively. The keyword `empty` on the position of a `LeftChild` or a `RightChild` indicates, that the node does not have such a child node.

Listing 31: AVL Representations by SICStus Prolog's AVL Library

```
1: % node(Key, Value, Weight, LeftChild, RightChild)
2: node('K1',1,1,empty,node('K2',2,0,empty,empty))
3: node('K2',2,-1,node('K1',1,0,empty,empty),empty)
```

Even if both trees contain the same nodes, their structure differ. This is caused by the insertion order of the nodes. ProB compares the states between transitions and decides, if a state has been visited before. In this comparison the syntactic structure and the content of the states are compared. Since the AVL trees represent the same state in different ways, ProB treats both trees as different states.

This led to the replacement of AVL trees with ordered sets in MPPCG's Prolog implementation of the B library. MPPCG uses ordered sets instead of lists, as the access times of ordered sets are faster. This is, because it can be determined if an ordered set (or a subset) does definitely not contain an item `X`, by comparing the first element of the set with `X`. Sets have in Prolog a first element, as they are represented as lists with distinct elements. Their worst case access times are O(n) for insertion and deletion, as well as O(n) for searching nodes, as probably the entire set has to be visited. The time complexity of ordered sets is worse compared to the time complexity of AVL trees, however, ordered sets represent same states in the same syntactic structure. Most of the implemented B operators in Prolog depend on the state's structure. Hence, changing the state representation implies changing most B operator implementations.

To increase the performance for Prolog in the future, there are different possibilities. One could either use AVL trees again, and insert during the setup phase the possible variables with uninitialized values. This would set up the same structure for different states, as all keys are present from the beginning. But as B supports relations, which might be represented as AVL trees again, this could lead to further problems, as large or even infinite relations can not create their AVL tree with every possible key.

Another approach could be mutable dictionaries as their access times are faster than the access times of ordered sets. However, creating mutable dictionaries and calculating hash values for possibly very large sets or relations, could again diminish this improved performance.

6.5 Comparing MPPCG with B2Program

B2Program uses a template-based approach instead of using intermediate code. This is, to simplify the implementation of multiple output languages and to achieve software engineering principles [46] such as *generic programming*[5] and *don't repeat yourself*[42]. A pure intermediate code approach of code generation does not support multiple output languages as the intermediate code is translated directly to output code. However, MPPCG combines intermediate code with a template-based approach of code generation. Thus, MPPCG does not only support multiple output languages, but also multiple input languages.

Listing 32 shows, how B2Program visits its abstract syntax tree nodes, in this case the initialisation node. It shows further, how B2Program instantiates and fills the templates. In comparison, listing 33 shows how MPPCG implements this logic.

Listing 32: Rendering of B's Machine Initialization in B2Program

```
private String visitInitialisation(MachineNode node) {
    String machineName = ...
    ST initialization = group.getInstanceOf("initialisation");
    TemplateHandler.add(initialization, "machine", machineName);
    TemplateHandler.add(initialization, "properties",
        generateConstantsInitializations(node));
    TemplateHandler.add(initialization, "values", generateValues(node));
    if (node.getInitialisation() != null) {
        TemplateHandler.add(initialization, "body",
            machineGenerator.visitSubstitutionNode(...));
    }
    return initialization.render();
}
```

Listing 33: Rendering of B's Machine Initialization in MPPCG

```
override fun Initialization.renderSelf(): RenderResult {
    val map = mapOf(
        "substitutions" to substitutions.render(),
        "name" to currentProgram.name
    )

    return RenderResult(renderTemplate(map))
}
```

When we compare listing 32 and listing 33, we can observe general differences in how both code generators visit nodes and fill templates. As MPPCG executes the extension function `renderTemplate` on each node, inside the function the fields of the extended node can be accessed. This enables the reduction of lines of codes, as the StringTemplate template does not need to be gathered inside the `renderSelf` method. In contrast, B2Program retrieves the template instance as shown in line 3 of listing 32. Instead, nodes of MPPCG contain the template's name as a field, such that the correct template can be obtained in a more dynamic, instead of hard coded, way inside `renderTemplate`.

In this example, MPPCG contains less information that need to be passed to the template. This depends on the underlying AST and intermediate code representation and is therefore not considered as an advantage or disadvantage. B2Program fills the template with its `TemplateHandler`. This handler receives a template instance, a parameter name, and a parameter value. It checks, whether the template instance expects the specified parameter name, and inserts then the parameter value. MPPCG does also encapsulate the filling of the template, but also executes additional operation on the parameters. This way, a `RenderResult` (line 3 of listing 33) is a valid parameter, as only it's rendered string is extracted and passed to the template.

Listing 34: Rendering of a Tuple/Couple in B2Program (Listing 5 in [46])

```
private String generateTuple(
        List<String> args,
        BType leftType,
        BType rightType
)
    ST tuple = currentGroup.getInstanceOf("tuple_create");
    TemplateHandler.add(
        tuple,
        "leftType",
        typeGenerator.generate(leftType)
    );
    TemplateHandler.add(
        tuple,
        "rightType",
        typeGenerator.generate(rightType)
    );
    TemplateHandler.add(tuple, "arg1", args.get(0));
    TemplateHandler.add(tuple, "arg2", args.get(1));
    return tuple.render();
}
```

Listing 34 shows how B2Program renders a tuple, and listing 35 shows how MPPCG does it, respectively. Note, that in MPPCG tuples are called couple. Listing 34 was presented in [46] to demonstrate the necessary changes when adding support for C++ as output language. As in the original listing, the highlighted code of listing 34 is the C++ specific code.

As MPPCG provides a custom `renderSelf` method for each node in each output language, it is not necessary to modify an existing method when implementing a new output language.

Listing 35: Rendering of a Tuple/Couple in MPPCG

```
override fun Couple.renderSelf(): RenderResult {
    val map = mapOf(
        "from" to from.render(),
        "to" to to.render()
    )
    return RenderResult(renderTemplate(map))
}
```

In B2Program, however, it is necessary to add additional lines of code inside a method. Further, when an existing method needs additional parameters, these parameters might have to be passed through multiple methods, starting from their origin. This is the main reason B2Program gets more complex with each new supported target language. MPPCG implies a certain level of redundancy as the `renderSelf` methods for different output languages might be equal. However, this redundancy comes along with complete control over the environment. To summarize, it can be stated that while adding a new language to both code generators, B2Program increases in terms of *complexity*, while MPPCG increases in terms of *redundancy*.

As future improvement, MPPCG could implement interfaces or superclasses like for example an `ObjectOrientedOutputEnvironment` to reduce this redundancy and simplify the process of adding new output languages even further. Object-oriented languages which need a different `renderSelf` implementation than the provided one, could then override the *wrong* implementations. This, however, needs a prior analysis of each node's `renderSelf` implementation in order to determine which implementations might be equal for most object-oriented languages.

The results can also inspire B2Program in terms of improvements. For example, B2Program could start using structures like MPPCG's `RenderResult`, to pass more information between rendering methods than just plain Strings. However, this change would require major refactoring. B2Program's `TemplateHandler` would need to support these new structures, and the rendering methods need to be adapted to return these structures. This refactoring could be a solution for the *coloring*-problem mentioned in chapter 3.

Another idea of improvement for B2Program is to introduce environments, similar to MPPCG. For B2Program these environments could be objects containing various additional information for the current target language, and could be either statically or created while initializing the code generator. With that change, language specific information, like the current expression count which might be required for generating Prolog code, would be available without passing them through deep nested method calls. It can then be studied, whether these two improvements enable B2Program to generate Prolog code.

Listing 36 shows an excerpt of the Java code for the CAN Bus model, generated by B2Program. This listing shows also, that B2Program renders model checking code. While the generated substitution code in Java is similar in B2Program and MPPCG, the generated guards differ in their signature.

Listing 36: Excerpt of the CAN Bus Java Code of B2Program

```java
// rest of class ...

public BSet<BTuple<BInteger, BInteger>> _tr_T3writebus() {
    BSet<BTuple<BInteger, BInteger>> _ic_set_14 =
        ↪new BSet<BTuple<BInteger, BInteger>>();
    for(BInteger _ic_ppriority_1 : Arrays.asList(new BInteger(4))) {
        for(BInteger _ic_pv_1 : Arrays.asList(new BInteger(0))) {
            if((T3_state.equal(T3state.T3_WRITE)).booleanValue()) {
                _ic_set_14 = _ic_set_14.union(
                    ↪new BSet<BTuple<BInteger, BInteger>>(
                        ↪new BTuple<>(_ic_ppriority_1, _ic_pv_1)));
            }
        }
    }
    return _ic_set_14;
}
```

MPPCG returns a boolean value to indicate whether the substitution can be executed. In contrast, B2Program returns a set of possible parameter combinations, for which the guard is true. This is similar to the unification done by Prolog, to receive the possible parameter combinations. B2Program requires this signature for the generated model checker, as each possible parameter combination returned by each transition guard is being used to execute the substitution. Such a code generation is more complex than the generation done by MPPCG (for the guards in Java), as iteration constructs need to be generated and handled. While MPPCG focuses more on a 1-to-1 translation of the B operations to Java code, B2Program focuses on the usability with respect to model checking.

7 Experiments

This chapter compares the model checking performance of the generated Prolog code executed in ProB, with the model checking performance of ProB executing machine files, and with the model checking performance of Java. For this comparison, the `ProB CLI` was used. As mentioned in section 6.1, the generated Java code does not contain any model checking code so far, as the results would be similar to the generated Java model checker of B2Program. Thus, the generated Java code of B2Program was used for the benchmarks. A performance comparison of the Java model checker, ProB, and other generated model checkers, was already done and discussed in chapter 4 of [44].

All benchmarks are run on a MacBook Pro with 16 GB of RAM and a M1 Pro chip. `ProB CLI` has been built locally, to run an ARM version instead of an emulated Intel version of

ProB. Hence, version 1.12.1-final[15] was used with `SICStus`[16] 4.8.0[17]. `SICStus` states, that its emulated Intel/x86 version “works well on newer Macs with Apple Silicon hardware, using the macOS built-in Intel emulator”[41]. They further say, that the native Apple Silicon version lacks the JIT compiler, and thus, the emulated Intel version is faster[41]. To determine the performance difference between the two versions, and to select the best version for the final benchmarks, I executed model checking on some of the benchmarked machines and measured the time. During these model checks, the ARM version performed much better compared to the emulated Intel version. To be precise, the ARM version performed on most machines almost twice as fast as the Intel version [18]. This suggests that ProB does not benefit much from the JIT compiler. As a result, the ARM version was used for the benchmarks.

For the final benchmarks, each file was model checked ten times with a timeout of one hour for each run. From these model checks, the median runtimes were then measured and compared. The median was chosen as the metric, as it represents the middle value where half of the results have longer runtimes and the other half have shorter runtimes. This approach helps to reduce the impact of outliers or significantly deviating runtimes, ensuring they have less influence on the overall comparison. ProB was benchmarked with machine files twice: with the parameter `-p OPERATION_REUSE full`, and without. The same applies for ProB’s model checking on the generated Prolog files. With this parameter set, ProB tries to reuse previously computed operation effects to avoid redundant calculations.

10 different machines have been benchmarked, most of them have already been benchmarked and described in [44]. Both Train machines and the CAN Bus model implement set and relational operations. As in [44], a modified version of [32] of the Train interlocking from [1] has been used. This version has only ten routes. A second modified version of Train with 9 routes has also been benchmarked. In the second version (Train_POR), partial order reduction is applied manually, as B2Program can currently not handle some of the original constructs.

Other machines are the Volvo Cruise Controller and the Landing Gear model (from Event-B [30]) which focus mainly on boolean variables and logical operations [44]. The Landing Gear model does also contain a large number of set operations.

An insertion sort algorithm (originally from Event-B [40]) with 1000 elements, which contains relational operations has also been benchmarked.

A version of Lift from [46], which increments and decrements a counter between 0 and 1,000,000 (instead of 0 and 100) is being used as a simple model with a large state space.

The N-Queens problem is implemented in two B models, for $N = 4$ and $N = 8$. These

[15] Revision 56aa9d8bf76e49a7f300a3a550352c3877e13d1b

[16] https://sicstus.sics.se/index.html

[17] arm64-darwin-20.1.0 version

[18] When model checking with ProB. This might be a ProB issue or could be related to the generated Prolog code.

models have been benchmarked as they focus on constraint solving capabilities.

Lastly, Nokia's NoTa (Network on Terminal architecture) model [38] was benchmarked as in [44]. It contains many set operations, power sets, and quantified constructs, and was rewritten to apply B2Program. This modified version has fewer transitions but does not affect the performance of ProB.

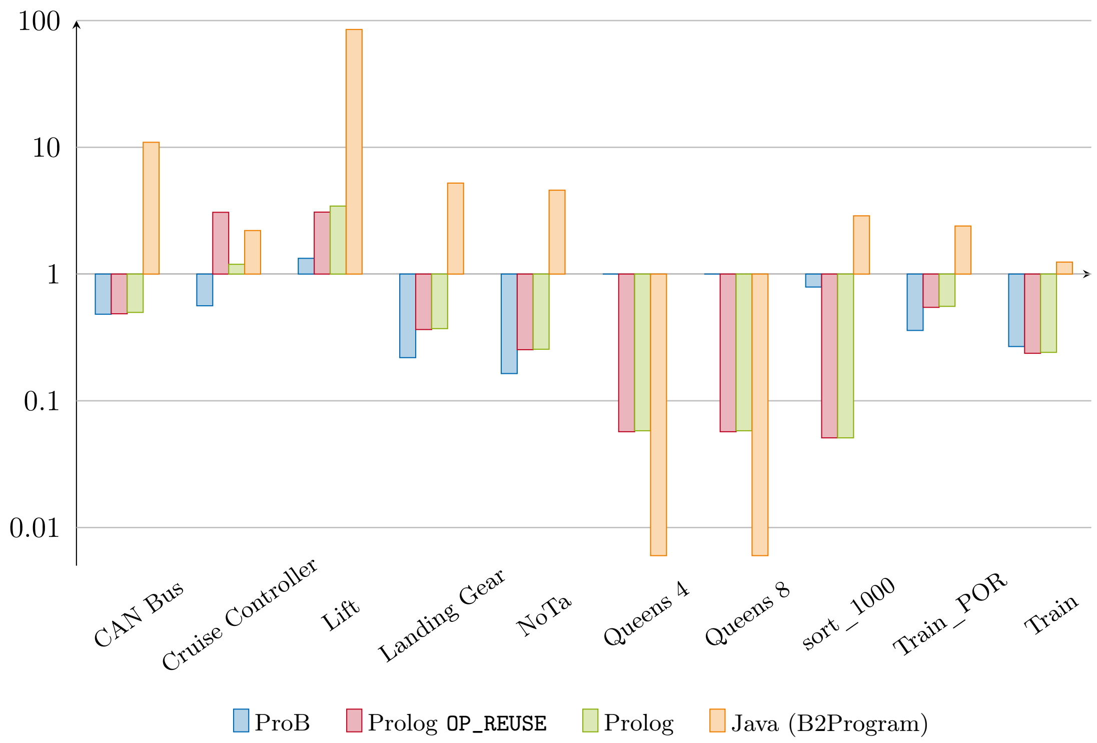

Figure 8: Speedups Relative to ProB with `OP_REUSE`

The results of the benchmarks are presented in figure 8 and listed in more detail in table 1 to 5 in Appendix A. On the one hand, the impact of ProB's operation reuse can be seen, when both of ProB's benchmarks are compared.

ProB with and without Operation Reuse. First, we will compare ProB with `Operation Reuse` to ProB without this property. In average, model checking with operation reuse is about two to three times as fast, as without operation reuse. Both Queens machines seem to be not impacted by the flag, which might be lead back to their small state space. The most notable speedup can be seen on the NoTa machine, where model checking with operation reuse is about 6 times as fast as without. However, on the Lift model, model checking with operation reuse performs worse than without.

ProB Running Prolog with and without Operation Reuse. On the other hand, when comparing both Prolog benchmarks, operation reuse has almost no notable impact. The only machine profiting from the flag is Volvo's Cruise Controller. With operation reuse, ProB model checks the Prolog code of this Cruise Controller about 40 percent faster as without. Similar to the comparison of ProB's benchmarks, it can again be seen that the Lift model performs slightly better without operation reuse. This comparison leads to the assumption, that operation reuse might not be fully supported for Prolog code using the XTL interface.

ProB Running Prolog and Machine Files with Operation Reuse. The performance increase in ProB with operation reuse is notable, resulting in the best results overall when comparing with Prolog. The only machines, where Prolog outperforms ProB are the Cruise Controller and the Lift model.

ProB Running Prolog and Machine Files without Operation Reuse. As ProB may not fully support operation reuse for Prolog yet, we will also compare ProB's and Prolog's benchmarks, both without operation reuse. The results are shown in figure 9. Here, the

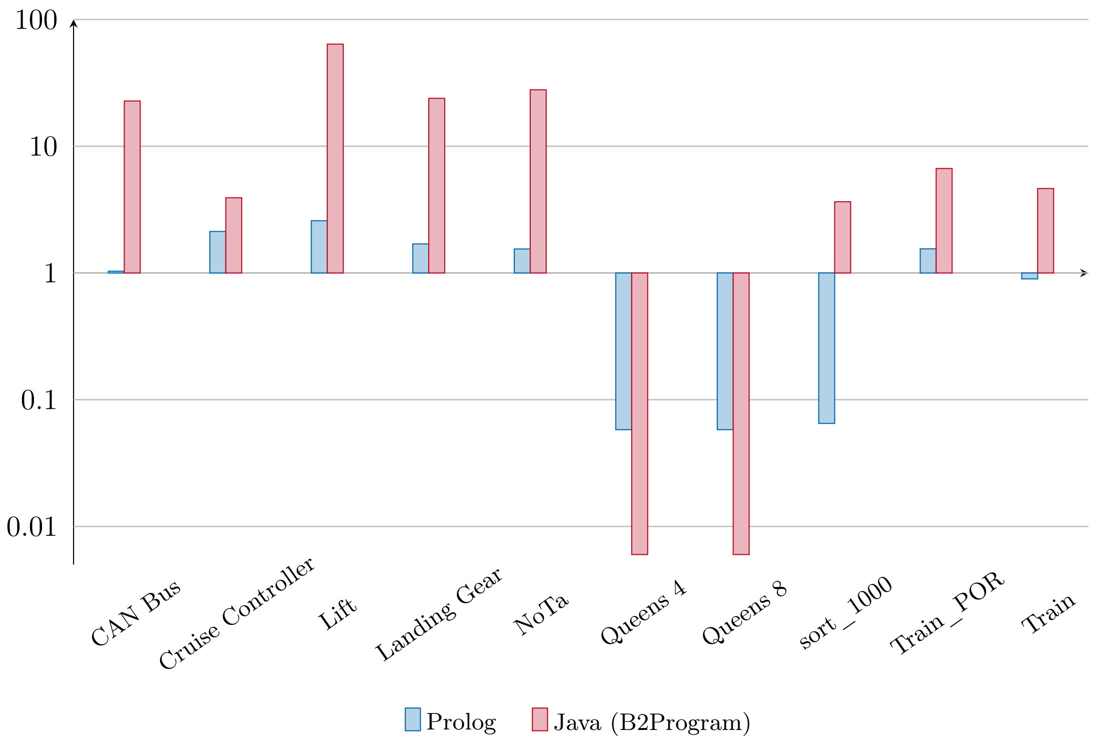

Figure 9: Speedups Relative to ProB without `OP_REUSE`

results are more balanced, as ProB executing the generated Prolog code outperforms ProB running the machine files in 6 machines, and vice versa in 4 machines. The highest speedup is again achieved when benchmarking the Lift model, which is for Prolog about 2.6 times as fast as ProB model checking the machine file. Model checking `Train_POR`, where partial order reduction has been applied manually, yields additionally better results for Prolog and ProB than model checking `Train`. Notable are also both benchmarks on the Queens models, where ProB is much faster when model checking the machine files. In numbers, ProB needs for both versions just 50 milliseconds. Prolog in contrast needs 8.6 seconds, which is a speedup of more than two orders of magnitude for ProB. On the one hand, this is due to ProB's constraint solving capabilities[44]. On the other hand, to recheck the results I implemented a simple model checker in Prolog.

Examining the Results of the Queens Benchmarks. The queens machine has only one transition: from the initial state to the state where the modeled problem is solved. After the problem is solved, this transition can be executed again, but does not alter the state further. This single relevant transition enabled it to implement a simple model checker. First, the model checker calculated the initial state. Then, this state was checked for invariant violations using `prop/2`. Using Prolog's `findall`, every valid transition and hence every successor state has been calculated and checked again with `prop/2`. The performance was measured using `statistics(walltime, ...)` which provided completely different results than the previous benchmarks. Using this simple model checker, the Queens machines have been model checked in Prolog within 60 milliseconds. ProB includes also many other checks and calculations, which this simple model checker does not implement. However, the difference in performance is remarkable. As ProB executes the same generated Prolog code as this simple model checker, one reason for this behavior could be the used Prolog data structure to represent machine states. The generated Prolog code represents machines as ordered lists, as already discussed in section 6.4. It might be the case, that this data structure is not optimal for ProB combined with the XTL interface. However, if this is the case, this would imply that the other benchmarked machines would also have a much higher possible speedup. This performance issue can be further examined using different data structures representing the machines in Prolog.

Comparison with Java. When comparing the results with the Java model checking code generated by B2Program, we see that this Java model checker dominates in most machines. When model checking the CAN Bus, the Java model checker performs more than an order of magnitude faster than ProB with operation reuse, and the relative speedup is even faster when comparing with Prolog. Notable are also the benchmarks of the Lift model, where Java performs almost two orders of magnitude faster than ProB with operation reuse. However, when solving the Queens Problems, Java is more than two orders of magnitude slower than ProB, and one order of magnitude slower than the Prolog benchmarks.

While benchmarking the Cruise Controller, ProB model checking the generated Prolog code with operation reuse outperforms every other model checking benchmark. Meanwhile, this is the only machine where Prolog achieved the fastest results. MPPCG's generated Prolog code can be further optimized in terms of performance. That could be achieved by reducing the lines of generated Prolog code. This requires an improved detection of evaluated expressions in the `PrologOutputEnvironment`. The invariant checking can also be optimized, since at the current state, the invariants have been split into different predicates. Thus, repetitive evaluation of code might occur, as the variables keeping the evaluation of expressions are not passed between the different Prolog predicates. And, as already mentioned, the implementation of the B types for Prolog, as well as the state representation, can be further optimized.

8 Related Work

Partial Evaluation. An example use of partial evaluation are the three Futamura projections introduced in [15]. The first Futamura projection is defined by specializing an interpreter for given source code. This results in an executable interpreter that only runs this given source code and is therefore faster than the original interpreter. The second Futamura projection is the specialization of the specializer for that interpreter of the first Futamura projection. This projection yields a resulting compiler. The third and last Futamura projection is specializing the specializer of the second projection. Hence, a tool to convert any interpreter to a compiler is created.

Some of the existing partial evaluation tools for Prolog are the online partial evaluator `ECCE` [35] and the offline partial evaluator `Logen` [35].

Code Generators. B2Program is a code generator that inspired the development of MPPCG. B2Program has far more features than MPPCG has, like generating model checking code for Java, C++, Rust, and JavaScript. Further, it implements animation of the generated JavaScript code [45]. However, B2Program is restricted to a subset of the B Method as input language, while MPPCG implements the foundation of supporting different input languages. There are also many other code generators for the B Method (or Event-B) [8, 14, 12, 36, 40], but also code generators for various other use cases, like generating parser generators [39] or reverse engineering tools assisting and simplifying the development process [23].

Model Checker. MPPCG does currently not generate executable model checking code. However, the generated Prolog code supports the XTL interface, such that model checking can be executed by `ProB`[33, 34]. ProB is a model checker and animator with several features and was already described in the previous chapters. Another existing model checkers is `SPIN`[25, 26] which is used for model checking concurrent systems. The systems are modeled using the Promela language and properties are specified in linear temporal logic (LTL). `JavaPathFinder`[22] is a model checker for Java programs. It checks race conditions and deadlocks and implements backtracking features. The Java code can also be translated to Promela, to model check the code with `SPIN`.

9 Conclusion and Future Work

This thesis presented MPPCG, a code generator capable of generating code for programming languages of different programming paradigms. MPPCG demonstrated that it is possible to create such an extendable, maintainable, and versatile code generator, with some advantages but also some drawbacks. We discussed the flexibility compared to B2Program and have seen, that Java and Prolog code can be generated efficiently. However, this approach faces issues in terms of redundancy. Separating the generator code of the programming languages enables the implementation of new output languages, but requires redundant code to keep this separation. This redundancy increases the development time for new languages, but grants an improved maintainability.

We have seen in listing 34, that B2Program adds additional parameters to existing methods, in order to support new languages. However, with an increasing amount of supported languages methods get complex over time and it might become difficult to determine which parameter belongs to which output language. With the language separation of MPPCG, each output language contains only its required variables and maintaining the language is less difficult. In the future, it can be studied, if it is possible or even recommended to weaken the separation, at least between output languages of similar programming paradigms, to avoid the discussed redundancy. This could either be by using for example an `ObjectOrientedOutputEnvironment` or by utilizing `StringTemplate`'s template inheritance, to minimize redundancy across templates. We also discussed possible improvements for B2Program in section 6.5 which could enable B2Program to support other programming paradigms.

The generation of Prolog code implements some optimizations in terms of a reduced number of generated lines of code. MPPCG's design enables also the development of code generation for different versions of the same language. This can for example be an optimized and *regular* version of the same output language. In the performance analysis, we have seen some strengths and weaknesses of the generated Prolog code. We have seen, that it does not profit from ProB's operation reuse property and that model checking the Prolog code in ProB performs similar to model checking the machine files. The performance of the generated code can be improved by a different state representation and by enhanced implementations of the B operators. We have also seen, that B2Program generates fast model checking code, which outperforms the other model checking benchmarks on many models.

In the future, MPPCG will be able to generate model checking code for Java. However, this will require more implementations regarding the analysis of the machine files to efficiently use various model checking techniques. This includes also constraint solving, which is currently not implemented in the generated Prolog code. At the current state MPPCG supports only a subset of B, which will be enlarged in the future.

Appendices

A Experiment Results

Table 1: ProB with Operation Reuse

Machine	Median (sec)	States	Transitions
CAN Bus	15.40	132,598	340,264
Cruise Controller	0.86	1,360	26,148
Lift	65.94	1,000,001	2,000,000
Landing Gear	25.09	131,328	884,368
NoTa	16.68	80,718	1,797,352
Queens 4	0.50	3	4
Queens 8	0.50	3	4
Sort 1000	184.51	500,500	500,500
Train POR	14.56	24,635	62,224
Train	535.07	1,044,335	4,515,172

Table 2: ProB without Operation Reuse, Speedup Relative to ProB with Operation Reuse

Machine	Median (sec)	Speedup	States	Transitions
CAN Bus	31.96	0.482	132,598	340,264
Cruise Controller	1.53	0.562	1,360	26,148
Lift	49.53	1.331	1,000,001	2,000,000
Landing Gear	114.73	0.219	131,328	884,368
NoTa	101.40	0.164	80,718	1,797,352
Queens 4	0.50	1.000	3	4
Queens 8	0.50	1.000	3	4
Sort 1000	233.70	0.790	500,500	500,500
Train POR	40.58	0.359	24,635	62,224
Train	1995.90	0.268	1,044,335	4,515,172

Table 3: Prolog in ProB with Operation Reuse, Speedup Relative to ProB with Operation Reuse

Machine	Median (sec)	Speedup	States	Transitions
CAN Bus	31.69	0.486	132,598	340,264
Cruise Controller	0.28	3.071	1,360	26,148
Lift	21.40	3.081	1,000,001	2,000,000
Landing Gear	68.71	0.365	131,328	884,368
NoTa	66.05	0.253	80,718	1,797,352
Queens 4	8.71	0.057	3	4
Queens 8	8.70	0.057	3	4
Sort 1000	> 3600	< 0.051	500,500	500,500
Train POR	26.66	0.546	24,635	62,224
Train	2256.72	0.237	1,044,335	4,515,172

Table 4: Prolog in ProB without Operation Reuse, Speedup Relative to ProB with and without Operation Reuse

Machine	Median (sec)	Speedup (`OP_REUSE`)	Speedup	States	Transitions
CAN Bus	30.93	0.498	1.033	132,598	340,264
Cruise Controller	0.72	1.194	2.125	1,360	26,148
Lift	19.18	3.439	2.583	1,000,001	2,000,000
Landing Gear	67.67	0.371	1.696	131,328	884,368
NoTa	65.54	0.255	1.547	80,718	1,797,352
Queens 4	8.61	0.058	0.058	3	4
Queens 8	8.61	0.058	0.058	3	4
Sort 1000	> 3600	< 0.051	< 0.0649	500,500	500,500
Train POR	26.16	0.556	1.551	24,635	62,224
Train	2,222.95	0.241	0.898	1,044,335	4,515,172

Table 5: B2Program's Generated Java Code, Speedup Relative to ProB with and without Operation Reuse

Machine	Median (sec)	Speedup (`OP_REUSE`)	Speedup	States	Transitions
CAN Bus	1.41	10.957	22.747	132,598	340,264
Cruise Controller	0.39	2.205	3.923	1,360	26,148
Lift	0.78	85.077	63.903	1,000,001	2,000,000
Landing Gear	4.81	5.221	23.877	131,328	884,368
NoTa	3.64	4.589	27.895	80,718	1,797,352
Queens 4	82.66	0.006	0.006	3	4
Queens 8	80.26	0.006	0.006	3	4
Sort 1000	64.05	2.881	3.649	500,500	500,500
Train POR	6.08	2.394	6.674	24,635	62,224
Train	430.40	1.243	4.637	1,044,335	4,515,172

List of Figures

List of Tables

List of Listings

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 2010.

[2] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996.

[3] G. M. Adelson-Velskii and E. M. Landis. “An Algorithm for Organization of Information”. In: *Proceedings of the USSR Academy of Sciences 146*. Ed. by Myron J. Ricci. 1962, pp. 263–266.

[4] Alfred V. Aho and Monica S. Lam. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Addison-Wesley, 1986. Chap. The Structure of a Compiler.

[5] R. Backhouse et al. “Generic Programming”. In: *Advanced Functional Programming*. Vol. 1608. LNCS. Berlin, Heidelberg: Springer, 1999, pp. 28–115. DOI: 10.1007/10704973_2.

[6] Patrick Behm et al. “METEOR: A Successful Application of B in a Large Project”. In: *Formal Methods*. Vol. 1708. LNCS. Berlin, Heidelberg: Springer, Jan. 1999, pp. 369–387. DOI: 10.1007/3-540-48119-2_22.

[7] Richard Bonichon et al. “LLVM-Based Code Generation for B”. In: *Formal Methods: Foundations and Applications*. Vol. 8941. LNCS. Cham: Springer, Jan. 2015, pp. 1–16. DOI: 10.1007/978-3-319-15075-8_1.

[8] Néstor Catano and Víctor Rivera. “EventB2Java: A code generator for Event-B”. In: *NASA Formal Methods*. Vol. 9690. LNCS. Cham: Springer, June 2016, pp. 166–171.

[9] Edmund M. Clarke et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification*. Vol. 7682. LNCS. Berlin, Heidelberg: Springer, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1.

[10] ClearSy. *Atelier B, User and Reference Manuals*. 06.2023. URL: http://www.atelierb.eu.

[11] Luís Damas and Robin Milner. “Principal Type-Schemes for Functional Programs”. In: *Symposium on Principles of programming languages*. POPL. ACM, Jan. 1982, pp. 207–212. DOI: 10.1145/582153.582176.

[12] Andrew Edmunds. “Templates for Event-B Code Generation”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Vol. 8477. LNCS. Berlin, Heidelberg: Springer, 2014, pp. 284–289. ISBN: 978-3-662-43652-3.

[13] Martin Fowler. *Patterns of enterprise application architecture*. Boston, MA: Addison-Wesley, 2015.

[14] Andreas Fürst et al. “Code generation for Event-B”. In: *Integrated Formal Methods*. Vol. 8739. LNCS. Cham: Springer, 2014, pp. 323–338.

[15] Yoshihiko Futamura. “Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler”. In: *Higher-Order and Symbolic Computation* 12 (Dec. 1999), pp. 381–391.

[16] E.M. Gagnon and L.J. Hendren. “SableCC, an object-oriented compiler framework”. In: *Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No.98EX176).* Santa Barbara, CA, USA: IEEE, Aug. 1998, pp. 140–154. DOI: 10.1109/TOOLS.1998.711009.

[17] Erich Gamma et al. *Design Patterns. Elements of Reusable Object-Oriented Software.* Prentice Hall, July 1997.

[18] Susan Gerhart, D. Craigen, and Ted Ralston. “Case study: Paris metro signaling system”. In: *IEEE Software* 11.1 (Jan. 1994), pp. 28–32. DOI: 10.1109/MS.1994.1279941.

[19] *GitHub StringTemplate 4.* 06.2023. URL: https://github.com/antlr/stringtemplate4/blob/master/doc/index.md.

[20] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem.* Vol. 1032. LNCS. Springer, 1996.

[21] James Gosling. *Java: an Overview.* Feb. 1995.

[22] Klaus Havelund and Thomoas Pressburger. “Model checking JAVA programs using JAVA PathFinder”. In: *International Journal on Software Tools for Technology Transfer* (Mar. 2000), pp. 366–381. DOI: 10.1007/s100090050043.

[23] *Hibernate.* 07.2023. URL: https://hibernate.org/tools/.

[24] Roger Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (Dec. 1969), pp. 29–60.

[25] Gerard J. Holzmann. “The model checker SPIN”. In: *IEEE Transactions on Software Engineering* 23.5 (May 1997), pp. 279–295. DOI: 10.1109/32.588521.

[26] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, Sept. 2003. ISBN: 0-321-22862-6.

[27] Paul Hudak. “Conception, Evolution, and Application of Functional Programming Languages”. In: *ACM Computing Surveys* 21.3 (Sept. 1989), pp. 359–411. DOI: 10.1145/72551.72554.

[28] *Kotlin.* 06.2023. URL: https://kotlinlang.org/docs/comparison-to-java.html.

[29] *Kotlin Multiplatform.* 06.2023. URL: https://kotlinlang.org/lp/multiplatform/.

[30] Lukas Ladenberger et al. “Validation of the ABZ landing gear system using ProB”. In: *International Journal on Software Tools for Technology Transfer* 19 (Apr. 2017), pp. 187–203. DOI: 10.1007/s10009-015-0395-9.

[31] Michael Leuschel. “Operation Caching and State Compression for Model Checking of High-Level Models”. In: *Integrated Formal Methods.* Vol. 13274. LNCS. Cham: Springer, June 2022, pp. 129–145. ISBN: 978-3-031-07727-2.

[32] Michael Leuschel, Jens Bendisposto, and Dominik Hansen. "Unlocking the Mysteries of a Formal Model of an Interlocking System". In: *Proceedings Rodin Workshop*. 2014.

[33] Michael Leuschel and Michael Butler. "ProB: A Model Checker for B". In: *FME 2003: Formal Methods*. Vol. 2805. LNCS. Berlin, Heidelberg: Springer, Sept. 2003, pp. 855–874.

[34] Michael Leuschel and Michael Butler. "ProB: An Automated Analysis Toolset for the B Method". In: *International Journal on Software Tools for Technology Transfer* 10.2 (Mar. 2008), pp. 185–203.

[35] Michael Leuschel et al. "The Ecce and Logen Partial Evaluators and their Web Interfaces". In: *Partial Evaluation and Semantics-Based Program Manipulation*. New York, NY, USA: Association for Computing Machinery, Jan. 2006, pp. 88–94. DOI: 10.1145/1111542.1111557.

[36] Dominique Méry and Neeraj Kumar Singh. "Automatic code generation from Event-B models". In: *Proceedings of the 2nd Symposium on Information and Communication Technology*. New York, NY, USA: Association for Computing Machinery, Oct. 2011, pp. 179–188.

[37] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. ISBN: 0-13-629155-4.

[38] Ian Oliver. "Experiences in Using B and UML in Industrial Development". In: *Formal Specification and Development in B*. Berlin, Heidelberg: Springer, 2007, pp. 248–251. DOI: 10.1007/11955757_20.

[39] Terence J. Parr and Russell W. Quong. "ANTLR: A predicated-LL (k) parser generator". In: *Software: Practice and Experience* 25.7 (July 1995), pp. 789–810.

[40] Victor Rivera et al. "Code generation for Event-B". In: *International Journal on Software Tools for Technology Transfer* 19 (May 2015), pp. 31–52. DOI: 10.1007/s10009-015-0381-2.

[41] *SICStus Prolog*. 07.2023. URL: https://sicstus.sics.se/download4.html.

[42] *The Pragmatic Programmer: From Journeyman to Master*. Amsterdam: Addison-Wesley, 1999.

[43] Jean-Christophe Voisinet. "JBTools: An experimental platform for the formal B method". In: *Proceedings of the inaugural conference on the Principles and Practice of programming*. Maynooth, County Kildare, Ireland: National University of Ireland, June 2022, pp. 137–139.

[44] Fabian Vu, Dominik Brandt, and Michael Leuschel. "Model Checking B Models via High-Level Code Generation". In: *Formal Methods and Software Engineering*. Vol. 13478. LNCS. Cham: Springer, Oct. 2022, pp. 334–351. DOI: 10.1007/978-3-031-17244-1_20.

[45] Fabian Vu, Christopher Happe, and Michael Leuschel. "Generating Domain-Specific Interactive Validation Documents". In: *Formal Methods for Industrial Critical Systems*. Vol. 13487. LNCS. Cham: Springer, Sept. 2022, pp. 32–49. ISBN: 978-3-031-15007-4. DOI: 10.1007/978-3-031-15008-1_4.

[46] Fabian Vu et al. "A Multi-target Code Generator for High-Level B". In: *Proceedings iFM 2019*. Vol. 11918. LNCS. Cham: Springer, Nov. 2019, pp. 456–473. DOI: 10.1007/978-3-030-34968-4_25.

[47] Robert G. Wilson Leslie B. amd Clark. *Comparative Programming Languages*. 3rd ed. Addison-Wesley, 2000. ISBN: 0201710129.