Statically Safe Program Generation with SafeGen

Shan Shan Huang * David Zook * Yannis Smaragdakis **

Abstract

SafeGen is a meta-programming language for writing statically safe generators of Java programs. If a program generator written in SafeGen passes the checks of the SafeGen compiler, then the generator will only generate well-formed Java programs, for any generator input. In other words, statically checking the generator guarantees the correctness of any generated program, with respect to static checks commonly performed by a conventional compiler (including type safety, existence of a superclass, etc.). To achieve this guarantee, SafeGen supports only language primitives for reflection over an existing well-formed Java program, primitives for creating program fragments, and a restricted set of constructs for iteration, conditional actions, and name generation. SafeGen's static checking algorithm is a combination of traditional type checking for Java, and a series of calls to a theorem prover to check the validity of first-order logical sentences constructed to represent well-formedness properties of the generated program under all inputs. The approach has worked quite well in our tests, providing proofs for correct generators or pointing out interesting bugs.

1 Introduction

Program generators can play an important role in automating software engineering tasks. A large amount of research has concentrated on meta-programming tools for writing program generators and specializers more conveniently, more

* College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332 USA
**Department of Computer and Information Science, University of Oregon, Eugene, OR 97403 USA

Email addresses: ssh@cc.gatech.edu (Shan Shan Huang), dzook@cc.gatech.edu (David Zook), yannis@cs.uoregon.edu (Yannis Smaragdakis).

Preprint submitted to Elsevier

efficiently, or more safely [3–6,8,14,26,28,30,33,34]. Nevertheless, such tools have not enjoyed much practical adoption. Programming language designers typically find meta-programming to be too unwieldy and undisciplined to be added as a general-purpose language feature. Working programmers who routinely use and write generators seem to find that advanced meta-programming infrastructure adds very little to what they can do with simple, text-based tools. For instance, many tens of thousands of programmers worldwide use code templates in the text-based XDoclet tool [27] to generate code for interfacing with J2EE application servers.

If a sophisticated meta-programming tool is to become mainstream, it should offer significant value-added for the generator programmer, comparable to the value added by high-level programming languages over assembly programming. In this paper, we explore one possible direction towards adding such value. We present SafeGen: a meta-programming language that offers static guarantees on the correctness of the generator, yet is expressive enough for many practical applications. That is, a generator written in SafeGen is analyzed statically and its correctness is examined under *all* possible legal inputs, where the user specifies what constitutes a legal input. If the analysis succeeds, the generator is guaranteed to only produce well-formed Java code. This addresses a common problem in generator development and a major reason why meta-programming often appears too unwieldy and undisciplined: a generator may have bugs that cause it to produce illegal programs but only under certain inputs. Such bugs can stay undetected for a long time and may only be found by end users and not by the generator writer.

To achieve well-formedness guarantees, SafeGen has an easy-to-analyze language for describing generators. SafeGen offers restricted syntax for describing control flow, iteration, and name generation. Inputs to a SafeGen generator are limited to legal Java programs. That is, SafeGen generates programs by examining existing Java programs at a level comparable to that of Java reflection. All SafeGen reasoning is done in a logic that deals with program entities that are normally exposed through Java reflection (e.g., methods of a class, argument types of a method, etc.), as opposed to, say, integer numbers. Intuitively, this makes SafeGen ideal for XDoclet-like [27] tasks. For instance, SafeGen is appropriate for going over an existing Java class and creating a delegator, or wrapper, or interface, or GUI class that will work correctly with the original class. In contrast, SafeGen is *not* appropriate for generation tasks such as creating specialized versions of the FFT transformation for specific matrix sizes and dimensions.

SafeGen statically checks the legality of code templates by combining traditional Java type checking algorithms with automated proofs of the validity of logical sentences. That is, SafeGen expresses the structure of the generator as a collection of first-order logic formulas, treated as axioms. Further axioms,

also in first-order logic, encode standard properties of Java at the static typing level (e.g., the fact that a final class cannot be extended). Finally, correctness conditions of the generator are described as first-order logic conjectures. SafeGen uses an automated theorem prover, SPASS [32], to attempt to prove these correctness conditions under all inputs, based on the axioms.

SafeGen's contribution to the meta-programming research community is its novel approach of combining logic on reflexive properties of valid programs with program generation, to guarantee the legality of programs that do not exist until the run-time of the generator. This general logic-based approach is not specific to SafeGen's current target language, Java, but could be adapted to other settings. The approach makes SafeGen the only meta-programming tool we know of that both guarantees at compile time of the generator the type-correctness of the generated program, and allows generation of arbitrary pieces of code (e.g., that can generate an unbounded number of new types and variables, as well as references to them). SafeGen's generation language is highly expressive and useful for many program generation needs.

2 Motivation and Background

One can question whether static checking of a generator is a valuable feature. After all, once the generator is used, the generated program will be checked statically before it runs. So why try to catch these errors before the program is even generated? The answer is that static checking is not intended to detect errors in the generated program or even errors in the generator input, but errors in the generator itself. Although these errors will be detected at compile-time of the generated program, this is at least as late as the run-time of the generator. Thus, static legality checking for generators is analogous to static typing for regular programs. It is a desirable property because it increases confidence in the correctness of the generator under all inputs (and not just the inputs with which the generator was tested). To see the problem in an example, consider a program generator that emits programs depending on two input-related conditions: (We use MAJ [34] syntax: code inside a quote, `‘[...]`, is generated. The unquote operator, `#[...]`, is used to splice the result of an evaluated expression inside quoted code.)

```
if (pred1()) emit( ‘[int i;] );
...
if (pred2()) emit( ‘[i++;] );
```

If, for some input, `pred2()` evaluates to `true`, while `pred1()` evaluates to `false` (i.e., `pred2()` does not imply `pred1()`), the generator can emit the reference to variable `i` without having generated the definition of `i`. This

is an error in the generator. However, it might not surface until after the generator writer has tested and widely deployed the generator. This error will then be detected by an end user. It should be the responsibility of a good meta-programming language to prevent such errors by statically examining the generator.

The problem of guaranteeing the well-formedness of generated programs is essentially a problem of analyzing the control-flow and data-flow of the generator. For instance, in the above code fragment, the question is whether there is a valid program path that reaches the second `emit` statement without passing through the first. Similarly, consider a generator that introduces two new names in the same lexical context:

```
emit( `[ int #[name1], #[name2]; ] );
```

For static well-formedness checking, we need to know that `name1` and `name2` do not hold the same value (or we will end up with an illegal duplicate variable definition in the generated program). This is a data-flow property.

We should note that an interesting special case of program generation already offers strong legality guarantees for generated programs. Specifically, statically typed multi-stage languages, such as MetaML[28], MetaOCaml[8] or MetaD[25] guarantee that the generated program is type-correct by statically checking the generator. In this sense, these multi-stage languages represent the state of the art in static safety checking of generators. Nevertheless, staging applies restrictions on the structure of the generator and prohibits the expression of arbitrary generators. Both of our above code examples are not possible in a multi-stage language. In the first example, identifiers in generated code (e.g., `i`) cannot refer to generated variable definitions that are not in an enclosing lexical scope inside the generator text. This is a drawback, even if the final program is expressible in a multi-stage language: ideally, a good meta-programming language should allow its user to express a generator in the style the user finds most convenient. In the second example, it is not possible in a multi-stage language to have the name of a generated definition vary depending on generator input. (Concretely, in MetaOCaml syntax, we cannot write, say, `.<let .~name:int = 0 in .~name + .~name)>.` since binding instances cannot be escaped. Similarly, we cannot escape a type, e.g., `.<let i:.~typename = 0 in i+i>.`)

These restrictions mean that multi-stage languages are ideal for program specialization where the entire code to specialize is available, but not for program generation where the generated program may be partial and may need to cooperate with other parts whose structure is not known until generator run-time. For example, a common generation task for J2EE applications is to take as input an arbitrary Java class and produce a Java interface that contains all of

the class's public methods [29]. In this case, there is no code to specialize that is statically known to the generator. If the generator is to reason about the well-formedness of its output, it needs to do so using abstract properties of yet-unknown program entities, such as "no two methods in the input class can have the same type signatures". This is exactly the kind of program generation that SafeGen intends to support. [1] From a technical standpoint, the problem is harder than multi-stage programming, since there are no restrictions as to how the control and data-flow of the generator can influence the contents of the generated program parts.

3 SafeGen Design

In this section we describe the main design of the SafeGen language. We first give a high-level overview of SafeGen and then present the language in detail.

3.1 Overview of the Approach

Before we discuss the specifics of the SafeGen language, we will offer a quick example of what SafeGen can do, which will hopefully illuminate the role of all the distinct language features described in detail in the next sections. As we have not yet defined all the elements of SafeGen syntax and functionality, we will appeal to the reader's intuition for our example.

A basic, but not too interesting, SafeGen generator is the following:

```
#defgen MakeInterface (Class c) {
  interface I {
    #foreach(Method m : MethodOf(m,c)) { void #[m] (); }
  }
}
```

The above code defines a generator named "`MakeInterface`". It accepts a Java class as its argument. It generates an interface named `I` (this name may be hygienically renamed at generator runtime if the generator is used multiple times in the same lexical context, as we explain in Section 3.2.5). For each method of the input class, the generator produces a void-returning, no-argument method by the same name in the generated interface.

[1] We expect that the general approach used in SafeGen could also apply to program specialization tasks. Nevertheless, as mentioned earlier, SafeGen's current input language and reasoning engine is limited to reflection-like properties, and cannot apply to, say, generating specialized numerical code for given array size and dimensions.

Although this generator is almost trivial, it is still challenging to determine automatically whether it will output a valid interface for every input class. For example, do all the declared methods have unique signatures? In its attempt to prove that the generated code is well-formed, SafeGen relies on three kinds of knowledge: assumptions about the input (in this example there are none other than the fact that it is a well-formed Java class), general knowledge of all well-formed Java programs (e.g., all well-formed Java classes have unique method signatures), and knowledge about the output (in this case, that it is an interface with methods named after the methods in the input class). SafeGen represents all its knowledge and assumptions, as well as the well-formedness properties that *should* hold in the output, in first-order logic sentences. It then attempts to prove that these properties hold under the given assumptions, for *any* possible input, by using the SPASS theorem prover to prove the validity of first-order logic sentences. For instance, the following formula is part of SafeGen's knowledge about well-formed Java programs—it states that any two members (either classes or interfaces) in a well-formed Java package have different names. (We show here the formula in SPASS syntax in order to be concrete about the level of interfacing with the theorem prover.)

```
formula(forall([c1, c2],
  implies(and(wellformed(c1), wellformed(c2),
              equal(DeclaringPackage(c1),DeclaringPackage(c2)),
              not(equal(c1,c2))),
          not(equal(Name(c1), Name(c2)))),
         MEMBERS_IN_PACKAGE_DIFF_NAME).
```

The following sentence states what SafeGen knows about the generated code, in relation to the input. It says that there is a well-formed class `c`, and an interface `I`, which has methods with the same name as the methods in the class. Note that the sentence does not assert that the generated interface is `wellformed`, but only that it exists:

```
exists([c],
  and(class(c), wellformed(c),
      exists([i], and(interface(i),
                      forall([m],
                        and(method(m), equal(DeclaringClass(m), c),
                            exists([m2],
                              and(method(m2),
                                  equal(DeclaringClass(m2), i),
                                  equal(Name(m), Name(m2)))))))))).
```

And finally, the following is a property that should hold for the generated class. It states that generated methods cannot have the same name and type signatures if they are in the same class.

```
forall([m1, m2],
  implies(and(method(m1),
              method(m2),
              equal(DeclaringClass(m1), DeclaringClass(m2)),
              not(equal(m1, m2))),
          not(and(equal(Name(m1), Name(m2)),
                  equal(Formals(m1),Formals(m2))))))
```

In fact, this conjecture cannot be proven for the above generator, `MakeInterface`. All generated methods have the same signature and methods can have the same names, since the same method name can be overloaded in the input class, `c`. Therefore, in this example we see that the output is potentially ill-formed.

3.2 Language Design

Figure 1 presents a formal syntax for SafeGen. We use the shorthand $\overline{\texttt{T}}$ to represent a possibly empty sequence of `T`, where `T` is some syntactical entity. We also use `IDENT` to represent any legal Java identifier. For clarity, we use `G_NAME` to indicate generator name, and `P_NAME` to indicate predicate name. Both are simply identifiers, as shown by the last two rules. We use (...)? to indicate that there can be either zero or one occurrences of the construct represented by "...".

SafeGen can be thought of as two languages (and thus two syntaxes), with constructs for passing control back and forth between the two. There is the meta-language, which allows for the definition of generators and meta-variables, provides constructs for directing the control and data flow of the generator and for abstracting values from meta-variables that can be used in the code to be generated. Then there is the object-language for defining the code templates—code to be generated when the generator is run. The object-language is essentially Java, with additional "escape" operators for passing control back into the generator. Most of the rules in Figure 1 are defined for the meta-language. The only syntax rule for the object language is `CODE_TEMPLATE`. It is simply the Java syntax (as defined by the Java Language Specification [15]), with the addition of three "escape" operators. We elided the exact definition of where the escape operators are allowed in the Java syntax for brevity. However, this should become clear in our subsequent examples of the syntax and main concepts of SafeGen.

```
GENERATOR_DEF   ::=   #defgen G_NAME ( INPUT ) { GEN_BODY }
INPUT           ::=   CURSOR_DEC | INPUT_PRED_DEC
CURSOR_DEC      ::=   META_TYPE IDENT ( : LOGIC )?
INPUT_PRED_DEC  ::=   P_NAME ( PRED_ARG ) => LOGIC
PRED_ARG        ::=   META_TYPE IDENT
GEN_BODY        ::=   CODE_TEMPLATE
                 |    GENERATOR_DEF
                 |    PRED_DEF
CODE_TEMPLATE   ::=   Java syntax
                 |    #[ META_EXPR ]
                 |    #foreach ( CURSOR_DEC ) { CODE_TEMPLATE }
                 |    #when ( LOGIC ) { CODE_TEMPLATE }
                      ( #else { CODE_TEMPLATE } )?
META_TYPE       ::=   Class | Interface | Method
                 |    Constructor | Field
META_EXPR       ::=   IDENT
                 |    META_EXPR.META_FUN
META_FUN        ::=   Name | Type | Formals
                 |    ArgTypes | ArgNames
PRED_DEF        ::=   #defpred P_NAME ( PRED_ARG ) = LOGIC
LOGIC           ::=   forall META_TYPE IDENT : ( LOGIC )
                 |    exists META_TYPE IDENT : ( LOGIC )
                 |    P_NAME ( IDENT )
                 |    IDENT = IDENT
                 |    ! LOGIC
                 |    LOGIC & LOGIC
                 |    LOGIC "|" LOGIC
                 |    LOGIC => LOGIC
G_NAME          ::=   IDENT
P_NAME          ::=   IDENT
```

Fig. 1. SafeGen syntax

3.2.1 Cursors.

The two main concepts in the SafeGen language are those of a *cursor* and a *generator*. A cursor is a variable ranging over all entities satisfying a first-order logic formula over the input program. Thus, the input program is viewed as a collection of logical facts about its type declarations. For instance, a cursor expression in SafeGen would be:

```
Method m : MethodOf(m,c) & Public(m) & !Abstract(m)
```

This cursor, `m`, describes all non-abstract, public methods in class `c` (`c` is a cursor assumed to have been defined earlier). In general, the values of cursors are type-level entities in the input program (methods, arguments, classes, interfaces, etc.). The logical predicates used to build cursors can be viewed best as a reflection mechanism over Java programs. SafeGen has several predefined predicates that correspond to Java reflection information and the user can create new predicate symbols that represent arbitrary first-order logic formulas over the predefined predicates. Since the logical sub-language used to define cursors in SafeGen is a standard first-order logic, we postpone describing its specifics in detail until later in the paper.

3.2.2 Generators.

A SafeGen generator is a way to express generated Java code. Generators are defined with the `#defgen` primitive, followed by zero or more input arguments, and the code fragment to be generated for the inputs. The following is a trivial generator taking no inputs, and always producing a constant piece of code:

```
#defgen TrivialGen () {
   class C { public void meth() {} }
}
```

To make a generator's output more interesting, we would have to supply some inputs. A generator can receive input parameters that are either a single reflection-level entity (e.g., class, method, field, etc.), or a collection of reflection-level entities constrained by a predicate. For instance, the following is a generator that accepts a single non-abstract class as an argument. (The body of the generator is elided.)

```
#defgen MyGen1 (Class c : !Abstract(c)) { ... }
```

Similarly, the following generator takes as input a collection of classes, constrained by the predicate `input`:

```
#defgen MyGen2 (input(Class c) => !Abstract(c)) { ... }
```

Note that `input` is a new predicate defined right inside the definition of this generator. Unlike predicate definitions that we will see later, the "implies" (`=>`) operator is used for predicates defining generator inputs. The syntax difference also serves to illustrate a subtle semantic difference: the input is not *all* classes that are non-abstract, just some classes that are guaranteed to be non-abstract.

The body of a generator (enclosed in `{...}` delimiters) can contain any legal Java syntax. This Java code (*object-level* code) is "quoted"—that is, it gets generated when the generator executes. Quoted code can contain three SafeGen constructs that serve as "escapes": they direct the control and data-flow of the generator, allowing configuration of the quoted code. These three SafeGen constructs are `#[...]` (pronounced "unquote"), `#foreach`, and `#when`. Before getting into the details of these escape operators, it is important to recognize the distinction between meta- and object-level variables. The quoted code can contain variable declarations of its own—these are regular Java variables, e.g., `int i;`. However, since the escape operators pass control back to the meta-level code, escaped code can only refer to meta-level entities. (More precisely, the part of the program's syntax tree that has `#[...]`, `#foreach`, or `#when` at its root can only refer to meta-level variables or functions.) Meta-level variables are those defined as inputs to the generator (e.g., `Class c` in the generator `MyGen1`), or those introduced by cursor definitions. Meta-level functions are other declared generators and predicates.

The `#[...]` operator is used for adding fragments of Java code inside a larger fragment. A generator can derive code fragments by applying several built-in functions to meta-variables. Available functions are: `Name`, `Type`, `Formals`, `ArgNames`, `ArgTypes`, and `Modifiers`. Consider the example of the following generator:

```
#defgen MyGen (Class c : !Abstract(c)) {
   #[c.Modifiers] class #[c.Name] {  }
}
```

This generates a new (empty) class with the same name and modifiers as the input class.

Functions `Name` and `Type` generate one identifier. The rest of the functions generate arrays—`Formals` generates an array of $\langle Type, Identifier \rangle$ pairs, whereas `ArgNames`, `ArgTypes`, and `Modifiers` generate arrays of names, types, and modifiers, respectively. When unquoting arrays, SafeGen splices the arrays into the abstract syntax tree of the code being generated. This means that proper separators will be generated in the final code, even though they are

syntactically elided in the code template. For example, consider the following SafeGen template:

```
#defgen GenMeth ( Method m ) {
   class OneMethod {
      void foo ( #[m.Formals] ) { ... }
   }
}
```

The code generated by this template is a class containing only one method, `foo`, which accepts the same arguments as the input method `m`, with commas properly placed to separate each argument type/name pair.

Clearly, not all functions can be applied to all meta-variables. `Formals`, `ArgNames`, and `ArgTypes` can only be applied to meta-variables of `Method` type. SafeGen also allows the syntax `#[c]` on a meta-variable `c`. This is a shortcut for `#[c.Name]`.

3.2.3 Control Flow

The control flow of the generator is affected by primitives `#foreach` and `#when`, allowing iteration and conditional execution, respectively. The `#foreach` construct takes as argument a cursor definition. As we saw previously, a cursor ranges over reflection-level elements retrieved from the generator input, constrained by some criteria defined using SafeGen logical formulas. Since generator inputs are well-formed Java program entities, and no Java program has an infinite structure, all `#foreach` iteration terminates.

Inside the body of the `#foreach`, the cursor name can be used to refer to the current element in the range of the formula used to define the cursor. For instance, consider the following generator:

```
#defgen AddFields (Class c) {
   #foreach ( Field f : FieldOf(f,c) ) { int #[f]; }
}
```

This creates a sequence of definitions of integer variables, each named after a field in the input class, `c`.

The `#when` construct's syntax is `#when ( LOGIC ) { CODE_TEMPLATE }`, optionally followed by `#else { CODE_TEMPLATE }`. That is, `#when` takes a logic formula as a parameter. If the formula evaluates to true at generation time, the first code template is generated. Otherwise, the code template following the `#else` is generated. In the example below, the argument to the generator

is a set of Java interfaces (with no other constraints on them). If the set is not empty, then the "implements" clause gets generated, followed by all the names of interfaces. Otherwise, nothing gets generated.

```
#defgen MaybeImplements ( input(Interface i) => true ) {
   #when ( exists (Interface in) : input(in) ) {
      implements #foreach(Interface i) { #[i] }
   }
}
```

Again, the generator's model ignores low-level separator tokens—our generators operate on abstract syntax trees, not parse trees. Thus, when the `#foreach` construct above generates multiple interface names, they get added to an AST. But when actual code is generated, they will be separated by commas, as Java requires.

3.2.4 User-Defined Predicates.

For modularity and code reuse, SafeGen also allows definitions of new predicates both inside and outside the body of a generator. `#defpred` is used to give a name to a frequently used logic formula. The following example declares a predicate `myPred` that can be used in logic formulas, just like built-in predicates:

```
#defgen MyGen3 ( ... ) {
   #defpred myPred ( Class c ) = Public(c) & !Final(c);  ...
}
```

3.2.5 Name Management and Hygiene.

In the body of a generator, identifiers that correspond to generated definitions are hygienically renamed to avoid name conflicts. (More precisely, this refers to "quoted", constant-name declarations—i.e., identifiers that do not occur under an escape operator.) For example:

```
#defgen RenameGen (input(Method m) => (m.Type = int) & noArg(m)) {
   #foreach( Method m: input(m) ) { int result = #[m](); }
}
```

(For convenience, the generator uses a predicate `noArg`, which we can define using `#defpred`. This constrains the input methods to accept no arguments.)

The result of the above generator will not be multiple definitions of variable `result`. Instead, at generation time, the actual variables generated will have fresh names. Any references to these variables under the same cursor (or a cursor defined over a sub-range) will be consistently renamed to refer to the right variable. Since the renaming is only performed at the final output phase (i.e., when all generators have been called and the result is a complete Java compilation unit) SafeGen can tell which identifiers need renaming. Sometimes, a generator writer might indeed want to specify a name for a particular declaration, without renaming. In these cases, we provide the keyword `#name["..."]`. The identifier between quotes is generated as is.

3.2.6 Predicates, Cursors, and Logic in Detail.

The logic underlying SafeGen is a sorted logic, with the basic sorts being: `Class`, `Interface`, `Method`, `Constructor`, `Field`, `Identifier`. Accordingly, all variables and constants in our domain are of one of these sorts. SafeGen does not provide any built-in constants. However, the user implicitly "creates" constants of the `Identifier` sort as needed. For example, if a user wishes to find all classes that implement `java.io.Serializable`, she writes the logical sentence:

```
forall (Class c) : (exists (Interface i) :
    ( InterfaceOf(i, c) & i.Name = "java.io.Serializable"))
```

`java.io.Serializable` is then declared as a constant in the domain during the compilation process.

The syntax for SafeGen logical sentences closely follows the syntax for first-order logic sentences (with the addition of sorts for declared variables). SafeGen provides logical operators `forall`, `exists`, `=`, `&`, `|` , `=>`, `!`, which correspond to all the operators available in first-order logic. The full list of available predicates and functions is shown below:

- Unary predicates: `Public`, `Private`, `Protected`, `Static`, `Final`, `Abstract`, `Transient`, `Strictfp`, `Synchronized`, `Volatile`, `Native`
- Binary predicates: `PackageOf`, `ClassOf`, `InnerClassOf`, `InterfaceOf`, `SuperClassOf`, `ConstructorOf`, `MethodOf`, `FieldOf`, `ExceptionOf`, `ArgTypeOf`
- Functions: `Name`, `Type`, `Formals`, `ArgNames`, `ArgTypes`, and `Modifiers`.

For readers unfamiliar with first order logic syntax, please refer to Figure 1, rule `LOGIC` for details.

3.2.7 Example.

We can now consider a non-trivial generator written in SafeGen. This is a realistic example, yet one that is short enough to study here and to use later for illustrating SafeGen's static checking process. The generator in Figure 2 takes a set of non-abstract classes as input and creates subclasses of the input classes with methods that just delegate to those of the superclasses. (As explained earlier, the identifier `Delegator` is going to be renamed for each of the generated classes as to not induce name conflicts.)

```
1. #defgen MakeDelegator ( input(Class c) => !Abstract(c) ) {
2.    #foreach( Class c : input(c) ) {
3.       public class Delegator extends #[c] {
4.          #foreach(Method m : MethodOf(m, c) & !Private(m)) {
5.             #[m.Modifiers] #[m.Type] #[m] ( #[m.Formals] ) {
6.                return super.#[m](#[m.ArgNames]);
7.             }
8.          }
9.       }
10.   }
11. }
```

Fig. 2. A generator that generates a delegator class for an input class.

3.3 Static Checking

We can now see how our approach can reason about a generator and guarantee that it produces well-formed programs under all inputs. Every well-formedness property of the output program is expressed as a logical formula. For instance, consider again our Section 2 example generator, for which we want to guarantee that a generated reference is always bound to a definition:

```
if (pred1()) emit( `[int i;] );
...
if (pred2()) emit( `[i++;] );
```

The above example written in SafeGen is:

```
#when(logic_1) { int i; }
...
#when(logic_2) { i++; }
```

where `logic_1` and `logic_2` are first-order logic formulas defined in the fashion described in the previous section. Checking whether variable `i` is declared before use becomes checking the validity of the logical implication `logic_2` →

`logic_1`. If the theorem prover proves validity, we know that under *any* input to the generator, the variable `i` would always be declared before it is used.

Other program well-formedness properties are also expressible in a similar fashion. Determining how to translate a given program property into a logical sentence is the role of the SafeGen implementation, described in the next section.

We should be explicit in that implementing checks for all well-formedness properties of Java programs is a heavy engineering task. SafeGen currently does not support all possible checks but we believe the omission is just a matter of engineering. [2] The currently supported checks in SafeGen are fairly representative in difficulty of the task and correspond to many valuable program correctness properties (e.g., method typechecking). Specifically, the currently fully supported tests are for the following properties.

- A declared superclass exists.
- A declared superclass is not `final`.
- Method argument types are valid.
- A returned value's type is compatible with the method return type.
- The return statement for a `void`-returning method has no argument.

Notably missing checks include access control (e.g., no access to "private" variables outside class); checking for subtyping restrictions (e.g., a non-abstract class supplies definitions for all its superclass's abstract methods); checking for referring only to defined variables; checks for duplicate definitions; checking for correct declaration of exceptions; etc.

4 Application

We demonstrated with the `MakeDelegator` example, in Figure 2, a programming pattern (Adaptor [12]) used in many common software engineering tasks. The same pattern can be applied to other use cases. For example, we can provide a generator that takes as input a class and creates a wrapper for it so

[2] First-order logic can express any computable property and the only question is whether a theorem prover can reason about such properties effectively. For several yet-unsupported properties (i.e., properties for which SafeGen does not generate conjectures automatically) we have hand-produced logic formulas corresponding to example SafeGen programs and we have confirmed that we can reason about them in SPASS effectively. For instance, the conjecture in Section 3.1 was hand-produced, although our longer example in Figure 4 was automatically produced by the SafeGen compiler.

that each execution of a method is logged. The implementation of this generator is very similar to the `MakeDelegator` example, with some additional logging code before invoking the `super` method call. We will not belabor the point by showing it again in detail. Similarly, we can create a generator that returns a "synchronization" wrapper for the input class, so that each method is wrapped in a `synchronize(mutex) {...}` block, where `mutex` is an object created just for synchronization. This is in fact the exact technique used by `java.util.Collections` (part of the Java Collections Framework [1]) to make synchronized data structures from unsynchronized ones. Instead of repetitively writing the same boiler-plate code for each class that needs to be synchronized, one generator can generate synchronized definitions for all classes.

The two examples given so far have a strong aspect-oriented [23] flavor. Indeed, since SafeGen allows generated code to be applied uniformly to methods (or fields, or any other reflection level entity), it has the ability to "cross-cut" object-oriented entities. However, SafeGen's expressive logic language allows programmers to do more than what is provided by the current aspect-oriented programming tools, such as AspectJ [22] or JBoss AOP [7]. For example, a task regularly encountered by programmers is to adapt an existing class to implement an interface. Oftentimes, the class already has all the meaningful implementation details in place. For the methods in the interface not already implemented by the class, the class simply needs to provide empty, or default, implementations. This is a rather tedious task and should have easy, automated solutions. Other research has also tried to address this problem [17,24,31]. For illustration, below is a piece of code often seen in Swing graphics programming—when a class needs to implement a listener interface, it often provides meaningful implementations only for a few methods, and empty bodies for the rest:

```
private class SomeListener
                implements MouseListener, MouseMotionListener {
  public void mousePressed (MouseEvent event) {
    ... // do something
  }
  public void mouseDragged (MouseEvent event) {
    ... // do something
  }

  // the rest are not needed. Provide empty bodies.
  public void mouseClicked (MouseEvent event) {}
  public void mouseReleased (MouseEvent event) {}
  public void mouseEntered (MouseEvent event) {}
  public void mouseExited (MouseEvent event) {}
  public void mouseMoved (MouseEvent event) {}
}
```

The solutions that previous research has provided either are not general enough to be applied to all classes that need such adaptation [31], or require separate implementations for different interfaces [24], or provide no static guarantee that, after adaptation, the class will not have type errors [17]. The SafeGen solution, however, is general enough that given any class and any interface, a new class can be generated so that it adapts the class to the interface, *and* there is a compile-time guarantee that the generated class will have no well-formedness errors. The implementation is shown in Figure 3. It defines a generator, `MakeImplement`, which takes two inputs—a class (`c`) and an interface (`i`). It produces a new class that extends `c`, and implements `i`. For all the methods that `c` already provides implementations for, the new class simply forwards the call, very much like the `MakeDelegator` example. In addition, for each method in interface `i` that is *not* declared in `c`, the generator generates a default implementation for that method.

5 SafeGen Implementation

The most interesting part of the SafeGen implementation is the static checker. Therefore in this section we discuss how SafeGen produces axioms and proof obligations for a theorem prover, based on the structure of the SafeGen program.

5.1 SafeGen Static Checking

Although the SafeGen checking algorithm is not a traditional type-checker, it is easiest to present it in terms of type-checking, where both the names and the types of the various entities can depend on logic predicates.

SafeGen has two type-checking processes. The first is type checking for the meta-language: legality of references to meta-variables, meta-level predicates, functions, and generators. The second, and much more complex one, is type checking for generated Java code. SafeGen's type system keeps two separate environments to support these two processes: the meta scope, for the generator, and the object scope, for the generated program.

5.1.1 Environment.

A meta scope keeps track of meta level declarations: generators, predicates, and meta-variables. A new meta scope is created by the following keywords: `#defgen`, `#defpred`, `#foreach`, `#when`, and quantifier keywords `forall` and

```
#defgen MakeImplement ( Class c, Interface i ) {
   // Define new class that extends c, implements i
   #[c.Modifiers] class NewC extends #[c] implements #[i] {
      // For all methods in c that implement methods in i
      #foreach(Method m : MethodOf(m, c)  && !Private(m) &&
                          (exists Method im : MethodOf(im, i) &&
                                              im.Name == m.Name &&
                                              im.Formals == m.Formals &&
                                              im.RetType == m.RetType)) {
         #[m.Modifiers] #[m.RetType] #[m] ( #[m.Formals] ) {
            #when( m.RetType == void ) { super.#[m](#[m.ArgNames]); }
            #else { return super.#[m](#[m.ArgNames]); }
         }
      }

      // For all methods in i that do not exist in c
      #foreach(Method im : MethodOf(im, i) &&
                           !(exists Method m : MethodOf(m, c) &&
                                               im.Name == m.Name &&
                                               im.Formals == m.Formals)) {
         // generate a default implementation
         #[im.Modifiers] #[im.RetType] #[im] ( #[im.Formals] ) {
            // for all primitive return types, return their default values.
            #when( im.RetType == int ) { return 0; }
            #else {
               #when( im.RetType == boolean ) { return false; }
               ... // repeat for all primitive types
               // for void-returning methods, do nothing.
               #when( im.RetType == void ) {}
               // finally, for non-native type, return null.
               #else { return null; }
            }
         }
       }

       // For all methods in c that are not in i, and do not conflict with i
      #foreach(Method m : MethodOf(m, c) &&
                          !(exists Method im: MethodOf(mi, i) &&
                                              im.Name == m.Name &&
                                              im.Formals == m.Formals)) {
         #[m.Modifiers] #[m.RetType] #[m] ( #[m.Formals] ) {
            #when( m.RetType == void ) { super.#[m](#[m.ArgNames]); }
            #else { return super.#[m](#[m.ArgNames]); }
         }
      }
   }
}
```

Fig. 3. Default implementation example.

`exists`. With the exception of `#when`, all of the keywords above create new meta-variable declarations.[3] In addition to keeping track of declarations, `#when` and `#foreach` meta scopes are also associated with the logical sentences under which they are created. Each meta scope is linked to at most one parent meta scope. For example, in Figure 2, the meta scope created by `#foreach` on line 4 has the `#foreach` scope created on line 2 as a parent. The declarations in parent meta scopes are visible in the children scopes.

An object scope (i.e., a scope for generated code) is very much like a type environment for regular Java type checking. It contains symbol tables for types, variables, and methods. However, there are two unique elements of our object scope. First, the entries in the symbol table (e.g., names of variables or methods declared in the scope, and the types these map to) may not be constants but instead refer to a cursor over the input program. Second, each entry in the symbol tables has a link to a meta scope within which the entry is declared. Intuitively, each entry has references that describe *what* are the variable's name and type, as well as a meta scope link that describes *when* (i.e., under what conditions) this variable declaration is generated. For example, consider the method declared on line 5 of Figure 2. The entry in the symbol table will not contain a constant method name, but the information that the method name is equal to the value of `m.Name`. Of course, `m` only makes sense in the context of the meta scope that this declaration is made under. Thus, the entry for this method links to the meta scope defined by the `#foreach` on line 4 (with parent meta scopes those on lines 2 and 1). Only meta scopes created with `#defgen`, `#foreach`, `#when` can be linked from object scope entries.

5.1.2 Algorithm.

SafeGen's type checking algorithm involves two phases. Phase I accomplishes the following two tasks:

Phase I.a) Fully populate meta scopes and type check the meta language. Type checking the meta language is simply ensuring that, first, every use of a meta-variable, predicate, function, or generator is defined, and secondly if a meta variable is used as an argument to predicates, functions, or generator calls, it has the correct type. For example, if meta-variables `m`, `c` are used in predicate `MethodOf(m, c)`, `m` should have a `Method` type, and `c` should have a `Class` or `Interface` type. This is an instance of standard and straightforward monomorphic type checking.

[3] The reader may wonder why `exists` and `forall` introduce a meta scope, since the quantified variables they introduce cannot appear in generated code. Nevertheless, we still have to check that references to these variables *inside* the constraint formulas are valid. This can be viewed as a third kind of scope: the lexical scope of SafeGen's first-order logic sub-language.

Phase I.b) Collect type information in code templates. Object scopes are partially populated with only type information for declared types, their methods, fields, and inner types. No statements are inspected. There is no legality checking done in this phase. This step is analogous to a conventional type checking algorithm, where a first pass is necessary to generate all the type information needed to type check the statements inside of method bodies and static initializers. After the object scopes are populated, we generate a logical representation of what is in the object scopes: a sentence in first-order logic describing the types available, their methods, fields, inner classes, etc. For the example in Figure 2, the initial segment of this sentence is:

```
forall([c],
  implies(and(Class(c), input(c)),
          exists([c'], and(Class(c'), Name(c')=Delegator, ...)))))
```

We call this sentence *fact*. It will be used in Phase II of the type checking algorithm, as described next.

Phase II is responsible for checking the type correctness of templated Java code. The algorithm resembles regular Java type checking in that it utilizes the symbol tables to look up information on variables, methods, and types. However, the algorithm is complicated by the use of meta-variables and functions in object-level declarations and references. Therefore, SafeGen's type system combines the use of object scope symbol tables with the building of logical sentences using the meta scopes (i.e., the meta scope associated with the current object scope and all its parent meta scopes). For example, in Figure 2, we need to check whether the method call, `super.#[m](#[m.ArgNames])` on line 6 is a valid call. The first step is to look up the superclass of the current class using the symbol table. However, we find that `super` does not point to an actual class with its own symbol tables, but to a meta-variable, `#[c]`. In order to check whether `super.#[m](#[m.ArgNames])` is a valid call, we must construct a logical sentence to inquire: under all legal inputs to this generator (any class that is `!abstract`), and under the logical context (encoded by the meta scope) in which this method call is used:

```
#foreach(Class c:input(c)) {
   ...
   #foreach(Method m:MethodOf(m,c) & !Private(m)) { ... }
}
```

does class `#[c]`'s superclass always have a method with name `#[m]`, and argument types matching the type of `#[m.ArgNames]`? SafeGen expresses this property as a first-order logic sentence, *test*. We present this sentence is SPASS format, shown in Figure 4.

```
implies(
forall([c],
  implies(
   and(input(c), class(c)),
   exists([del],
     and(class(del),
         equal(Name(del), Delegator),
         forall([sc], equiv(equal(SuperClass(del), sc), equal(c, sc))),
         forall([m],
           implies(and(equal(DeclaringClass(m), c), method(m)),
                   exists([del_meth],
                     and(method(del_meth),
                         equal(DeclaringClass(del_meth), del),
                         equal(Name(m), Name(del_meth)),
                         equal(RetType(m), RetType(del_meth)),
                         equal(Formals(m), Formals(del_meth)))))))))),
forall([c],
  implies(and(input(c), class(c)),
  forall([m],
    implies(
     and(equal(DeclaringClass(m), c), method(m), not(private(m)),
     exists([meth],
       and(method(meth), equal(Name(meth), Name(m)),
           exists([sc],
             and(equal(DeclaringClass(meth), sc),
                 exists([c'],
                   and(equal(DeclaringClass(meth), c'),
                       equal(SuperClass(c'), sc),
                       equal(Name(sc'), Delegator))))),
         equal(Formals(meth), Formals(m)))))))))
```

Fig. 4. Logical representation of the *test* property that the example "super" call is valid.

As the reader may notice, the sentence in Figure 4 does not correspond exactly to the logical language used in the definition of the generator in Figure 2. For example, there is no predicate `MethodOf(m, c)` in the sentence presented to SPASS. This is because the underlying logic SafeGen type checker uses does not match the SafeGen language constructs exactly. Several low-level conversions are performed in order to interface with the theorem prover. This is expected, as the logic is designed for maximal reasoning power, while the language is designed for ease of use. For instance, at the logic level we have more sorts and sub-sort hierarchies for logical entities. Furthermore, many of the concepts appearing as binary predicates at the SafeGen language level are expressed as functions in the logic. For example, the `MethodOf(m, c)` binary predicate, used earlier, is ideally represented more strictly by a function returning the class of each method, since a method cannot be in a `MethodOf` relation-

ship with more than one class. The same is true of predicates `SuperClassOf`, `InnerClassOf`, `ConstructorOf`, `FieldOf`, `ExceptionOf`, etc.

We then construct the sentence *fact* → *test*, where *fact* was constructed in Phase I, as described earlier. *fact* needs to be the condition in the implication because it states the existence of classes and methods that *test* might refer to. Facts about the well-formedness of generator inputs are also part of the theorem prover input, supplied as axioms. We next feed this sentence to the theorem prover to test its validity. The full input to the theorem prover includes the logic definition (i.e., predicates, functions, sorts), axioms about Java, and the *fact* → *test* conjecture. This is typically many hundreds of lines long.

5.1.3 Translation

A SafeGen generator interfaces with the outside world through Java reflection entities and strings. For instance, a generator that takes a `Class` argument, as above, is implemented as a Java method that accepts a `java.lang.Class` object as argument. Similarly, a generator that takes a collection of `Class`es, as in our second example, is implemented as a Java method that accepts a `java.lang.Class[]` as argument.

6 Discussion

We next discuss our design decisions and experience with SafeGen.

6.1 Choice of Logic

The design of the logic language allowed in SafeGen resulted from striving for a balance between expressiveness and our type checker's ability to reason about the logic. Clearly, a decidable fragment of logic would have been ideal, since our type checker would be able to say with certainty whether a type error exists. There are a number of general fragments of first-order logic that are known to be decidable [16]. However, we find unary functions (e.g., `SuperClass(c)`) as well as equality to be indispensible for the SafeGen language to be useful. These two requirements alone take us outside of any known decidable (prefix-)fragments of first-order logic. Thus, we decided early on in our design process to allow the full syntax of first-order logic to be applied, and appeal to the power of a theorem prover.

Within the confines of first-order logic, we can still tune our logic to limit

its expressiveness, and thus maximize the number of proofs we can produce completely automatically. That is, when we find in our examples that a specific pattern causes consistent difficulties in reasoning, we remove the logic feature it depends on. For instance, transitivity is very hard to reason about. The superclass relation is transitive, but instead of specifying the transitivity fully in our logic axioms, we only expand it three levels. As a result, if the validity of a generator depends on a subtyping relation between classes more than 3 links away in the subtyping hierarchy, then our logic cannot express the proof and SafeGen will issue a spurious warning.

Ordering is another property that is particularly difficult to reason automatically (partly because it involves reasoning about transitivity). In type checking, one place where ordering matters is the argument types of a method. For example, if a user defines a method `void foo ( int i, String s) { ... }` and later invokes it with `foo("bar", 2)`, the type checker should be able to catch this invalid reference. However, without a notion of ordering, the logic would only be able to express that there exists a method `foo`, with argument types `int` and `String`. Using this as a known fact, the type checker would erroneously infer that the call is valid. For this reason, we explicitly disallow referring to individual elements of a method's argument list. We provide only functions to refer to the argument list as one entity: `Formals`, `ArgTypes`, and `ArgNames`. The results of applying these functions cannot serve as the base collection for cursor definitions, either. For example, one *cannot* create a method with only the `public` members of another method's arguments:

```
void foo ( #foreach (Type t : m.ArgTypes & public(t)) { #[t] arg } )
{ ... }
```

6.2 Using the Theorem Prover.

There are two approaches to using the theorem prover to verify the correctness properties of generated code. We could construct a large sentence that is the conjunction of all the type-correctness properties the generated code should preserve, and ask the prover whether these properties hold given the facts produced by the code templates. While this approach simplifies our language implementation by delegating all type checking duties to the theorem prover, it has a major disadvantage. The checking would be all-or-nothing and it would not produce very useful error messages to the users. When one of the properties in the conjunction fails to be valid due to a contradiction, all we receive from the theorem prover is a series of syntactic maneuvers that arrived at the contradiction. It is very difficult to decipher these messages to determine the exact property that failed. We can only inform the user that, *somewhere* in their program, there is an error. The problem is exacerbated by spurious

errors due to valid formulas that could not be proven: the user would be unable to tell that the error is spurious if we just reject the entire program.

Therefore, we have chosen a second approach. SafeGen's type checking algorithm consists of a large number of simpler calls to the theorem prover. The calls check the validity of very specific properties. For example, when we are type-checking a class declaration, and we reach the declaration of a superclass, we make two calls to the theorem prover. One is to check that the declared superclass exists. Another is to check that the superclass is a non-final class. This approach yields simpler logic formulas to prove. At the same time, we are able to produce very precise error messages to the user regarding exactly which property the code template failed to establish.

The one disadvantage of our approach is that we must make many calls to the theorem prover in the process of compiling just one generator. There might be a potential performance hit depending on how long the theorem prover takes to return answers. However, as discussed next, we have not yet found this to be a major cause of concern.

6.3 Experience.

SafeGen is still work in progress. Nevertheless, we have experimented extensively with the checking process for formulas that correspond to SafeGen programs. In fact, we first chose example SafeGen programs and expressed in logic their properties that we wanted to check, before trying different theorem provers and eventually choosing SPASS.

The choice of theorem prover is largely orthogonal to the overall approach, and we may switch in the future. The overriding factor we used in choosing a theorem prover was its ability to arrive at a result without human guidance. We cannot expect the user of SafeGen to hand-tune the logic whenever the theorem prover fails. A theorem prover that fails to find either a definite proof of validity or a counterexample would cause SafeGen to produce lots of spurious warnings to users. After trying several (4) theorem provers, we chose SPASS because (in our tests) it demonstrated the best ability to terminate much of the time without human guidance. With our limited set of example validity tests, SPASS always finds a proof for the valid sentences. For sentences that are not valid, SPASS terminates with a decision roughly 50% of the time. It fails to terminate (during the several minutes we observed it) the other 50% of the time. This means that, for our examples, SafeGen issues no false positive errors. However, for half of the true type errors SafeGen reported, SafeGen was only able to report a "possible error", because SPASS did not terminate with a decision (i.e., a counterexample) that the sentence is not valid.

Because SafeGen makes a large number of calls to the theorem prover during type-checking, the performance of the theorem prover was a consideration, as well. So far, for the cases that SPASS was able to terminate, it terminates in a small fraction of the timeout we set for each proof attempt. (Timeouts are discussed in detail later.) This is hardly surprising: most of the properties we want to prove are quite shallow. For instance, for many type-checking tests, the types and meta scopes match exactly (i.e., the *test* logic sentence is verbatim, modulo variable names, a part of *fact*) even though they are complex expressions involving cursors and logic predicates.

It is worth noting that our delegator example in Figure 2 has a bug that SafeGen readily detects: the superclass method is not always guaranteed to have a return type. If the return type of method `m`, called in line 6, is `void`, then the statement `return super.#[m](#[m.ArgNames])` is not legal. The user should instead use a `#when` clause, to detect whether the superclass method has a returnable result and if not to just call it without attempting to return its value.

6.4 Cost of Compilation

The repeated calls to SPASS are responsible for the main portion of SafeGen's compilation cost. We ran the SafeGen compiler on three generators: the `MakeDelegator` example in Figure 2, `MakeImplement` in Figure 3, as well as an unshown generator, `SynchronizeMe` which generates a "synchronized" version of the input class, as described briefly in Section 4. (The generated class declares the same methods as the input class. Each method in the generated class synchronizes against a mutex before delegating the call to an object of the input class type.)

We show in Figure 5 compilation times collected for these generators. LOC denotes the lines of code for each generator; t_{total} denotes the total compilation time; t_{SPASS} denotes the time spent by SPASS proving FOL sentences; *# timeouts* denotes the number of times that SPASS is unable to terminate with a decision given the time limit. The current time limit for proof attempts is 3 seconds. Of all the FOL sentences for which SPASS is able to terminate with a definitive decision of validity, *ave.* t_{term} denotes the average time taken for SPASS to terminate; t_{max} denotes the maximum amount of time to termination; t_{min} denotes the minimum amount of time to termination. All times are in milliseconds.

As the data show, time spent in SPASS takes up roughly 94% of the total compilation time. Furthermore, the majority of the time spent in SPASS was spent on queries that never terminated before reaching the timeout limit: 92%.

	LOC	t_{total}	t_{SPASS}	$\#timeouts$	*ave.* t_{term}	t_{max}	t_{min}
`MakeDelegator`	11	4,044	3,684	1	217	420	19
`SynchronizeMe`	12	3,973	3,623	1	201	431	20
`MakeImplement`	33	12,933	12,406	4	144	266	31
Overall	56	20,950	19,713	6	181	431	19

Fig. 5. Compile-time statistics.

For those queries that *did* terminate, the average time per query is only 181 milliseconds. Given that our current timeout is set at 3 seconds, the time spent in SPASS (and thus the total compilation time) reduces dramatically if we reduce the timeout limit. The maximum time taken for a successful proof was less than half a second for the above generators, and generally we never observed a SPASS proof that successfully terminated in more than 1 second. Thus, reducing the timeout to 1 second might be reasonable if time becomes an issue.

An interesting point exhibited by the data is that the size of a SafeGen program does not have any correlation to the average time SPASS needs *per sentence*. In fact, the longest program we have, `MakeImplement`, has the shortest average termination time per sentence. This is not surprising given that a longer generator is not necessarily a more complex generator, and proving each local property depends only on the context (i.e., the contents of the meta scope). Thus, the theorem prover's performance for any given sentence is affected by the structure of the sentence, rather than its size. For instance, a sentence with more nested existential or universal quantification would be harder to prove than a simple, non-nested sentence of much greater size. Furthermore, an implication `A implies B` is much harder to automatically prove when `A` and `B` have different structures in terms of quantification nestings. Fortunately for SafeGen, most of the sentences the compiler generates are of the form `A implies B`, where `A` and `B` share largely the same structure. This could be the reason that the increase in problem size does not manifest itself in the average query time.

The SafeGen compiler is still a prototype. Type checking in imcompletely implemented. As we state in Section 3.3, the current implementation only checks for five types of errors. However, since the query time does not seem to increase with problem size, we expect the time for compilation to grow linearly with the number of calls made out to the compiler for generators of typical complexity.

The SafeGen static checking algorithm is intended to be sound: if a generator is approved by SafeGen, it is guaranteed to be correct (with respect to the supported tests, of course—but with no fundamental reason why these tests cannot in the future be all possible Java well-formedness tests). As in any static checking system, however, what matters most is not soundness but usefulness. After all, soundness is easy to achieve by just rejecting all programs. In the static checking arena, tools like ESC/Java [11] have garnered a lot of attention by trying to be useful, even though they are not sound.

We view the soundness argument as tied to another major decision, namely whether to support a hard-to-analyze programming language like Java as the meta-language, or to design a small, specialized language like SafeGen. If we were to implement our checking approach in a meta-programming system built on top of Java (such as our MAJ system [34]), we would certainly have sacrificed soundness to achieve usefulness. Java has several language constructs (including dynamic dispatch, aliasing and assignments, exceptions) that make it hard to be sound (i.e., guarantee correctness) while allowing a large percentage of the correct programs. Instead, our choice of creating a new language was largely so that we could be sound, yet useful. We believe that soundness is not a goal by itself, yet it is valuable in terms of user perception. Sound static checking mechanisms (such as type systems) are much more easily accepted by programmers than unsound tools (like ESC/Java) because they feel more disciplined. At the same time, we have aimed to make SafeGen expressive enough for most program generation tasks that depend on reflection over existing programs.

Of course, SafeGen checking offers no guarantees of completeness: if we find no proof of the correctness of the generator, it is by no means certain that it is erroneous. Since most interesting properties on first-order logic are undecidable, the proof process will not always terminate. We have examined the possibility of restricting our language to a broad but decidable fragment of first-order logic, such as the guarded fragment [2]. (In fact, SPASS, with the right choice of parameters is a decision procedure for the guarded fragment [13].) Nevertheless, we believe that this would limit significantly the expressiveness of our logic. Furthermore, it is not clear whether a guarantee of termination of the proof process with a decision is a very important property in practice, unless it is a guarantee of termination in a very short time, which seems impossible: such decision procedures typically have super-exponential complexity.

7 Related Work

We have mentioned throughout the paper some related work, such as work on multi-stage languages, and other solutions to class adaptation and expansion. Nevertheless, this does not cover the work most closely related to SafeGen. Concurrently or subsequently to the original SafeGen publication [19], other work has appeared in the area of combining reflection with generation and much of it has drawn inspiration from SafeGen.

Two such projects are Genoupe [9] and Compile-Time Reflection (CTR) [10]. Both of these are extentions to the C# language. Genoupe allows users to write programs by reflecting over C# types, using controlled `@foreach` and `@if` constructs very similar to SafeGen's `#foreach` and `#when`. Genoupe provides an integrated way to both generate and reference the newly generated type in the program. However, Genoupe iteration and conditionals are based on values that may change at runtime, which makes the type system unsound. CTR provides tighter control over its iteration and branching conditions such that they can only be based on values known at compile-time. Furthermore, CTR introduces the use of *patterns* for expressing the properties of generated code, which is a major syntactic simplification. For instance, instead of a logical property that states that a method should have any name and an `int` type argument, the user can just write a matching method signature and use a pattern variable for the name. In terms of checking, however, CTR elides an important issue: the uniqueness of declarations. A transform written in CTR could declare a variable with potential naming conflict to the program it may transform, but this is not caught at the compile-time of the transform itself, but at the time the transform is *applied*. (In fact, the potential conflict is not reported as an error, but just used to determine that the transform is not applicable.) This is precisely the point we argued against earlier—the compile-time of the resulting, transformed program is the run-time of the generator, and the naming conflict in the generated program is really a bug in the generator itself, and should be caught by the generator writer.

SafeGen has also inspired our subsequent work on integrating reflection-based program construction tightly with Java generics. cJ [20] is an extension of Java where methods, fields, and supertypes of a class (or interface) can be declared based on static type conditionals. These type conditionals are only subtyping conditions (i.e., is one type the subtype of another?). This is analogous to a `#when` condition in SafeGen. MorphJ (a.k.a. MJ) [21,18] is more sophisticated in that it combines static iteration with pattern-matching. For example, in MorphJ, a class may contain a series of methods defined in a static loop, based on the methods of another type, matching a particular pattern on the method signature. Method signature patterns can involve the use of both type and name variables. This is analogous to a `#foreach` loop in SafeGen, where a

pattern is converted to a conjunctive formula in logic. SafeGen is a superset of both cJ and MorphJ in the sense that it allows all the program configurations that these languages allow, and much more (e.g., negation in static conditions, unlimited levels of reflection). What SafeGen does not provide is the seamless integration into the object language (Java) that both cJ and MorphJ provide. SafeGen is very much a separate generation language. Because of their limited expressiveness (as compared to SafeGen), cJ and MorphJ are able to have relatively simple to state typing algorithms.

8 Conclusions

In this paper we presented SafeGen, a meta-programming language with the distinguishing feature that it offers powerful correctness guarantees for generators expressed in it. SafeGen statically checks its input to guarantee that only well-formed code will be generated at the generator's runtime. We demonstrated a novel approach that combines traditional static type checking with representing program correctness properties in logic. We believe that SafeGen is expressive and useful, even though its syntax is restricted so we can represent all program correctness properties logically. We also believe that the approach of using logic to control and reason about code generation is one that extends beyond the implementation of SafeGen. It can be used for a different target language (from Java), and with a different logic (from one based on Java reflective properties), suitable for other broad categories of generation needs.

Acknowledgments.

This research was supported by the National Science Foundation under Grant No. CCR-0735267.

References

[1] Java Collections Framework Web site, http://java.sun.com/j2se/1.5.0/docs/guide/collections/, Accessed Feb. 2008.

[2] H. Andreka, J. van Benthem, I. Nemeti, Modal languages and bounded fragments of predicate logic, Journal of Philosophical Logic 27 (3) (1998) 217–274.

[3] J. Bachrach, K. Playford, The Java syntactic extender (JSE), in: Proc. of the 16th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications, ACM Press, Tampa Bay, FL, USA, 2001.

[4] J. Baker, W. C. Hsieh, Maya: multiple-dispatch syntax extension in Java, in: Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, ACM Press, Berlin, Germany, 2002.

[5] D. Batory, B. Lofaso, Y. Smaragdakis, JTS: tools for implementing domain-specific languages, in: Proc. Fifth Intl. Conf. on Software Reuse, IEEE, Victoria, BC, Canada, 1998.

[6] A. Bryant, A. Catton, K. De Volder, G. C. Murphy, Explicit programming, in: Proc. of the 1st international conference on Aspect-Oriented Software Development, ACM Press, Enschede, The Netherlands, 2002.

[7] B. Burke, et al., JBoss AOP Web site, http://labs.jboss.com/portal/jbossaop, accessed Feb. 2008.

[8] C. Calcagno, W. Taha, L. Huang, X. Leroy, Implementing multi-stage languages using ASTs, gensym, and reflection, in: Generative Programming and Component Engineering (GPCE) Conf., LNCS 2830, Springer, 2003.

[9] D. Draheim, C. Lutteroth, G. Weber, A type system for reflective program generators, in: Proc. of the 4th Intl. Conf. on Generative Programming and Component Engineering, LNCS 3676, Springer-Verlag, Tallin, Estonia, 2005.

[10] M. Fähndrich, M. Carbin, J. R. Larus, Reflective program generation with patterns, in: Proc. of the 5th Intl. conference on Generative Programming and Component Engineering, ACM Press, Portland, OR, USA, 2006.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended static checking for Java, in: Proc. of the ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation, ACM Press, 2002.

[12] E. Gamma, R. Helm, R. Johnson, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, Addison-Wesley, 1995.

[13] H. Ganzinger, H. de Nivelle, A superposition decision procedure for the guarded fragment with equality., in: LICS, 1999.

[14] R. Glück, J. Jørgensen, Efficient multi-level generating extensions for program specialization, in: Programming Languages, Implementations, Logics, and Programs, Utrecht, The Netherlands, September 1995. (Lecture Notes in Computer Science, vol. 982), Berlin: Springer-Verlag, 1995.

[15] J. Gosling, et al., The Java Language Specification, GOTOP Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan.

[16] E. Grädel, Decidable fragments of first-order and fixed-point logic. From prefix-vocabulary classes to guarded logics, in: Proc. of Kalmár Workshop on Logic and Computer Science, Szeged, 2003.

[17] S. S. Huang, Y. Smaragdakis, Easy language extension with Meta-AspectJ, in: Proc. of International Conference on Software Engineering (ICSE), 2006.

[18] S. S. Huang, Y. Smaragdakis, Class morphing: Expressive and safe static reflection, in: Conf. on Programming Language Design and Implementation (PLDI), ACM, 2008.

[19] S. S. Huang, D. Zook, Y. Smaragdakis, Statically safe program generation with SafeGen., in: Proc. of the 4th Intl. Conf. on Generative Programming and Component Engineering, LNCS 3676, Springer-Verlag, Tallin, Estonia, 2005.

[20] S. S. Huang, D. Zook, Y. Smaragdakis, cJ: Enhancing Java with safe type conditions, in: Proc. of the 6th Intl. Conf. on Aspect-Oriented Software Development, ACM Press, Vancouver, British Columbia, Canada, 2007.

[21] S. S. Huang, D. Zook, Y. Smaragdakis, Morphing: Safely shaping a class in the image of others, in: E. Ernst (ed.), Proc. of the European Conf. on Object-Oriented Programming (ECOOP), LNCS, Springer-Verlag, 2007.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: Proc. of the 15th European Conf. on Object-Oriented Programming, Springer-Verlag, London, UK, 2001.

[23] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (eds.), Proc. of the 11th European Conf. on Object-Oriented Programming, vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[24] M. Mohnen, Interfaces with default implementations in Java, in: Proc. of the inaugural conference on the Principles and Practice of Programming, 2002, National University of Ireland, Maynooth, County Kildare, Ireland, Ireland, 2002.

[25] E. Pasalic, W. Taha, T. Sheard, Tagless staged interpreters for typed languages, in: ICFP '02: Proc. of the seventh ACM SIGPLAN international conference on Functional programming, ACM Press, New York, NY, USA, 2002.

[26] T. Sheard, S. P. Jones, Template meta-programming for Haskell, in: Proc. of the ACM SIGPLAN workshop on Haskell, ACM Press, Pittsburgh, Pennsylvania, 2002.

[27] A. Stevens, et al., XDoclet Web site, http://xdoclet.sourceforge.net/, accessed Feb. 2008.

[28] W. Taha, T. Sheard, Multi-stage programming with explicit annotations, in: Proc. of the 1997 ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation, ACM Press, Amsterdam, The Netherlands, 1997.

[29] E. Tilevich, S. Urbanski, Y. Smaragdakis, M. Fleury, Aspectizing server-side distribution, in: Proc. of the Automated Software Engineering (ASE) Conf., IEEE Press, 2003.

[30] E. Visser, Meta-programming with concrete object syntax, in: Generative Programming and Component Engineering (GPCE) Conf., LNCS 2487, Springer, 2002.

[31] A. Warth, M. Stanojevic, T. Millstein, Statically scoped object adaptation with expanders, in: Proc. of the 21st annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications, ACM Press, Portland, OR, USA, 2006.

[32] C. Weidenbach, The theory of spass version 2.0, Tech. rep., Max-Planck Institute fur Informatik.

[33] D. Weise, R. F. Crew, Programmable syntax macros, in: SIGPLAN Conf. on Programming Language Design and Implementation, 1993.

[34] D. Zook, S. S. Huang, Y. Smaragdakis, Generating AspectJ programs with meta-AspectJ, in: Proc. of the 3rd Intl. Conf. on Generative Programming and Component Engineering, Springer-Verlag, Vancouver, British Columbia, Canada, 2004.