A Multi-Target Code Generator for High-Level B

Fabian Vu[0000−0003−2556−5553], Dominik Hansen, Philipp Körner[0000−0001−7256−9560], and Michael Leuschel[0000−0002−4595−1518]

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{fabian.vu, dominik.hansen, p.koerner, leuschel}@uni-duesseldorf.de

Abstract. Within high-level specification languages such as B, code is refined in many steps until a small "implementable" subset of the language is reached. Then, code generators are used, targeting programming languages such as C or Ada.
We aim to diminish the number of refinement steps needed, by providing an improved code generator. Indeed, many high-level operations and data types, such as sets, can be dealt with in programming languages such as Java and C++. We present a code generator for B named B2Program with two distinct features. Firstly, it targets multiple (high-level) languages via a template-based approach to compilation. In addition to flexibility, this also enables one to safeguard against errors in the individual compilers and standard libraries, by generating multiple implementations of the same formal model. Secondly, it supports higher-level constructs compared to existing code generators. This enables new uses of formal models, as prototypes, demonstrators or simply as very high-level programming languages, by directly embedding formal models as components into software systems. In the article, we discuss the implementation of our code generator, evaluate it using B models taken from literature and compare its performance with simulation in ProB.

1 Introduction and Motivation

Models written in formal specification languages, such as B, can be verified via proof obligation generation and proving (e.g. by using AtelierB [9]) and animation and model checking (e.g. by using ProB [25]). Once a B model is verified, it is often desirable to derive executable code from the model. This might be a standalone binary or code that can be used as a library. Re-implementing the code by hand, however, is cumbersome and might introduce new errors. Instead, for safety-critical applications, code generators are typically applied.
Yet, existing code generators do not work on just any B model but only support a very limited subset of B, often referred to as "B0" [9] or *implementation language*. Refining the model to B0 often requires many refinement steps and, again, is very cumbersome. We can make two observations: firstly, translation of higher-level constructs, e.g. sets, to modern languages is straightforward. This allows translation of a larger subset than B0 and reduction of effort due to refinement of the model. Secondly, *flexible output* is desirable: while software can run as low-level program on some embedded systems, there are safety-critical components

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

Listing 1. Example of a State Machine of a Lift Controller in B

implemented in different languages, e.g. web applications written in Java or software written in C++.
In this paper, we present B2Program[1], which is a code generator that, technically, works on any abstraction level and is able to target multiple high-level programming languages using a template-based approach. Following this approach, B2Program supports higher-level B constructs than other code generators. None of the examples used in Table 2 are in the B0 language, but code can be generated from these models by B2Program without any refinement steps. Most other code generators would require refining these models to B0 before code generation can be applied.
In the following, we briefly introduce the B method and B language as well as ProB, which is a tool that we build upon. We explain best practices from compiler engineering that are foundations for our own code generator and discuss concerns regarding correctness in Section 2. Afterwards, in Section 3, our template-based approach to code generation is described in detail. In Section 4, the performance of the generated code is compared to ProB and trade-offs in our standard libraries are analysed. Finally, we compare our approach with existing work in Section 5.

The B-Method, B and ProB The B-Method [1] is a method that is mainly used for specification and verification of software systems. The B method enforces a "correct-by-construction" approach. Many safety-critical system applications, e.g. the *Paris Métro Line 14* [10], the New York Canarsie Line [12] and around 95 installations of Alstom's U400 CBTC system contain code generated from verified B models. In some more recent applications such as the *ETCS Hybrid Level 3 Concept* [16] formal B models have been executed at runtime.
Part of the B-Method is the B specification language which is based on set theory and first-order logic. A component in the B language is called a machine, which contains declaration of constants, variables and sets along with initialisation and operations to modify the machine's state. Furthermore, there are constructs relevant for verification, such as preconditions or invariants, i.e. a predicate that must be true in each reachable state. Listing 1 shows a simple specification of a lift in B containing substitutions (aka statements), expressions, preconditions and an invariant.

[1] Available at: `https://github.com/favu100/b2program`

ProB [25] is an animator, constraint solver and model checker for B models. It allows automatic animation along with model checking using different techniques [29,21]. In particular, ProB supports checking invariants and absence of deadlocks, but also custom assertions and LTL formulas.

2 Steps of Code Generation

A compiler is typically separated into two parts [2]: the front-end performing an *analysis*, and the back-end performing code *synthesis*. Within the context of formal methods, the model must be verified before it is passed to the code generator. An overview is given in Figure 1. In the following, these three phases are described in more detail.

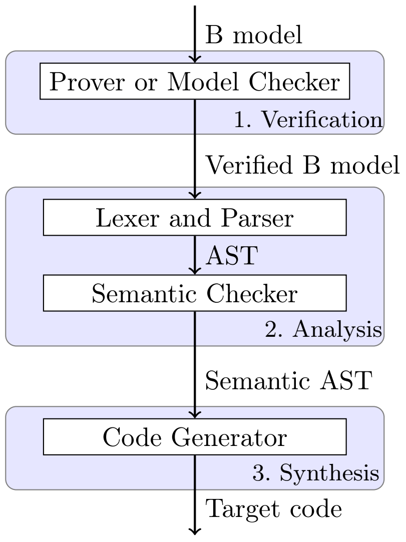

Fig. 1. B Model to Generated Code

Verification of the B Model Verification can be done either by proving generated POs [9] or by model checking [5]. This is of utmost importance, as generating code from an incorrect model may eventually lead to undesired or incorrect behaviour. Furthermore, well-definedness, as well as the absence of infinite loops and integer overflows has to be checked. This can be done with tools such as ProB or AtelierB.
Note that our code generator currently does not check that verification has been successfully carried out. This phase is currently merely an item on the checklist of a user's workflow.

Analysis Phase The next step assumes that the given B model already is verified. First, the B model is passed to the lexer, which divides the B code into tokens defined for B, with categories such as identifiers, separators, operators, keywords, literals etc. After this step, the tokens are passed to the parser, which applies the defined context-free grammar rules to create the abstract-syntax tree (AST) for semantic checks. Semantic analysis consists of scoping and type checking. Scoping ensures that variables and operations were defined before they are used. The type checker assigns a type for all appropriate nodes in the AST. After that, the typed AST is checked for type errors (for more details on best practices of compiler front-end design, see [2]).

Synthesis Phase During the synthesis phase, the semantic AST of the B model is used to generate code. We decided to use a *template-based approach*, which allows taking advantage of similarities of several programming languages. Compared to an approach with intermediate code generation, this renders it easier to target different languages and aligns with the best software engineering principles, e.g. generic programming [3] and don't repeat yourself [18]. Furthermore, an intermediate code representation does not assist the extensibility of the code generator concerning additional target languages in any way.

Correctness of Code Generation A big question is: how can one trust the output of the code generator, and the underlying hardware and compiler used to process the code generator's output? While there are some efforts to produce formally verified compilers [24], the industry practice is to use at least two different code generators, developed using different techniques and developed by different teams. The purpose of the second piece of code is to validate the output of the main code generator. The second translation is typically run on a different hardware, safeguarding against faults in the hardware as well. If the output of the two translations differ, the system has to go into fail-safe, degraded mode.
Our code generator is arguably more complex than the ones derived for B0 [9]. These code generators, however, are also not proven and require to be complemented with a second code generator for high-integrity systems, as described above. Our code generator would hence have to follow this approach when being used for SIL-3 or SIL-4 components.[2] If performance is sufficient, one could investigate using, e.g., ProB as the complementary high-level code generator.
One drawback, however, is the dynamic use of heap allocated memory by the respective standard libraries used by our code generator. This will preclude its use in some settings, such as embedded systems, in its current form.
Anyway, the main target of our code generator is not embedded systems, but prototypes, demonstrators, business-critical applications or applications such as data validation. Still, note that our code generator can target different languages. By producing multiple translations for different programming languages, we could safeguard against errors in individual compilers and the respective standard libraries used (but not against errors in the language independent part of our code generator).

3 Code Generation in Practice

In this section, we describe code generation with the use of the semantic B AST generated after the analysis phase and templates written in the language of StringTemplate (`https://www.stringtemplate.org`).

3.1 Template-based code generation

A *template* is a document with holes, which are filled by a *template engine* using provided parameters. The idea of template-based code generation (cf. Figure 2) is to provide templates for possible operators of the AST. AST nodes are then

[2] SIL stands for Safety Integrity Level. SIL-4 is the highest level of integrity for railway systems. See `https://en.wikipedia.org/wiki/Safety_integrity_level`.

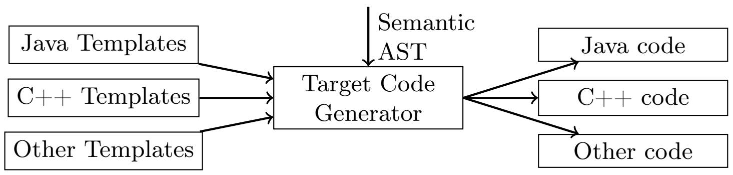

Fig. 2. Template-based Code Generation

translated by the template engine to code, by filling out the holes in the associated template; the content of the holes is derived from the concrete attributes and parameters of the AST nodes. A new language can be targeted by providing a new set of templates. Our code generator uses the StringTemplate engine. It was initially applied for generating dynamic web pages [28], but it now complements the ANTLR parser generator and is well-suited for code generation[3]. Parameters of templates in StringTemplate can also be booleans that decide which part of the template is used to generate the resulting code. In StringTemplate all templates for a programming language are stored in a separate file, named group file.
In order to target an additional programming language, the following steps must be followed:

1. Create all templates for the programming language and implement the mapping of semantic AST nodes to the templates. E.g. an operation node is mapped to the operation template with placeholders for the operation name, parameters, return type etc. These placeholders are replaced by strings that are generated from the semantic information of the operation node. So, rendering the operation template with the required placeholders results in the generated code for an operation. Two different AST nodes can also be mapped to the same template e.g. expression nodes and most predicate nodes with binary operators are mapped to the *binary template*. Some templates require information from many AST nodes, as in Section 3.2.
   Furthermore, templates in two different programming languages that are associated with the same AST node must have the same name. E.g. an operation node is mapped to a template with the name "operation" in both Java and C++. This is required to keep code generation for both programming languages generic. Thus, there is only one implementation for each AST node in B2Program to generate code for many programming languages.
2. Implement B data types in the target language as described in Section 3.4. These types are used by the generated code.
3. Solve the collision problem between keywords and identifiers as described in Section 3.6.

[3] `https://web.archive.org/web/20170723204548/http://pjmolina.com/metalevel/2010/11/stringtemplate-a-great-template-engine-for-code-generation/`

```
initialisation(machine, properties, values, body) ::= <<
public <machine>() {
    <properties; separator="\n">
    <values>
    <body>
}
>>
```

Listing 2. Template for Generating from the `INITIALISATION`, `PROPERTIES` and `VALUES` Clauses in Java

3.2 Code Generation with StringTemplate and B AST

Based on Listing 2, we explain how Java code is generated from the `INITIALISATION` clause of a B machine. The goal is to generate a valid Java class constructor.
Assume that the `INITIALISATION`, `PROPERTIES` and `VALUES` clauses of a B machine are not implemented in the code generator yet. The template in Listing 2 contains placeholders for the machine name, the body of the `INITIALISATION` clause and the `PROPERTIES` and `VALUES` clause. Until the placeholders are substituted, it only outlines what a Java class constructor may look like.
In addition to the template definition, translation must be implemented for the target language, which is outlined in Listing 3. After the machine name is extracted from the AST and the identifier template is applied, the placeholder *machine* is replaced with the result. The body of the initialisation, which is represented by a *SubstitutionNode* in the AST, is passed to another template that belongs to the *substitution*. Code is generated recursively from this node by generating code from all children of the AST node, each yielding an assignment. It finally results in a string that replaces the placeholder *body*. B2Program has the restriction that the `VALUES` and `PROPERTIES` clauses must assign a value to all constants via "=". The `VALUES` clause is represented by a list of substitutions in the AST. Each of the substitutions are generated like other substitutions. In contrast, the `PROPERTIES` clause contains only a single predicate that must be a conjunction. Then, each conjunct that contains the operator "=" is generated as an assignment.
The final Java constructor code for the B machine in Listing 1 is as follows:

```
public Lift() {
    floor = new BInteger(0);
}
```

The generated code uses the type `BInteger` which has to be implemented as described in Section 3.4. Variable declarations are generated beforehand when handling the `VARIABLES` clause of the B machine.

3.3 Extensibility for Other Programming Languages

Adding support for another language, e.g. C++, works similarly to code generation for Java. Some templates in C++ require only a subset of semantic information that is required by the same template in Java. In this case, superfluous

```
private String visitInitialisation(MachineNode node) {
    String machineName = ...
    ST initialisation = 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 initialisation.render();
}
```

Listing 3. Implementation within B2PROGRAM for the `INITIALISATION`, `PROPERTIES` and `VALUES` Clauses

```
// Java
tuple_create(arg1, arg2) ::= <<
new BTuple\<>(<arg1>, <arg2>)
>>

// C++
tuple_create(leftType, rightType, arg1, arg2) ::= <<
(BTuple\<<leftType>, <rightType> >(<arg1>, <arg2>))
>>
```

Listing 4. Template for Creating a Tuple in Java and C++

information is simply ignored. For some constructs however, additional semantic information may be required to generate C++ code, e.g. type information for the C++ STL. Thus, supporting another programming language requires writing templates for this language and extending the *TemplateHandler* only. So code generation for different programming languages is done by the same *TemplateHandler*, but with different templates. A concrete example are maplets (aka tuples), which are represented by the type `BTuple` in the generated code. `BTuple` is a class containing two generic types (one for the first entry and another for the second). Both templates need placeholders for the actual values. While Java can infer both types from the arguments of the constructor, C++ requires both types written in the code explicitly, as shown in Listing 4.
Listing 5 shows the implementation in B2PROGRAM for generating code from a tuple. The highlighted code is added in order to support C++. The additional semantic information does not affect handling the Java template as the function *add* in *TemplateHandler* ensures that there are no additional arguments passed to the Java template. So code generation from a tuple for both languages is done by the same function *generateTuple*. Code generation for the tuple `1|->2` to Java and C++ finally results in:

```
/* Java */ new BTuple<>(new BInteger(1), new BInteger(2))
/* C++  */ (BTuple<BInteger, BInteger>((BInteger(1)), (BInteger(2))))
```

```
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 5. Implementation in B2Program to Generate Code from a Tuple

```
_ld_x = x;
_ld_y = y;
x = _ld_y;
y = _ld_x;
```

Listing 6. Translation of `x := y || y := x`

3.4 Implementation of B Data Types

The B data types are implemented and provided as a library that is included in the generated code. B2Program supports the *scalar* types *Integer* and *Boolean* and *compound* types such as *Set*, *Tuple*, *Relation*, *Sequence*, *Struct* and *Record*. Instead of implementing these types ourselves, it would also be possible to use existing equivalent types in the target language (e.g. implementations of `java.util.Set`). But the API of the B types library must contain all operations that can be used in B, e.g. the operation *relational image* in the class `BRelation`. This approach enables the support for high-level data structures, which are not part of B0.

In addition to sequential substitutions, which evaluate statements one after another, B also allows parallel substitutions. The latter is not part of B0 and poses two interesting challenges for code generation. First, it means we need to keep access to the original, unmodified data structures; we cannot modify sets or relations in place. For efficiency, we have used *immutable data structures* (aka persistent data structures, see [20]). Take, for example, the assignment `x := x \/ {max(x)+1}`, where x is a very large set. Using a traditional mutable data structure, we would have to generate a copy of x for read access for other parts of the B operation. With an immutable data structure we can create a new version of x, while keeping the old value of x and while sharing values between the old and new value of x. Second, B variables that are re-used in another expression are assigned to temporary variables first before the actual assignment is executed.[4] An example is the parallel substitution `x := y || y := x`. This statement swaps both values, where a sequential substitution would ensure both `x` and `y` have the same value afterwards. Instead, it is translated to the Java code shown in Listing 6.

[4] Note that assignment does not copy the data structure; it copies just the reference.

Table 1. Supported Subset of B Types and Operators

B Type	Class	Supported Operators
Integer	BInteger	$x + y$, $x - y$, $x * y$, $x \mathit{ mod } y$, x/y, $-x$, $x < y$, $x \leq y$, $x = y$, $x \neq y$, $x > y$, $x \geq y$, $\mathit{succ}(x)$, $\mathit{pred}(x)$
Boolean	BBoolean	$p \land q$, $p \lor q$, $\neg p$, $p \Rightarrow q$, $p \Leftrightarrow q$, $p = q$, $p \neq q$
Set	BSet	$s \cup t$, $\bigcup_{t \in s} t$, $s \cap t$, $\bigcap_{t \in s} t$, $s \setminus t$, $s \times t$, $\lvert s \rvert$, $x \in s$, $x \notin s$, $x \,..\, y$, $\mathit{min}(s)$, $\mathit{max}(s)$, $\mathbb{P}(s)$, $\mathbb{P}_1(s)$, $s \subseteq t$, $s \not\subseteq t$, $s \subset t$, $s \not\subset t$, $s = t$, $s \neq t$
Tuple	BTuple	prj_1, prj_2, $s = t$, $s \neq t$
Relation	BRelation	r(s), r[S], $\mathit{dom}(r)$, $\mathit{ran}(r)$, r˜, $S \lhd r$, $S ⩤ r$, $r \rhd S$, $r ⩥ S$, $r \mathbin{<\!\!+} s$, $r \otimes s$, $r \parallel s$, id, $r \circ s$, $r \,;\, s$, $r \,..\, n$, r^*, r^+, prj_1, prj_2, $\mathit{fnc}(r)$, $\mathit{rel}(r)$, $r = s$, $r \neq s$
Sequence	BRelation	$\mathit{first}(s)$, $\mathit{last}(s)$, $\mathit{size}(s)$, $\mathit{rev}(s)$, $\mathit{front}(s)$, $\mathit{tail}(s)$, $\mathit{take}(s, n)$, $\mathit{drop}(s, n)$, s^t, $\mathit{conc}(S)$, $E \rightarrow s$, $s \leftarrow E$

Scalar Types Integers have functions for arithmetic operations and comparisons as shown in Table 1. The execution of B2Program has the option to use *primitive integers*, where the language primitive is used, or *big integers*, which allows arbitrary-sized integer values, for the generated code. Creating `BInteger` as a big integer is done by invoking the constructor with a String. The use of big integers avoids exceptions or unsound behaviour in the presence of under- or overflows, at the cost of performance (memory and speed-wise). If it is proven that in a machine integer overflows cannot occur, primitive integer can be used for better performance. Booleans implement functions for logical operations. All operations on integers and booleans that are part of the B language are supported by B2Program.

Compound Types A set in B is represented by a `BSet` in the supported programming languages. The `BSet` class consists of functions for operations on sets (e.g. union, intersection, difference) and an immutable data structure. Thus, applying a binary set operation creates a new `BSet` without changing any of the provided arguments. Deferred sets are also supported by B2Program. The size of each deferred set is fixed and either defined in the `PROPERTIES` clause or taken from the settings for code generation. This makes it possible to interpret deferred sets as enumerated sets.
A relation in B is represented as `BRelation`. As a relation is a set of tuples in B, all implemented operators for a set are available for relations as well. API functions that are exclusive to relations are implemented in `BRelation`, but not in `BSet`. In earlier implementations, `BRelation` extended `BSet`. But representing `BRelation` as a persistent set of tuples resulted in slow performance of operations on relations. In the current version of B2Program, `BRelation` is implemented as a persistent map where each element in the domain is mapped to a persistent set containing all belonging mapped elements in the range. This makes it possible to improve the performance of operations on relations significantly.

Functions are special cases of relations where each element in the domain is mapped to at most one element in the range. As long as a function call is well-defined, the associated value of the only matching tuple is returned.
There are two possible errors when applying a function; normally these should be caught during verification (see Section 2). Firstly, invoking a function with an argument that is outside of the domain raises an exception at runtime. In contrast, calling a function with a value mapped to more than one element returns the first associated value, without raising an error.
Sequences in B are instances of `BRelation` where the domain is always a contiguous set of integers starting at 1. The implementation of sequence operations assumes that this property is fulfilled. Applying sequence operations on relations that are not sequences should also be caught during verification (see Section 2) lest they lead to undefined behaviour at runtime.
Structs and Records are also supported in the generated code. While a struct declares the given fields and field types, a record is an instance of a construct with the given field and the belonging values. In contrast to other compound types, structs are generated at code generation. The generated structs must extend `BStruct` where the needed functions of all structs are implemented. As a struct can have a various number of fields, it would also be possible to implement `BStruct` as a hash map. In this case, each field with its value in a record would be represented as a key-value pair in the hash map. But as the fields can have different types, the implementation would not fulfil type safety. The generated class for a struct contains the belonging fields and functions for accessing or overriding them. The function for overriding is implemented having no side effects on the fields. Instead, a new instance with updated values is returned.
All compound types apart from structs are implemented using generics (aka templates in C++) to specify the type of the elements in `BSet`, `BTuple` and `BRelation`. For example, `BRelation` is a type containing two generic types, one for the type of the elements in the domain and another for the elements in the range. It extends `BSet` where the elements are tuples with the same generic types as the corresponding `BRelation`. Using generics avoids casting and ensures type safety. Table 1 also shows all operations on sets, tuples (i.e., nested pairs), relations and sequences that are supported in B2Program now. The operations that are not listed in Table 1 are not implemented yet.

3.5 Quantified and Non-Deterministic Constructs

Quantified constructs are set comprehensions, lambdas, quantified predicates, quantified expressions as well as `ANY` and "becomes such that" substitutions. They consist of variables constrained by a predicate.
Let $a_1 \dots a_n$ be the bounded variables with $n \in \mathbb{N}$. B2Program has the restriction that the first n sub-predicates must be joined as a conjunction where the *i-th* conjunct must assign or constrain (e.g. via $\subset$, $\subseteq$, $\in$) the *i-th* bounded variables with $i \in \{1, \dots, n\}$. Moreover, the sets that are used to constrain the bounded variables must be finite in order to avoid infinite loops. Additional conditions are joined to the other n sub-predicates as a conjunction. Furthermore,

```
BSet<BInteger> _ic_set_0 = new BSet<>();
for(BInteger _ic_x : BSet.interval(new BInteger(0),new BInteger(5))) {
    if((_ic_x.modulo(new BInteger(2)).equal(new BInteger(1))).booleanValue()){
        _ic_set_0 = _ic_set_0.union(new BSet<>(_ic_x));
    }
}
set = _ic_set_0;
```

Listing 7. Generated Java Code for Set Comprehension

the sub-predicates constraining or assigning a variable are only allowed to use other bounded variables if the used bounded variables are constrained or assigned before. These properties provide the opportunity to generate each of the first n predicates as an assignment or a for-loop to iterate over the sets with a conditional check whether the values of the constrained variables satisfy the entire predicate. The restriction is also necessary because otherwise a constraint solver would be needed to solve all quantified constructs in the generated code. E.g. the predicate `x:INTEGER & x > z & x < z*z` can be solved by a constraint solver but is not supported in a quantified construct for this code generator; the type of `x` is infinite and a generated loop would not terminate. We decided against the usage of a constraint solver as it cannot give any performance guarantees.
A fresh variable storing the result of a quantified construct is defined when necessary. Primed variables in "becomes such that" substitutions are generated to temporary variables that are assigned to their belonging variables before constraining the results. Once a solution for the constraint is found, it is assigned or added to the result. The substitution `set := {x|x : 0..5 & x mod 2 = 1}`, for example, results in the Java code shown in Listing 7.
Non-deterministic constructs such as "becomes element of", "becomes such that" and `ANY` substitutions are also implemented in B2Program. "Becomes element of" substitutions generate invocations of a special function `nondeterminism` on the given `BSet` or `BRelation` on the right-hand side. The implementation of `nondeterminism` chooses an element randomly. `ANY` and "becomes such that" substitutions are generated in the same way as quantified constructs with one difference: they are executed with the solution of the predicate that is found first.

3.6 Identifier Clash Problem with Keywords

Different programming languages use different keywords and different regular expressions for identifiers. In particular, some identifiers in B can be keywords in other languages, e.g. `new` in Java. We store the keywords for each target language in a *keywords* template. Identifiers themselves can collide with each other as well, e.g., local variables due to machine inclusion or other local variables and operation names. Thus, some identifiers have to be re-named during code generation, in case scoping rules differ with B.

4 Performance Considerations and Evaluation

In this section, we discuss the performance of the generated code. We target two languages, Java and C++. The actual implementation of the B types, i.e. the representation of integer, boolean and set, has a major impact on performance. This will be discussed in more detail before we compare the results with simulation in ProB.

B Data Type Implementation There are many subtleties when aiming for a suitable implementation of B Data Types.
Boolean Values are fairly straightforward. They are implemented as classes that wrap a native boolean in both C++ and Java.
Integers can be implemented similarly as long as the absence of over- and underflows is guaranteed. Otherwise, e.g. in C++, this might trigger undefined behaviour. To avoid overflow issues, our code generator also supports arbitrary-sized integer values. In Java, we use the big integer implementation from Clojure [17], as we found operations to be about twice as fast as the one from the Java Class Library. For C++, we use the big integer implementation provided by the GMP library (GNU Multiple Precision Arithmetic Library) [14].
Sets are, as discussed in Section 3.4, assumed to be immutable to render a correct translation easier. Java hash sets or sets from the C++ STL, however, are mutable. Initial versions of our code generator used these along with copying upon modification, which did not perform well.[5] Yet, there are immutable set implementations based on Bagwell's Hash Array Mapped Tries [4]. Due to structural sharing, only a small amount of internal nodes has to be copied in order to create a "changed" copy, e.g. where an element is added. Copying six nodes suffices for a perfectly balanced hash trie with 10 billion elements. We have considered several immutable set implementations for Java. By default, we use sets as provided by Clojure, while we use the state-of-the-art library Immer [30] for immutable sets in C++. They are both stable implementations providing very good performance. For both Java and C++, there is the opportunity to change the set implementation at compile-time.
Analogous to the representation of *Sets*, we use persistent hash maps provided by Clojure and the library Immer for *relations*.

Empirical Evaluation The B data types used in the generated code for the performance analysis are implemented as described above. Generated Java code was executed on the *Java HotSpot(TM) 64-Bit Server VM (build 12.0.1+12, mixed mode, sharing)*. In order to compile C++ code, the *clang compiler (Version: Apple LLVM version 10.0.1 (clang-1001.0.46.4))* was used with the optimisation options `-O1` and `-O2` respectively. As a baseline, we use ProB in the version *1.9.0-nightly (c5a6e9d31022d0bfe40cbcdf68e910041665ec41)*. The complete benchmark code can be found in the B2Program repository.

[5] Several benchmarks ran slower than simulation with ProB.

These benchmarks range from simple machines such as Lift and TrafficLight to complex machines with large state spaces such as CAN bus, Train or Cruise Controller. While Lift and TrafficLight contain arithmetic and logical operations only, Train and CAN bus consist of many set operations. Again, Cruise Controller is a machine having many assignments and logical operations which are more complicated in comparison to other machines. The performance of set operations is also investigated by the benchmarks Sieve, sort_m2_data1000 and scheduler. So the selected benchmarks cover different aspects of the performance.
For the empirical evaluation, an execution trace with a cycle is used for each machine listed in Table 2. The cyclic part of the trace is executed several times within a while loop. Cycles are selected such that each operation is executed at least once. The exceptions are CAN bus and sort_m2_data1000. While the cycle in CAN bus does not contain all operations, sort_m2_data1000 is a quadratic sorting algorithm and does not have any cycles. The state space in sort_m2_data1000 consists of one path from a state representing an unsorted array of 1000 elements to a state representing a sorted array.
Each generated program is executed ten times measuring runtime and memory (maximum resident set size). Table 2 shows the median of all measurements for both runtime and memory. We set a timeout at 30 minutes execution time. The speedup relative to ProB is given as well. As the translation is only run once, the time utilised by B2Program is not measured. Since execution is long enough, the start-up and parsing time of ProB are not relevant (but are included). Note that ProB does variant checking when executing while loops; this cannot be turned off. ProB was run using the command-line version probcli using the `-execute` command. All measurements are done on a MacBook Air with 8 GB of RAM and a 1.6 GHz Intel i5 processor with two cores.
As can be seen, for most machines, generated Java and C++ code can be one to two orders of magnitudes faster than execution in ProB. This comes to no surprise, as interpretation overhead can be quite large. Furthermore, generated C++ code uses only a small percentage of memory compared to Java and ProB. The reason is that both Java and SICStus Prolog make use of garbage collection and both were running in unconstrained memory. Memory consumption can be heavily reduced at the cost of enormous penalties concerning runtime in Java.
The difference between primitive and big integers concerning runtime is comparatively low impact for most machines. For the traffic light and lift examples however, there is an approximately 5× speed-up. Because the considered loops in the machines are very small, significant overhead is caused by incrementing the loop counter which is also a `BInteger`. There can also be significant performance increases depending on compiler optimisations: as there is neither user input during execution nor program parameters, *clang* is able to optimise aggressively. Similarly, most parts of the cruise controller and the sorting example can be optimised. In the other benchmarks, optimisation is more conservative and increases performance up to a factor of two.

Table 2. Runtimes of ProB and Generated Code in Seconds with Number of Operation Calls (OP calls), Speed-Up Relative to ProB, Memory Usage in KB, BI = Big Integer, PI = Primitive Integer

Lift (2×10^9 op calls)		ProB	Java BI	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	> 1800	156.63	27.43	78.42	0.00
	Speed-up	1	> 11.49	> 65.62	> 22.95	$> 180\,000$
	Memory	-	735 188	785 628	756	736
Traffic Light (1.8×10^9 op calls)		**ProB**	**Java BI**	**Java PI**	**C++ PI -O1**	**C++ PI -O2**
	Runtime	> 1800	47.04	9.05	69.09	0.00
	Speed-up	1	> 38.27	> 198.9	> 26.05	$> 180\,000$
	Memory	-	855 112	447 828	756	736
Sieve (1 op call, primes until 2 Million)		**ProB**	**Java BI**	**Java PI**	**C++ PI -O1**	**C++ PI -O2**
	Runtime	76.31	7.71	6.49	14.63	8.94
	Speed-up	1	9.9	11.76	5.22	8.54
	Memory	398 980	1 415 428	1 096 284	32 472	35 732
Scheduler (9.6×10^9 op calls)		**ProB**	**Java BI**	**Java PI**	**C++ PI -O1**	**C++ PI -O2**
	Runtime	786.74	10.62	10.49	21.57	10.32
	Speed-up	1	74.08	74.99	36.47	76.23
	Memory	5 341 316	414 772	398 924	816	820
sort_m2_data1000 [32] (500×10^3 op calls)		**ProB**	**Java BI**	**Java PI**	**C++ PI -O1**	**C++ PI -O2**
	Runtime	17.27	3.27	2.10	0.2	0.03
	Speed-up	1	5.28	8.22	86.35	575.67
	Memory	577 808	191 280	143 864	1192	1104
CAN Bus (J. Colley, 15×10^6 op calls)		**ProB**	**Java BI**	**Java PI**	**C++ PI -O1**	**C++ PI -O2**
	Runtime	273.58	7.23	6.81	7.23	2.91
	Speed-up	1	37.84	40.17	37.84	94.01
	Memory	167 284	428 084	402 432	968	952
Train [1] (940×10^3 op calls)		**ProB**	**Java BI**	**Java PI**	**C++ PI -O1**	**C++ PI -O2**
	Runtime	241.16	13.31	12.83	18.55	8.10
	Speed-up	1	18.11	18.8	13	29.77
	Memory	163 476	377 292	376 540	984	1016
Cruise Controller (136.1×10^6 op calls)		**ProB**	**Java BI**	**Java PI**	**C++ PI -O1**	**C++ PI -O2**
	Runtime	> 1800	21.26	15.26	11.90	0.30
	Speed-up	1	> 84.67	> 117.96	> 151.26	> 6000
	Memory	-	750 816	484 948	864	820

5 Related Work

Low-Level Code Generators There are a variety of code generators that work on a low-level subset of B or Event-B and emit low-level code. This includes the code generators in AtelierB [9], which emit C, low-level C++ or Ada code, B2LLVM [7], which generates the LLVM intermediate representation and jBTools [33] that generates low-level Java code. In contrast to B2Program, these code generators usually only support primitive integers, boolean values as well as enumerated sets which are translated to enums. Higher-level constructs are not supported in order to avoid run-time memory allocation. This also means that most B models cannot be translated without (several) additional refinement steps. Apart from B2LLVM, which can use the LLVM infrastructure in order to emit code in many programming languages, these code generators feature only a single output language.

Automatic Refiner AtelierB also provides an automatic refiner called BART [31] which can help perform data-refinement and makes it much easier to reach the B0 level. BART can be used to generate code for SIL4 components, as the refinement steps are still validated by AtelierB like regular refinement steps. BART, however, still requires user interaction and may require discharging of proof obligations. We also doubt that BART can be applied to the high-level models in our experiments (cf. Table 2).

Event-B Code Generators There are also code generators for Event-B to other programming languages. The code generators in the EB2ALL tool-set presented in [27] generate code from Event-B to Java, C, C++ and C#. Like the other code generators for B, these code generators only support a subset of Event-B at implementation level.
In contrast, the code generator EventB2Java presented in [8,32] generates B models from all abstraction levels to Java just like B2Program. While EventB2Java generates JML contracts additionally to Java code, B2Program requires a verified B model as input. JML supports quantified constructs, which is used by EventB2Java to generate code for B quantifications. In contrast to B2Program, EventB2Java does not support non-deterministic constructs and it uses mutable data structures (e.g., java.util.TreeSet). Note that the sorting example from Table 2 stems from [32].[6]
Another code generator for Event-B [11] generates Java, ADA and C for OpenMP and also C for the Functional Mock-up Interface. Code Generation depends on manual annotation of tasks via a Rodin plugin. This code generator also uses a template-based approach to store boilerplate code in templates and re-use them. B2Program does not only use templates to store boilerplate code only, but also to target various programming languages.
Finally, [13] is focused on extracting a scheduling of events from an Event-B model. It does not seem to be publicly available (see Section 3.6.2 of [26]).

Execution in ProB *and TLC* Another approach to utilise formal models in software is to skip code generation altogether and to directly execute the model using an animator, such as ProB and its Java API [22]. An example model and application is an implementation of the ETCS Hybrid Level 3 principle [16]. There, a Java application interacted with a non-deterministic model using input from different sources in order to calculate new signal states.
The TLC model checker also has library for TLA operators [23]. [15] provided a translation from B to TLA+ and has added some TLC libraries for B data types. The way TLC deals with quantification is reminiscent of Section 3.5. On the other hand, the translation provides limited support for composition and refinement and allows no sequential composition. The speed for lower-level

[6] But note that the timings reported in [32] are incorrect. In our experiments the EventB2Java generated code seems to be about twice as fast as execution with ProB, taking in the order of 8.15 seconds to sort 1000 elements. In [32] it is reported that sorting a 100,000 element array takes 0.023 seconds, and a 200,000 element array 0.028 seconds which is impossible using a quadratic insertion sort.

models of TLC is faster than ProB, but one cannot easily use TLC to execute a formal model.

Code Generators for other state-based formal methods [6] uses the Xtend technology provided by the Xtext framework for domain specific languages to generate C++ code for ASM formal models. This work [19], contains a code generator for VDM capable of producing Java and C++ code. There seem to be no code generators available for Z, see Section 3.9.2 of [26]. The commercial products Matlab/Simulink and Scade come with code generators, which are widely used (despite any guarantees of correctness).

6 Discussion, Conclusion and Future Work

In this paper, we presented B2Program, a code generator for high-level B models. Compared to existing work, more data types can be translated: e.g., sets, tuples, relations, sequences, records, many quantified constructs, and even some instances of non-determinism. Our code generator makes use of efficient immutable data structures to encode sets and relations.
B2Program does not cover the entirety of B. Table 1 shows all supported operators for the supported B types. Those constructs that would require constraint solving techniques to deal with, e.g. infinite sets, are intentionally not supported; we do not wish to embed a constraint solver into the generated code as explained in 3.5. Substitutions are covered by B2Program completely, except for *becomes such that* constructs that would require a constraint solver. For now, the only supported machine inclusion clauses are `INCLUDES` and `EXTENDS`. Supporting other machine inclusion clauses will be implemented in the future.
The generated code often runs one or two orders of magnitude faster than interpretation using ProB. For some benchmarks, making heavy use of set operations on large sets, the difference is less marked. Initial versions of our code generator used mutable data structures, and was in many cases slower than ProB. Representing relations as a set of tuples using persistent sets was still slower than ProB for the sorting example, possibly because operations on relations such the override operator were still inefficient. But in the current version of B2Program, the performance for operations on relations is improved by representing relations as a persistent map.
Another aspect that makes B2Program unique is the flexible output: using templates with StringTemplate provides the opportunity, to exploit similarities of programming languages. This results in code generation for several target languages with less effort. In the future, we want to explore code generation for other target languages, including declarative languages such as Haskell, Clojure and Prolog. Finally, B2Program might be used to rewrite B models by targeting the B language itself, potentially allowing optimisations for model checking.
To be able to use B2Program for SIL-3 or SIL-4 systems, the independent development of another high-level code generator would be required. But we hope that our code generator will already on its own enable new applications of

formal models, putting formal models into the loop and connecting them with other software components and controlling or monitoring systems in real time.

References

1. J. Abrial and A. Hoare. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 2005.
2. R. S. Alfred V. Aho, Monica S. Lam. The Structure of a Compiler. In *Compilers Principles, Techniques & Tools 2nd Edition.* Addison Wesley, 1986.
3. R. Backhouse and J. Gibbons. *Generic Programming.* Springer-Verlag, 2003.
4. P. Bagwell. Ideal Hash Trees. *Es Grands Champs*, 1195, 2001.
5. C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.
6. S. Bonfanti, M. Carissoni, A. Gargantini, and A. Mashkoor. Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino. In *Proceedings NFM 2017*, volume 10227 of *LNCS*, pages 295–301. Springer-Verlag, 2017.
7. R. Bonichon, D. Déharbe, T. Lecomte, and V. Medeiros Jr. LLVM-based code generation for B. In *Proceedings SBMF 2014*, volume 8941 of *LNCS*, pages 1–16. Springer-Verlag, 09 2014.
8. N. Cataño and V. Rivera. EventB2Java: A Code Generator for Event-B. In *Proceedings NFM 2016*, volume 9690 of *LNCS*, pages 166–171. Springer-Verlag, 2016.
9. ClearSy. *Atelier B, User and Reference Manuals.* Aix-en-Provence, France, 2016. Available at `http://www.atelierb.eu/`.
10. D. Dollé, D. Essamé, and J. Falampin. B dans le transport ferroviaire. L'expérience de Siemens Transportation Systems. *Technique et Science Informatiques*, 22(1):11–32, 2003.
11. A. Edmunds. Templates for Event-B Code Generation. In *Proceedings ABZ 2014*, volume 8477 of *LNCS*, pages 284–289, 2014.
12. D. Essamé and D. Dollé. B in Large Scale Projects: The Canarsie Line CBTC Experience. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, pages 252–254, Besancon, France, 2007. Springer-Verlag.
13. A. Fürst, T. S. Hoang, D. A. Basin, K. Desai, N. Sato, and K. Miyazaki. Code Generation for Event-B. In *Proceedings iFM 2014*, volume 8739 of *LNCS*, pages 323–338. Springer-Verlag.
14. T. Granlund. GNU MP. *The GNU Multiple Precision Arithmetic Library*, 2(2), 1996.
15. D. Hansen and M. Leuschel. Translating B to TLA + for Validation with TLC. In *Proceedings ABZ 2014*, volume 8477 of *LNCS*, pages 40–55. Springer-Verlag.
16. D. Hansen, M. Leuschel, D. Schneider, S. Krings, P. Körner, T. Naulin, N. Nayeri, and F. Skowron. Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains. In M. Butler, A. Raschke, T. S. Hoang, and K. Reichl, editors, *Proceedings ABZ 2018*, volume 10817 of *LNCS*, pages 292–306. Springer-Verlag.
17. R. Hickey. The Clojure programming language. In *Proceedings DLS.* ACM, 2008.
18. A. Hunt and D. Thomas. The Evils of Duplication. In *The Pragmatic Programmer: From Journeyman to Master*, page 26. The Pragmatic Bookshelf, 1999.
19. P. W. V. Jørgensen, M. Larsen, and L. D. Couto. A Code Generation Platform for VDM. In N. Battle and J. Fitzgerald, editors, *Proceedings of the 12th Overture Workshop.* School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446, January 2015.

20. H. Kaplan. Persistent Data Structures. In *Handbook of Data Structures and Applications.* 2004.
21. S. Krings. *Towards Infinite-State Symbolic Model Checking for B and Event-B.* PhD thesis, Heinrich Heine Universität Düsseldorf, Aug. 2017.
22. P. Körner, J. Bendisposto, J. Dunkelau, S. Krings, and M. Leuschel. Embedding High-Level Formal Specifications into Applications. In *Proceedings FM 2019*, volume 11800 of *LNCS*. Springer-Verlag.
23. L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers.* Addison-Wesley Longman Publishing Co., Inc., 2002.
24. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, POPL, pages 42–54. ACM Press, 2006.
25. M. Leuschel and M. Butler. ProB: A Model Checker for B. In A. Keijiro, S. Gnesi, and M. Dino, editors, *FME*, volume 2805 of *LNCS*, pages 855–874. Springer-Verlag, 2003.
26. A. Mashkoor, F. Kossak, and A. Egyed. Evaluating the suitability of state-based formal methods for industrial deployment. *Softw., Pract. Exper.*, 48(12):2350–2379, 2018.
27. D. Méry and N. K. Singh. Automatic code generation from event-B models. In *Proceedings SoICT 2011*, pages 179–188. ACM ICPS.
28. T. Parr. Enforcing Strict Model-View Separation in Template Engines. `https://www.cs.usfca.edu/~parrt/papers/mvc.templates.pdf`. Accessed: 2019-05-14.
29. D. Plagge and M. Leuschel. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. *International Journal on Software Tools for Technology Transfer*, 12:9–21, 01 2007.
30. J. P. B. Puente. Persistence for the Masses: RRB-vectors in a Systems Language. *Proc. ACM Program. Lang.*, 1(ICFP):16:1–16:28, 2017.
31. A. Requet. BART: A tool for automatic refinement. In *Proceedings ABZ 2008*, volume 5238 of *LNCS*, page 345. Spirnger-Verlag.
32. V. Rivera, N. Cataño, T. Wahls, and C. Rueda. Code generation for Event-B. *STTT*, 19(1):31–52, 2017.
33. J.-C. Voisinet. JBTools: an experimental platform for the formal B method. *Proceedings of the naugural International Symposium on Principles and Practice of Programming in Java*, pages 137–139, 01 2002.