LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
SOFTWARE AND COMPUTATIONAL SYSTEMS LAB

BACHELORARBEIT

Extending the Framework JavaSMT with the SMT Solver Yices2

Michael Obermeier

Prüfer: Prof. Dr. Dirk Beyer
Mentor: Karlheinz Friedberger

Abgabetermin: 13.03.2020

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 13.03.2020

. . . . . .
(Unterschrift des Kandidaten)

Abstract

JavaSMT, developed at the Software and Computational Systems Lab at the Ludwig-Maximilians-Universität München, is a common API for SMT solvers. It offers access to a selection of solvers developed in Java as well as other programming languages. Support for those non-Java solvers is achieved through either existing or self-developed language bindings. While most solvers have a mostly identical core set of supported theories and features, they still differ by availability of additional theories and performance. As such adding more solvers to the framework will always be beneficial to the users. The Yices2 SMT solver, developed at SRI International's Computer Science Laboratory was chosen as an addition because of its large feature set and extensive, well documented API. In this paper we will go over how the needed Java binding was developed and the integration with the JavaSMT API works. We will also cover the problems that were encountered while adapting the Yices2 API to JavaSMT's and the solutions that were implemented. After covering the implementation, we will evaluate the performance of Yices2 against existing solvers in JavaSMT using the CPAchecker software verification framework and the SV-benchmarks set of verification tasks, which are also maintained at the SoSy-Lab.

Contents

1 Introduction

Today software and hardware are becoming ever more complex due to the advent of cloud services, internet of things, software as a service and rising hardware performance requirements for tasks like machine learning or weather predictions. As software gains more features increasing the code base size and hardware contains more different circuits than before, both become increasingly difficult to verify in a manual review or through test cases. Thus additional tools like SMT solvers are needed to help ensure a piece of hard- or software works like it should. Since these SMT solvers have varying degrees of capabilities and performance and may not accept the same input, various frameworks that offer a unified API to several SMT solvers were created. One of these frameworks is JavaSMT[1] to which the solver Yices2 was added as part of this work. JavaSMT is a common API for multiple SMT solvers written in Java. For an overview which solvers are available in JavaSMT compared to other frameworks see table 1.1. It is developed by the Software and Computational Systems Lab (SoSy-Lab)[2] at Ludwig-Maximilians-Universtität München and distributed under the Apache 2.0 license. If the license of a solver allows it, compiled binaries are also distributed for a simpler setup. The goal of JavaSMT is to provide a performance optimized, customizable and type safe interface for various solvers. A simple example of solving a formula with Yices2 is given in listing 1.1. For comparison an equivalent example in SMT-LIB2 notation is given in listing 1.2. Additionally to simple solving JavaSMT can provide additional features such as Interpolation or AssumptionSolving when the selected solver supports them. For an overview of these features and their support see table 1.2. A short explanation of each feature can be found in 2.5. Since the theory support differs by solver and JavaSMT does not necessarily support all theories offered by a solver, an overview of available theories can be found in table 1.3. For a closer look at JavaSMT see [KFB16].

This work is divided into three major parts. Chapter 2 gives a short introduction of Satisfiability Modulo Theories, the JavaSMT framework, the Yices2 solver and the Java Native Interface, which was used to build a wrapper between the C-Code of Yices2 and the Java based JavaSMT. The Chapters following 3 will then cover how the implementation was done, the encountered problems and how they were solved. The final chapter 7 then explains the programs used for evaluation and compares the performance of Yices2 to the already available solvers.

[1] `https://github.com/sosy-lab/java-smt`, 12.2019
[2] `https://www.sosy-lab.org`, 12.2019

`Highlighted text` refers to relevant program parts, such as variables, methods and macros. A prefix `yices_` refers to methods from Yices2's API.[3]. `UPPER-CASE TEXT` indicates a macro found in the JNI binding(see 3.1). The majority of remaining highlights refer to methods within the Yices2 adapter classes in JavaSMT and are usually referred to in context of their containing class.

	Boolector[4]	Z3[5]	MathSAT5[6]	CVC4[7]	Princess[8]	SMTInterpol[9]	Yices2[10]	PicoSAT[11]	SWORD[12]
JavaSMT[13]	✓	✓	✓	✓	✓	✓	✓	✗	✗
ScalaSMT[14]	✗	✓	✓	✓	✗	✓	✓	✗	✗
MetaSMT[15]	✓	✓	✗	✓	✗	✗	✗	✓	✓
PySMT[16]	✓	✓	✓	✓	✗	✗	✓	✓	✗

Table 1.1: Solvers supported by JavaSMT and other frameworks. Other frameworks may support additional solvers.

[3] An API reference can be found under: `https://yices.csl.sri.com/doc/index.html`, 02.2020
[4] `https://boolector.github.io`, 03.2020
[5] `https://github.com/Z3Prover`, 03.2020
[6] `http://mathsat.fbk.eu/`, 03.2020
[7] `https://cvc4.github.io`, 03.2020
[8] `http://www.philipp.ruemmer.org/princess.shtml`, 03.2020
[9] `https://ultimate.informatik.uni-freiburg.de/smtinterpol/`, 03.2020
[10] `https://yices.csl.sri.com`, 03.2020
[11] `http://fmv.jku.at/picosat/`, 03.2020
[12] `http://www.informatik.uni-bremen.de/agra/eng/sword.php`, 03.2020
[13] `https://github.com/sosy-lab/java-smt`, 12.2019
[14] `https://bitbucket.org/franck44/scalasmt/src/master/`, 12.2019
[15] `http://www.informatik.uni-bremen.de/agra/eng/metasmt.php`, 12.2019
[16] `https://github.com/pysmt/pysmt`, 12.2019

	Boolector	CVC4	MathSAT5	Princess	SMTInterpol	Z3	**Yices2**
AllSAT	✗	✗	✓	✓	✓	✓	✗
AssumptionSolving	✓	✗	✓	✗	✗	✓	✓
Interpolation	✗	✗	✓	✓	✓	✓	✗
Optimization	✗	✗	✓	✗	✗	✓	✗
UnsatCore	✗	✓	✓	✓	✓	✓	✓
UnsatCore with Assumptions	✗	✗	✓	✗	✗	✓	✓

Table 1.2: Additional features supported by the available solvers. See section 2.5 for explanations.

	Boolector	CVC4	MathSAT5	Princess	SMTInterpol	Z3	**Yices2**
Bool	✓	✓	✓	✓	✓	✓	✓
BV	✓	✓	✓	✓	✗	✓	✓
Int	✗	✓	✓	✓	✓	✓	✓
Real	✗	✓	✓	✗	✓	✓	✓
Float	✗	✓	✓	✗	✗	✓	✗
UF	✓	✓	✓	✓	✓	✓	✓
Array	✓	✓	✓	✓	✓	✓	✗
QF	✓	✗	✗	✓	✗	✓	✗

Table 1.3: Theories available in JavaSMT.

Listing 1.1: Solving a simple formula with Yices2 using JavaSMT

```
// Instantiate JavaSMT with Yices2 as backend
try (SolverContext context = SolverContextFactory.
    ↪ createSolverContext(config, logger, shutdownNotifier
    ↪ , Solvers.YICES2)) {
  IntegerFormulaManager imgr = context.getFormulaManager()
     ↪ .getIntegerFormulaManager();

  // Create formula "a = b" with two integer variables
  IntegerFormula a = imgr.makeVariable("a");
  IntegerFormula b = imgr.makeVariable("b");
  BooleanFormula f = imgr.equal(a, b);

  // Solve formula, get model, and print variable
     ↪ assignment
  try (ProverEnvironment prover = context.
     ↪ newProverEnvironment(ProverOptions.GENERATE_MODELS
     ↪ )) {
    prover.addConstraint(f);
    boolean isUnsat = prover.isUnsat();
    assert !isUnsat;
    try (Model model = prover.getModel()) {
      System.out.printf("SAT with a = %s, b = %s",
         ↪ model.evaluate(a),
         ↪ model.evaluate(b));
    }
  }
}
```

Listing 1.2: SMT-LIB2 equivalent to listing 1.1

```
(set-logic QF_LIA)
(declare-const a Int)
(declare-const b Int)
(assert (= a b))
(check-sat)
(get-value (a b))
(exit)
```

2 Background

In this first chapter we will give a short introduction of Satisfiability Modulo Theories and the SMT-LIB2 Standard, as well as the used programs and their capabilities.

2.1 Satisfiability Modulo Theories

Satisfiability Modulo Theories (short SMT) are a class of decision problem (Yes-No answer) used in various tasks of computer science such as software verification. They are an extension of the Boolean satisfiability problem (SAT) with predicates and theories, other than the boolean theory, such as the theory of integers or bitvectors. Because of SMT extending SAT, solving SMT formulae is also a NP-complete problem. This relationship also makes it possible to translate SMT formulae into SAT formulae and use proven SAT solving techniques like the Davis–Putnam–Logemann–Loveland (DPLL) algorithm. But due to DPLL being inefficient in discovering theory specific facts, like that $x + y = y + x$ holds for integer values, most solvers now use a more refined approach called DPLL(T). This approach allows the DPLL-based SAT solver to interact with a theory specific solver T to 'learn' more about the given formula. For details on the DPLL(T) approach see [NOT06].

2.2 SMT-LIB2

SMT-LIB2[1] is a standard aiming to provide common definitions for SMT theories and associated logics as well as a unified language for writing input formulae for SMT solvers.

2.3 Yices2

Yices2[2] is a SMT solver mainly written in C that is being developed at SRI International's Computer Science Laboratory[3]. It is open source software distributed under the GPLv3 license. Since it is incompatible with the Apache 2.0 license, that JavaSMT is under, a compatible one, that would allow the non-commercial use of Yices2 as part of JavaSMT and CPAchecker, was requested. But due to the developers not wanting to change the license as requested, Yices2 can only be offered as an optional part of JavaSMT. Yices2 offers support for a wide range of SMT theories, model generation and exploration as well as some additional features for solving as listed in table 1.2. It can use the DPLL(T) approach or Model Constructing Satisfiability Calculus (MCSAT)[4] for solving with the

[1] `http://smtlib.cs.uiowa.edu/index.shtml`, 12.2019
[2] `https://yices.csl.sri.com/index.html`, 12.2019
[3] `https://www.sri.com/about/organization/information-computing-sciences/computer-science-laboratory`, 12.2019
[4] For details on MCSAT see [JBd13] and [JdM12]

latter supporting non-linear arithmetic. When compiling Yices2 from source MCSAT is optional and was left out in this implementation as it currently only supports the one-shot mode for solving, while JavaSMT requires push-pop. For more information on the architecture of Yices2 the tool paper [Dut14] can be read.

2.4 Java Native Interface

The Java Native Interface(JNI)[5] allows Java programs to interact with programs written in another language. This helps if a certain program can reach significant performance improvements in a lower level language or an existing library/program can be used in Java without re-implementing. Additionally to just calling a method, it also enables the native method to access and manipulate Java objects, call Java methods and more. For Java to be able to call a native method a binding, such as the one shown in listing 3.1, has to be written and compiled into a library, which then has to be loaded in Java (see 3.2) before calling the native method.

2.5 Explanations for Table 1.2

AllSAT The solver can find all satisfying assignments to a given set of predicates with a set of formulae.

AssumptionSolving The solver can use additional assumptions during solving without adding them to the context beforehand.

Interpolation The solver can compute a formula ψ from φ_1 and φ_2 where the following holds: $\varphi_1 \wedge \varphi_2$ is unsatisfiable, $\varphi_1 \Rightarrow \psi$ holds, $\psi \wedge \varphi_2$ is unsatisfiable and all variables in ψ occur in both φ_1 and φ_2.

Optimization The solver can be told to maximize or minimize the value of a given term during solving.

UnsatCore For an unsatisfiable problem, the solver can give a small set of constraints causing the unsatisfiability.

UnsatCore with Assumptions The solver can generate an UnsatCore when assumptions were used during solving.

[5] `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html`, 12.2019

3 Basic Implementation

In this chapter we will cover the initialization of Yices2 for use with JavaSMT and the handling of types and terms. But first we will explain some quality of life improvements that were used for the C part of the wrapper.

3.1 JNI Macros

Manually writing a JNI wrapper for an existing API requires defining the method signature, meaning name, argument types and return type, for each needed method. Additionally arguments must be copied from the Java argument into a native equivalent, before using them to call a native method. Likewise it needs to be ensured that the return of a method can be handled by Java. For an example of how such a JNI binding for JavaSMT of the simple method `yices_not`, that takes one term and returns its negation, would look like, see Listing 3.1. As can be seen in the example, even ignoring the return error handling, this is a lot of code for one method call. It also contains unnecessary information like the full Java package name, which in the case of JavaSMT is the same for each method. Also since many methods of the Yices2 API have similar or even identical signatures, for example all functions for checking if a term is a certain type, writing out each method like in 3.1 would cause a lot of code duplication, making the resulting code harder to read, understand and maintain. Thus we used C macros to replace multiple occurrences of (near) identical code to alleviate this issue. The listing 3.2 shows the same `yices_not` method as before, although this time using these macros. It will result in the same code as 3.1, but is more compact and the key parts, such as the method name, argument type, number of arguments and return type, are more easily identifiable. This also has the benefit of only requiring editing one macro instead of each declaration, if something like the package name were to change or additional error checking is needed for a common return type. The macros and the JNI binding can be found under `/lib/native/source/yices2j/`.

3.2 Initializing

Because Yices2 is a native library it first needs to be loaded in Java before any of its methods can be called. This is done in the `create()` method in `Yices2SolverContext` with the call to `NativeLibraries.loadLibrary("yices2j");`. Due to the platform independent nature of Java, the given name is first expanded to fit the platform specific naming scheme of the current environment, e.g. `libyices2j.so` for Linux. Afterwards Java searches for a matching library in and if one is found loads it. Now that the library is loaded the next step is to initialize Yices2 with a call to the native method `yices_init()`. This allows Yices2 to setup its internal data structures for storing type,

Listing 3.1: JNI binding

```
JNIEXPORT jint JNICALL
    ↪ Java_org_sosy_1lab_java_1smt_solvers_yices2_
    ↪ Yices2NativeApi_yices_1not (JNIEnv *jenv, jclass
    ↪ jcls, jint arg1) {
term_t m_arg1 = arg1;
term_t retval = yices_not(m_arg1);
if (retval <= 0 && yices_error_code() != 0){
  const char *msg = yices_error_string();
  throwException(jenv, "java/lang/IllegalArgumentException
    ↪ ", msg);
  return -1;
}
return (jint)retval;
}
```

Listing 3.2: JNI binding with Macros

```
DEFINE_FUNC(jterm, 1not) WITH_ONE_ARG(jterm)
TERM_ARG(1)
CALL1(term_t, not)
TERM_RETURN
```

term, model information and so on. After setting up Yices2 the `create()` method initializes the theory specific formula managers and passes them to the `Yices2FormulaManager` as arguments, before finally creating a `Yices2SolverContext`. Now that we have a solver environment we will go over creating types and terms.

3.3 Types

In this part we will explain how Yices2 handles types and how they are created in JavaSMT.

3.3.1 Types in Yices2

Yices2 stores type information internally and offers type creation, testing and access methods. All these functions either return or accept a value `type_t` that serves as a reference to the internal table. Since `type_t` is defined as `typedef int32_t type_t;` it can be easily passed back and forth between C and Java. Since, as described earlier, JavaSMT uses only a handful of the types supported by Yices2, only the methods needed for those were implemented. The methods for getting the Bool, Int and Real type are a simple call with no arguments, since these are non-parametric. The method for Bitvectors is also very simple, as it just needs a positive integer as a parameter to define the size. Finally the function for getting a Function type of arbitrary arity is the most complex,

as it requires passing a positive integer n defining the arity, an array of types of length n containing the domain types and a range type. Each of these types has an associated method for testing if a given `type_t` is of said type. The access functions allow to retrieve the size of a Bitvector type as well as the number and type(s) of children of a Function type.

3.3.2 Types in JavaSMT

In JavaSMT the `Yices2FormulaCreator` is responsible for handling the types. The values for Bool, Int and Real type are initialized along with the `Yices2FormulaCreator` itself as these remain the same throughout the execution. As the Bitvector type is parametric it has its own method `getBitvectorType` for getting different sizes of Bitvector types. Since the creation of Function types is closely tied to uninterpreted functions it will be covered in Section 3.4.6. The type tests and access functions for a type's child(ren) are used while visiting a formula, dumping formulas in SMT-LIB2 format and during Model exploration.

3.4 Terms

This section will first cover how terms are handled by Yices2, then cover how the different theories were implemented in JavaSMT.

3.4.1 Terms in Yices2

Similar to types Yices2 stores the contents of terms internally and only allows manipulation through some generic as well as theory specific constructors, test and access functions. These return and/or accept terms as a `term_t` which is also defined with `typedef int32_t term_t;` and thus is also easily transferred between C and Java.

3.4.2 Symbols

Yices2 offers three different kinds of symbols for use in terms. The first are 'constants' which are used for scalar and uninterpreted types. As these types are not used in JavaSMT, they are not relevant for this implementation. 'Variables' are the second kind and used in lambda terms and quantifiers. While JavaSMT can use quantified formulas, there currently is no way to create specific symbols for use as quantifiers. A solution for this would be building a map between quantifier symbols and general symbols when creating quantified formulae, but it was chosen not to do this for now. The final kind of symbol are 'uninterpreted terms' which are usable with all remaining types. When creating these Yices2 by default does not set a name, but one can be set afterwards. It is of note that if two symbols are created and the same name is set for both, the first will lose its associated name. As it is possible to call the `makeVariable` function in the `Yices2FormulaCreator` multiple times with the same type and name values, this would create a set of symbols with different ids, of which only the last would carry the name. Since JavaSMT expects to get the same actual symbol in this scenario, a cache was added, which either returns an already existing symbol of same name and type, creates

and names a new symbol if no symbol with this name exists or throws an exception if a symbol with an existing name but different type is requested for creation.

3.4.3 Boolean terms

Boolean terms are created in the `Yices2BooleanFormulaManager` and since the Boolean theory is the most basic one, the methods simply needed to be matched with their Yices2 equivalents. Note that for *and*, *or* and *xor* the 2-ary versions were used. The super implementation of `Yices2BooleanFormulaManager` contains wrappers for n-ary *and* as well as n-ary *or*, which also simplify them to Boolean constants if appropriate, e.g. an *and* term containing a $false$ constant. Additionally functions for creating Boolean constants and variables exist.

3.4.4 Bitvector terms

Just like for the Boolean theory, implementing the `Yices2BitvectorFormulaManager` mostly just required matching the appropriate signed or unsigned Yices2 functions to the JavaSMT methods. However the two methods equivalent to SMT-LIB's *bv*2*int* and *int*2*bv* could not be implemented as Yices2 lacks support for these as of version 2.6.1. Another more complex function is `makeBitvectorImpl` which creates a bitvector of a certain length from a `BigInteger` value. First the value is range checked and if negative transformed into the matching positive representation using the super method `transformValueToRange`. The result is then converted to a binary string and zero padded as needed. Finally the string is passed to `yices_parse_bvbin` to create the bitvector.

3.4.5 Arithmetic terms

The creation of arithmetic terms in JavaSMT is split in three classes. The abstract `Yices2NumeralFormulaManager` containing the arithmetic and comparative operations, which are usable for both Integer and Rational, as well as general functions for making arithmetic constants. Extending this base class the `Yices2IntegerFormulaManager` and `Yices2RationalFormulaManager` then add theory specific functions such as modular congruence and the appropriate division type. Also note that all arithmetic operations, that would require nonlinear arithmetic to solve, are checked eagerly if they contain a constant before creating the formula. This is done to ensure solvability as the DPLL(T) solver of Yices2 can not solve nonlinear constraints.

3.4.6 Uninterpreted functions

The `Yices2UFManager` extends the `AbstractUFManager` and only contains a constructor, delegating the creating and calling of uninterpreted functions entirely to its super class. The `AbstractUFManager` in turn uses the methods `declareUFImpl` and `callFunctionImpl` implemented in the `Yices2FormulaCreator` for this purpose.

declareUFImpl The `declareUFImpl` method is responsible for declaring an uninterpreted function from a list of argument types, a return type and a name. If the ar-

gument list is empty the function is nullary and thus is treated as a variable with the type being the return type. When it is not empty a function type is created using `yices_function_type`. This method gets passed an array of the argument types, its length as well as the return type and returns a function type, the arity of which is defined by the length of the array. The return value and the given name can then be used with `yices_named_variable` to declare the uninterpreted function of this this type.

callFunctionImpl The method `callFunctionImpl` calls a given uninterpreted or built-in function identified by a term constructor (see 4.1.2 for details) with the given list of parameters. Since there exists a collision between term constructor values and UF ids as described in 4.4.1 the method first checks if the received declaration is less than zero. In this case the declaration is a built-in function and gets matched against the possible term constructor values in a switch case. When a match is found the list of arguments is tested for correct length if the particular function requires a fixed argument count and then the appropriate term construction method is called. If the declaration is positive it is an uninterpreted function declaration and `yices_application` is called to apply the list of arguments to the UF.

4 Visiting and dumping

JavaSMT supports visitation, dumping and parsing of SMT formulae. In this chapter we will talk about the term properties in Yices2, give an overview of the process for visiting terms in JavaSMT then finally cover dumping and parsing in SMT-LIB2 notation.

4.1 Term properties in Yices2

Apart from the type a term in Yices2 has several more properties that can be tested or retrieved. We will explain the properties used for visiting in this section.

4.1.1 Term classes

In Yices2 terms are categorized in four classes, which are *atomic*, *composite*, *projection* and *polynomial*. These classes determine the kind of term and which property access functions can be used, e.g. `yices_term_child` can only be used for *composite* terms. The *atomic* class contains the different types of constants and variables/uninterpreted terms. Most built-in functions are in the *composite* class as these have one or more child terms. The *projection* class contains one method specific to tuples and one for extracting bits from a bitvector. Finally arithmetic and bitvector sums as well as power products are classed as *polynomial* and have their own specific functions for retrieving their components.

4.1.2 Term constructor

To determine which function a term is `yices_term_constructor` is used. This method returns a `term_constructor_t`, which is defined as an `enum` containing a list of the built-in functions, constants and variables as well as a special error value for invalid terms. When visiting a term in JavaSMT the term constructor is used in most cases to determine the function kind and match it to a `FunctionDeclarationKind` defined by JavaSMT.

4.1.3 Value

Yices2 offers a method in the form of `yices_TYPE_const_value` to retrieve the value of the Bool, Bitvector, Arithmetic and Scalar (unused in JavaSMT) constant type. While the retrieved values for the first two types can be easily passed back to Java as they are represented by an `int32_t` and an array of `int32_t` values respectively, the value of arithmetic constants is given as a GMP[1] rational `mpq_t`. Since these GMP rationals are structs and cannot be handled by Java in this form, the `MPQ_RETURN` macro converts them to strings using the GMP method `mpq_get_str`.

[1] GNU Multiple Precision Arithmetic Library, `https://gmplib.org`, 02.2020

4.2 Basic visiting principle

When a given formula is visited, it gets analyzed from the outside in. In each step of the analysis, information about the current part of the formula, such as values for constants, names for variables and child terms of functions, is collected until the formula is fully explored. The collected data can then be used to simplify other actions, like dumping the formula in SMT-LIB2 format for use with another solver as described in 4.7.

4.3 Constants and variables

The `visit` function in the `Yices2FormulaCreator` checks if the given formula is a constant, an uninterpreted term or a function based on the term constructor. If it is a function the analysis is delegated to the `visitFunctionApplication` method 4.4. For an uninterpreted term the formula and the name, obtained with `yices_get_term_name`, are recorded. Finally for constants the value and formula are stored. Obtaining the value of boolean constants is simply done using `yices_bool_const_value`, while the values of arithmetic and bitvector constants require conversion to their matching Java type, which is done in the `convertValue` method. This method in turn passes the conversion task to an appropriate method, based on the type of the formula. If the constant is arithmetic, the conversion is done in the `parseNumeralValue` function which obtains the value as a `String` (see 4.1.3) using `yices_rational_const_value` and parses it as either a `Rational` or `BigInteger` based on the type of the formula. For bitvector constants the `parseBitvector` function takes the integer array returned by `yices_bv_const_value`, which is in little endian order, reverses and makes a `String` out of it. This `String`, now in big endian order, is then converted to a BigInteger.

4.4 Functions

This section describes the handling of most built-in Yices2 functions and uninterpreted functions, while the following sections describe special cases and how they are handled on JavaSMT's side. If a formula is recognized as a function it is analyzed by the `visitFunctionApplication` method. The following section will go over what kind of information is collected and how.

4.4.1 Built-in

This section describes how the information is obtained for non special cases.

FunctionDeclarationKind The `FunctionDeclarationKind` is an `enum` defined in JavaSMT to uniformly identify which operation the function represents, e.g. OR for a boolean or operation. The kind of most functions is simply determined by matching the Yices constructor values to the appropriate `FunctionDeclarationKind` in a switch case. Some special cases require further information to accurately identify them and are described later.

Function declaration The function declaration is a value that can be used in the method `callFunctionImpl` 3.4.6 to call a function of the same kind is the currently analyzed one. For built-in functions the term constructor given by Yices2 is used. However uninterpreted functions are a special case, as for them the term id replaces the term constructor for this purpose (see 4.4.2. As both term constructors and term ids use positive integers a collision between them was found during testing. To mitigate this issue the term constructor values are negated when used as a function declaration, such that they use negative integers and UFs/ term ids use positive integers.

Arguments For most functions obtaining the arguments is handled by the `getArgs` method. It uses `yices_term_num_children` to determine the number of arguments of the given function and then runs a loop from 0 to less than the argument count, which simply retrieves each child term using `yices_term_child` and adds it to a list. Just like for getting the FunctionDeclarationKind some special cases exist for getting the arguments, which are detailed below.

4.4.2 UFs

During visiting function applications are treated as uninterpreted functions. For function applications in Yices2 the first child term is the function the arguments are applied to and the remaining child terms are the arguments. Thus the list of arguments is retrieved like normal, but then the first argument is set as the function declaration and dropped from the list. The remaining list is used as the arguments to the uninterpreted function.

4.5 Transformed terms

Yices2 performs several simplifications after a term is created to speed up and simplify solving. Some of these simplifications cause the terms to appear in an alternate form during visiting. Thus additional measures need to be taken to detect their kind and extract the arguments in a sensible way.

4.5.1 And

When a term in the form $AND(X, Y, ...)$ is created, Yices2 transforms it into the alternative form $NOT(OR(NOT(X), NOT(Y), NOT(...)))$, thus when visiting such a term would be classified as a NOT term based on the outer constructor. To enable JavaSMT to distinguish between a true NOT term and a transformed AND term the method `isNestedConjunction` is used. This method labels the term as an AND term if the first child of an outer NOT is an OR. If a term is recognized as an AND, its function declaration is set to a special value as Yices2 does not have a term constructor value for AND. Additionally the arguments are extracted using `nestedConjunctionArgs`, which takes the arguments of the inner OR and negates each one such that the retrieved arguments can be used normally.

4.5.2 Bitvector functions

Some bitvector operations such as sign- or zero-extension are modeled as an array of boolean terms and/or bit extraction operations, which extract a certain bit from a bitvector. For example the extension of a bitvector b of length m with n sign bits would result in an array of n extraction operations on the highest bit of b followed by m extraction operations on each bit of b. As it would be highly involved and potentially unreliable to try to match these back to the original operation type, these arrays are simply considered a concatenation of 1-bit bitvectors. Since the arguments of the bit extraction operation consist of a bitvector term and an integer index of the bit to extract, the index is stored as an integer term, which is unpacked into an integer when a bit extract function is called.

4.6 Sums and products

Like mentioned in 4.1.1 sums and products of both arithmetic and bitvector have their own unique argument retrieval methods due to their polynomial nature. These methods also require some work on the JNI side as they all return a term, which is easily returnable as an integer, and a coefficient or an exponent, which also need to be returned to Java.

4.6.1 Arithmetic sums

The components of arithmetic sums can only be accessed through `yices_sum_component`. If successful the method fills a given `mpq_t` variable with the coefficient and a given `term_t` variable with the term. To be able to return both values back to Java they are converted to strings and returned in a string array. On JavaSMT's side the method `getSumArgs` is responsible for collecting all components of a sum. It iterates over the sum's child terms and adds either the multiplication of the coefficient and the term or if the term is an error term just the term representation of the coefficient to the list of arguments.

4.6.2 Bitvector sums

The arguments of a bitvector sum can only be retrieved using `yices_bvsum_component`. If successful an integer array equivalent to the length of the bitvector is filled for the coefficient and a `term_t` for the accompanying term. To facilitate the return of both values from JNI the length of the integer array given to the method is deliberately one integer longer than needed and the term is stored in the last free field of this array. On JavaSMT's side the method `getBvSumArgs` loops over the child terms of the bitvector sum and retrieves its arguments. Equivalent to the handling of arithmetic sums, if an error term is encountered only the coefficient is stored as a bitvector constant, otherwise a multiplication of coefficient and term is stored.

4.6.3 Products

Both arithmetic and bitvector products are handled by `yices_product_component`. If successful both the term and the exponent are passed back to Java in an integer array. On JavaSMT's side the method `getMultiplyArgs` iterates over the children of the product

term and stores them in a list either as a boolean or an arithmetic power term based on a boolean value, that the method `visitFunctionApplication` determines form the type of the product term.

4.7 Dumping/Parsing

Dumping Dumping a formula from Yices2 in SMT-LIB2 is a relatively simple process due the previously described visiting capabilities of JavaSMT. The method `dumpFormula` in the `Yices2FormulaManager` first retrieves a map of the variables and UFs contained in the to be dumped formula with the help of the `extractVariablesAndUFs`. This function is part of JavaSMT and uses the visiting process to find all variables and UFs in a formula. Each entry in the map contains the the name as a key and the corresponding term as a value. For each entry the type of the term is determined using `yices_type_of_term` (for application terms the first child is used for this as it represents the UF). If this type has no children as tested via `yices_type_num_children` this type is used directly as return type, otherwise the child types obtained with `yices_type_children` are used for declaring. After this the function declaration is built, beginning with $(declare - fun$, followed by the name, then if needed the input type(s) and finally the return type. The strings representing the types is obtained by using `yices_type_to_string` and uppercasing the first letter of the returned string. Once all declarations are processed, ($assert$ followed by the string from `yices_term_to_string` and the final closing bracket are appended to finish the declaration.

Parsing Parsing SMM-LIB2 into Yices2 could not be implemented as Yices2 itself supports parsing SMT-LIB2, but the methods offered by the API expect the formulas in Yices2's own input language.

5 Solving and stack manipulation

This chapter will go over creating a context for solving, adding and removing formulas to solve from the stack of this context and finally solving the pushed formulas.

5.1 Creating a solver environment

A new context for solving is created by calling the `newProverEnvironment0` method in `Yices2SolverContext` which creates a `Yices2TheoremProver` instance and returns it. In the constructor of `Yices2TheoremProver` a new configuration for Yices2 is created using `yices_new_config`, then, using `yices_set_config`, the configuration is set to use the DPLL(T) solving approach and use push-pop mode for adding or removing levels from the stack. Finally the new context is created using `yices_new_context` with the configuration as an argument and a first level is pushed to the internal stack described below.

5.2 Push and pop

This section will briefly describe why an internal stack is needed and how the push and pop operations interact with it.

5.2.1 Internal stack

The internal stack in JavaSMT is needed for two reasons. The first is that Yices2 detects obvious falsities while adding formulas to the stack. This can cause the context to become unsatisfiable (UNSAT) before solving and pushing a new level in this state causes an error. The second reason is that Yices2 will not generate unsat cores, if the context is solved without assumptions. Thus if unsat cores should be generated, the internal stack collects the formulas and uses them as an assumption for solving.In conjunction with the internal stack an integer value `stackSizeToUnsat` is used to keep track of the lowest level of the stack causing the context to become UNSAT. In the beginning its value is set to the maximum integer value.

5.2.2 Push

Before a push of Yices2's stack is done, the `push` method checks if the internal stacks level is less than or even to `stackSizeToUnsat`. It also checks if the context's status is something else than UNSAT using `yices_context_status`. If both conditions are met, the stack Yices2's stack can be pushed to the next level with `yices_push`. Should one of the conditions fail the current level of the internal stack is set as the new `stackSizeToUnsat`, provided its value is still the max integer value meaning it has not

been set before. Regardless of the previous actions the level of the internal stack is always increased, to keep track of the stack level.

5.2.3 Pop

Similar to the `push` method, the `pop` checks if the internal stack's level is less than or equal to `stackSizeToUnsat`. When this condition is met, both stacks are on the same level and a level of Yices2's stack can be removed using `yices_pop`. SInce this action will bring the context back into a non UNSAT state, if it was previously UNSAT, `stackSizeToUnsat` is reset to its initial value to indicate this. The internal stack is always popped regardless.

5.3 Adding constraints

Constraints are usually added to both Yices2, using `yices_assert_formula`, and the internal stack. Note that the state of Yices2's context does not matter here, as it will simply do nothing if a constraint is added to an UNSAT context. But due to the problems regarding unsat cores described in 5.2.1, if the flag to generate unsat cores is set, the constraint is not added to Yices2's context and only kept in the internal stack for use during solving.

5.4 Solving and UnsatCores

This section describes how the context can be solved and how to obtain the unsat core if the context is unsatisifiable.

5.4.1 Solving

JavaSMT provides two methods `isUnsat` and `isUnsatWithAssumptions` to check if a context is unsatisfiable. These methods call `yices_check_context` respectively `yices_check_context_with_assumptions` via an intermediary method. This method converts Yices2's numerical status code for UNSAT/SAT to a boolean value. Should the context be in another state an exception is thrown. Note that due the inability to generate an unsat core after solving with `yices_check_context`, `isUnsat` will use assumption solving with all constraints collected on the internal stack if an unsat core is needed.

5.4.2 UnsatCores

Yices2 is able to generate unsat cores from an unsatisfiable context after it was solved using `yices_check_context_with_assumptions`. The returned unsat core then contains the terms that cause the context to be unsatisfiable. The method `yices_get_unsat_core`, which is responsible for this, requires a struct `term_vector_t` to output the unsat core. This struct consists of two unsigned integers, indicating the capacity and the actual size, and an array of `term_t`. Since it needs to be initialized before use, a macro `TERM_VECTOR_ARG` that creates a `term_vector_t` and uses `yices_init_term_vector` on it was needed. After the method has filled the struct with the unsat core another macro

`TERM_VECTOR_ARG_RETURN` unpacks the contained `term_t` array into an integer array for returning to Java and deletes the `term_vector_t` using `yices_delete_term_vector`.

6 Model exploration

A model can be created from a satisfiable context and holds information on the terms in the context and the values that their variables were assigned during solving. This chapter details the structure of the model in Yices2 and how JavaSMT traverses and evaluates it.

6.1 Model in Yices2

6.1.1 Structure

Yices2 stores the model as a directed acyclic graph (DAG). In this DAG leaf nodes represent atomic values, while non-leaf nodes can either represent a tuple, a function or a mapping, that serves as an auxillary node for describing functions. For terms of atomic type the value can be either obtained directly through provided functions or exploring the DAG, while evaluating the value of a tuple or function requires exploration. In the DAG a function is represented by a node for the function, a node for the function's default value and a set of mapping nodes. Each of these mapping nodes represents a tuple of arguments of the function and the returned value for these arguments.

6.1.2 Yval_t and yval_vector_t

Programmatically each node of the DAG is represented as a `yval_t`. It is a struct containing a `node_id`, identifying the node, and a `node_tag`, representing the type of the node. Since Java can not handle structs directly a pair of macros `YVAL_RETURN` and `YVAL_ARG` were created, where `YVAL_RETURN` packs the `node_id` and the `node_tag` into an int array for Java and `YVAL_ARG` repacks two int values from Java into an `yval_t` argument. The `yval_vector_t` struct has the same structure as `term_vector_t` and also needs to be initialized, which is done by the `YVAL_VECTOR_ARG` macro with the use of `yices_init_yval_vector`. Since it is only used for the `yices_val_expand_function` method, its return is done inside the JNI wrapper of said function by unpacking the `yval_t` values from the array, unpacking these like above and concatenating them into an integer array for return.

6.2 Creating the model

The model of a satisfiable context can be created by calling the `getModel` method of a `Yices2TheoremProver`. This method creates and returns an instance of `Yices2Model` with one of the constructors being a pointer to the model. This pointer is obtained by calling `yices_get_model` and passing the returned pointer back to Java as a `long` value. This method also takes a flag that tells it to either disregard or keep substituted variables

in the model. Per the developer's recommendation this flag is set to keep substitutions in JavaSMT.

6.3 Evaluating a term

A single term can be evaluated in the model by using `evalImpl` in the `Yices2Model`. This method calls `yices_value_as_term` which evaluates the term's value and returns it as a term of an appropriate type. This term can then be converted into a value by using `convertValue` in the `Yices2FormulaCreator`. Since the returned term's type does not necessarily match the evaluated term's type, `convertValue` uses the latter's type for the conversion decision to ensure consistent typing.

6.4 Traversing the model

Additionally to evaluating a single formula with `evalImpl` the method `toList` is available for gathering a `ValueAssignment` for each defined term in the model. Each of these assignments consist of the evaluated term, a term representing the value, a term representing the equality between evaluated term and value term, the term's name, its value as a Java object and Java objects for the value of arguments if applicable. To facilitate this first all defined terms in the model are collected using `yices_model_collect_defined_terms` which returns a `term_vector_t` (see 5.4.2) as an integer array, which contains all terms that have a value in the model. Then for each term its `yval_t` is obtained using `yices_get_value`. Depending on the tag of the `yval_t` one of three actions is selected. For tags, that indicate an unused constant type or are otherwise unexpected, an exception is thrown. For terms with `YVAL_FUNCTION` tag further evaluation is done by the `getFunctionAssignment` method 6.4.2. Finally terms with an expected constant tag are evaluated using `getSimpleAssignment` 6.4.1.

6.4.1 Evaluating constants

The method `getSimpleAssignment` is responsible for obtaining the `ValueAssignment` for terms representing constants during traversal of the model. It retrieves the value of the term and the term representing this value the same way the `evalImpl` method does. The only additional operations are getting the term's name using `yices_get_term_name` and creating the equality between evaluated term and value term using `yices_eq`.

6.4.2 Evaluating functions

Due to its structure in the model DAG (see 6.1.1) a function can not be evaluated directly and instead must be evaluated through expanding and exploring the function node and its child nodes. Therefore the `getFunctionAssignment` analyzes each function in a three step process.

Function expansion In the first step the `yval_t` corresponding to the function is passed to the `yices_yval_expand_function` method. This method returns the child nodes of

the function node, which contain one node describing the function's default value and one or more nodes describing the mapping of argument values to a return value. The default value is currently not used in JavaSMT and thus is ignored.

Mapping expansion In the second step the mapping nodes are expanded into their child nodes using `yices_val_expand_mapping`. These child nodes contain argument nodes, equal in count to the function's arity and returned as an array of `yval_t`, and one node for the return value, returned as a separate `yval_t`. For easier handling in Java all returned `yval_t` are unpacked into a single integer array similar to to the handling of the `yval_vector_t` struct6.1.2.

Yval conversion In the final step the helper method `valueFromYval` is used to retrieve the value of each argument and return value node as an appropriate Java object. The Java object is then used with a second helper function `valueAsTerm` to create a value term as no direct way to convert the value of an `yval_t` to a term exists in Yices2. The `valueFromYval` method uses the `node_tag` of the `yval_t` to identify which `yices_val_get_TYPE` to use. For boolean and bitvector constant nodes their associated methods `yices_val_get_bool` and `yices_val_get_bv` are used. For arithmetic constant nodes `yices_val_get_mpq` is used as it returns a `mpq_t` (see 4.1.3, which is not subject to possible overflows and can represent both integer and rational values. To decide whether the value should be stored as a `BigInteger` or a `Rational` the type of the argument in the evaluated function is used, as the `node_tag` is the same for both integer and rational constants. Once the value is stored as a Java object `valueAsTerm` uses it to generate a matching term. To ensue an appropriate term is created the method uses the type of the argument, which is retrieved from the evaluated function's type, as a decider. For boolean type values either `yices_true` or `yices_false` is used, for bitvector type values a similar approach to `makeBitvectorImpl` 3.4.4 is used and for arithmetic type values `yices_parse_rational` is used, if the value string contains a slash, otherwise `yices_parse_float` is used.

7 Evaluation

JavaSMT is used as backend for several analyses in CPAchecker. This allows a real-world evaluation of Yices2 on a large set of tasks. In this chapter we will evaluate the performance of Yices2 in comparison to the existing solvers in JavaSMT. First we will give a short overview of CPAchecker and sv-benchmark, which were used for the testing. Then we will give details about the used configuration, before presenting the results.

7.1 Software

7.1.1 CPAchecker

CPAchecker[1], the Configurable Software-Verification Platform, is a framework developed at SoSy-Lab written in Java for verifying programs. It offers a highly customizable verification process and has won several awards for many different software verification tasks. See [BK11] for details on its architecture. Additionally it offers benchmarking capabilities in conjunction with sv-benchmarks7.1.2 and BenchExec[2].CPAchecker makes use of JavaSMT for the analysis steps that require SMT solving. One of the analysis techniques, that use SMT, is Bounded Model Checking (BMC) , which was selected for the benchmark because it can be used with all solvers that are currently available in JavaSMT. BMC is used for more effectively checking programs with a large set of execution paths. Consider a program similar to listing 7.1 where an error happens early in the loop. Analyzing every possible path using SMT would need a large amount of SMT statements describing every loop from 0 to the analyzed language's maximum integer value. Even for a such a simple program, this could cause the used solver to not be capable of solving the formula created by this approach due to time and/or resource constraints. BMC prevents this scenario by limiting the number of checked paths to a certain number. For this evaluation we allowed checking a loop once, respectively 10 times (see 7.2). Only checking the first 10 iterations of the sample program's loop would still lead to the discovery of the error, while significantly reducing the required resources. For a more in-depth explanation of BMC and the other analysis techniques employed by CPAchecker one can read this paper [BDW18].

7.1.2 SV-benchmarks

SV-benchmarks[3] is a large collection of verification tasks aiming to provide a common basis to evaluate software evaluation techniques. The collection is divided into sets, allowing to test against a subset of the tasks rather than every task. This is helpful if

[1] `https://cpachecker.sosy-lab.org`, 12.2019

[2] A framework for reliable benchmarking developed at SoSy-Lab. For details on the technical background see [BLW19]

[3] `https://github.com/sosy-lab/sv-benchmarks`, 12.2019

one wants to test recognition of specific error types or not all tasks are supported by the tested program.

Listing 7.1: Sample program in pseudo code

```
for x from 0 to Max_Integer{
        if (x = 7){
                ERROR
        }
// do nothing
}
```

7.2 Configuration

The following configurations were used during the evaluation:

Execution platform The benchmark was executed on a cluster of Intel Xeon E3-1230 v5 processors running Ubuntu 18.04 and clocked at 3.8 GHz with Turbo Boost disabled. Each task of the set was ran on 2 Cores of the Cluster with a memory limit of 15000 MB and a timeout of 900 seconds.

Task set The task set that was used for the evaluation is the BMC set of SV-benchmarks. This set contains the ReachSafety-ControlFlow, ReachSafety-ECA, SoftwareSystems-DeviceDriversLinux64-ReachSafety, ReachSafety-Heap, ReachSafety-Loops, ReachSafety-ProductLines and ReachSafety-Sequentialized subsets. Each subset was run with the unreach-call specification. All tasks were run twice, the first time unrolling each loop once (k1) and the second time unrolling each loop 10 times (k10).

Solvers The solvers that were compared for this evaluation are Boolector, CVC4, MathSAT5, Yices2 and Z3. Due to an incompatibility regarding Yices2's implementation of arrays, the CPAchecker option useArraysForHeap was disabled for all solvers as to not punish Yices2 with the associated performance decrease. Additionally for Yices2 the option encodeFloatAs=RATIONAL was set because Yices2 does not support the floating point theory. Lastly for Boolector the options encodeFloatAs=INTEGER, encodeIntegerAs=BITVECTOR, createFormulaEncodingEagerly=false and handlePointerAliasing=false were set due to reasons explained in [Bai19].

7.3 Results

This section gives a short overview of the evaluation results. Note that due to BMC not necessarily reaching a definite answer within the loop limit, not only correct results, but also incorrect and unknown ones were taken into account for the plots. This means that the plots contain values for all tasks, except for those that ran into the memory/time limit or another error occurred during execution. As can be seen in figure 7.1 for BMC with 1 loop unfold Yices2's cpu times lie between Boolector's and MathSAT5's and tend

to stay closer to Boolector's with a clear separation only in the mid range of execution times. For memory figure 7.2 shows Yices2's memory consumption in this scenario is near identical to Boolector's for most tasks with a only slightly higher usage on memory intensive tasks. Moving on to BMC with 10 loop unrolls figure 7.3 once again shows Yices2 very close in execution times to Boolector and MathSAT5 but with MathSAT5's executions times being considerably higher for long tasks. The memory consumption for these tasks, as plotted in figure 7.4, shows Yices2 taking a small lead on Boolector for tasks in the lower memory usage range, with the mid-range being nearly identical and Boolector only getting a clear lead the highest memory range. Also even though for the most part competitive to Yices2 and Boolector in execution times, MathSAT5's memory usage is considerably higher for these tasks.

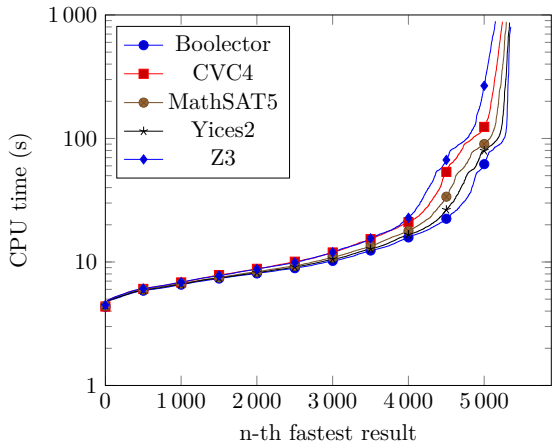

Figure 7.1: Results for k1 sorted by lowest CPU time.

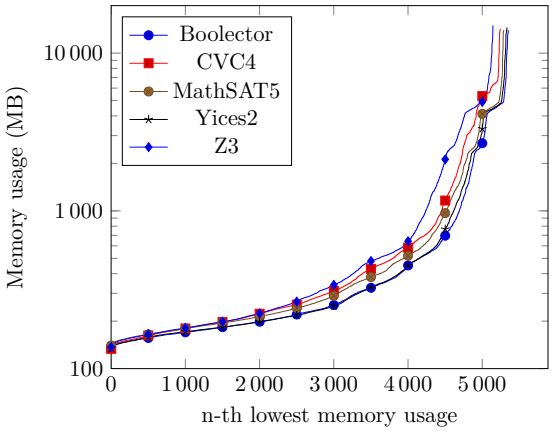

Figure 7.2: Results for k1 sorted by lowest memory usage.

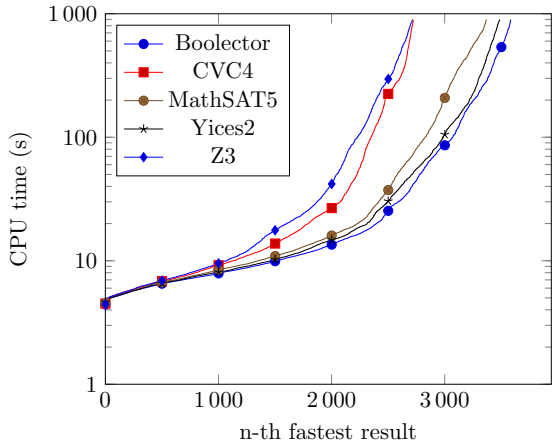

Figure 7.3: Results for k10 sorted by lowest CPU time.

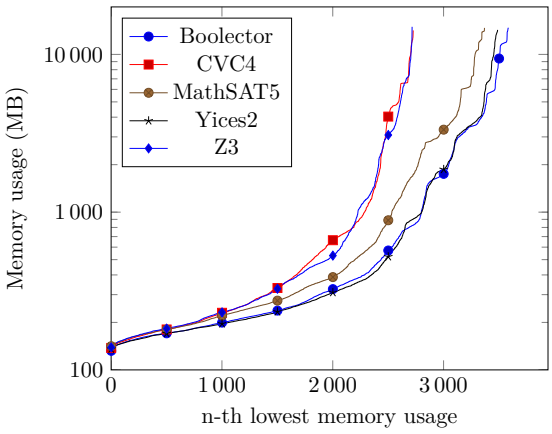

Figure 7.4: Results for k10 sorted by lowest memory usage.

8 Conclusion

This work has shown that extending JavaSMT with the SMT solver Yices2 was mostly straightforward thanks to Yices2's clearly defined, documented and comparatively abstract API. Paired with a good feature set and extensive theory support this meant that Yices2 could be very well integrated into JavaSMT. Only array and quantified formula support could not be implemented at this time. For the array theory differences in handling between Yices2 and JavaSMT were problematic, while supporting quantified formulas was not possible due to Yices2 requiring a special symbol for quantified terms. For the latter a solution on JavaSMT's side is planned as another solver has similar problems. Another small problem is that Yices2's API does not allow parsing SMT-LIB2 at this point, while Yices2 itself does. Despite these little annoyances Yices2 showed a very solid performance in the evaluation, putting it very close to Boolector, the currently fastest solver in JavaSMT, while offering more features. In the future we hope to be able to fully support all theories offered by Yices2. We would also like to see Yices2 adding support for interpolation and optimization while maintaining or topping its current performance.

List of Figures

List of Tables

Listings

Bibliography

[Bai19] Daniel Baier. Integration des SMT-Solvers Boolector in das Framework JavaSMT und Evaluation mit CPAchecker. Bachelor's Thesis, LMU Munich, Software Systems Lab, 2019.

[BDW18] Dirk Beyer, Matthias Dangl, and Philipp Wendler. A unifying view on SMT-based software verification. *Journal of Automated Reasoning*, 60(3):299–335, 2018.

[BK11] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, July 14-20)*, LNCS 6806, pages 184–190. Springer-Verlag, Heidelberg, 2011.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer (STTT)*, 21(1):1–29, 2019.

[Dut14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[JBd13] D. Jovanovic, C. Barrett, and L. de Moura. The design and implementation of the model constructing satisfiability calculus. In *2013 Formal Methods in Computer-Aided Design*, pages 173–180, Oct 2013.

[JdM12] Dejan Jovanović and Leonardo de Moura. Solving non-linear arithmetic. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, IJCAR'12, pages 339–354, Berlin, Heidelberg, 2012. Springer-Verlag.

[KFB16] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. JavaSMT: A unified interface for SMT solvers in Java. In *Proc. VSTTE*, LNCS 9971, pages 139–148. Springer, 2016.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.