Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 99

Serialization of Foreign Types with SKilL

Constantin Michael Weißer

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Dipl.-Inf. Timm Felden
Beginn am:	April 19, 2016
Beendet am:	October 19, 2016
CR-Nummer:	D.2.12, D.3.4

Abstract

SKilL provides a language independent means to specify serialisable data types. Bindings for these types can be generated automatically for multiple supported languages based on this specification. However, using these bindings is only hassle-free for newly written code, because existing types must be replaced by the controlled generated bindings. This impedes the usefulness of SKilL in old projects.

We investigate a new approach. SKilL is to be extended in order to support the reuse of existing Java classes, so-called *foreign types*. The tool shall be able to analyse classes, associate them with specified SKilL types, verify the type correctness of this association and generate the required code in order to serialise objects of these types. This thesis points out the occurring challenges and discusses potential solutions. We experiment with several alternatives and provide insight into their pros and cons, as well as justification for our design. Functional and performance tests assess our implementation and shortcomings are addressed in detail.

Kurzfassung

SKilL bietet sprachunabhängige Mittel um serialisierbare Datentypen zu spezifizieren. Anbindungen für diese Typen in verschiedenen Sprachen können automatisch mithilfe eines Werkzeugs generiert werden. Allerdings können diese generierten Datentypen nur in Neuentwicklungen mühelos benutzt werden, da in Bestandsprojekten die existierenden Klassen durch die generierten ausgetauscht werden müssten. Dies beeinträchtigt die Nützlichkeit SKilLs für ältere Projekte.

In dieser Masterarbeit soll ein neuer Ansatz untersucht werden. Das SKilL-Werkzeug soll erweitert werden, sodass es bestehende Java-Klassen wiederverwenden kann. Solche Klassen werden *fremde Typen* (engl. *foreign types*) genannt. Das Werkzeug soll in der Lage sein, Klassen zu analysieren, diese mit SKilL-Typen zu assoziieren, die Typkorrektheit dieser Beziehung nachzuprüfen und schließlich den nötigen Code zu generieren, damit Objekte dieser fremden Typen serialisert werden können. In dieser Arbeit werden die auftretenden Probleme aufgezeigt und mögliche Lösungen diskutiert. Es werden mehrere Lösungsansätze ausprobiert um deren Vor- und Nachteile zu ermitteln. Die anschließenden Designentscheidungen werden ausführlich begründet. Funktionale Tests und Leistungstests beurteilen die Implementierung und auftretende Mängel werden detailliert diskutiert.

Contents

1 Introduction

Many software projects contain source code in different languages. This can have a variety of reasons. The developers might want to include a useful library written in another language or exploit certain strengths and advantages of a particular language. For example, software analysis tools often exhibit multiple processing steps. It might be beneficial to reuse an existing analysis on the intermediate results of such a step, even though this analysis is written in another language. It might even be useful, to develop analysis in a specific language to exploit its strengths in terms of fast prototyping, for example.

SKilL is a domain-specific language that provides the means to specify types independently of any programming language [Fel13]. The SKilL tool parses such a specification and then generates bindings for any of the supported target languages. Along with the bindings, the tool also generates code for serialisation. Objects of these generated types can be translated into a specified binary format and back again. The binary format “does encode the type system and can therefore provide maximum of upward and downward compatibility” [Fel13].

1.1 Motivation

The use of SKilL promises many merits. Types can be specified before the implementation languages for the software are known. The types can be changed over time without breaking compatibility with existing programs, making SKilL very flexible. The serialisation and deserialisation method provides “high decoding and encoding speeds” [Fel13]. However, working with existing code-bases is not supported automatically. If the user wants to benefit from SKilL’s advantages, they must change existing code and use the generated types. In large projects, this can be a tedious and time-consuming task.

In this thesis, we experiment with a new approach. We extend the SKilL tool by support for *foreign types* in Java. Foreign types are already part of the user’s code-base and need not be generated by SKilL. The SKilL tool is improved in order to recognise Java classes, analyse their structure and subsequently generate the appropriate serialisation code. Hence, the foreign classes act as the language bindings.

Serialisation of foreign types in general can have several advantages over using the conventional generated types:

- Existing software without SKilL bindings can become a "SKilL-supported" project. The original code-base can be maintained as before and new languages can easily be used within an old project.
- The architectural design of a program can be maintained, as foreign types are allowed to be located in different packages or even libraries.
- Types can also be serialised even though the user cannot access their source code.
- Foreign types support may facilitate experimenting with SKilL in a real project. Getting started might be easier and less of a burden. This can increase the attractiveness of SKilL over its competitors.

1.2 Task Description

The following tasks are to be accomplished in this thesis:

1. Develop a method to map SKilL type definitions to Java classes, such that Java data can be serialised and deserialised as SKilL data.
2. The type correctness of the specified mapping must be guaranteed by the implementation.
3. The implementation must be able to access fields via getter and setter methods as well as directly.
4. If necessary, the state management must be revised in order to compensate for the lack of certain SKilL-specific fields, such as the internal "skillID".

1.3 Contributions

We contribute the following enhancements and related findings in order to solve the tasks described above:

- A very simple and extendible domain-specific language to define mappings between types of a SKilL specification and Java classes. The language aims to minimise the manual work of the user. This solves the task 1 mentioned above.

- A fast type checker which is able to detect semantic errors within the specified mapping and inform the user about those errors. This solves task 2.
- A class file analyser which translates Java classes into SKilL's internal type system. It also records properties of the Java classes in order to generate appropriate serialisation code, such as direct field accesses or the use of existing getter and setter methods. This solves task 3.
- A new code generator, that produces serialisation code concerted to operate with foreign types. It also generates AspectJ code to inject the missing SKilL ID and other required properties into foreign classes during compilation. This approach promises to minimise runtime overhead and maintain high serialisation and deserialisation speeds. This solves task 4.
- We introduce a new way to mark entire object graphs for serialisation at once. This aims to minimise the programming effort for the user, when adopting SKilL serialisation.
- We provide an extensive report about the success of our approach by means of a realistic test case. We explain the reasons of shortcomings and how they might be solved.
- We conduct performance tests to show the high performance of our extension and compare the results with the performance of generated SKilL types.

1.4 Basic Principles

In this section, we briefly explain the basis for this thesis. We name the relevant technologies that are used, define some basic terms and names and describe the assumed restrictions under which our implementation operates.

1.4.1 SKilL

As mentioned shortly in the introduction, *SKilL* designates a specification language for types. The implementation of SKilL parses and interprets the language and generates bindings for the types in requested target programming languages. The features of the SKilL language are described in [Fel13]. For simplicity, throughout this thesis we use the term *SKilL* for both the language and the software tool. Figure 1.1 shows a short example for the conventional use of SKilL. Besides these generated bindings, the tool also produces management and serialisation code.

```
SimpleType {
   i32 anInt;
   string aString;
   list<string> listOfStrings;
}
```

```
public class SimpleType extends SkillObject {
   private static final long
       serialVersionUID = 0x5c11L + ((long)
       "simpletype".hashCode()) << 32;

   protected int anInt = 0;

   protected java.lang.String aString = null;

   protected
       java.util.LinkedList<java.lang.String>
       listOfStrings = null;

   // ...
}
```

Figure 1.1: A very simple example for the use of SKilL. The left shows a SKilL specification and the right the resulting Java binding. All methods were removed for brevity.

1.4.2 Terms and Special Names

In this section we list some terms and names which are used in this thesis. This is not a complete reference but merely a clarification for the meaning of these terms.

Java[1] is a general-purpose programming language developed by *Sun Microsystems* which is part of the *Oracle Corporation*. It is a registered trademark of Oracle[2]. The Java compiler (`javac`) translates Java source code into class files. The class file's binary format is called *bytecode*. The Java Virtual Machine (*jvm*) is part of the Java platform and executes this bytecode.

AspectJ is a language extension for Java, which provides aspect-oriented programming features. It originates from the *Paolo Alto Research Center Incorporated* and is part of the Eclipse Project[3]. We cover the relevant AspectJ features in Section 1.4.3.

Types and Java classes which are provided by the user and are to be serialised are called *foreign types*, because they are foreign to the SKilL tool. In contrast, we call the language bindings produced by SKilL *generated types*. The term *container types* refers to the three

[1] https://www.oracle.com/java/index.html

[2] https://www.oracle.com/legal/trademarks.html

[3] http://www.eclipse.org/aspectj/

container types supported by SKilL, namely *list*[4], *set*[5] and *map*[6]. *Type context* refers to a closed set of related types within SKilL's type system.

1.4.3 AspectJ

AspectJ features aspect-oriented programming (AOP) in Java. In AOP, the developer can define cross-cutting behaviour which may influence many parts of the code, without manipulating the code itself. Thus, cross-cutting concerns can be described once in a central place rather than repeatedly throughout the code-base.

An *aspect* is a grouping for several such concerns. Its structure is similar to a Java class. It is introduced by the keyword `aspect` and a name. E.g. `public aspect MyAspect`, followed by a block in curly braces. Within the braces, there can be two kinds of definitions: *inter-type declaration* and *advice*.

Inter-type declarations allow for structural changes in other classes. It is possible to declare fields, add implemented interfaces or set parent classes. It is also possible to define new methods. For example, the declaration `public long SomeClass.x;` adds a public field of type `long` to `SomeClass`. Whereas `public long SomeClass.getX() { return this.x; }` adds a method to `SomeClass` with the given name and body.

A *join point* is a position within the control-flow of a program. A *pointcut* is a set of such join points. With AspectJ the user has the means to describe pointcuts in order to select a subset of join points. The description is declarative. It can involve the name and signature of functions as well as certain stages during a method call, such as *call* when the call itself takes place or *execution* when the body is executed, i.e. after the call mechanism is completed. For example "`target(Point) && call(int *())`" means "any call to an int method with no arguments on an instance of Point, regardless of its name" [Xer03]. An *advice* is a block of code, similar to a method body which can be executed whenever a join point of a specified pointcut is reached.

A scientific overview of AspectJ can be found in [KHH+01]. Technical implementation details are described in [HH04]. A complete programming guide can be found online in [Xer03].

[4] https://en.wikipedia.org/wiki/List_(abstract_data_type)
[5] https://en.wikipedia.org/wiki/Set_(abstract_data_type)
[6] https://en.wikipedia.org/wiki/Associative_array

1.5 Assumptions and Restrictions

In real-life, there is a vast amount of scenarios that SKilL might be confronted with. In order to keep the complexity at a manageable level, we make several assumptions about the use cases. The assumptions cover properties of the Java code and classes at hand, the build process and the scenarios in which SKilL with foreign types might be used. Altogether, the assumptions pose some restrictions on the use cases and input. Generally speaking, we strive for the goal:

> SKilL shall support all properties and constructs in *foreign types* if they can also be produced in *generated types* by an appropriate SKilL specification. SKilL may support additional constructs, only if they can be mapped properly to SKilL's internal model. This implies that for any foreign type, SKilL can generate compatible bindings in all supported languages, including Java itself.

1.5.1 Class Files, Build Process

We assume, that the user has full access to the class files of the types which are to be serialized. As the analysing module only reads class files for analysis, we do not require the source files to be available. Furthermore, we assume, that the build process can be controlled and modified by the user. The code generator will produce Java and AspectJ code as output. This code must be compiled using the AspectJ compiler in place of *javac*. The aspects will induce the compiler to weave the foreign types' class files. It is possible to provide either the original source files or the Java class files as input to the AspectJ compiler. Since class files are always available, this provides maximum flexibility for foreign types support.

1.5.2 Constructors

The deserialisation code must create actual in-memory objects of the serialised data. This involves instantiating foreign types in Java using the "**new**" keyword. This, in turn, calls a constructor of the respective class, depending on the types of the provided arguments. The deserialisation code calls the *default constructor*. If the user does not define any constructor at all, Java implicitly adds a default constructor. However, if the user adds customised constructors the default constructor is not added automatically. It must be added explicitly in the source code. We assume the following properties about constructors:

- If the default constructor is available, either added implicitly by Java or defined explicitly by the user, we assume that deserialisation can use this constructor safely. In particular, we assume that using the default constructor will leave the object in a valid state. This way, deserialisation will return usable and valid objects. SKilL will not modify existing default constructors in any way.
- If the public default constructor is not available, we assume that an aspect may add one safely. That is, there must not be any conflicting definitions, such as private or protected constructors with the same signature. The added constructor will be empty, except for a call to *super()*.

1.5.3 Generics

SKilL does not know generic types, except for some basic container types as described in Section 1.5.4. Consequently, generic types in Java cannot be mapped properly to SKilL's type system. In foreign types, generic classes and fields are permitted, however when mapped to SKilL types, the generic information will be lost. If a field f is of type T, where T is a type variable, the type of f cannot be determined statically. SKilL can only assume that f will be of type `Object`. However, `Object` does not carry any information worth serialising. We ignore fields in this case, even though they could reference a known foreign type at runtime.

1.5.4 Container Types

In SKilL, `list`, `set` and `map` are called container types. We associate the `List`, `Set` and `Map` interfaces from the package `java.util` with SKilL's container types respectively. We allow them to be generic, because they can be properly mapped to SKilL's type system.

Foreign types may contain fields with container types. That is, fields may be of any type that implements either `List`, `Set` or `Map`. SKilL's type system does not allow for nested lists or sets. Consequently, our extension also refuses nested `Lists` or `Sets` in foreign types.

However, SKilL provides maps with variable amount of type arguments, i.e. two or more. They act as a right-associative chain of mappings. Say $T \rightarrow U$ is the map type that maps elements of type T to one element of type U each. Then, the SKilL type `map<T, U>` is $T \rightarrow U$. Say further that chains of mappings are right-associative. Then the SKilL type `map<T, U, V>` becomes $T \rightarrow (U \rightarrow V)$.

In order to maintain compatibility with generated types, we only allow right-associatively nested maps in foreign types. That is, while the Java type

```
HashMap<Integer, HashMap<Integer, Integer> >
```

is legal,

```
HashMap<HashMap<Integer, Integer>, Intger>
```

is not. Furthermore, the following examples will be refused by our foreign types extension as well:

```
Set<Set<Integer> >
List<Set<Integer> >
HashMap<Integer, List<Integer> >
```

1.5.5 Interface Types

At the time when this thesis started, SKilL's interfaces were not mapped to Java interfaces but resolved before the generation. That is, the fields in SKilL interfaces were added to every implementing type but no code containing a Java interface was generated. Therefore, it did not seem reasonable at the time, to map Java interfaces back to SKilL interfaces in the context of foreign types. Because Java interfaces do not carry fields, we ignore them. However, this restricts the foreign types, because we cannot serialise a field which has an interface type. After recent advancements in SKilL, it would now be possible to map Java interfaces to SKilL interfaces and the restriction could likely be abolished.

1.5.6 Serialization of Standard types

As the serialisation of foreign types relies on compile-time class file weaving, SKilL cannot serialise the types from the standard library (with the only exception of the aforementioned container types).

The standard library is shipped with a Java implementation in form of the "rt.jar"-file. It is a regular jar-archive. However, Java's class loader treats this library differently. Firstly, it is not looked up in the usual class path. The archive normally resides within the directory of the Java installation. Thus, the class loader does not expect it to be in the current application's path. It is rather looked up from the "boot strap class path". The Java SE documentation [Orac] by Oracle states: "It is very unlikely that you will need to redefine the bootstrap class path. The nonstandard option, -Xbootclasspath, allows you to do so in those rare cicrcumstances in which it is necessary to use a different set of core classes." OpenJDK's `java` command also provides this option, but other implementations might not.

In case the user wants to serialise classes from the standard library using SKilL's foreign types extension, they can weave the class files in the shipped `rt.jar`[7]. The altered class files must be packaged separately from the application's "regular" class files. The Java Virtual Machine must load classes from the standard library before it loads any other class. E.g. since all types inherit from `Object`, it must be loaded and available when any other type's class file is read. Consequently, the custom `rt.jar` containing the weaved standard types must be loaded from the redirected bootstrap class path.

While not entirely impossible, it is a rather cumbersome way of serializing classes from the standard library. We assume that most of the time the user wants to serialise their own classes. Consequently, we make a compromise which favours the most frequent use cases.

1.5.7 Type Correspondence

In case that types from a SKilL specification are to be associated with Java classes, we require that field mappings are a one-to-one relationship. Every SKilL or Java field can participate in zero or one mapping. That is, each SKilL field may be mapped once or not at all. Zero or one SKilL field can be mapped to one Java field. These two restrictions arise naturally. If a SKilL field is mapped twice, during deserialisation the value of the SKilL field would be stored in two Java fields. Consequently, they hold identical values. However, during serialisation, there is no general way to determine which of the two fields to serialise and store as the SKilL field, if the two Java fields hold different values. The analogous problem occurs during *deserialisation*, if multiple SKilL fields are mapped to one Java field. In the same way, SKilL types can only be mapped in a one-to-one relationship to Java types.

[7] AspectJ conveniently allows for jar-files as input as well.

2 Implementation

A conventional execution of SKilL mainly requires a SKilL specification as input. The specification describes the types for which SKilL shall generate both language-specific bindings and the serialisation code. An execution of SKilL for *foreign types* in Java takes the following input:

- A Java code base that contains said types.
- A *mapping* that describes which Java types are to be serialised. This mapping may also state how a type relates to a type of a SKilL specification.
- A SKilL specification, which may be empty.

Evidently, when generating serialisation code for foreign types, several processing steps are quite similar or even identical, despite the different nature of the user input. Thus, these extensions to the SKilL tool are naturally implemented based on SKilL's code base. We reuse as much of the tested code as possible to preserve maintainability.

This chapter first describes the architecture and how it integrates with SKilL's existing architecture. Subsequently we detail on the single processing steps. We focus on the challenges and their proposed solutions in contrast to some alternatives.

2.1 Architecture

Figure 2.1 shows a rough overview of the architecture. The *parser* analyses a SKilL specification, which is basically a description of types. The result is an *intermediate representation (IR)* of the types. Subsequently, the IR is passed to all *code generators* of requested output languages. That is, depending on the requested output languages, multiple code generators may be invoked. They all operate on the same IR as input. A code generator produces the necessary code for a particular language. That includes, for

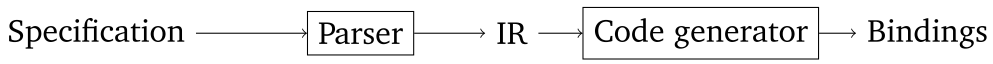

Figure 2.1: Rough overview of the SKilL architecture. Arrows indicate information flow.

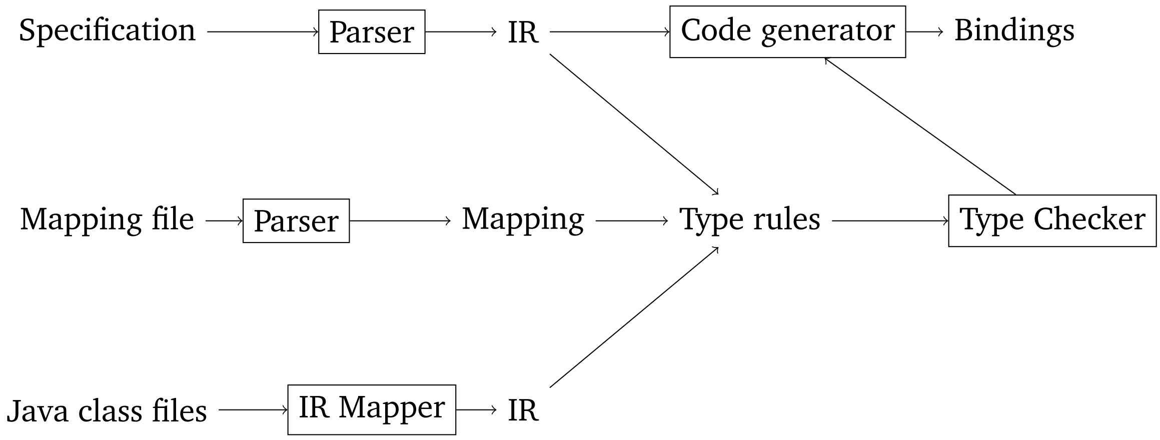

Figure 2.2: Overview of the extension's impact on SKilL's architecture. Arrows indicate information flow. The top line of the original architecture remains unchanged.

conventional generated types, the language bindings for the types, a reflection interface for client code and administrative code that handles serialisation, deserialisation and the memory management.

The foreign types extension must leave this existing architecture unaltered. The SKilL language specification [Fel13] defines the valid input to SKilL and extending upon it must not break compatibility with any previously valid input. Foreign types ought to interoperate seamlessly with generated types, as long as there exists a compatible mapping (see Section 2.5). This implies, that foreign types must be representable in SKilL's type system. If there are foreign types which SKilL's type system cannot represent, it would not be able to generate bindings for those types. This contradicts the requirement for interoperability, as one could not deserialise data in any other language.

As depicted in Figure 2.2, SKilL's original architecture remains unaltered. For "conventional" invocations of SKilL the user can provide the same input as before. In order to use the foreign types extension, the user must additionally provide Java class files of the types which are to be serialised and a mapping file as input. The *IR Mapper* translates a list of Java classes into the intermediate representation. This process is described in detail in Section 2.2. A second parser translates mapping files into an internal representation of mappings. Such a mapping merely consists of two strings which represent names of SKilL and Java types. During a binding step, the names are translated to type objects of the intermediate representation. The output of this step is a set of type rules. The rules are used to determine the correctness of a mapping. The mapping specification, name binding and type safety are explained in detail in Section 2.3, Section 2.4 and Section 2.5.

Language bindings generated by SKilL exhibit some properties that foreign types do not. In particular, the user types do not inherit from the known Java class `SkillObject` as the generated types do. However, SKilL's serialisation code relies heavily on the fact, that all objects possess a field `skillID`. Section 2.6 describes how SKilL's code generator for Java needs to be adapted in general to produce code which interoperate with foreign types. Section 2.7 explains how the SKilL ID is added to foreign classes without interfering with the user's source code. In Section 2.8 we present an approach to facilitate the adoption of SKilL in old projects by marking entire graphs of foreign type objects for serialisation.

2.2 Analysing Java Class Files

SKilL requires information about foreign types for several reasons:

1. The generators for serialisation and management code needs to know all serialisable types and their fields' types.
2. Foreign types that ought to be compatible with generated SKilL types must be type checked. The type checker has to verify whether a serialised foreign type can be deserialised into a generated type and vice versa.
3. SKilL must verify if the foreign type is compatible with SKilL's type system. SKilL poses several restrictions on types, such as limited nesting of containers (see Section 1.5.4). If a foreign type uses types that cannot be generated by SKilL, they must be refused.

Assuming that the Java source code is available, SKilL has essentially two possibilities to analyse existing types: analyse the source code itself or analyse class files. We decided to analyse class files for several reasons:

- Class files are always available, even if the source code is not.
- There is a plenitude of Java class file libraries freely available.
- Class files are more robust against version changes in the language. I.e. many language updates involve syntax changes or extensions but do not affect the layout in the class files. Even if they change, well-maintained class file libraries would be updated promptly.

Any foreign Java type must be compatible with conventional generated types. Thus, we are able to reuse SKilL's IR to represent Java types internally. We extend SKilL by the module *IR Mapper*. It takes a list of class paths and a list of Java (fully qualified) class names as input and produces the respective IR representation for each of these classes as output.

From an architectural point of view, the IR Mapper should be able to map *any* class to the IR. If a class is not compatible with SKilL's restrictions, only the type checker should decline such classes in a later processing step. These restrictions include nesting of containers: SKilL does not allow lists of lists, or lists of sets for example. Merely maps of maps are allowed, if the value type of the outer map is a map type itself. However, SKilL's IR is not able to model those invalid type combinations. In order to keep code changes to a minimum, we allow the IR Mapper itself to refuse those invalid type combinations. This slight break in the architecture prevents a lot of code changes that could potentially introduce many bugs.

2.2.1 Principle of Operation

We use *javassist*[1] to dissect class files and extract the essential type information. *javassist* provides a simple API to lookup, load and analyse Java class files. A `ClassPool` can be configured with a classpath, which specifies from which locations class files will be loaded. With help of the `ClassPool`, classes can be loaded simply by calling `get()` with their name as argument. For each loaded class, the *javassist* API instantiates a `CtClass` object. This object acts as a convenient wrapper for Java's reflection API.

Now that the IR Mapper can load a reflective representation of any Java class, it can start mapping them to SKilL's IR. That is, the `CtClass` reflection objects are to be translated into type objects of the IR. This translation may be partial. Classes may contain fields with types which are unknown to SKilL. Such fields cannot be considered for serialisation. Thus, they will be left out.

The IR Mapper performs the translation of a Java type T_j into SKilL type T_s in two basic steps:

1. Collect all explicitly requested types and the transitive closure over the parent type relation.
2. Translate all types to SKilL types. For a type T this includes translating parent types first recursively and translating all field types of T's fields before adding the result of T's translation to the set of translated types.

[1] http://jboss-javassist.github.io/javassist/

In the first step the types are only collected, not translated yet. “Collecting” a type T_j means to create an empty prototype T_s in the IR as representative for T_j. Then, the pair (T_j, T_s) is added to a map of *known types*. This map links a `CtClass` object with its IR representation. Whenever the IR Mapper is asked to map T_j again (for example if a field with this type occurs), it will simply retrieve T_s from the map. This can also safely be done when T_s is yet incomplete.

After collecting a type, the algorithm collects the parent type recursively, until it reaches Java’s top type `Object`. Thus, after the first step, all user types are created as empty prototypes and their relationship to the original Java class is stored within a map. The remaining (not collected) types include containers, arrays and Java’s primitive standard types.

This dedicated collecting step breaks up possible cycles. Say there are the two types T and its parent type T_p. Now assume T_p has a field with type T. If T is translated first, its parent type cannot be set, because it yet has to be translated. If then T_p is translated, its field type T is not translated yet and missing. Collecting all types in the hierarchy will ensure that for each type there is at least a prototype available during the translation step.
Furthermore, the transitive walk along the parent references are a convenient way to add new types in the correct order. In SKilL’s intermediate representation, the types’ order is meaningful. The code generators produce a SKilL-specific reflection interface for all types. In this interface, any type’s reflection object also contains a reference to the parent’s type reflection object. As they are instantiated one after the other, parent classes must be instantiated first, so that the child class can obtain the respective reference for the inheritance relationship. By recursively collecting parent objects first, this order is produced naturally. Translating the types directly could destroy the order due to the translation of field types.

The second step actually translates a type T_j to T_s, which means the empty prototype is filled with fields and possibly the parent type is set, if any. In order to add the fields to T_s, the IR Mapper iterates over T_j’s fields and checks the map with known types for each field type. If a type is not found in the map, it is either a primitive type, a container type or the type is a missing user type. The latter case will cause the field to be omitted in the translation. Primitive types are translated to SKilL’s primitive types, which match the ones available in Java. Container types are handled separately. Afterwards T_s is an equivalent representation of T_j in SKilL.

```
HashMap<Integer, HashMap<Integer, Integer>> maps;

Ljava/util/HashMap;

Ljava/util/HashMap<Ljava/lang/Integer;
    Ljava/util/HashMap<Ljava/lang/Integer;Ljava/lang/Integer;>;>;
```

Figure 2.3: From top to bottom: a field declaration in Java, its non-generic type descriptor and its generic type signature as found in a Java class file. The line break is only added for formatting purposes.

2.2.2 Container field types

SKilL knows three container types: lists, sets and maps. These types are all generic in nature and accept one or in the case of maps two or more type arguments. Evidently, container type fields occur frequently in real-life software. SKilL can serialise containers and thus it must support the such in foreign types as well. In generated types, the code generator can pick which concrete container implementation is used. For example, lists are mapped to `LinkedList`, whereas SKilL's map type is implemented by the `HashMap` class in Java. However, in foreign types, containers can be much more diverse. The user might have picked any of the classes that implement the interfaces from Java's standard library: `List`, `Set` and `Map`[2]. This includes both the default implementations from Java's standard library and any custom implementation by a third-party or the user. For foreign types, we allow any class as a container type, as long as it implements the respective interface. See Section 1.5.4 for details on restrictions for container types.

In order to maintain type safety in the presence of container types, we must analyse and interpret the generic type arguments of such fields. Recall, that we analyse class files rather than Java source code. Thus, the IR Mapper must analyse the type signatures encoded in the Java bytecode. "Signatures are used to encode Java programming language type information that is not part of the Java Virtual Machine type system, such as generic type and method declarations and parameterised types." [LYBB13]. Section 4.3.4 in the jvm specification describes in detail how signatures are constructed from a set of formal grammar rules.

Figure 2.3 shows a generic signature of a field of type `HashMap`. Here, the second type argument (i.e. the value type in the map) is in turn a `HashMap`. It is obvious that in such a case the generic part can be syntactically nested multiple times. The given example is

[2] these can be found in package `java.util`

the only valid nesting of two maps which can be mapped to SKilL's map type. Obviously, the nesting can be much deeper, as long as it is right associative. That is, the value type of a Map can be another Map, but the key type cannot. The IR Mapper must verify that the combination of container types is valid. Thus, the entire generic signature must be analysed. The signature in Figure 2.3 is equivalent to the output of the Java code generator for the SKilL type:

```
map<i32, i32, i32>;
```

Say the function m maps any SKilL user type to its corresponding Java class and any primitive SKilL type to its respective Java primitive type. We extend the definition of m to clarify how SKilL's map corresponds to Java's `Map`:

Let m be defined for SKilL types $T_1, T_2, \ldots, T_n, n \geq 2$.

Then $m(map < T_1, T_2, \ldots, T_n >) :=$

$Map < m(T_1), Map < \ldots, Map < m(T_{n-2}), Map < m(T_{n-1}), m(T_n) >> \cdots >>$

We use Sun's SignatureParser[3] to analyse the generic signature. The parser is run on the generic signature and returns a tree representing the dissected signature. We define our own custom signature visitor[4]. The visitor then analyses the field's generic type and maps it to an equivalent SKilL type. If unsupported combinations are detected (such as nested lists or sets or illegally nested maps) the visitor will emit an error. The exact type of the container is not determined. The visitor rather checks if the field type implements the `List`, `Set` or `Map` interface and maps the type to the its representative in the SKilL type system. That way our extension can provide the largest degree of freedom for foreign types. The container must merely implement the respective interface but can otherwise be chosen freely by the user.

2.2.3 The Reflection Context

In SKilL's original architecture, the IR is the major means of communication between the front-end and the code generator. A specification file is parsed and translated into the IR which is then passed to the code generators. We strive to keep changes to the IR minimal, because it constitutes the common base for all languages. Language specific properties should not pollute the architecture too much. However, our code generator

[3] can be found in package sun.reflect.generics.parser.SignatureParser and is shipped with Java's runtime.

[4] implementing the abstract `sun.reflect.generics.visitor.Visitor<T>`.

needs additional information which is not represented in the IR. E.g. for container types, it must determine the actual used type.

Say, the user declared a field of type `TreeMap<String, Integer>`. The IR Mapper will translate this type into the type context as `map<string, i32>`. When SKilL deserialises data, it must instantiate a `TreeMap`, because the resulting object is to be stored in the field of type `TreeMap` again. Simply instantiating a "default implementation" such as `HashMap` would lead to incorrect code which cannot be compiled. Hence, the IR Mapper must inform the code generator about a field's concrete type. The IR does not provide means to store such information.
Furthermore, the code generators for foreign types must also be able to determine if a field can be accessed directly or only via getter/setter methods. Again, it is not possible to store such a property in the IR.

Therefore, in addition to the *type context*, we introduce a *reflection context*. It mostly consists of two maps:

- A map from IR types to their reflection object.
- Another map from IR fields to the field type's reflection object.

This context is passed along to our code generator. With help of the two maps, it can determine the original type of fields (such as concrete container implementations) and generate field accessing code by directly accessing public fields or using getter/setter methods, whichever is appropriate.

2.3 Mapping

There are several cases when a user might want to serialise existing types. Firstly, the user might have created a Java code base and, of late, wants to send data back and forth to another programming language. In this case the user could simply use the existing Java code base, analyse it with SKilL and derive the matching SKilL specification. With the help of this specification, they now can generate bindings for other languages supported by SKilL. In this case the specification is derived from the Java code base. The Java code is restricted by SKilL's general assumptions, but the mapping itself poses no additional restrictions. I.e. there is no forced correspondence between certain types or fields. We call foreign types which are not bound to any SKilL specification *unbound* foreign types.

If the user derives the SKilL specification automatically, they might also want to preserve it. While the derived specification can be reused directly by other code generators in form of the internal representation, SKilL can also write out the specification to a plain text file. This file can be parsed by the SKilL specification parser again to recreate the same type context at a later point.

Sometimes, there might be a preexisting SKilL specification and a Java code base that matches the specification completely or partially. In that case, the user might want to serialise those data types without changing the code base. Compliance with the SKilL specification, however, poses some restrictions on the serialised data. E.g. the naming of the serialised types must match the names in the SKilL specification. If neither the specification nor the code base should be changed, the generated code must translate the Java names to SKilL names during serialisation, and back again during deserialisation.

Furthermore, in case of differing naming between a SKilL specification and a set of Java classes, it is not guaranteed that the correspondence between types and fields can be determined automatically. E.g. if a type contains two fields of integer type, there are two possible mappings, which would be correctly typed. It could be possible to guess the intended mapping automatically using the fields' names. However, such an automatic guessing can always fail if none of the predefined patterns are used for naming. Moreover, it could be that the guessed correspondence is simply not the intended one.

In short, the user must have a means to instruct SKilL on how to tie a foreign type to a specification. We call this instruction a *mapping* and it is provided via a mapping file. We extended the command line interface by the `-OJavaForeign:M` option, to specify the path to the mapping file. This option has no effect on any other of SKilL's language generators.

In the previous section, we described how a set of Java types is translated to the core data structure of SKilL's type system, the internal representation of types (IR). After this step, there are two independent type contexts. One stems from the compiled specification file. The other is produced by the IR Mapper and represents the Java classes. The mapping is the user's tool to connect both worlds. They can freely pick which types on both sides are connected and correspond to each other (*bound mappings*), which types shall be ignored entirely and which Java types must be serialised without corresponding SKilL equivalent (*unbound mappings*). In the next sections we first describe the general structure of a mapping file, followed by different variants of *bound* and *unbound* mappings. We parse the mapping file and translate it into our internal representation. It is then bound to actual types in the SKilL and Java type contexts. We describe this process and its consequences in Section 2.4.

2.3.1 General Structure

A mapping file contains a series of type mappings. Type mappings can be of various natures, as explained below. The order of the mappings is irrelevant. As Java type names

⟨*mapping-file*⟩ ::= ⟨*mapping*⟩*

⟨*mapping*⟩ ::= ⟨*explicit-mapping*⟩ | ⟨*unbound-mapping*⟩ | ⟨*unbound-total-mapping*⟩

Figure 2.4: Syntax of a mapping file in EBNF.

```
map Point3D -> my.package.Point {
   x -> coordX;
   y -> coordY;
   z -> coordZ;
}

map Line -> my.package.Line {
   start -> startPoint;
   end -> endPoint;
}
```

Figure 2.5: Example for a simple mapping file. We assume two SKilL types and two corresponding Java types.

are case sensitive, the mapping file is also case sensitive. Figure 2.4 gives an overview of the mapping file's structure. The missing definitions are explained in the following sections.

2.3.2 Explicit mapping

Explicit mappings state the name of both the SKilL and Java type and the names for each pair of corresponding fields. An explicit mapping is introduced by the keyword **map**. We describe the syntax in EBNF as seen in Figure 2.6.

⟨*explicit-mapping*⟩ ::= '`map`' ⟨*skill-name*⟩ '->' ⟨*java-name*⟩ '{' ⟨*field-mapping*⟩* '}'

⟨*field-mapping*⟩ ::= ⟨*name*⟩ '->' ⟨*name*⟩ ';'

Figure 2.6: EBNF excerpt for explicit type mappings. The definitions for the non-terminals `name`, `java-name` and `skill-name` are left out. These rules produce valid names as expected. Note, that Java names include '.' because they are fully qualified.

We use '->' as "mapping operator" to emphasise, that mappings are directed: the left side is a SKilL type and the right side a Java type. Similarly for field mappings. Every SKilL or Java field can participate in zero or one mapping. That is, each SKilL field may be mapped once or not at all. Zero or one SKilL field can be mapped to one Java field.

These two restrictions arise naturally. If a SKilL field is mapped twice, during deserialisation the value of the SKilL field would be stored in two Java fields. Consequently, they hold identical values. However, during serialisation, there is no general way to determine which of the two fields to serialise and store as the SKilL field if the two Java fields hold different values. The analogous problem occurs during *deserialisation*, if multiple SKilL fields are mapped to one Java field.

Due to the fact, that fields on either side do not *need* to participate in a mapping, both sides can be mapped partially. That is, SKilL may ignore existing Java fields and may simply not serialise them. At the same time, it may deserialise data into Java types which lack certain fields that are available in the serialised data. The latter is possible because of SKilL's flexibility with version changes. The binary representation of data "does encode the type system and can therefore provide a maximum of upward and downward compatibility, while maintaining type safety at the same time." [Fel13].

Figure 2.5 shows an example mapping for a type `Point3D` with three field mappings. In case that the SKilL type `Point3D` or the Java type Point have more fields, they will be ignored by SKilL.

2.3.3 Unbound mapping

The previously described mappings are used to tie a Java type to an existing SKilL specification. However, sometimes there might not be any SKilL specification available. A user might simply want to serialise Java classes without any prior restrictions. Moreover, the user might want to automatically generate a SKilL specification derived from the Java classes. It would be nonsensical to write the specification by hand. An additional type of mapping is necessary. Thus, we introduce *unbound mappings*. An unbound mapping only specifies which Java types are to be serialised. The Java types are not tied to the SKilL type context. The mapping is necessary nonetheless, because the user has to specify the set of classes selected for serialisation.

Figure 2.7 shows a grammar in EBNF describing the structure of unbound mappings. In Figure 2.8 an exemplary use of unbound mappings is depicted for the previously mentioned `Point` class. Below the "regular" mapping, there is an unbound total mapping using the "new!" keyword instead of "new". A total mapping implies that the entire class is to be serialised. It is a useful shortcut, as it is shorter than listing all fields explicitly.

⟨*unbound-mapping*⟩ ::= ‘new’ ⟨*java-name*⟩ ‘{’ ⟨*unbound-field-mapping*⟩
  (‘,’ ⟨*unbound-field-mapping*⟩)* ‘}’

⟨*unbound-total-mapping*⟩ ::= ‘new!’ ⟨*java-name*⟩ ‘;’

⟨*unbound-field-mapping*⟩ ::= ⟨*name*⟩

Figure 2.7: Excerpt of the EBNF that describes unbound and unbound total mappings.

```
new my.package.Point {
   coordX,
   coordY,
   coordZ
}

new! my.package.Point;
```

Figure 2.8: Example for an unbound mapping and an equivalent unbound total mapping, assuming that `my.package.Point` only contains those three fields.

Further, it is robust to changes made in the Java class. If a field is removed or added to the Java class, the user does not need to adapt the mapping file.

However, note that even for a “total mapping” the restriction holds, that the IR Mapper will only serialise known types. Thus, if the user requests a total mapping, but a class contains fields whose type is not requested for serialisation, the field will still be dropped.

2.4 Name Binding

Recall Figure 2.2 which outlines SKilL’s architecture including the foreign types extension. So far, the original parser read in a SKilL specification and the IR Mapper translated Java class files into the IR. The mapping file as described in Section 2.3 is also parsed and translated to an internal representation. However, up until now all three internal representations are unconnected. The mapping merely contains type and field names as strings. A lookup via string, however, is expensive. Hence, SKilL performs name binding once and subsequently only works with references.

The name binding translates names represented as strings into references to their type representatives from the intermediate representation. The respective type context is searched for either side of the mapping using the specified name. A successful lookup results in a reference to the internal representative of a type. Note, that unbound mapping rules only contain a Java name and thus there is only need for one lookup in the Java type context. If a lookup fails, SKilL will produce an error as output to the user. The name binding algorithm will not be canceled after the first error. It rather performs all name bindings and produces a list of missing types. That way, the user can try to fix multiple errors at a time, rather then invoking the tool after only one eliminated issue.

2.5 Type System and Type Checking

SKilL might encounter arbitrary classes as foreign types. Because of Java's strong static type system and the fact that we operate on bytecode level, we can safely assume that those types are sane at least in the Java world. In order to avoid compile-time and run-time errors, the SKilL tool must now verify that those types are also valid within the SKilL type system, i.e. that they can be modelled and represented thereby. This is not guaranteed, because SKilL is slightly more restrictive in respect to some typing properties, such as generics or nested containers.

Beyond that, it must also be guaranteed that the mapping between SKilL and Java types does not cause any conflicts. Both mappings between types and mappings of fields imply constraints for type correspondence. Illegal mappings must be detected and reported to the user.

2.5.1 Derivation of Type Rules

Besides finding type representatives for a given name, name binding serves another purpose: it produces type rules that can be evaluated by the type checker.

In case of SKilL with foreign types, the type correctness does not only depend on the specification of SKilL types. It is also influenced by the correctness of the mapping. However, the mapping file is merely a character string. The user can specify corresponding types by using their name. In order to determine the mapping's correctness, SKilL verifies if the mapping abides a set of rules. These rules are derived naturally from the mapping. An instance of such a rule contains one or more references to an actual type using SKilL's intermediate representation. This has several reasons: firstly, due to the fact that every type has exactly one object in memory representing it, equality and inequality are simply implemented as pointer comparison. Secondly, some of the necessary type rules require knowledge of the respective type in order to check if they are obeyed. Hence, the type checker must resolve names to types in any case. Performing this resolution during a dedicated name binding step, guarantees that each name is only looked up once.

After successfully binding a SKilL or Java name to a type, the references for these types can be stored in one or more type rule instances. The type checker verifies the correctness of the rule by operating on actual type objects.

2.5.2 Type Rules

The correctness of an entire mapping is dissected into a series of statements. A statement declares a single atomic fact. For example, a valid statement R could be "Type T_S corresponds to type T_J", where T_S is a type derived from a SKilL specification and T_J is a type produced by the IR Mapper (see Section 2.2).

The statements are partitioned into different categories. In the example above, R belongs to the category "TypeEquation". We call these categories "type rules" and the concrete statements (such as R) an instance of a type rule.

Our foreign types extension for SKilL supports the following type rules:

- TypeEquation
- TargetTypeExists
- TypeMappedOnce
- FieldMappedOnce
- FieldAccessible

For a mapping, there may be multiple instances of any of those rules. In the following, we describe type rules, what they mean and how they are derived.

TypeEquation

A TypeEquation signifies the correspondence of two types, one from the SKilL world and one from the Java world. We use the following notation for TypeEquations: $Eq(T_S, T_J)$. Thus an instance of this rule contains two references to type objects. As opposed to a "mathematical equation", this equation is not commutative: $Eq(T, U) \neq Eq(U, T)$. Its left side should always be the SKilL type and its right side the Java type. This negligible restriction simplifies the process of checking this rule significantly.

The TypeEquation implies a one-to-one relationship between the types. That is, if any type T participates in any two TypeEquations: $Eq(T, T_J^1)$ and $Eq(T, T_J^2)$ then $T_J^1 = T_J^2$ must hold. This is checked and verified by the type checker. TypeEquations are produced by the following parts of a mapping:

- When a type is mapped to another type using their type names. For example `map Point -> example.Point3D { ...` produces an instance of such a rule with these two types.
- Furthermore, the parent types of any two mapped types also form a TypeEquation.
- When a field of a SKilL type is mapped to a field of a Java type, the two types of these fields form a TypeEquation.

TargetTypeExists

Recall unbound mappings as described in 2.3.3. They only specify that the mentioned Java type shall be serialised. This type, however, is not bound to any SKilL type. Thus, a TypeEquation is not necessary to guarantee a correct mapping. However, the type must exist within the specified class path. A TargetTypeExists-rule declares, that the type checker must verify the existence of the mentioned Java type.

TypeMappedOnce

Section 1.5.7 describes the restriction that any type on either side (SKilL or Java) can at most participate in one mapping. In order to guarantee this property, the type checker must refuse any mapping in which a type participates in more than one mapping. The TypeMappedOnce-rule signifies that a type participates in a mapping. We write for a type T: $Once(T)$. For a bound mapping between T_S and T_J, this rule is emitted twice: $Once(T_S)$ and $Once(T_J)$. Both T_S and T_J may only participate in this mapping and not in any other. For an unbound mapping, it is emitted only once for the Java type T: $Once(T)$. This prevents that a Java type is part of both a bound and an unbound mapping at the same time, for example. Note that this rule is never emitted for field mappings. There can be many field mappings with the same field types, as long as every SKilL type is always mapped to the same Java type.

FieldMappedOnce

This rule is analogous to the TypeMappedOnce rule. For bound mappings the rule is emitted twice for each field mapping, once for either side of the mapping. For unbound mappings, it is emitted once for each field in the Java type. The rule states that a field participates in one mapping. Consequently there must not be two instances of the rule with the same field. Note, that the rule actually refers to fields and not the fields' types. This way, obviously, a TypeMappedOnce rule and a FieldMappedOnce do not conflict.

FieldAccessible

The FieldAccessible rule states that a field of a Java type must be accessible from outside the class. This rule is emitted once for every field in a Java type, which is to be serialised. Our code generator produces serialisation code which has to access the actual values stored in a field. Hence, the field must either be `public` or a pair of public getter/setter methods with the conventional naming pattern must exist.

2.5.3 Type Checker

The type rules described above can be considered statements or facts which must hold true in order to realise the requested mapping. The type checker's task is, to prove that the entire set of rule instances is free of contradictions. A contradiction implies that the user requested a mapping that collides with our restrictions for foreign types or with the type system. Again we break this section down to the different rules, how they are verified and how they may contradict instances of the same or of other rules. The type checker iterates over the set of all type rule instances. For each instance, it checks which category the instance falls in. Then, the rule is processed accordingly. Note, that the order of rule instances is irrelevant. The type checker will try to verify that *all* instances are free of conflicts with other instances. That is, if a contradiction is detected, the checker will continue with the following rule instances and report all errors before failing. In the following we describe the verification steps for each rule.

TypeEquation

The TypeEquation rule can only conflict with itself, but not with instances of other type rules. The two rules $Eq(T, U)$ and $Eq(T, V)$ for $U \neq V$ form a minimal contradiction. The first rule clearly states that T corresponds with U. However, the second rule claims that T corresponds with V. Since $U \neq V$, this is not possible. We require the mapping to be a one-to-one relationship.

The origin of this contradiction is also obvious: the user (accidentally) mapped type T twice. The type checker cannot determine which of the two mappings was the intended one. Thus, it reports the error and SKilL does not proceed to code generation.

Conflicts due to TypeEquation rules can simply be determined using a HashMap. Whenever a TypeEquation rule is processed, the type checker tests if the SKilL type is stored as a key in the HashMap. If the type is still missing, the checker will insert it with the Java type from the rule instance as value. In case that the type is found in the HashMap, the checker will compare its mapped value to the Java type in the TypeEquation instance.

If they are equal, there is no contradiction and the processing for this rule instance is complete. If they are not equal, it is a contradiction and the type checker will report the problem to the user.

TargetTypeExists

TargetTypeExists rules are very simple to verify: the checker simply looks up the respective type in the Java type context. If the type is not part of the type context, it might have been erased during a processing step after the IR Mapper completed.

TypeMappedOnce

An instance of a TypeMappedOnce rule can only contradict an instance of the same category. Whenever a TypeMappedOnce rule is encountered, the checker will test if the respective type is stored in a set. If it is, there was at least one other mapping for the same type. The reason of this contradiction stems from the requested mapping: the user made a single type part of more than one mapping. If the type is not found in the set, it is now added and the checker continues with the next rule instance.

FieldMappedOnce

The FieldMappedOnce rule is verified analogously to the TypeMappedOnce rule using a set. A contradiction can only arise between two instances of FieldMappedOnce rules. The reason for such a contradiction is, that a field is part of more than one field mapping. This can happen if the user mentions a field twice within one type mapping, or if they map entire types more than once.

FieldAccessible

The FieldAccessible rule is verified with help of the Reflection Context (see Section 2.2.3). The type checker analyses the original Java class, and checks for possibilities to access the respective field. I.e. either the field is found to be public or a matching pair of getter/setter methods is found. The FieldAccessible rule is used to ensure a certain property about the input itself (in this case the Java classes) but they are never in conflict with any other rules.

The reason for a conflict lies in the original Java class. It defines private or protected fields that do not have a getter/setter pair. Thus, SKilL cannot access the values of the field during serialisation and deserialisation. Because the serialisation code cannot operate efficiently under these circumstances, fields for which the FieldAccessible instance does not hold true are considered to be an error.

2.5.4 Unused Types and Fields

The type checker's primary task is to determine whether the requested mapping is typed correctly or not. However, in the process, it also computes another useful information. Due to the fact, that TypeMappedOnce and FieldMappedOnce rules are emitted for every type and every field which is actually part of a mapping, the type checker automatically knows about the set of all used types and the set of all used fields. The code generator for foreign types shall only produce serialisation code for these types and fields. Non-mapped entities shall be ignored entirely.

When using the foreign types extension, the IR is produced by translating Java classes in the IR Mapper. Thus, the derived types contain all fields which are found in the Java classes, including all the unmapped ones. We extend the type checkers output by these two sets and use them to filter out unused fields and types.

2.6 Code Generation

When SKilL is used the conventional way, language bindings are generated alongside the management and serialisation code. For foreign types, SKilL obviously does not have to generate language bindings. The classes and methods handling serialisation, state management and SKilL's reflection interface are required nonetheless. In order to develop the foreign types generator, we copied the Java generator and adapted it to the slightly different requirements. Copying the code and using it as a basis kept the development effort at a manageable level, since the generator is complex.

The code generators for all target languages are roughly of the same structure. They produce several kinds of classes, each with a specific purpose such as providing access to object pools, a reflection-like interface, state management and so on. For each kind of output code file there is one module in the generator, whose name usually ends in "Maker". In the following, whenever we refer to these different *makers*, we always refer to our specific version which generates the accompanying code for foreign types. Other target languages have their own versions of these makers. They are not considered here.

There is a special class GeneralOutputMaker which is the base class for the other makers. It does not correspond to a class in the generated output. Our GeneralOutputMaker remains mostly unaltered. We added some useful helper methods, for example the getterOrFieldAccess and setterOrFieldAccess methods, which take a SKilL field and produce Java code for direct member access or a getter/setter call, depending on the accessibility of the respective field. That is, if the field is public, a direct member access is generated, otherwise the getter or setter methods are used.

In the Java generator, the “TypesMaker” produced the language bindings for the types, i.e. the Java classes derived from the input SKilL specification. We removed this maker entirely, because we do not need to generate types. Furthermore, the existing code within the SkillFileMaker, FileParserMaker and StateMaker are only adapted insignificantly. We extend the StateMaker in order to generate the addAll() methods, which we describe later in Section 2.8.

The AccessMaker produces an “access class” for each user type. As the name suggests, this class provides access to SKilL’s object pools, provides make() methods which instantiate a user type and allows to mark objects for serialisation with the add() method. A series of minor changes are necessary in this maker. Many are caused by differing package names because we provide our own adapted version of SKilL’s common library in which the package is called jforeign instead of java to avoid conflicts, if both are linked. Additionally, this maker produces several code lines containing the fully qualified name of user types. Since the foreign types are not necessarily in the same package as the resulting “access class” we adapt the generated package path. The correct package path for the foreign types is stored in an optional field in the IR, for each type individually. The generated make() method needs minor adaption. It calls a constructor of the user type. We cannot assume, that the respective constructor exists in a foreign type. Consequently, it needs to be added during compilation, with a slightly altered signature to avoid conflicts. The details are explained in Section 2.7.

The FieldDeclarationMaker produces part of SKilL’s own reflection interface. Each generated class represents a specific field. They are heavily used by the serialisation code to read and write field contents. In that, they also function as an interface to the low-level implementation details of SKilL. This maker shows the most deep-rooted changes.

Again, many occurrences of package paths are adapted. The original maker for Java extensively uses the method mapType(). Its purpose is to map a type object of SKilL’s type system to the name of the corresponding type in the generated code. We overload this method for fields. In case the field has a container type, we map it to the Java type originally declared in the user’s class. E.g. a field of type TreeMap in the user’s class is translated to SKilL’s map type by the IR Mapper. When the IR is passed to our code generator, the type is translated back to the original, i.e. TreeMap. This double translation might seem unnecessary. However, as all types are translated to the internal representation, they can easily be passed to any of the other code generators. Furthermore, it makes the reuse of the generator code possible, with only minor adjustments. If the field is of a non-container type, the original `mapType()` is called and types are translated as usual. Due to this minor extension, the user has a freedom of choice for implementations of the container interfaces. In case the user declared a field using the interface type, say `Map`, we pick a default implementation, in this case `HashMap`.

The `DependenciesMaker` is a small maker which creates a libs directory in the output path and copies SKilL's common runtime library. We adjusted the code to copy our `skill.jforeign.common.jar` rather than the library for Java.

We introduce a new `AspectMaker`. Along with all the previously described makers, this maker produces an aspect, intended for the AspectJ compiler. The generated aspect is best explained using a realistic example. Section 2.7 covers all details about AspectJ. We first describe why we chose AspectJ over some alternatives and then focus on what aspects are generated and what effects they have.

2.7 Providing a SKilL Interface in Foreign Types

Foreign types are created by the user, without SKilL serialisation in mind. Consequently, one issue arises for the serialisation code: it cannot identify objects as "SKilL objects". That is, generated types inherit from an abstract class `SkillObject` which carries a unique ID, as well as a method to determine the type's *SKilL name*. This basic information is essential for the serialisation code to work. The name is used to identify types in serialised data. When an object is deserialised, it can be associated with its runtime representation, a class in the case of Java, which is then instantiated in order to hold the deserialised data. The unique ID is used to identify objects at runtime and determine their state, i.e. whether they are serialised already or not. A negative ID indicates that the object is not yet serialised.

Foreign types do not carry these fields. Consequently, SKilL must provide means to associate an object with its SKilL-specific data, namely the ID and the SKilL name. There are several ways to store this data and link it to the objects. We describe three alternatives that were considered for the implementation of foreign types in SKilL, two of which place the information within the foreign type's object.

2.7.1 Custom Class Loaders

For one, SKilL could generate a custom class loader. A class loader's purpose is to load class files at runtime and link them to the Java Virtual Machine. Java naturally features a hierarchy of class loaders rather than a single one. Their intended behaviour is to firstly delegate class loading to the parent loader, and subsequently, if the parent loader fails to do so, attempt to load the class. By default, the hierarchy contains at least the bootstrap loader (top level) which loads classes from the standard library, an "extension class loader" and one for the user classes. However, this hierarchy may be extended by user code in that it adds another class loader at the bottom of the hierarchy. This custom loader can break the convention to delegate first. In our case, it would rather attempt to load a respective class first, if this class is a "SKilL type" that can be serialised. Otherwise, class loading would be delegated as usual.

In order to use such a class loader, it must be configured at startup *before* any relevant class is loaded. Therefore, our code generator for foreign types would have to generate an alternative `main()` method. This method firstly instantiates and configures the custom class loader. It then calls the "regular" main method of the program. The custom class loader will intercept every attempt to load a new class file. It can then check if the class belongs to a *foreign type*. If it does, it alters the class file by adding an ID and a name field and appropriate getter/setter methods to access the fields. In order to identify the type as a foreign type (a `SkillObject`), SKilL also adds an interface to the class which declares the signatures of the getter/setter methods. When the alterations are completed, the resulting class is linked to the application. There are several downsides to this approach. The main issues are slower class loading, potentially risky code[5], interference with "execution conventions" of the original application[6] or potential conflicts with other custom class loaders[7].

2.7.2 Storing SKilL Data Externally

One option is not to alter the foreign types' classes at all. However, since SKilL still relies on the information, we must somehow provide means to associate the externally stored meta data with a SKilL object. The obvious way is to add every object to a hash map

[5] Changing class files in the class loader effectively invalidates signatures for signed classes.

[6] Because we provide a different `main()` method which must be called in order for SKilL code to work, the user must either specify this class as the main class using command line arguments or the manifest file must be altered.

[7] In case the user already has a custom class loader in place which must be used for all classes including the SKilL types in order for the application to work, the only solution would be to merge SKilL's and the other custom class loader.

with the SKilL meta data as value[8]. For any object we can then retrieve the data by a simple hash map lookup.

However, this solution has major disadvantages. First of all, hash table lookups are slower than a field access by several orders of magnitude. The more objects are serialised, the more lookups are necessary. We want to avoid this slowdown. While fields of different objects can be accessed in parallel, even a concurrent hash table implementation will lock the table for write operations. The deserialisation code works in parallel and the hash table might become a bottleneck. Moreover, externally stored information requires another Java object just to enclose the two variables.

Therefore, with this alternative, both the memory consumption and the computation time will be increased. Efficiency is one of SKilL's major advantages and diminishing it should be avoided whenever possible.

2.7.3 Compile-time Weaving

We chose a solution which both promises speed and safety while keeping as much of the management code unchanged as possible. In order to inject SKilL meta data into foreign types, we alter the compilation process rather than class loading or runtime behaviour.

We use AspectJ to change specific classes during compilation. AspectJ provides its own compiler which takes an extension of Java as input. Besides regular Java classes, we can also specify *aspects*. Section 1.4.3 explains the possibilities of AspectJ in sufficient depth to understand the constructs we use in SKilL with foreign types.

Thanks to AspectJ's ability to work directly on class files, we can even employ compile-time weaving when no source files are available. The alteration of class files will only happen once during compilation. Hence, this approach is superior to the runtime weaving with custom class loaders in terms of speed. If desired, the compilation can be separated only for SKilL-relevant classes. That is, the user may compile the classes of foreign types and the generated code using the AspectJ compiler. The resulting class files could be packaged into a JAR-file, for example. The remaining code of a larger software project can be compiled using `javac` as usual[9].

Consider Figure 2.10. It shows AspectJ code produced by our code generator for foreign types. An aspect is introduced by the keyword `aspect` followed by the aspect's name. The

[8] E.g. introduce a small class only containing those two fields for the ID and the name.

[9] If development tools such as Integrated Development Environments (IDEs) are configured appropriately, they can even provide completion for the classes that are altered by AspectJ. With this completion based on class files, even the injected fields become visible to the user.

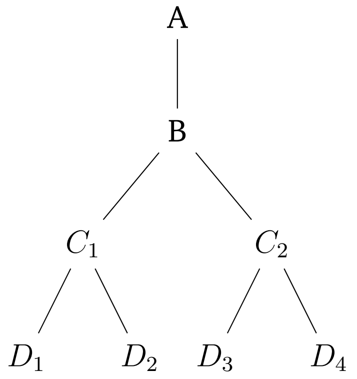

Figure 2.9: Exemplary class hierarchy. Say D_2 and D_3 are to be serialised. If they shall have a common ancestor containing the SKilL ID, we must insert it between A and B. Consequently, B, C_1, C_2, D_1, D_4 also inherit the SKilL ID field even though they do not need it.

aspect's content is enclosed by curly brackets, similar to Java classes. In the following sections, we discuss the different parts of the aspect on the basis of the example in Figure 2.10. All line numbers refer to this figure. The aspects of all types look very similar, except for type specifics such as the fields.

2.7.4 Injecting the SKilL ID into Foreign Types

We use AspectJ to weave in the SKilL ID into foreign types. As previously mentioned, SKilL's generated management code relies on the presence of this special field. The SKilL name must also be accessible. However, since the SKilL name is actually a constant, we return it directly from a getter method.

The original Java generator produced types that all inherit (directly or indirectly) from an abstract class `SkillObject`. However, for foreign types, this is not as appropriate. For example, say the user wants to serialise a few classes $C_1 \ldots C_n$ that are at the bottom of a rather large type hierarchy. If we want to insert a single class C in this type hierarchy which carries the SKilL ID, it must become an ancestor of the common supertype of $C_1 \ldots C_n$. This implies that a potentially big amount of classes in between will also carry the SKilLID unnecessarily. Thus, a lot of memory is wasted. Figure 2.9 illustrates the problem.

We are rather looking for a way to insert the SKilL ID only in such classes that are to be serialised. Luckily, AspectJ allows for insertion of fields directly into a specific type. Line 4 in the example in Figure 2.10 shows the insertion. AspectJ will parse and interpret this aspect and will weave the ID field into the class's bytecode. Additionally to the field itself, we also define getter/setter methods to access it. These are added by lines 6 and 7. In line 9 the `skillName()` method is added to the foreign type. This method defines which name is used for the type in the serialised format. The returned value can be different from the class's name. This happens if a mapping between a SKilL type and a Java class is used. SKilL identifies this type by its SKilL name.

Since we do not use an abstract base class `SkillObject`, foreign types' only known common base type would be `Object`. However, sometimes it is necessary to declare variables of "any type known to SKilL". This type is also used in the common library that is shipped and linked to a program using SKilL serialisation. We thus replaced the abstract class `SkillObject` by an interface with the same name. The interface defines the same methods with identical signatures as the original abstract class. Consequently, much of the existing code can be left unaltered. The advantage of the interface is, that it can be added to any class without altering the existing type hierarchy or changing an object's memory representation. The interface is added to the type in line 35 using the `declare parents` directive. Together with the methods from lines 6, 7 and 9, the foreign type actually implements the interface correctly after AspectJ's weaving.

2.7.5 Initialising the SKilL ID

SKilL IDs of SKilL objects are usually positive numbers. However, objects that are not yet marked for serialisation (and thus yet unknown to the serialisation code) exhibit a negative SKilL ID. All newly created objects are unmarked, at first. Hence, we want to make sure that the SKilL ID is always initialised to a negative number. The user might be calling their own constructors which do not set the SKilL ID to any particular value. In order to always initialise the SKilL ID no matter which of the constructors is called, we introduce a pointcut which executes the initialisation before any constructor code is executed. This can be seen in lines 38 to 40 in Figure 2.10. As the code shows, the point cut applies to objects x of type `Simple` whenever a constructor (`Simple.new`) with *any* signature (the two dots) is called. If the point cut applied, the code in line 39 is executed, i.e. the field `skillID` is set to -1.

2.7.6 Constructors

Section 1.5.2 describes our general assumptions about constructors for foreign types. We require a default constructor in all types. Another constructor which sets all fields to initial values is useful in the generated management code. Our aspect generator checks if the foreign type already provides a standard constructor. In this case, we assume that this constructor can be used to instantiate the type during deserialisation

```
public aspect SimpleAspects {

   // add skillID to Simple
   public long Simple.skillID;
   // getter and setter for skillID
   public long Simple.getSkillID() { return this.skillID; }
   public void Simple.setSkillID(long skillID) { this.skillID = skillID; }
   // Access to skillName
   public String Simple.skillName() { return "Simple"; }


   public Simple.new() { super(); }

   public Simple.new(long x, long y, byte b, short s, int i, long l, float f, double d,
       java.lang.String str, long skillID, SkillObject ignored) {
      super();
      this.x = x;
      this.y = y;
      this.b = b;
      this.s = s;
      this.i = i;
      this.l = l;
      this.f = f;
      this.d = d;
      this.str = str;
      this.skillID = skillID;
      // this is stupid, but the parameter makes sure that there are no signature
          conflicts.
      assert ignored == null;
   }

   public void Simple.selfAdd(SkillState state) {
      state.addAll(this);
   }

   // Add SkillObject interface
   declare parents : Simple implements SkillObject;

   // Initialize skillID no matter which constructor is called
   before(Simple x): target(x) && execution(Simple.new(..)) {
      x.skillID = -1;
   }
}
```

Figure 2.10: Example for a generated aspect for the type `Simple`. Imports are left out for brevity.

(see Section 1.5.2). If it does not exist, we add one using an aspect. Line 12 shows the insertion of the standard constructor.

Line 14 adds the field-initialising constructor. All fields known to SKilL cause a formal parameter which accepts the initial value for this field. We also append another parameter for the SKilL ID to be set directly by the constructor. The last parameter is of type `SkillObject` which we assume is not known anywhere in the user's code. By using this type in the last formal parameter, we ensure that the constructors signature is unique and will not cause any conflicts with existing constructors. In the management code we simply pass `null` as the last actual parameters. Line 27 tries to prevent any surprises on the unknowing user's side.

2.7.7 Injecting the selfAdd method

In Section 2.8 we describe a newly added mechanism to mark entire data structures for serialisation. The code, which handles this, needs a dispatching call on the first parameter's type. Since Java only supports single dispatch (i.e. on the receiver "`this`" of a method call), we need to add a method to the SKilL objects themselves. Calling this method will invoke the dispatching mechanism and result in a call to the method of the actual type rather than the static type. This method than calls the intended method, effectively acting as a proxy. The insertion of this simple method can be seen in the lines 30 to 32.

2.8 Marking Objects for Serialisation

There is one significant difference between foreign and generated types. Since foreign types exist before SKilL comes into play, they might be instantiated anywhere in the code. The user probably just calls the regular constructors of the class. Generated types, on the other hand, are accompanied by the respective type access classes. They allow for instantiation of user types with `make()`. Calling this method will return an instance that is already marked for serialisation.

The foreign types however, are likely instantiated by constructors and thus not automatically marked for serialisation. The user must call the `add()` method from the type's access class. However, the `add()` method exhibits a shortcoming. It only adds the passed object itself and none of the objects referenced in the fields. Hence, the user must manually keep track of all objects they create and add them one by one to the SKilL context. This can become very tedious, especially when big and complex data structures are involved. Moreover, constructors will often create other objects automatically. So the user must check and potentially adjust all code positions where any of the foreign types are instantiated. Alternatively, they might write their own algorithm that traverses the object graph and adds all reachable objects. Again, this would be a time consuming and error-prone task.

We want to provide an easy to use alternative. We extend the code generator to provide methods `addAll()`. Similarly to `add()` they mark an object for serialisation. However, beyond that, they also call `addAll()` for all fields that contain references to further SKilL objects. If a field is of a container type, `addAll()` will iterate over all contained elements and call `addAll()` on each. Consequently, by calling `addAll()` once for a SKilL object, it will add the entire transitive closure, that is, all SKilL objects which are reachable from the initial one.

There are two pitfalls that must be considered:

1. The object graph may (and often will) contain cycles. Obviously, while traversing the graph, the algorithm will encounter objects multiple times. We must break up these cycles. Otherwise, the program will eventually crash due to endless recursion.
2. An object O may be reachable from two objects O_1 and O_2. However, O_1 and O_2 are not reachable from each other. Thus, the user will likely call `addAll()` for both O_1 and O_2. Both transitive closures will eventually try to mark O for serialisation. We must ensure that this only succeeds the first time. The second time the algorithm should simply do nothing.

We use the SKilL ID as an indicator of the SKilL object's state. Note, that the SKilL ID is always initialised to -1 for new objects. A negative ID signifies that the object is not yet serialised. Fortunately, the code only uses -1 and no other negative value. We extend upon this convention. We define, that an ID of -2 signifies "not yet serialised but already seen by `addAll()`". When calling `addAll()` for an object, it first checks if the ID is -2. If so it returns without performing any further action. If not, it calls `add()` for the object and subsequently calls `addAll()` for all fields that may contain SKilL objects, as long as they are not `null`.

In case that the user calls `addAll()` on an object O and subsequently changes it, our algorithm cannot detect that O's updated parts might have to be added again. We oblige the user to call `addAll()` for all references in O which have not been added yet. In most cases, the user can just make the call to `addAll()` when none of the objects change anymore.

Note that the `addAll()` method is overloaded for every user type. Consequently, one such method only needs to know about the fields of this particular type. With all the type information at hand, the generation can simply be done by iterating over all fields and generating the respective code for each field.

However, Java does not feature dispatching function calls on the arguments. Only the receiver of method calls (“`this`”) is considered for dispatching calls[10].

Say we have SKilL types A and B, where B inherits from A. If a field is of type A but contains an object of type B, calling `addAll()` for this field will result in a call to the overloaded version for A. This is not the intended behaviour. The user wants that an object of type B is added, including all of B’s fields. The method for A does not know about these fields. Hence, we must force Java to dispatch on the parameter by employing a widely-used workaround[11]. As described in Section 2.7.7, all user types receive a method `selfAdd()`. Calling this method will be a dispatching call. Thus within the respective version of `selfAdd()`, we know the actual type of the object. The `selfAdd()` method then calls the correct overload of `addAll()` in a non-dispatching call.

Figure 2.11 shows an example for a generated `addAll()` method. Lines 2 to 5 break up cycles in the object graph. Line 6 adds the object itself. The lines 8 to 12 iterate over a map field, adding all the keys and values. The remaining lines iterate over other container fields (lists or sets) and add all their contained objects. Note, that `addAll()` is not called directly for the fields. Rather the dispatching `e.selfAdd(this)` is used. The proxy method `selfAdd()` will call the correct `addAll()` method.

[10] For details on dispatching calls in Java see the section “invokevirtual” of the Java Virtual Machine Specification [LYBB13]: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.1.9.invokevirtual

[11] Programming patterns like the *visitor pattern* are based on the same mechanism. See the literature for details about the visitor pattern [GHJV95].

```
public void addAll(de.ust.skill.ir.TypeContext x) {
   if (x.getSkillID() == -2) {
      return;
   }
   x.setSkillID(-2);
   TypeContexts().add(x);

   if (x.types != null) {
      x.types.forEach( (k1, v1) -> {
         Strings().add(k1);
         v1.selfAdd(this);
      });
   }
   if (x.declarations != null) {
      x.declarations.forEach(e -> e.selfAdd(this));
   }
   if (x.getUsertypes() != null) {
      x.getUsertypes().forEach(e -> e.selfAdd(this));
   }
   if (x.getInterfaces() != null) {
      x.getInterfaces().forEach(e -> e.selfAdd(this));
   }
   if (x.getEnums() != null) {
      x.getEnums().forEach(e -> e.selfAdd(this));
   }
   if (x.getTypedefs() != null) {
      x.getTypedefs().forEach(e -> e.selfAdd(this));
   }
}
```

Figure 2.11: Example for a generated `addAll()` method for the type `TypeContext`.

3 Evaluation

The purpose of this thesis is to investigate serialisation of foreign types with SKilL. In the previous chapter we describe the details on how our implementation works. We try to make serialisation of existing types seamless to the user. That is, we try to minimise manual adaptions which the user must perform in order to serialise data with SKilL. In this chapter we want to examine the extent of our success. We first describe a series of targeted tests we designed in order to investigate if our extension to SKilL works correctly. This is covered in Section 3.2. Section 3.3 describes the serialisation of a "real-life" data structure. This test's purpose is to determine how useful our foreign types extension is in a real-life scenario. We describe the problems and shortcomings, their causes and how they can be solved or mitigated. In Section 3.4 we show the results of a few performance tests. The generated code for foreign types is assessed. We compare its efficiency with the performance of conventially generated types.

3.1 Testing

During the course of implementation, it is useful to design test cases alongside the code itself. This is a widely used practice to find programming and design mistakes as early as possible. By adding more and more test cases the programmer increases the chance to find mistakes in new features or errors that interfere with previously implemented functionality.

SKilL already has a set of several hundreds of tests which cover the core functionality as well as the language specific code generators. While adding new features and extending SKilL by foreign types support, we always test against this set. That way, we ensure that none of the core functionalities or "conventional" generators are interfered with. Simultaneously, we add new test cases which cover the recently changed or newly added code.

There are multiple working steps, starting with the processing of user input and ending in the rather complex code generators, one for each supported programming language. There are multiple error sources along this path:

- The user can provide erroneous input which violates the syntax or semantic rules defined by SKilL's type system.
- There might be programming errors anywhere from the input parser to the code generator. Errors in one working step can obviously cause malfunctioning of following steps.

The result of such errors' presence might be, that SKilL crashes or, similarly, does not produce any output. Such errors can be detected simply by running SKilL. However, even if SKilL generates code and seems to perform without errors, the output code might be incorrect. It is obvious, that testing such an application actually requires another stage in the testing harness. By compiling the *generated* code and testing it against a set of targeted tests for the respective language, we can increase the probability to find errors in the code generators.

In the case of SKilL with foreign types, we have yet another potential error source. SKilL takes additional input which may lead to errors. Firstly, the *foreign types* are read and analysed in the form of class files. While class files can be assumed to be correct, they may still exhibit properties which are not supported by SKilL. These properties should be detected while analysing and should be properly rejected by SKilL. More importantly, SKilL reads the mapping file which may be syntactically or semantically incorrect. All these potential errors should lead SKilL to reject the input rather than generating incorrect code on the output.

3.2 Targeted Test Cases

We design targeted test cases whose purpose is to detect the discussed error sources. In this section we describe each category of test cases and how they are designed. All our tests integrate with the existing testing framework. Whenever the build system's *test* target is executed, all tests are compiled and executed. Thus, the application can be tested using continuous integration tools, which facilitate development considerably.

3.2.1 Mapping Tests

As discussed before, there are two types of errors which a mapping file may introduce: syntactical and semantical errors. Both types are tested in the same way. A test case contains a Java class and a SKilL specification. Additionally, many mapping files can be provided. The mapping files are organised in subdirectories "succeed" and "fail" based on the expected outcome. That is, "succeed" contains many mapping files which are all considered to be correct mappings. Mapping files in “fail” are expected to cause an exception either in the mapping file parser or the type checker.

Syntax tests are mostly modelled as violations of the syntactic rules for the mapping files, i.e. they are mostly *fail*-tests. The fact that SKilL accepts the correct syntax is tested by cases which target the semantic features of the mapping already. The *fail*-tests include:

- missing arrow
- missing braces
- wrong braces
- missing mapping keyword
- file containing complete garbage

Because the mapping file syntax is very simple, the expected syntax errors are also rather simple. Semantic errors add a level of complexity, because some of these errors are only detected once the names in the mapping are bound to actual types. Thus, we additionally introduce test cases which all feature syntactically correct mapping files, which introduce semantical errors, including:

- mapping a *type* twice
- mapping a type twice with different kinds of mappings (e.g. unbound vs explicit)
- mapping a *field* twice
- mapping non-existent types
- mapping non-existent fields
- wrong casing (mapping files are case sensitive, because Java is case sensitive)

Furthermore, we also provide a set of tests, in which both the syntax and the semantics are considered correct. These mappings must obviously be accepted without producing any kind of error. They mostly cover all different variants of mappings, i.e. unbound mappings, explicit mappings and, for both forms, incomplete mappings which do not cover all fields.

3.2.2 Write/Read Tests

The mapping tests only cover the mapping parser and the semantics associated with a mapping. Those tests are important in order to make sure that the subsequent processing in SKilL is only fed error-free input. However, the code generators for foreign types are yet completely untested. Consequently, we introduce another series of tests targeted at the generated code. Java is a rather complex language and testing it simply by analysing the code is a time-consuming task. Rather than reading the generated code back in, we add another stage in our testing procedure. The components of such a test are threefold: the *generated* code, the existing *foreign types* and some dedicated test code which uses both parts together. The test code always follows the same pattern: instantiate the classes of foreign types, ideally set all serialised fields with some value and subsequently serialise the objects. The serialised data is stored to a temporary file. Afterwards, the file is read again and the objects are deserialised. We then verify that all objects are serialised and all the fields are set correctly. We call those tests "write/read tests".

Following this simple principle, we design tests that target different implementation details of SKilL with foreign types. SKilL supports language constructs such as primitive basic types, containers, and inheritance. While these features are well-tested in *generated* types, we must verify that they work well with foreign types. In the following we describe the most important test cases.

Simple Type

The purpose of this test is to verify that SKilL handles all "ground types" correctly. These ground types include integer types of different sizes, floating point types and strings. Since all of these types are somehow special both in Java and in SKilL we must check if they are mapped correctly and behave as expected. We create a new type called "SimpleType" both as SKilL specification (Figure 3.1a) and as Java class (Figure 3.1b). The explicit mapping simply maps all fields of same name. The test code creates multiple instances of class *Simple* and sets arbitrary values for the fields. Then the objects are deserialised and assertions for each field value check if the correct values have been restored.

We add another test for the same type using an unbound mapping. Generally, there is no SKilL type which corresponds to a Java type when using an unbound mapping. The whole purpose is to serialise Java types without tediously writing an equivalent SKilL specification. However, even in unbound mappings, the primitive Java types and strings must be associated with SKilL's ground types. These types are often treated separately in programming languages, including Java. For example, the primitive types cannot be instantiated using the *new* keyword. If they are to be used as "reference types", they

```
SimpleType {
   i64 x;
   i64 y;
   i8 b;
   i16 s;
   i32 i;
   i64 l;
   f32 f;
   f64 d;
   string str;
}
```

(a)

```
public class Simple {
      public long x;
      public long y;
      public byte b;
      public short s;
      public int i;
      public long l;
      public float f;
      public double d;
      public String str;
}
```

(b)

Figure 3.1: SKilL specification and Java class for simple type containing all ground types as fields.

must be "boxed"[1]. Whenever it is unnecessary, boxing should be avoided because it decreases performance[2]. In order to distinguish these types from the regular user types, they are treated specially in SKilL's type system. This additional test ensures that the special treatment works in the presence of unbound mappings.

Containers

The three container types *list, set* and *map* are supported by SKilL and Java provides these types as part of the standard library. There are several implementations for each kind of container. In generated types, SKilL can pick the implementation, which seems to suit the needs best. However, in foreign types, *the user* picks an arbitrary implementation. Since SKilL should be able to handle all of them, our support for container types needs to be tested specially.

We discriminate two general cases that may occur. The user may use the interface type (e.g. `java.util.List`) in the field declaration. In this case we assume that the user does not make any assumption about the concrete type of that list. That is, it is not allowed to cast the interface type to a concrete type, because the deserialisation code could use a different implementation. In the other case, the field declaration is made with a concrete type such as `java.util.concurrent.CopyOnWriteArrayList`. Obviously SKilL must then use exactly this type when deserialising.

[1] https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html
[2] http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html

In order to verify the correct behaviour in both of these cases, we design two targeted tests. In the first one, we declare all fields with the interface types `java.util.List`, `java.util.Set` and `java.util.Map`. In the standard constructor, the fields are then initialised using arbitrary implementations of the interfaces. The second test case uses arbitrary concrete implementations of containers directly in the field declarations. In both cases, we intentionally picked implementations which are not used in the original code generator for Java in order to detect any implementation-specific generated code. For example instead of `java.util.HashMap`, which is used by the Java code generator for SKilL's map type, we use `java.util.TreeMap`. Both tests also contain nested maps.

Inheritance

One of SKilL's defining features is inheritance among types. In Java, type inheritance is mapped directly into Java's type system. That is, if a SKilL type B inherits from another SKilL type A, the generated Java class B' will also inherit from the class A'. This enables SKilL's foreign type extension to recognise such an inheritance relationship and identify the correct class hierarchy.

We provide a simple test to verify that the class hierarchy is recognised correctly. That is, we introduce a few types with inheritance relationships. Each type gets some designated fields. In the test code, each type is instantiated and all fields are set to specific values, including the fields of parent types. After the deserialisation, we verify that all fields again exhibit the correct value. If the inherited fields are also deserialised correctly, SKilL recognised the type hierarchy properly.

Generics

SKilL does not support generics, but it may still encounter generic foreign types. In Section 1.5.3 we describe that generic classes are treated as if they were raw types. A test shall verify that our IR Mapper and type checker can deal with generic types. It shall further ensure that the generated code is compatible with the generic foreign class. We introduce a simple class with generic type arguments. The test code verifies that all the non-generic parts are serialised correctly while the generic parts are simply ignored.

3.2.3 Adding Reachable Objects Automatically

In Section 2.8 we described the generated methods `addAll()` which add a given object and all SKilL objects reachable from this one. We test these methods with the "realistic" types described in Section 3.3. The test's purpose is to verify, that all reachable objects are actually serialised. Thus, we build a connected graph of objects. We use multiple types to ensure that the generated code works for all types. We also use values whose dynamic type differs from the static types. That way, we can test if the code recognises the actual type correctly.

Once the object graph is build, we use the `addAll()` method on an object from which all other objects are reachable. We serialise the data to a temporary file. After that, the file is read and the data is deserialised again. With a series of checks, we assert that all objects are serialised correctly and their references to each other are correct.

3.3 Serialising Realistic Data

In Section 3.2 we described a series of targeted test cases. We developed these tests in order to detect errors in the respective features. However, those test cases are all rather limited. They can only find anticipated errors. Furthermore, most of them are very focused on one specific feature. Hence, they do not test the combination of different features extensively.

Another issue with targeted test cases is the fact that they are not necessarily realistic. They might detect if certain features behave as expected, but they cannot really verify that the set of features is sufficient. In order to address these shortcomings, we evaluate SKilL with foreign types on a realistic set of types. SKilL was originally designed with intermediate representations of languages in mind. It is used in conjunction with program analysis software. Hence, we use SKilL's intermediate representation as an example.

In the following of this section, we first describe the SKilL IR. Then, we describe our attempt to serialise the entire IR using our own foreign types extension. We describe the issues and shortcomings and by which circumstances they are caused.

3.3.1 SKilL's Intermediate Representation

The SKilL IR consists of roughly over 40 classes. The central class of SKilL's IR is `TypeContext`. A type context can contain different kinds of types, which are represented by different classes. Figure 3.2 shows the classes representing different types. The container types are generic in SKilL. The subclasses of `ContainerType` contain references to types that are stored in the container. In the case of `MapType`, this is a list `Types`. For the other containers, it is a single type. Furthermore, there are classes that represent

- `Type`
  - `ContainerType`
    - `ConstantLengthArrayType`
    - `ListType`
    - `MapType`
    - `SetType`
    - `VariableLengthArrayType`
  - `Declaration`
    - `EnumType`
    - `InterfaceType`
    - `Typedef`
    - `UserType`
  - `GroundType`
    - `PointerType`

Figure 3.2: Excerpt of the type hierarchy of SKilL's IR. These are all classes that inherit from the abstract class `Type`.

names, fields, comments and the restrictions for fields[3]. Finally, there are a few classes which represent type substitutions, a mechanism which the IR uses internally.

The SKilL IR features a rich type hierarchy, with multiple levels. There are interfaces and abstract classes. All container types (`Map, Set, List`) are used as field types. Access modifiers (`public, protected, private`) are all used appropriately.

3.3.2 Creating a Mapping File

We extract the SKilL IR from its larger code-base and put it into an isolated directory. It can be compiled independently, since it is a self-contained code-base. The compilation is necessary because SKilL will analyse the class files rather than the source code. We then create a mapping file in order to instruct SKilL about all classes that we want to serialise. We run `find -iname \*.class` in order to get a list of all class files. With the command line tool `sed` we replace slashes (`/`) by dots and adapt the beginning of each

[3] Explained in the SKilL specification document [Fel13].

line to read “new!”. The file extension “.class” is replaced by a semicolon. Because the file system path matches the package path, this results in valid fully qualified names for the classes. We store the result in a file. We then manually remove all interface types from the list[4].

Using the “new!” keyword, we only use unbound mappings. That way, we do not have to write the SKilL specification and an exact mapping in tedious manual work. We then make a first attempt at producing the serialisation code. With help of this mapping file, we run SKilL with our foreign types extension. The result is a series of errors due to unsupported language constructs in the Java code. In the following, we describe the specific errors that occurred.

3.3.3 Nested Containers

Several fields are declared with type `Map<String, List<String> >`. However, SKilL only supports maps of maps. Other combinations of container types are illegal. As a workaround, the user would have to transform the value type (`List<String>`) into a proper type. That is, they could define a class which implements the functionality of a list of strings. This class would likely have a field `List<String>`, which is legal as a field type. However, the class cannot be generic, because SKilL treats generic classes as raw types, in general. The workaround would obviously also cause a series of code changes, since the list cannot be accessed the same way as before. In order to continue testing without considering this issue any further, we remove the fields with this type.

3.3.4 Inaccessible Fields

The second cause of errors is the fact that many of the fields are not accessible in any way. There are private or protected fields for which there are no getter/setter methods. This is not surprising, as *information hiding* is considered a good practice in software development. SKilL’s IR is comprised of immutable types. To guarantee immutability, information hiding is crucial. However, it poses a problem for SKilL, because we want to serialise the data which is stored in private fields as well.

[4] At the starting time of this thesis, SKilL interfaces were not mapped to Java interfaces, but resolved before code generation. Hence, mapping foreign interfaces back to SKilL interfaces did not seem reasonable, because Java’s interfaces do not introduce fields. This poses a restriction: fields which are declared with an interface type cannot be serialised. Because SKilL is under ongoing development, our implementation could now be extended to support interfaces.

The problem could easily be solved by either making the fields public or by adding getter and setter methods for each non-public field. But more importantly, the extent of this issue leads to an important conclusion. Our initially posed restriction, that all relevant fields must be accessible, is not a practical solution. By means of this restriction, we effectively force the user to avoid good software patterns and make the code more prone to programming errors.

We propose to avoid the restriction and access fields even though they might be non-public in the source code. Since we use AspectJ to alter all user types, we can easily inject getter and setter methods for each non-public field. Because the aspects only take effect during compile time, the non-generated code still exhibits the usual encapsulation. That is, within the user's code, the fields are private (protected) and there are no getters or setters to read or modify them. Only after the compilation with AspectJ getter/setter methods exist. The generated serialisation code can simply access the fields via those methods. This also renders the type rules superfluous, which ensure that all fields are accessible. The type checker can be simplified.

3.3.5 Interfaces and Abstract Classes

As previously mentioned, interfaces are not considered in our implementation. Abstract classes on the other hand may introduce fields which all inheriting classes also carry. With abstract classes in the type hierarchy, SKilL generated illegal code. The abstract modifier was not considered and thus the classes were treated as regular concrete classes. Because abstract classes cannot be instantiated, the resulting code does not compile.

We consider two potential solutions. Firstly, we propose to enhance our IR Mapper and enabling it to recognise abstract classes. In conjunction, we can adapt the code generators to respect the limitations of abstract classes and avoid their instantiation.

Since abstract classes have no instances, there never exist objects to serialise. The only significance of abstract types results from inheritance. The abstract classes can have fields which are also part of all descendants. Consequently, we considered not regarding abstract classes as types at all, but rather as a set of fields which might be added to multiple subclasses. We could easily collect all fields from abstract classes and apply them to the first concrete subtype. This is also very similar to the way SKilL's interfaces work. However, there is one significant difference. When using SKilL's interfaces, the code generator produces these fields to be an actual member of the concrete types. That is, if multiple user types have the same SKilL interface, they all receive their own field with identical name. In our case, the field would be part of the abstract parent class. This in turn implies, that SKilL's generated reflection interface must exist for these abstract classes. Hence, it is only feasible to consider abstract classes as user types as well.

We extend the IR Mapper to add the AbstractRestriction to user types, whose Java equivalent carries the abstract modifier. We extend the AccessMaker to respect the AbstractRestriction. Whenever this restriction is present, the AccessMaker produces slightly different code. The method insertInstances(), which is called during deserialisation, has an empty body for abstract classes. There are no actual instances for these types, so there is nothing to do. Furthermore, we change the make() methods to throw exceptions in case they are called. These methods are merely convenient helpers, that instantiate a class and mark the object for serialisation. If the user calls such a method, it will now lead to an error. The methods could not be simply removed, because they are inherited. Furthermore, we remove the UnknownSubPool and SubType classes for abstract user types. The SubType classes actually inherit from the foreign class. Say the user's class is called MyClass. The generator produces a subclass called MyClassSubType. If MyClass contains abstract methods, MyClassSubType would have to implement those methods.

With these alterations, we first test the enhancement with a further targeted test. We provide three classes Grandpa, Parent, Child, of which Parent is an abstract class. All classes contain one field. We instantiate the Child class and serialise the object. After deserialisation, we verify that all three fields are set correctly. Subsequently, we continue testing the SKilL IR. Abstract classes are now recognised correctly and our write/read tests work as expected.

3.3.6 Final Fields

Another class of errors occurred due to the presence of final fields. SKilL's type system has constants, but they can only be of an integer type. In general, fields cannot be declared to be constant in a SKilL specification. Thus, the constant (final) fields in our foreign types lead to errors. The cause of this issue lies in the deserialisation code. It operates by firstly creating all the objects and subsequently by setting the fields to their respective values. If a reference is to be set within a deserialised object, the respective target object will also be present already. The IR's immutable types are in direct conflict with this, as all final fields must initialised in the constructor.

If the deserialisation code set all fields upon object creation, the order of deserialisation would be of significance. This might increase code complexity. Furthermore, cycles within the object graph would lead to further difficulties, because they cause cycles in the dependencies of objects. Bypassing the final modifiers is also heavily discouraged by the Java Language Specification. See Section 17.5.3 for comparison [Oraa]. Thus, it is not feasible to change the deserialisation code.

```
Compiled from "Hint.java"
public final class de.ust.skill.ir.Hint$Type extends
    java.lang.Enum<de.ust.skill.ir.Hint$Type> {
  public static final de.ust.skill.ir.Hint$Type owner;
  public static final de.ust.skill.ir.Hint$Type provider;
  public static final de.ust.skill.ir.Hint$Type removerestrictions;
  public static final de.ust.skill.ir.Hint$Type constantmutator;
  public static final de.ust.skill.ir.Hint$Type mixin;
  public static final de.ust.skill.ir.Hint$Type flat;
  public static final de.ust.skill.ir.Hint$Type unique;
  public static final de.ust.skill.ir.Hint$Type pure;
  public static final de.ust.skill.ir.Hint$Type distributed;
  public static final de.ust.skill.ir.Hint$Type ondemand;
  public static final de.ust.skill.ir.Hint$Type monotone;
  public static final de.ust.skill.ir.Hint$Type readonly;
  public static final de.ust.skill.ir.Hint$Type ignore;
  public static final de.ust.skill.ir.Hint$Type hide;
  public static final de.ust.skill.ir.Hint$Type pragma;
  public static de.ust.skill.ir.Hint$Type[] values();
  public static de.ust.skill.ir.Hint$Type valueOf(java.lang.String);
  static {};
}
```

Figure 3.3: `javap`'s output for the enum `Hint.Type` from SKilL's IR.

3.3.7 Enum Types

Even though SKilL has enumeration types, they are not compatible with Java's `enum`. In most languages, enumeration types are simply represented by integers, or in some rare cases as strings[5]. In Java, on the other hand, using the keyword `enum` will not produce an integer, which is treated as a separate type. Java rather creates a class which inherits from `java.lang.Enum`. The different options of the enum are singleton instances stored as constant fields of the class. Figure 3.3 shows `javap`'s output for an enum type from the SKilL IR.

This enum implementation is not compatible with SKilL. Firstly, the different options are singletons which are instantiated as soon as the enum's class is loaded. During deserialisation, SKilL would attempt to create an object of the type for every enum field. Thus, the value stored in the field is not equal to the singleton value created upon loading the enum class. This will effectively destroy the intended behaviour of enums.

Another issue is SKilL's back and forward compatibility. This could lead to differing version of the enum between the serialised data and the actual code. There is no obvious

[5] There are Perl packages using strings for enum values for example.

way how SKilL should handle a missing option in the current version of the enum, for example. Java's enums cannot be supported by SKilL without major code changes.

As a reference, we experiment with Java's built-in serialisation. We create an enumeration type and serialise it using an `java.io.ObjectOutputStream`. The serialised data is stored in a file. Then we change the enum type in the source code and attempt to deserialise the data from the file again. Our experiment shows, that this is possible, as long as all options that were stored are still available in the enum's current definition. This leads to the supposition, that Java serialises enumerations simply by using their name. Section 1.12 of the Java Object Serialization Specification confirms: "Enum constants are serialized differently than ordinary serializable or externalizable objects. The serialized form of an enum constant consists solely of its name; field values of the constant are not present in the form." [Ora10]. It then states further about deserialisation: "To deserialize an enum constant, `ObjectInputStream` reads the constant name from the stream; the deserialized constant is then obtained by calling the `java.lang.Enum.valueOf` method, passing the constant's enum type along with the received constant name as arguments." [Ora10]. Note, that the in-memory representation of enums is not based on strings, but on references. Only the serialised form of enums relies on strings.

We propose, that SKilL's enum implementation should also be based on strings. It is the only way to reliably distinguish between options that were introduced independently[6]. An integer-based implementation cannot detect this conflict. This would violate SKilL's forward compatibility.

3.3.8 Nested Classes

The SKilL IR also contains several nested classes. Within Java source code, those nested classes are referred to by their name in the scheme `SurroundingClass.NestedClass`. The Java compiler translates each nested class into its own class file. The naming changes to `SurroundingClass$NestedClass`. This implies, that the IR mapper must load class files using their bytecode name. The generator on the other hand must produce code with the Java name.

After adjusting the name handling within the generator and the IR mapper, we notice further issues. In Section 2.7.6 we describe how two kinds of constructors are inserted for every foreign type using AspectJ. For some reason, the generated aspect does not work for nested classes. While for regular classes it has the expected effects, for nested

[6] Options with the same name would be considered equal, even if they are introduced independently. This is also true for types.

classes the AspectJ compiler reports “can’t define constructors on nested types (compiler limitation)”. It might have to do with the way Java implements constructors of nested classes. If such a constructor is `private`, it can only be called by the surrounding class. During compilation, the nested class is transformed into its own proper class file. Hence, the compiler must establish a way that the private constructors can be called from the “outer” classfile. This is done by adding a public constructor, which calls the private one[7]. Evidently, this is not handled in the same way by all Java implementations.

Avoiding this issue would force us to adapt our code generator further. As we cannot inject the field initialising constructor, we must adapt the `AccessMaker`. For nested classes it would have to call the default constructor and subsequently set the fields one by one. Furthermore, this adds another restriction for nested classes. We cannot inject the default constructor using aspects either, so the user must ensure its existence for all nested classes.

Nested classes cannot be modeled with SKilL’s type system. This leads to an architectural problem. The code generator must produce import statements for foreign types in several of the code files. For nested classes, it should produce an import of the outer class. Consequently, each inner class should carry a reference to its surrounding class. We could extend the internal representation and store the information within each type’s representative. Alternatively, the *reflection context* (Section 2.2.3) could be extended.

3.4 Performance

We evaluate the performance of our foreign types extension in terms of serialisation and deserialisation speed. It is a primary objective to provide efficient serialisation and deserialisation for foreign types. However, measuring SKilL’s performance in general is not part of this thesis. In order to assess our success in terms of speed, we rather compare the performance of serialising foreign types with the performance of serialising their generated equivalent. For this purpose, we use SKilL’s IR as types. Our extension produces the serialisation code for these foreign types and also generates a SKilL specification which can be passed to the Java generator. Thus, we have the two set of types, foreign and generated ones.

Our test method creates a graph consisting of a `TypeContext` with several `UserTypes` and these types in turn contain `Field` objects. The `TypeContext` and `UserType` classes use several containers, such as lists and maps. `UserTypes` also contain references to `Comment`

[7] There is an article about another conflict with AspectJ and nested class constructors: http://andrewclement.blogspot.de/2009/03/compiler-variation.html?view=timeslide

objects. Name objects can be found in user types and fields alike. Name, Comment and TypeContext all contain references to strings. We parameterise the graph creation with the amount of types and fields. This way, we can easily control the amount of nodes that are to be serialised.

The foreign types contain user-implemented methods which allow us to create and use the types realistically. For example, creating a UserType within a TypeContext will automatically add the type to a hash map, associating its name with the type object. This logic is missing in the generated version, because SKilL only generates fields, getter and setter methods. Hence, after creating the respective objects, we must also ensure all the references are set, as they are using the original types. E.g. we add the user type manually to the hash map.

In order to obtain most reliable results possible, we perform all tests on the same machine and in the same general setup. We try to maintain a stable execution environment and try to eliminate potential sources of interference, such as other process executing on the system. One iteration of a test consists of *serialising* and *deserialising* an object graph. For foreign types, we also consider the use of addAll() (Section 2.8) as a separate step. We measure the execution time for each step using System.nanoTime(), which accesses "Java Virtual Machine's high-resolution time source" [Orab][8]. Each step is executed for multiple instance sizes.

3.4.1 Serialisation and Deserialisation

Figure 3.4 shows the time spent on serialisation. Note, that both axes have a logarithmic scale. Obviously, if the amount of nodes is increased, serialisation will take longer. But the chart shows, that for foreign and generated types, the increase is similar. In most cases, foreign types' serialisation is slightly slower. This could be caused by some of the many differences between generated and foreign types, such as the modified commons library, class file alterations through AspectJ, or the changed management code. Fortunately, the overhead seems to be constant and thus does not increase for larger amounts of objects.

Figure 3.5 compares execution times of the deserialisation for foreign and generated types. Again, the chart shows, that the increase for both is roughly similar. For millions of nodes, there are much more outliers. This might have to do with the deserialisation code's memory usage. We suppose when deserialising many nodes, the garbage collector might have been invoked several times, distorting our measurements. Repeating the performance tests showed, that this effect prevails.

[8] Direct link: https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime()

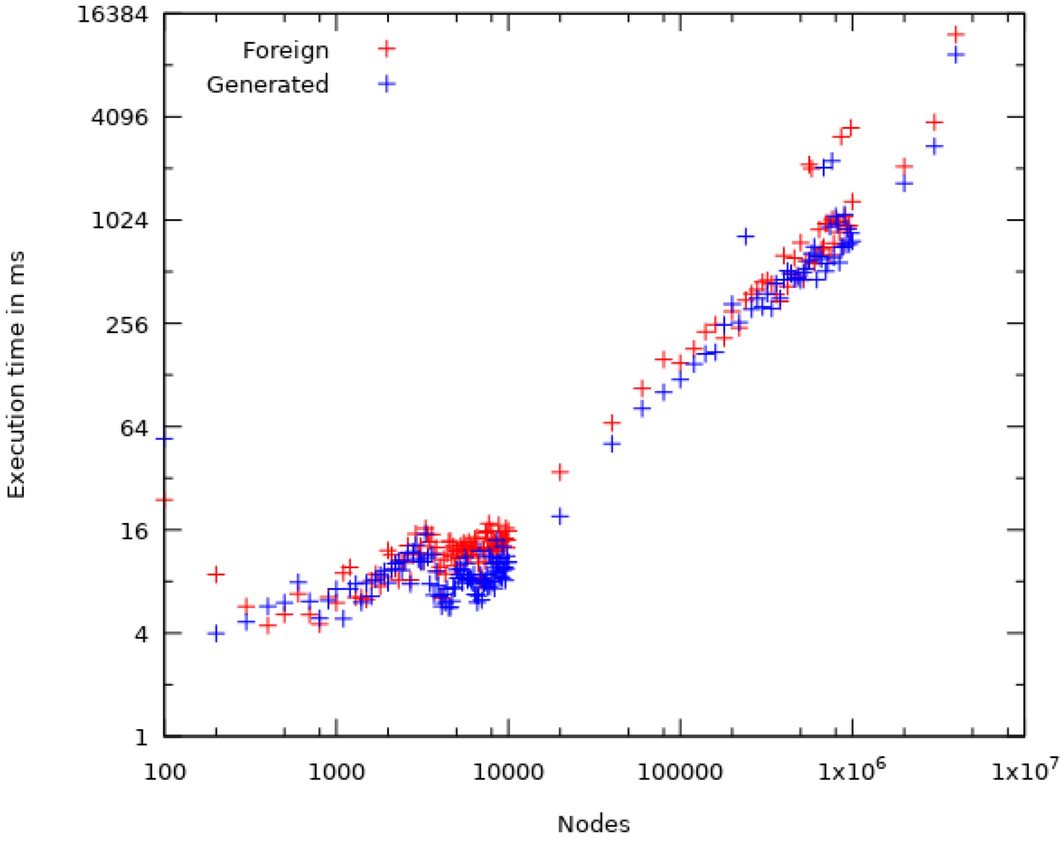

Figure 3.4: Time spent on serialisation for foreign and generated types with different amount of nodes.

Further, we test how much time is spent on marking objects for serialisation using our generated `addAll()` methods. Figure 3.6 compares the performance of `addAll()` with the performance of serialisation itself. It is evident, that adding is quite a bit faster than serialisation. Hence, in the worst-case scenario, the execution time would be at most doubled by our transitive add algorithm, but the average case is much better.

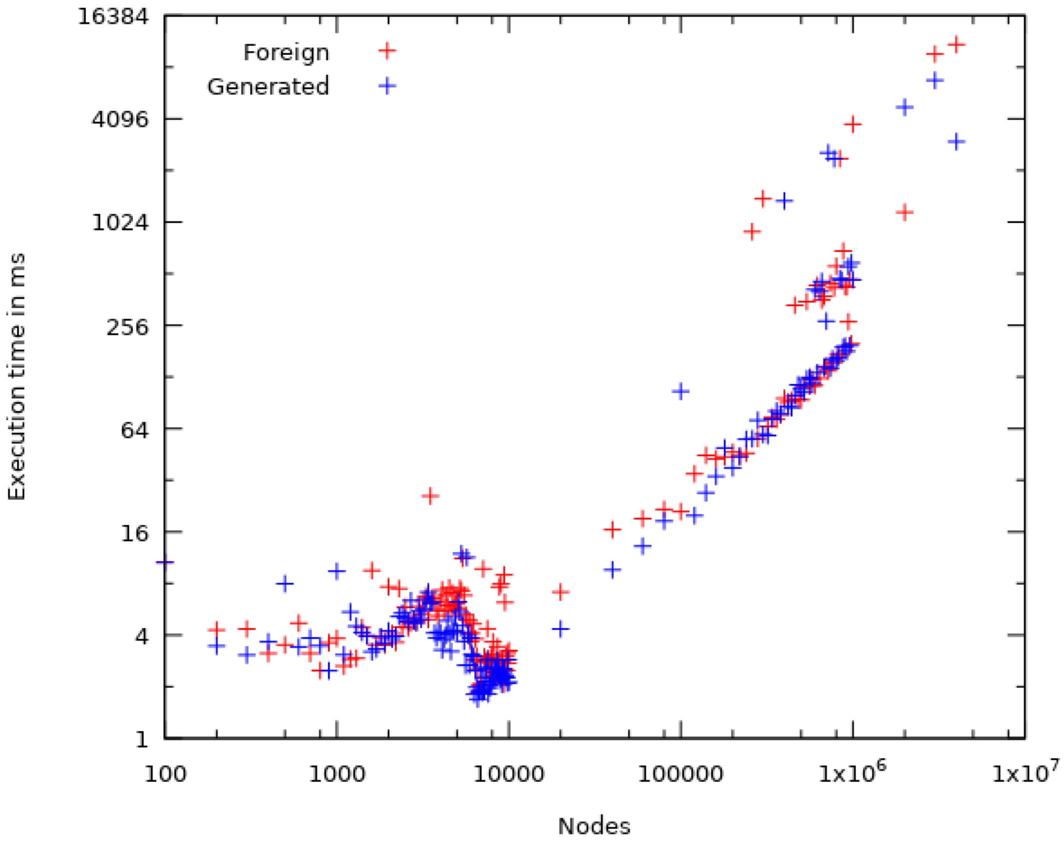

Figure 3.5: Time spent on deserialisation for foreign and generated types with different amount of nodes.

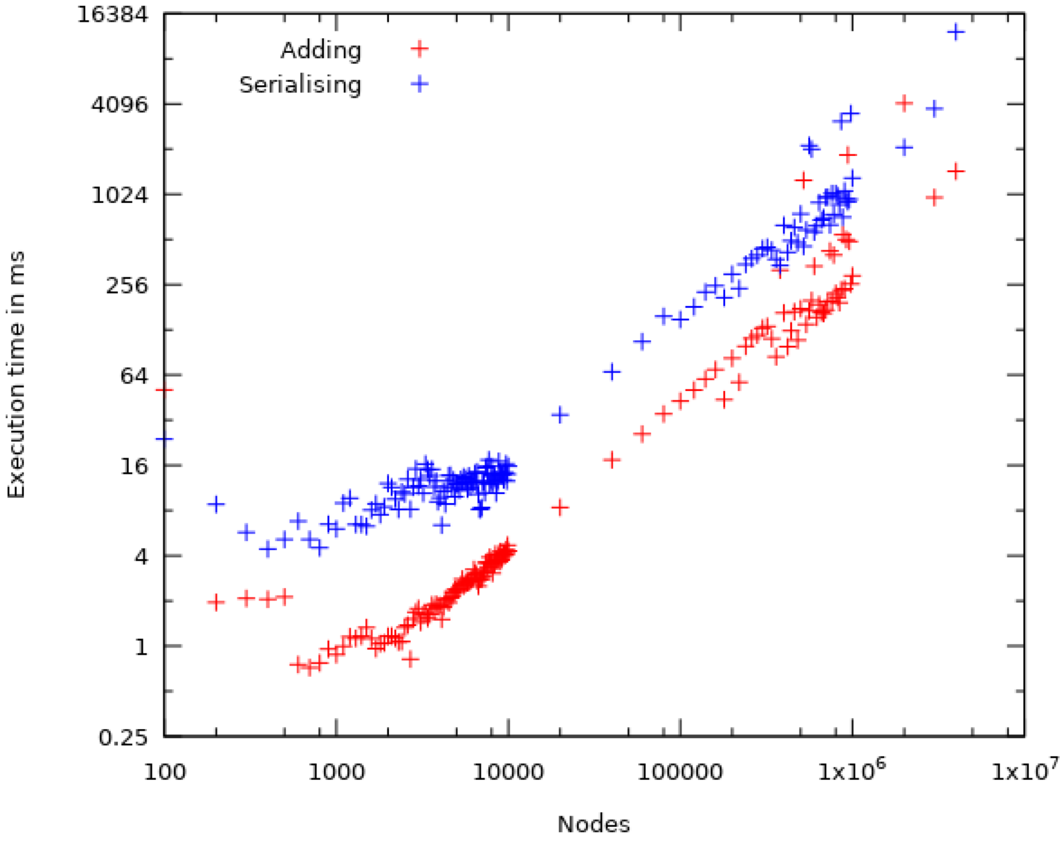

Figure 3.6: Time spent on marking objects for serialisation using the `addAll()` method compared with the actual serialisation time.

4 Summary and Conslusion

We have described a new approach that allows to serialise existing Java classes without using the conventional generated bindings. In order to provide a maximum of flexibility to the user, we have introduced two kinds of mappings, which are checked for type correctness by our type checker. We have introduced the IR Mapper, which analyses Java classes and translates them into SKilL's internal type system. This IR representation of the foreign types can also be used to generate language bindings in different target languages. That means, we have established the means to serialise foreign Java classes and make the data available in any other of SKilL's supported languages. We have discussed the challenges of developing such an extension to SKilL. This includes the architectural changes, analysis of Java classes, implementing the type safe mapping, compensating for the missing SKilL ID and developing the code generators. For several of these challenges, we have provided in-depth discussion of the alternatives and have reasoned our design choices.
The evaluation provides insight into our testing methodology. We have created a series of targeted test cases, as well as an assessment of the usability and utility of the foreign types extension based on a real-life scenario. Finally, the performance tests have shown that foreign types can be used with the usual high performance of SKilL serialisation.

We find, that serialising foreign types is a feasible and useful enhancement. The core features could be implemented with moderate effort. The extensions integrate well with SKilL's architecture. We also find, that the foreign type's serialisation and deserialisation performance is nearly identical to the one of generated types. Because we chose a compile-time method to inject the missing SKilL ID the runtime performance is hardly affected at all. We suppose that the remaining drawbacks, such as missing support of interfaces, could be implemented with moderate effort.

Future work might include reducing the amount of unsupported Java features. It would also be beneficial to the user, if type mappings were more convenient. Enhancements in the mapping parser could include the use of regular expressions or wildcards. Lastly, it could be interesting to investigate if support for foreign types is feasible in other languages as well.

Bibliography

[Fel13] T. Felden. *The SKilL Language*. Englisch. Technischer Bericht Informatik 2013/06. Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau: Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Sept. 2013, p. 43. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2013-06&engl=0 (cit. on pp. 7, 9, 18, 27, 54).

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on p. 44).

[HH04] E. Hilsdale, J. Hugunin. “Advice Weaving in AspectJ.” In: *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*. AOSD ’04. Lancaster, UK: ACM, 2004, pp. 26–35. URL: http://doi.acm.org/10.1145/976270.976276 (cit. on p. 11).

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. “An Overview of AspectJ.” In: *ECOOP 2001 — Object-Oriented Programming: 15th European Conference Budapest, Hungary, June 18–22, 2001 Proceedings*. Ed. by J. L. Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 327–354. URL: http://dx.doi.org/10.1007/3-540-45337-7_18 (cit. on p. 11).

[LYBB13] T. Lindholm, F. Yellin, G. Bracha, A. Buckley. *Java Virtual Machine Specification*. Feb. 28, 2013. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/index.html (cit. on pp. 22, 44).

[Oraa] Oracle. *Java Language Specification*. URL: http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.5.3 (cit. on p. 57).

[Orab] Oracle. *Java Platform, Standard Edition 7: API Specification*. URL: https://docs.oracle.com/javase/7/docs/api/ (cit. on p. 61).

[Orac] Oracle. *Java SE Documentation*. URL: http://docs.oracle.com/javase/7/docs/technotes/tools/findingclasses.html (cit. on p. 14).

[Ora10] Oracle. *Java Object Serialization Specification*. 2010. URL: https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html (cit. on p. 59).

[Xer03] Xerox Corporation, Palo Alto Research Center, Incorporated. *The AspectJ Programming Guide*. 2002 - 2003. URL: http://www.eclipse.org/aspectj/doc/released/progguide/index.html (cit. on p. 11).

All links were last followed on October 14, 2016.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

____

Ort, Datum, Unterschrift