UDC 004.05
doi:10.31799/1684-8853-2022-1-30-43

Articles

Kex: A platform for analysis of JVM programs

A. M. Abdullin[a,b], *Post-Graduate Student, Assistant Professor, orcid.org/0000-0002-9669-2587*
V. M. Itsykson[a,b], *PhD, Tech., Associate Professor, orcid.org/0000-0003-0276-4517, vlad@icc.spbstu.ru*
[a]*Peter the Great St.Petersburg Polytechnic University, 19, Politechnicheskaya St., 195251, Saint-Petersburg, Russian Federation*
[b]*JetBrains Co. Ltd., 70, Building 1, Primorskiy Av., 197374, Saint-Petersburg, Russian Federation*

Introduction: *Over the last years program analysis methods were widely used for software quality assurance. Different types of program analysis require various levels of program representation, analysis methods, etc. Platforms that provide utilities to implement different types of analysis on their basis become very important because they allow one to simplify the process of development.* ***Purpose:*** *Development of a platform for analysis of JVM programs.* ***Results:*** *In this paper we present Kex, a platform for building program analysis tools for JVM bytecode. Kex provides three abstraction levels. First is Kfg, which is an SSA-based control flow graph representation for bytecode-level analysis and transformation. Second is a symbolic program representation called Predicate State, which consists of first order logic predicates that represent instructions of the original program, constraints, etc. The final level is SMT integration layer for constraint solving. It currently provides an interface for interacting with three SMT solvers.* ***Practical relevance:*** *We have evaluated our platform by considering two prototypes. First prototype is an automatic test generation tool that has participated in SBST 2021 tool competition. Second prototype is a tool for detection of automatic library integration errors. Both prototypes have proved that Kex can be used to implement competitive and powerful program analysis tools.*

Keywords *— program analysis, analysis platform, test generation, symbolic execution.*

For citation: Abdullin A. M., Itsykson V. M. Kex: A platform for analysis of JVM programs. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2022, no. 1, pp. 30–43. doi:10.31799/1684-8853-2022-1-30-43

Introduction

Modern world is highly dependent on software: it controls almost every part of human life. Thus, errors in the modern-day software may lead to fatal consequences. To address that problem IT-industry adopts software quality assurance techniques.

Software quality assurance techniques could be divided into two main groups: manual techniques and automatic ones. Manual quality assurance techniques include software testing, code review, audits, etc. Those techniques have proven their effectiveness over time and are currently used in everyday development processes. However, they all share one significant weakness: manual quality assurance is very hard and time-consuming work [1].

Automatic software quality assurance techniques are trying to overcome that weakness. Alike manual techniques, automatic methods vary on the level of complexity and depth of the analysis: from simple and fast code smell detection [2] to resource intensive verification [3]. Over the last years, automatic quality assurance methods were widely used for automatic testing, automatic test generation, bug detection, etc. Most of the automatic quality assurance techniques are based on methods of static (e. g. symbolic execution [4, 5], bounded model checking [6], etc.) and dynamic (e. g. fuzzing [7], dynamic symbolic execution [8], concolic testing [9], etc.) program analysis. Many IT-companies are currently using program analysis methods as part of their everyday development process [10, 11].

For most widely used programming languages like Java, JavaScript, C/C++, etc. there already exists a large variety of tools for both static and dynamic analysis [12–15].

In this paper we present Kex (https://github.com/vorpal-research/kex), a platform for building various kinds of program analysis tools for Java Virtual Machine (JVM) based languages. Kex consists of three main components: Kfg library for JVM bytecode analysis and transformations, intermediate representation called Predicate State (PS) for symbolic program representation and constraint solving module based on satisfiability modulo theories (SMT) solvers. These modules allow one to build different types of analyses on top of Kex, both dynamic (based on bytecode instrumentation and execution) and static (based on symbolic execution and constraint solving). To showcase capabilities of Kex we have considered two prototypes of analysis tools: one for automatic test generation for Java language and the other for automatic integration errors detection. Evaluation results show that Kex can successfully be used to analyze JVM based languages on a variety of levels of depth, complexity and precision.

Related work

As we have mentioned earlier, there already exists a number of tools and frameworks for analysis of JVM bytecode. These frameworks differentiate by analyses they support which the underlying model of program representation inherently limits. Let us consider some of the most significant examples.

ASM [16] is an all-purpose Java bytecode analysis and manipulation framework that was introduced in 2002. The project is still under active development and the latest release of version 9.2 was in the summer of 2021. ASM provides a set of bytecode analyses and transformations and can be used both to modify existing classes and to dynamically generate new classes. ASM is focused on working with low-level representation of compiled classes and therefore is mainly used by many projects (https://asm.ow2.io/). The Kfg library also uses ASM for working with JVM bytecode.

Soot [17] is a Java bytecode optimization framework that was first introduced in 1999. Soot provides four intermediate representations for bytecode: *baf* — simplified stack based bytecode, *jimple* — 3-address code representation of bytecode, *shimple* — static single assignment (SSA) variation [18] of *jimple* and *grimp* — an aggregated version of *jimple* suitable for decompilation and code inspection. Each representation is more suited for its own kind of analyses and optimizations including points-to analysis, call-graph construction, data-flow analysis, etc. Both ASM and Soot are libraries which are mainly focused on bytecode-level optimizations and do not provide tools for more in-depth analysis.

Spoon [19] is a library for Java source code analysis and transformation. Spoon meta model consists of three parts:

— structural part contains the declarations of program elements (classes, interfaces, methods, etc.);

— code part contains executable Java code in the form of AST;

— reference part models references to program elements.

Limitations of Spoon come from its meta model. First, as it works at the source code level, it is only limited to work with one language, whereas bytecode level frameworks can work with any JVM based languages. Second, Spoon provides only one program model — AST — which is not always well-suited for various types of analyses.

JBSE [5] is a symbolic JVM for automated program analysis, verification and test generation. JBSE uses javassist [20] library to interact with the target classes, provides its own API for working with source code at a bytecode level and provides an implementation of *symbolic state* that represents the state of execution of a program. Symbolic state can be transformed into an SMT formula in smtlib2 format [21], which is then sent to an SMT solver to reason about reachability of that state. JBSE also provides utilities for automatically generating test cases that reach a given state using reflection [22]. Authors have also developed two tools on top of JBSE. SUSHI [23] is an automatic test case generator for Java programs that uses JBSE for symbolic execution and EvoSuite [24] for test generation, which allows it to generate tests that do not use reflection. TARDIS [25] is an extension of SUSHI that uses JBSE to perform concolic testing. Those tools confirm the applicability of JBSE; however, it still has some limitations. First, JBSE does not provide any utilities to work with more structured program representation rather than stack-based bytecode; hence, the symbolic state is also very close to low-level bytecode representation. Second, the internal structure of JBSE is more suited for easy usage of symbolic execution results, but it is hard to extend it.

JDQL [26] is a framework for Java static analysis that uses Datalog [27] query language for automatically detecting bad patterns in the program source code. JDQL works both with Java source code and JVM bytecode, provides utilities to perform flow analysis and intra-procedural analysis, and it is easy to extend with new error detectors. However, JDQL is limited in a sense that it is only suitable for a lightweight pattern recognition based static analysis and does not allow performing more precise and complex types of analyses.

As one can see, there already exist many frameworks for analysis of JVM programs (both on bytecode and source code level). Existing frameworks are well suited for building analysis tools at one specific program representation level. For example, symbolic execution engines do not provide access to underlying source code and bytecode analyses and manipulation frameworks do not provide utilities for more in-depth analysis. This limitation of the existing frameworks has inspired us to develop a new platform that will:

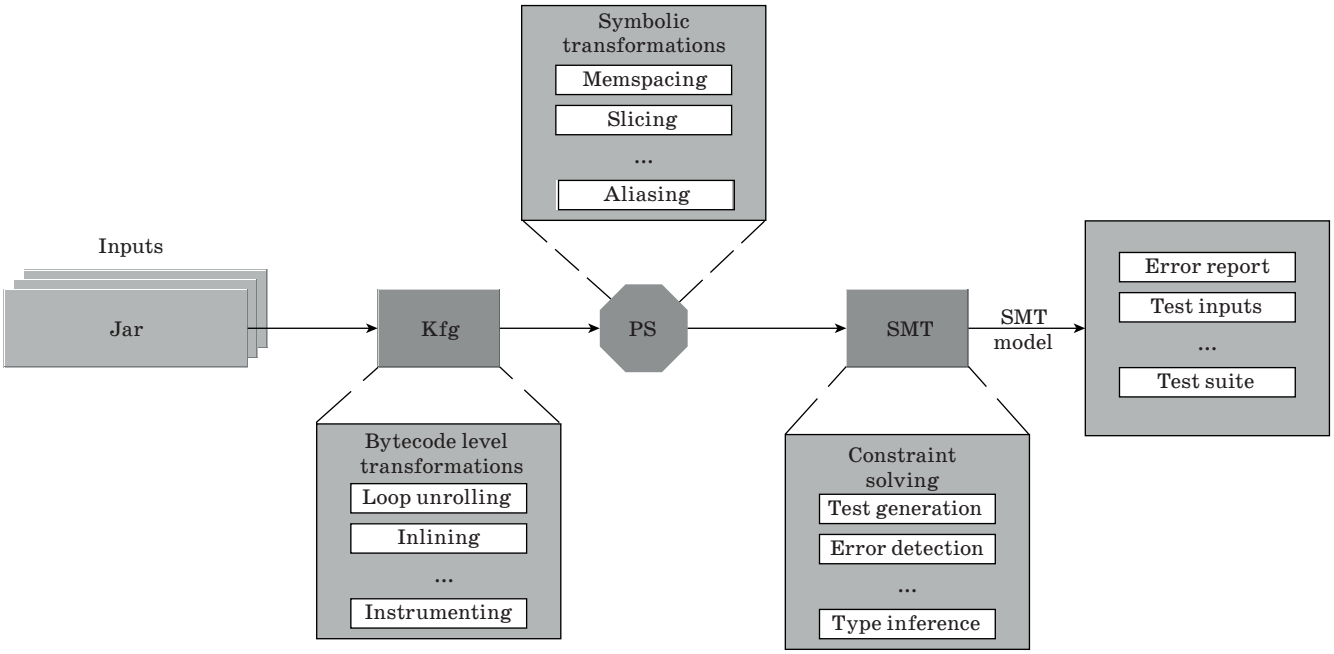
— provide utilities for various types of analyses (both static and dynamic);

— allow multi-level analysis;

— provide application programming interface (API) for constraint solving.

Kex in detail

Kex is a platform for analysis of JVM based languages. It takes a set of compiled classes and provides utilities to perform transformation and analyses on multiple levels of program representation: control flow graphs, PS and SMT formulae. Kex as-



■ ***Fig. 1.*** A high level overview of Kex

sumes a closed-world model during analysis, i. e., it has full access to all possible types, functions, etc. A high-level overview of Kex architecture can be found in Fig. 1. In this section we give a detailed description of every module of Kex.

Kfg: control flow graph for JVM bytecode

Program analysis requires having an informative and easy-to-use program model. JVM combines in itself features of stack machine and register machine: each execution frame has an operand stack and an array of local variables. The operand stack is used to provide operands for bytecode instructions and to receive results of their computations. The local variables array serves the same purpose as processor registers: to store quickly accessible data and to pass arguments for methods. While that model of computation is very effective for JVM purposes, it is not fitted for purposes of program analysis.

Kfg (https://github.com/vorpal-research/kfg) is a library for JVM bytecode analysis and transformation. Kfg builds control flow graphs (CFG) in SSA form [18] for each method of the target program. Kfg also provides utilities to create and modify classes and fields of a project. Kfg is built on top of the latest ASM version 9.2 — an all-purpose Java bytecode manipulation and analysis framework — and provides an API to directly access ASM representation for more low-level features. Currently Kfg supports JVM bytecode version 62 and lower (which corresponds to JVM version 18). Let us now consider the internal structure of Kfg in more detail.

Class management

The key concept of the Kfg is the *ClassManager*, which stores all the information about available classes and allows one to access those classes. Classes are the essential part of JVM and, therefore, Kfg: every project consists of a set of classes. Each class contains the following list of information: modifiers, superclass, interfaces, methods, etc. Each class also stores an instance of *ClassNode* — an ASM representation of class — allowing it to make low-level transformations.

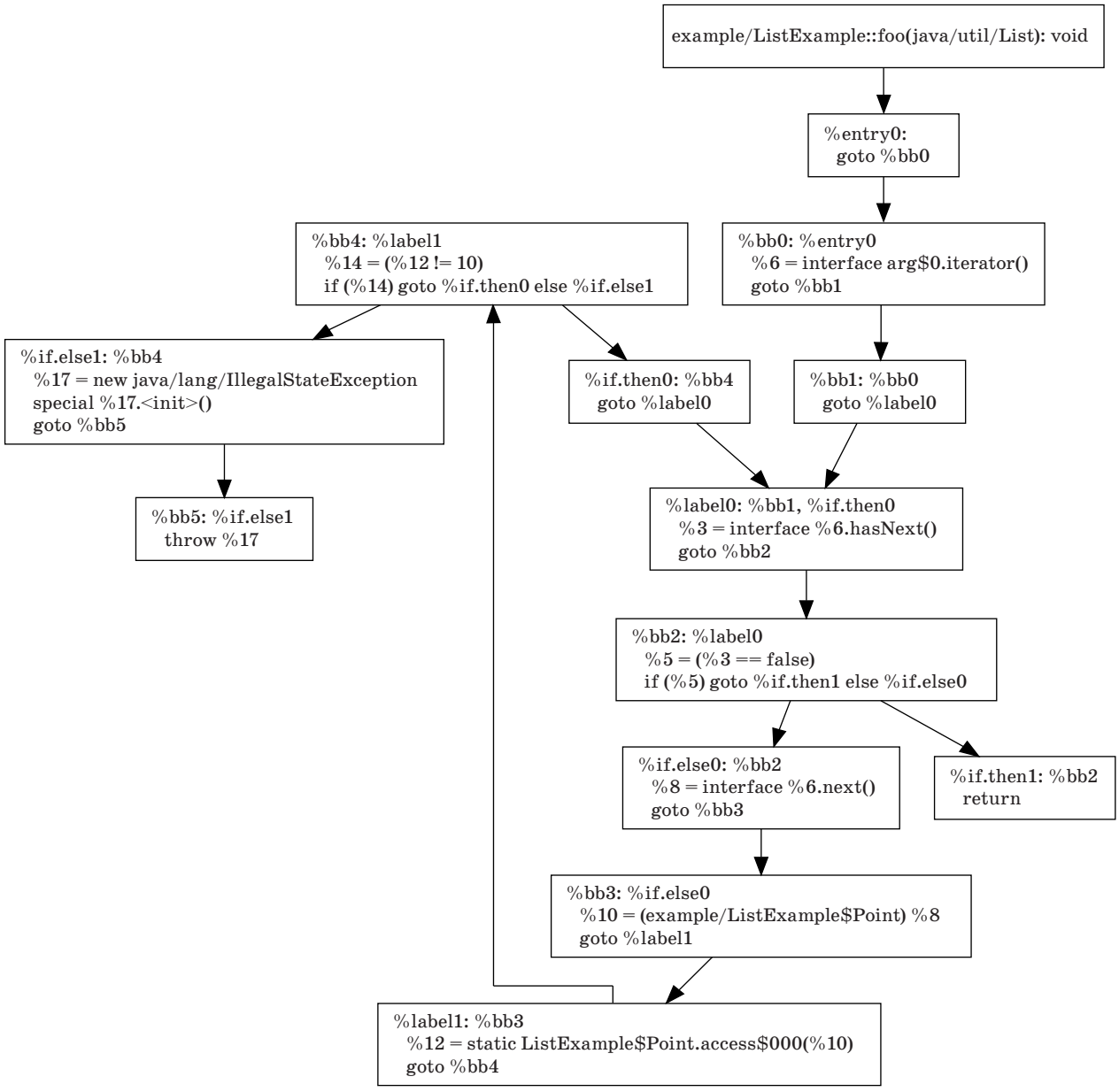
Kfg preserves connection between each class and its actual bytecode stored in the file system, thus allowing creating, modifying and updating the bytecode both on singular class level and on project level (e. g. modify jar files or directories with compiled sources). That connection is implemented through *Containers*.

To build a precise model of a project one needs to have access to all the libraries that it depends on. However, building that model for a large-scale project with many dependencies can be very resource intensive and even redundant in some cases. To have an ability to analyze projects without access to full class path, Kfg introduces an idea of *OuterClass*: a class which bytecode Kfg cannot access. When working with instances of *ConcreteClass* (i.e. class whose bytecode that Kfg has access to), Kfg checks validity and correctness of all operations. Downside of the *OuterClass* idea is that Kfg cannot guarantee correctness of the resulting bytecode and relies on the user to ensure it.

Type system of Kfg directly corresponds to JVM type system described in the JVM specification [28] and consists of integrals (boolean, byte, char, short, int and long), floating points (float and double), references (classes, arrays and null) and void type. Let us now consider the details of the control flow graph.

Methods and CFG

For each method of each *ConcreteClass* available to *ClassManager* Kfg builds a CFG in SSA form. An example of CFG built by Kfg is shown in Fig. 2. A control flow graph consists of basic blocks — a sequence of instructions that are executed one after other without any branching. Each basic block ends with terminating instruction; it can be simple jump, branching, return or throw. As one can see, basic blocks *%entry0, %bb0, %bb1* could have been united to a single basic block as they do not have any branching and seemingly are always directly following each other. However, CFG for JVM bytecode is more complex, as the instructions (and, therefore, basic blocks) of JVM bytecode may have *hidden* connections to exception catch blocks. JVM specification defines all the instructions that can potentially throw exceptions. Each basic block of Kfg ends either with branching instruction or with an exception throwing instruction. Figure 2 shows the examples of both of this cases: blocks *%bb2* and *%bb4* end with branch instruction (if — else); blocks



■ ***Fig. 2.*** An example of CFG built by Kfg

%bb0 and *%label0*, for example, end with simple jump instruction but the last "meaningful" instruction they contain is the call instruction, which potentially can throw an exception. To express those hidden connections, basic blocks also store a set of *handler* blocks in addition to a set of predecessors, successors and instructions. Basic blocks themselves are also divided into two categories:

— *BodyBlock* — default block that forms the CFG;

— *CatchBlock* — special block that handles thrown exceptions; that block does not have usual predecessors but a set of *thrower* blocks that it catches from; also, *CatchBlock* stores information about the catched exception type.

Values and instructions

Instructions of the Kfg operate on *Values*: representation of program variables. Values are divided into four categories: *this* reference, arguments, constants and instructions. Instruction set of Kfg is corresponding one-to-one to the instruction set of JVM bytecode with two exceptions:

— JSR instruction [28] is inlined before CFG building and therefore is not present in Kfg instructions set;

— Kfg adds a new *phi* instruction — a special instruction that represents φfunction of SSA form.

CFG analysis and modification

Along with the API to build a CFG model of a project and translate it back to JVM bytecode, Kfg provides an API to perform various analyses and transformations with the built model.

Kfg uses a visitor [29] pattern and provides a *NodeVisitor*, a *ClassVisitor* and a *MethodVisitor* allowing one to traverse all the classes, fields, methods and instructions of a project. In addition, Kfg also provides loop analysis for all of the methods of a project: information about each loop of a method is stored in a graph form. To simplify the work with loops Kfg also provides a *LoopVisitor*: an extension of *MethodVisitor* that allows to traverse all the loops of a method. *Pipelines* allow combining a set of visitors into a single instance that will apply each visitor to all the classes of *ClassManager* one after another.

At the instruction and value level Kfg implements the "user" pattern: each instruction, value and basic block contains a set of objects that it is used by. Any class that uses CFG elements should implement *BlockUser* or *ValueUser* interfaces. That idea was inspired by LLVM [30].

Kex currently uses Kfg for:

— loop canonicalization [31];

— loop unrolling;

— bytecode instrumentation on various levels;

— bytecode modification (e. g. all the *System.exit()* calls are replaced with a special *SystemExitCallException* to prevent JVM stopping during dynamic analysis);

— CFG modification, etc.

Predicate State representation

Predicate State is Kex's intermediate representation that is used to perform various types of analysis and that is designed to be easily converted into an SMT formula. This section describes details of PS implementation.

Basic PS structure

Predicate State is designed as a directed acyclic graph because SMT formulae cannot express loops. PS was originally introduced in Borealis bounded model checker [32]. Kex has adapted PS from Borealis and extended it to support Kfg instructions.

Predicate State is built from CFG and, therefore, CFG should be preprocessed in order to be convertible to PS. Preprocessing consists of two main steps:

— loop canonicalization;

— loop unrolling.

These two steps allow presenting a CFG in a form that is directly convertible to PS. Both of these operations are implemented as Kfg loop visitors. The format of PS in Backus — Naur form [33] is shown in listing 1 and an example of PS is shown in listing 2.

As one can see from listing 1, PS has three types:

— *BasicState* — PS represents a single basic block, basically just a list of predicates;

— *ChoiceState* — PS that represents branching, contains a list of branches (as PS);

— *ChainState* — PS that combines two states into a sequence, used to create full program representation from *BasicState* and *ChoiceState*.

Listing 1. PS format.

```
<PredicateState> ::= ChainState head:<PredicateState>
tail:<PredicateState>
    | ChoiceState choices:<ListOfPredicateStates>
    | BasicState data:<ListOfPredicates>

<ListOfPredicateStates> ::= <PredicateState>
<ListOfPredicateStates> | <empty>

<Predicate> ::= ArrayInitializerPredicate arrayRef:<Term>
value:<Term>
    | ArrayStorePredicate arrayRef:<Term> value:<Term>
    | CallPredicate lhv:<Term> call:<Term>
    | CatchPredicate throwable:<Term>
    | DefaultSwitchPredicate cond:<Term> cases:<ListOfTerms>
    | EnterMonitorPredicate monitor:<Term>
    | EqualityPredicate lhv:<Term> rhv:<Term>
    | ExitMonitorPredicate monitor:<Term>
    | FieldInitializerPredicate field:<Term> value:<Term>
    | FieldStorePredicate field:<Term> value:<Term>
    | GenerateArrayPredicate lhv:<Term> length:<Term>
generator:<Term>
    | InequalityPredicate lhv:<Term> rhv:<Term>
```

```
    | NewArrayPredicate lhv:<Term> dimensions:<ListOfTerms>
elementType:<Type>
    | NewPredicate lhv:<Term> type:<Type>

<Term> ::= ArgumentTerm index:Int type:<Type>
    | ArrayContainsTerm array:<Term> value:<Term>
    | ArrayIndexTerm array:<Term> index:<Term>
    | ArrayLengthTerm array:<Term>
    | ArrayLoadTerm arrayRef:<Term>
    | BinaryTerm op:<BinaryOp> lhv:<Term> rhv:<Term>
    | CallTerm owner:<Term> method:<Method>
arguments:<ListOfTerms>
    | CastTerm term:<Term>
    | CmpTerm op:<CmpOp> lhv:<Term> rhv:<Term>
    | ConstTerm
    | EqualsTerm lhv:<Term> rhv:<Term>
    | ExistsTerm start:<Term> end:<Term> body:<Term>
    | FieldTerm owner:<Term> fieldName:String
    | FieldLoadTerm field:<Term>
    | ForAllTerm start:<Term> end:<Term> body:<Term>
    | InstanceOfTerm term:<Term> type:<Type>
    | IteTerm cond:<Term> trueValue:<Term> falseValue:<Type>
    | LambdaTerm arguments:<ListOfTerms> body:<Term>
    | NegTerm term:<Term>
    | ReturnValueTerm method:<Method>
    | StaticClassRefTerm klass:<Type>
    | ValueTerm type:<Type> name:String

<ListOfTerms> ::= <Term> <ListOfTerms> | <empty>
```

Listing 2. PS example.

```
(
 @S kotlin/jvm/internal/Intrinsics.checkNotNullParameter(arg$0, 'a')
 @S %1 = arg$0.size()
 @S %3 = %1 != 2
 @P %3 = false
 @S %5 = arg$0.get(0)
 @S %7 = (%5 as example/ListExample$Point)
 @S %9 = %7.getX()
 @S %11 = %9 != 10
 @S %11 = false
 @S %13 = arg$0.get(1)
 @S %15 = (%13 as example/ListExample$Point)
 @S %17 = %15.getY()
 @S %19 = %17 != 11
 @S %19 = false
 @S %24 = new java/lang/IllegalStateException
 @S %23 = 'a'.toString()
 @S %24.<init>(%23)
 @S %26 = (%24 as java/lang/Thowable)
)
```

One may notice that current implementation of PS is limited because it does not handle try/catch blocks, i. e. exception handling is not supported. Potentially it can be implemented by adding *ChoiceState* at each predicate that leads to two branches: one to the next predicate in the program and one to a catch block that handles the exception. However, that will lead to an exponential growth of the state size. We consider adding exception control flow handling into PS as a part of our future work.

The design of PS is closer to SMT formulae than CFG and it introduces some of the concepts that are later passed on to an SMT solver. First, the PS model introduces a *memory* concept and explicitly separates expressions that change the memory from ones that do not: predicates and terms correspondingly. Thus, predicates are used to express actions that change the state and the memory of a program, e. g. *FieldStorePredicate* that writes value to some field. However, there are also predicates that allow us to express some additional constraints for a program. The type of predicate determines those properties:

— state — usual predicate that changes the state (and, therefore, the memory) of program;

— path — predicate that expresses the current path condition;

— assume — predicate that carries some additional information that Kex can assume is true;

— axiom — predicate that encodes some axioms that are always true (e. g. a class reference always being not null);

— require — predicate that encodes some properties that Kex should check for correctness.

The PS definition shows that most of the predicates directly correspond to Kfg instructions. However, there are some exceptions. For example, *FieldInitializerPredicate* that allows initializing the value of a field before actual program execution.

Terms mainly represent Kfg values and operations that do not change the memory state, e. g. arguments, constants, array index reads, etc. In JVM bytecode there are no ways to reference the memory address that holds the value of a field or an element of an array, one can only read the value stored in that location. However, during analysis one needs to differentiate between memory location and the value that it stores. For that reason, Kex adds two special pointer terms: *ArrayIndexTerm* and *FieldTerm*. To receive the value stored in a given location one needs to explicitly specify memory load action with *ArrayLoadTerm* and *FieldLoadTerm* correspondingly.

Type system of Kex extends the type system of Kfg by supporting special typing *ArrayIndexTerms* and *FieldTerms*. The type system consists of:

— integrals: bool, byte, char, short, int, long;

— reals: float and double;

— pointers — an equivalent of Kfg references:

  ○ object pointers;

  ○ array pointers;

  ○ references — types of array indexes and fields;

  ○ null;

— void.

PS modification and analysis

Analysis of a program suggests that one has an ability to traverse and modify the model, i. e. Predicate State. Kex provides a *Transformer* interface to traverse PS and a *RecollectingTransformer* interface to modify it. *Transformer* implements

CRTP pattern [34] and provides an API to dismantle each component of the PS and build it up again with the same or new structure. PS and its components are immutable and therefore if any transformer changes the state it returns a new copy of it.

Kex provides a set of transformer implementations:

— Stensgaard alias analysis [35];

— static backward slicing [36];

— constant propagation;

— inlining of various types: static fields initialization inlining, static method inlining, virtual method inlining (requires type resolving);

— reflection info inlining (e. g. Kotlin reflection provides a lot of useful type and nullity information);

— external information provider: e. g. annotation info inliner that adds method invocation info from JetBrains annotations (https://github.com/JetBrains/java-annotations), etc.

Symbolic execution using SMT solver

Kex uses SMT solvers for constraint solving and currently supports three solvers: Z3 [37], Boolector [38] and STP [39]. To simplify work with multiple solvers Kex uses automatically generated unified wrapper classes. Example of SMT wrapper API can be found in Fig. 3.

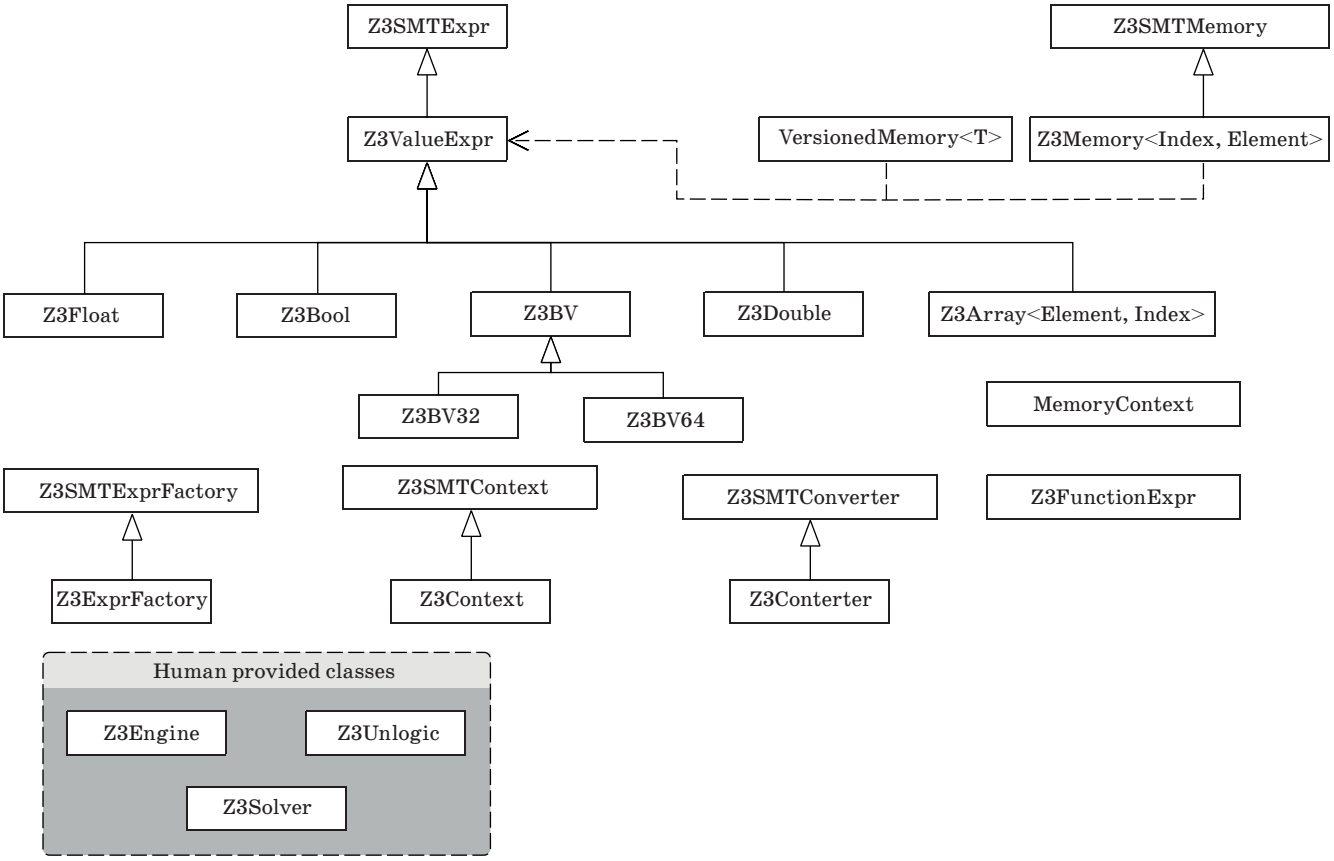
To add the new solver to Kex one needs to provide implementations of three classes: *Engine*, *Solver* and *Unlogic*. Engine class should provide bindings to the API of the solver. Solver class should implement methods that perform a query and return a model. Unlogic class should provide an interface to convert received model back into terms and predicates of PS.

In this section we will describe the model that Kex uses to express queries over PS in the SMT formula. First, however, let us describe the steps Kex uses to prepare PS.

Preparing PS

Predicate State preparation consists of two steps: reifying PS with additional information and complementing PS with necessary type information for SMT solving. The first step is optional and only used to give solver more information on the constraint solving: it inlines resolvable methods, includes available reflection and annotation information, propagates constants, etc. The result of the first step is the PS and the query over that state.

The main goal of the second step is to simplify the PS and the query so that SMT solver will be able



■ ***Fig. 3.*** Z3 solver wrapper classes

to solve it faster. It uses two techniques to reach its goal: memory spacing [40] and slicing.

Memory spacing is a technique that allows splitting all memory used in a program into a disjoint set of sub-memories. Each sub-memory is independent from the others and can be modeled separately. Each sub-memory is assigned a unique index, pointers referencing this memory are identified by the mentioned index. This reduces the complexity of solving the resulting formulae, as the disjoint set of memories decreases the search space SMT solver needs to work with.

Slicing is used to reduce the overall size of the PS. The main idea is to remove terms and predicates that are not "interesting" w.r.t. query from PS. The term is considered "interesting" if it affects or aliases any of the interesting terms. Aliasing is currently determined by Stensgaard alias analysis. Initial set of "interesting terms" contains all the variable (i. e. non-constant) terms from the query.

These preparation steps allow us to reinforce PS with additional information and simplify it w.r.t. SMT solving. Let us now consider how PS are encoded into an SMT formula.

Modeling program in SMT formulae

To be able to use SMT solver for constraint solving one needs to define a memory model suitable for representing the program and its variables as SMT formulae. In Kex we have used a memory model inspired by the work on bounded model checker Borealis. We have adapted its memory model to JVM and PS.

As was mentioned earlier, PS (like JVM bytecode) has several primitive data types: booleans, integers, floating point numbers. Each variable of a given type can be represented as an expression of corresponding SMT theory: booleans for boolean, bitvectors [41] for integers, floating point numbers [42] for float and double.

The more complex part, however, is modeling non-primitive data types: objects and arrays. To represent references in the heap we use a "property-based" memory model: memory is encoded as a collection of SMT arrays [43], each array corresponding to a disjoint partition of heap objects definitely not aliasing objects from other partitions. SMT arrays are immutable and each store operation returns a new version of the array. Therefore the memory model allows one to work with *versioned memory* i. e. one can potentially get the whole memory state of a program after execution of each instruction. Initial memory of the program is empty, it can be filled with special *FieldInitializer* and *ArrayInitializer* predicates.

This allows it to encode object references as 32-bit bitvector indices into their partition; arrays are represented as continuous chunks, with array reference pointing to its start index.

Object fields are represented in a similar fashion, using "property memories": each field is mapped to a separate SMT array, indexed by object references; to access field *x.y* one needs to work with property memory *typeOf(x).y* by index *x*.

Property memories also have one additional use case: they are used to calculate runtime type information of pointers. Resolving runtime type information is very important because it not only may affect control flow of a program (e. g. through *instanceof* instructions) but also is used to resolve virtual method calls. Each pointer variable of the program is assigned a special "type" property: each reference may be used as an index to this property memory to get its type. Kex analyses the program as a closed world model, therefore it can assign a constant to each defined type and encode subtyping via SMT axioms over a special *isSubtype* uninterpreted function:

$$\forall a,b \in types \begin{cases} isSubtype(a,b) = true, \\ if\ a \text{ is a subtype of } b \\ isSubtype(a,b) = false, \text{ otherwise} \end{cases}$$

All type-related operations in the program are expressed through *isSubtype*: casts and *instanceof* checks impose new constraints on the "type" property of the corresponding variable. That, together with the subtyping axioms, gives SMT solver enough information to correctly analyze types.

We have given a description of the memory model that Kex uses for symbolic execution. Given that, PS is a directed acyclic graph; its translation into SMT formula is straightforward, as predicates can be directly mapped to corresponding SMT expressions. One may vary the precision and complexity of SMT formulae by changing the depth of inlining and loop unrolling. An example of PS, query and corresponding SMT formula can be found in Fig. 4.

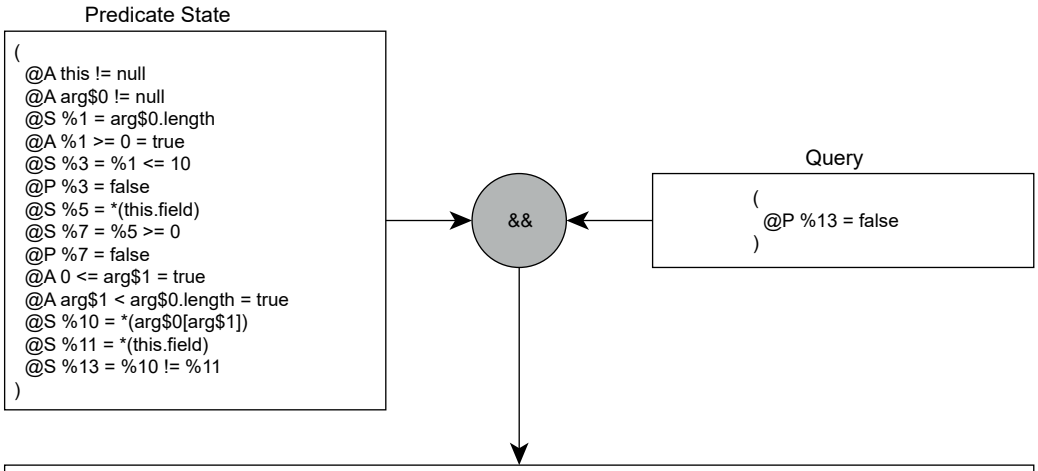
After encoding PS and query as SMT formulae Kex performs a request to SMT solver. SMT solver can return three types of answers:

— SAT — formula is satisfiable, solver also returns an SMT model containing counterexample that makes formula satisfiable;

— UNSAT — formula is unsatisfiable, solver also may return an unsatisfiable core [44], i. e. a minimal set of clauses that makes the formulae unsatisfiable;

— UNKNOWN — unknown result, returned if solver is terminated by timeout.

Depending on the query type, these results can be interpreted differently. However, if the formula is satisfiable, one needs to be able to raise the program state encoded in the SMT model to a higher level. Let us now describe how that is performed.




```
(declare-fun isSubtype ((_ BitVec 32) (_ BitVec 32)) Bool)
(declare-fun __word__property__type0!0 () (Array (_ BitVec 32) (_ BitVec 32)))
(declare-fun arg$0 () (_ BitVec 32))
(declare-fun __word__property__length0!3 () (Array (_ BitVec 32) (_ BitVec 32)))
(declare-fun |int[]!2| () (_ BitVec 32))
(declare-fun this () (_ BitVec 32))
(declare-fun PSExample!1 () (_ BitVec 32))
(declare-fun %13 () Bool)
(declare-fun %11 () (_ BitVec 32))
(declare-fun %10 () (_ BitVec 32))
(declare-fun __word__property__PSExample.field0!4 () (Array (_ BitVec 32) (_ BitVec 32)))
(declare-fun arg$1 () (_ BitVec 32))
(declare-fun __array__0!5 () (Array (_ BitVec 32) (Array (_ BitVec 32) (_ BitVec 64))))
(declare-fun %7 () Bool)
(declare-fun %5 () (_ BitVec 32))
(declare-fun %3 () Bool)
(declare-fun %1 () (_ BitVec 32))
(assert
  (and (not (= this #x00000000))
       (not (= arg$0 #x00000000))
       (= %1 (select __word__property__length0!3 arg$0))
       (bvsle #x00000000 %1)
       (= %3 (bvsle %1 #x0000000a))
       (not %3)
       (= %5 (select __word__property__PSExample.field0!4 this))
       (= %7 (bvsle #x00000000 %5))
       (not %7)
       (bvsle #x00000000 arg$1)
       (not (bvsle (select __word__property__length0!3 arg$0) arg$1))
       (= %10 ((_ extract 31 0) (select (select __array__0!5 arg$0) arg$1)))
       (= %11 (select __word__property__PSExample.field0!4 this))
       (= %13 (not (= %10 %11)))
       (bvsle #x00000000 (select __word__property__length0!3 arg$0))
       (not (bvsle #x000003e8 (select __word__property__length0!3 arg$0)))
       (bvsle #x00000000 this)
       (or (= this #x00000000)
           (= (select __word__property__type0!0 this)
              PSExample!1))
       (bvsle #x00000000 (select __word__property__type0!0 this))
       (not (bvsle #x3fffffff this))
       (bvsle #x00000000 arg$0)
       (or (= arg$0 #x00000000)
           (= (select __word__property__type0!0 arg$0) |int[]!2|))
       (bvsle #x00000000 (select __word__property__type0!0 arg$0))
       (bvsle #x00000000 (select __word__property__length0!3 arg$0))
       (not (bvsle #x3fffffff arg$0))
    )
  )
(assert (and true
     (= (isSubtype PSExample!1 PSExample!1) true)
     (= (isSubtype PSExample!1 |int[]!2|) false)
     (= (isSubtype |int[]!2| PSExample!1) false)
     (= (isSubtype |int[]!2| |int[]!2|) true))
  )
(assert (not %13))
(check-sat)
```

■ ***Fig. 4.*** An example of PS, query and SMT formula

Interpreting SMT model

An SMT model allows one to evaluate concrete values of formula expressions. In the context of program analysis those concrete values describe some interesting program state: counterexample that triggers a bug, test case that covers some branch, etc. Thus, to be useful, the SMT model eventually needs to be converted into Java objects or Java code that creates those objects. In Kex that conversion process consists of two steps: translating the model into terms and building Java objects from those terms.

The first step is straightforward: using special *unlogic* functions of each solver, Kex converts constants from the SMT model into constant terms. Kex model consists of three components:

— assignments — a map where each variable of a program is assigned a constant value evaluated from SMT model;

— memory shapes — each shape contains two memory states: initial and final. Each memory state maps concrete integer addresses into constants;

— type map — a map where each type of a program is assigned constant integer value. Type map is later used to evaluate runtime types of variables from type property memory.

An example of the Kex model corresponding to the SMT model is given in listing 3.

Listing 3. An example Kex model.

```
Model {
 this = 32768
 arg$0 = 1
 %1 = 35
 %3 = false
 %5 = -1
 this.field = 32768
 %7 = false
 arg$1 = 0
 %10 = -1
 %11 = -1
 %13 = false
 (1)<0> = 0
 (32768)<0> = 0
 (1073741823)<0> = 0
 PSExample.field(32768)<0> = -1
 length(1)<0> = 35
 type(1)<0> = 3
 type(32768)<0> = 4
 type(1073741823)<0> = 2
}
```

Although the Kex model is not very illustrative, it still captures the whole program state. For further transformation and analysis, Kex transforms SMT model into *descriptors*. Descriptors are used to represent the object shape; one may consider them trees that capture (nested) object states. The descriptor format for the JVM platform is given in listing 4.

Listing 4. JVM descriptor format.

```
<Descriptor> ::= “CostantDescriptor”
      | “ObjectDescriptor” fields:<ListOfFields>
      | “ArrayDescriptor” elements:<ListOfElements>
      | “StaticFieldDescriptor” field:<Field>

<ConstantDescriptor> ::= “NullDescriptor”
      | “BoolDescriptor” value:Boolean
      | “ByteDescriptor” value:Byte
      | “ShortDescriptor” value:Short
      | “CharDescriptor” value:Char
      | “IntDescriptor” value:Int
      | “LongDescriptor” value:Long
      | “FloatDescriptor” value:Float
      | “DoubleDescriptor” value:Double

<Field> ::= name:String klass:Class value:<Descriptor>

<Element> ::= index:Int value:<Descriptor>

<ListOfFields> ::= <Field> <ListOfFields> | <empty>

<ListOfElements> ::= <Element> <ListOfElements> | <empty>
```

Additionally Kex is able to build Java objects from the descriptors if needed. Kex collects all the variables from the program and builds Java objects for those projects using following algorithm:

— if a variable has primitive type, create a primitive Java variable with corresponding value;

— if a variable is an object, resolve its runtime type (using type map) and create Java object of resolved type (using Java reflection utilities);

— if a variable is an array, resolve its runtime type (using type map) and create Java array of resolved type (using Java reflection utilities);

— if a variable is a field, recursively create a Java object corresponding to its value and set the field value of an object (using Java reflection utilities);

— if a variable is an array element, recursively create a Java object corresponding to its value and set the element value of an array (using Java reflection utilities).

Kex also has techniques to generate a test case that recreates a program state corresponding to the SMT model, but its implementation details are left outside of this work.

Evaluation of Kex platform

The evaluation of our platform consists of two parts. First part is the qualitative comparison of Kex platform with other analogues considered earlier. Second part is the evaluation of Kex applicability for developing program analysis tools. In this part we will consider two prototypes of program analysis tools that were developed based on Kex platform.

Qualitative comparison with analogues

To evaluate our platform we decided to compare it with five other analogues, that were previously considered, based on seven criteria:

— source artifact: what artifacts does the tool takes as an input;

— source manipulation and transformation: does the tool provides utilities for transformation of the input sources;

— behavioral representation: whether tool provides behavioral program representation (like CFG, SSA, etc.) rather than simple stack-based bytecode;

— symbolic representation: does the tool provide a symbolic representation of a program that can be used for more in-depth analysis;

— constraint solving: does the tool provides API to work with any kind of constraint solvers;

— static analysis utilities: does the tool has built in utilities for static program analysis;

— dynamic program analysis: does the tool has built in utilities for static program analysis.

The results of the comparison can be found in the Table. As one can see from the results, ASM and Soot frameworks are libraries which are mainly focused on bytecode-level optimizations and do not provide tools for more in-depth analysis. Spoon is similar to ASM and Soot except that is concentrates on the Java source code analysis. JBSE is similar to Kex in almost every criteria; however, its main weakness is that it does not provide any utilities to work with behavioral program representations like CFG. JDQL is only tool that supports both source code and bytecode analysis and allows one to solve queries over program variables using Datalog. However, it is only suitable for a lightweight pattern recognition based static analysis and does not allow performing more precise and complex types of analyses. Judging by the results Kex is the tool that fits the most criteria; the only weakness is that Kex does not support source code analysis. That decision was intentional, because as source code analysis may provide more information about program (e. g. generics), it bounds the tool to only one programming language (or requires too much infrastructure for working with multiple languages).

Evaluation of prototypes

To evaluate our platform we have implemented a prototype of an automatic test generation tool for Java language based on Kex infrastructure. The prototype uses symbolic execution to analyze control flow graphs of the program under test (PUT) and produces interesting symbolic inputs for each basic block of PUT. Those symbolic inputs are then converted into JUnit test cases (either in Java or in Kotlin language). Prototype currently supports two modes of test case generation: basic, which generates reflection based test cases, and advanced, that tries to generate test cases using only public API's of the PUT. We have participated in the SBST 2021 Tool Competition [45, 46] with the described prototype. With an overall score of 44.21, Kex ranked fifth. Thorough analysis of the results has shown that the prototype had many technical issues due to a low degree of maturity of the project. On the *guava* project, Kex was able to reach ~20% line coverage, which is competitive

■ Qualitative comparison of Kex with analogues

Criteria	ASM	Soot	Spoon	JBSE	JDQL	Kex
Source artifact	JVM bytecode	JVM bytecode	Java	JVM bytecode	Java or JVM bytecode	JVM bytecode
Source manipulation and transformation	+	+	+	+	–	+
Behavioral representation	–	+	–	–	–	+
Symbolic representation	–	–	–	+	–	+
Constraint solving	–	–	–	SMTLib2 formulae for SMT solvers	Datalog queries	API for Z3, Boolector and STP SMT solvers
Static analysis utilities	–	–	–	+	+	+
Dynamic analysis utilities	–	–	–	+	–	+

with the results of other participating tools. After resolving all the technical issues with the prototype, it was able to reach ~25% average line coverage on the whole SBST 2021 competition benchmark (https://github.com/vorpal-research/kex/tree/sbst-21). We consider that as significant improvement.

Another application of Kex platform is Spider [47]. The authors had built a tool that allows them to find library integration errors using static analysis methods. Authors enrich the source code of external libraries with formal specifications written in LibSL specification language [48]. They use Kfg library to inject specification automata into the original library classes. All the necessary checks are marked with calls to a Kex intrinsics (https://github.com/vorpal-research/kex-intrinsics) library. Implemented analysis module finds the library API function calls and checks their conditions. If the condition can be false then intrinsic call is reachable, and the error occurs.

We conclude that Kex is an applicable and extendable platform for building various types of program analysis, both static and dynamic.

Conclusion

In this paper we presented a platform for analysis of Java programs called Kex. We have described all the main components of Kex, their implementation details and external APIs. Kex can be used to build tools for various types of program analysis, both lightweight and complex. During evaluation, we have considered two prototypes of program analysis tools: one for automatic test generation and one for integration errors detection. Evaluation has shown that Kex is applicable for creating program analysis tools for JVM platform.

In the future we plan to further work on improving capabilities of Kex. In terms of capabilities of Kex as a platform, we want to implement an exception handling mechanism in PS and improve lambda function support both on PS level and on SMT formulae level. Another area of our future work is the development of tools based on Kex. Currently we have two main priorities for the future: improve our automatic test generation tool for Java language and participate in the SBST 2022 competition and develop a concolic testing tool based on Kex.

References

1. Sharma R. M. Quantitative analysis of automation and manual testing. *International Journal of Engineering and Innovative Technology*, 2014, no. 1, pp. 252–257.
2. De Stefano M., Gambardella M. S., Pecorelli F., Palomba F., De Lucia A. cASpER: A Plug-in for automated code smell detection and refactoring. *Proceedings of the International Conference on Advanced Visual Interfaces*, 2020, pp. 1–3. doi:10.1145/3399715.3399955
3. Jhala R., Majumdar R. Software model checking. *ACM Computing Surveys (CSUR)*, 2009, no. 4, pp. 1–54. doi:10.1145/1592434.1592438
4. Zhang T., Wang P., Guo X. A survey of symbolic execution and its tool KLEE. *Procedia Computer Science*, 2020, pp. 330–334. doi:10.1016/j.procs.2020.02.090
5. Braione P., Denaro G., Pezzè M. JBSE: A symbolic executor for Java programs with complex heap inputs. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1018–1022. doi:10.1145/2950290.2983940
6. Gadelha M. R., Menezes R. S., Cordeiro L. C. ESBMC 6.1: automated test case generation using bounded model checking. *International Journal on Software Tools for Technology Transfer*, 2021, no. 6, pp. 857–861. doi:10.1007/s10009-020-00571-2
7. Klees G., Ruef A., Cooper B., Wei S., Hicks M. Evaluating fuzz testing. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138. doi:10.1145/3243734.3243804
8. Zhang L., Xie T., Zhang L., Tillmann N., De Halleux J., Mei H. Test generation via dynamic symbolic execution for mutation testing. *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10. doi:10.1109/icsm.2010.5609672
9. Sen K. Concolic testing. *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 571–572. doi:10.1145/1321631.1321746
10. Ayewah N., Pugh W., Hovemeyer D., Morgenthaler J. D., Penix J. Using static analysis to find bugs. *IEEE Software*, 2008, no. 5, pp. 22–29. doi:10.1109/ms.2008.130
11. Calcagno C., Distefano D., Dubreil J., Gabi D., Hooimeijer P., Luca M., O'Hearn P., Papakonstantinou I., Purbrick J., Rodriguez D. Moving fast with software verification. *NASA Formal Methods Symposium*, Cham, 2015, pp. 3–11. doi:10.1007/978-3-319-17524-9_1
12. Nielsen B. B., Møller A. Value Partitioning: A lightweight approach to relational static analysis for JavaScript. *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, 2020, pp. 16:1–16:28.
13. Böhme M., Pham V. T., Nguyen M. D., Roychoudhury A. Directed greybox fuzzing. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344. doi:10.1145/3133956.3134020
14. Kroening D., Tautschnig M. CBMC–C bounded model checker. *International Conference on Tools and Algo-*

rithms for the Construction and Analysis of Systems, Berlin, 2014, pp. 389–391. doi:10.1007/978-3-642-54862-8_26
15. Visser W., Păsăreanu C. S., Khurshid S. Test input generation with Java PathFinder. *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004, pp. 97–107. doi:10.1145/1007512.1007526
16. Bruneton E., Lenglet R., Coupaye T. ASM: A code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems*, 2002, no. 19. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.5769 (accessed 5 December 2021).
17. Vallée-Rai R., Co P., Gagnon E., Hendren L., Lam P., Sundaresan V. Soot: A Java bytecode optimization framework. *CASCON First Decade High Impact Papers*, 2010, pp. 214–224. doi:10.1145/1925805.1925818
18. Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991, no. 4, pp. 451–490. doi:10.1145/115372.115320
19. Pawlak R., Monperrus M., Petitprez N., Noguera C., Seinturier L. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 2016, no. 9, pp. 1155–1179. doi:10.1002/spe.2346
20. Shigeru C. Load-time structural reflection in Java. *14th European Conference on Object-Oriented Programming (ECOOP 2000)*, 2000, pp. 313–336. doi:10.1007/3-540-45102-1_16
21. Barrett C., Stump A., Tinelli C. The SMT-LIB Standard: Version 2.0. *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, Edinburgh, 2010. 14 p.
22. Demers F. N., Malenfant J. Reflection in logic, functional and object-oriented programming: A short comparative study. *Proceedings of the IJCAI*, 1995, pp. 29–38.
23. Braione P., Denaro G., Mattavelli A., Pezzè M. SUSHI: a test generator for programs with complex structured inputs. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 21–24. doi:10.1145/3183440.3183472
24. Fraser G., Arcuri A. Evosuite: automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 416–419. doi:10.1145/2025113.2025179
25. Braione P., Denaro G. SUSHI and TARDIS at the SBST2019 tool competition. *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, 2019, pp. 25–28. doi:10.1109/sbst.2019.00016
26. Saxena A., Soundrapandian P. D., Sharma V. S., Kaulgud V. JDQL: A framework for Java Static Analysis. *Proceedings of the 9th India Software Engineering Conference*, 2016, pp. 136–140. doi:10.1145/2856636.2856645
27. Huang S. S., Green T. J., Loo B. T. Datalog and emerging applications: an interactive tutorial. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011, pp. 1213–1216. doi:10.1145/1989323.1989456
28. Lindholm T., Yellin F., Bracha G., Buckley A. *The Java virtual machine specification*. Pearson Education, 2014. Available at: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html (accessed 5 December 2021).
29. Palsberg J., Jay C. B. The essence of the Visitor pattern. *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)*, 1998, pp. 9–15. doi:10.1109/cmpsac.1998.716629
30. Lattner C., Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization*, 2004, pp. 75–86. doi:10.1109/cgo.2004.1281665
31. Kaushik M. D. *Loop Fusion in LLVM Compiler*. Bach. of eng. Diss., Visvesvaraya Technological University, 2015. 39 p.
32. Akhin M., Belyaev M., Itsykson V. Borealis bounded model checker: The coming of age story. *Present and Ulterior Software Engineering*, Cham, 2017, pp. 119–137. doi:10.1007/978-3-319-67425-4_8
33. McCracken D. D., Reilly E. D. Backus-naur form (bnf). *Encyclopedia of Computer Science*, 2003, pp. 129–131.
34. Coplien J. O. Curiously recurring template patterns. *C++ gems*, May 1996, pp. 135–144.
35. Steensgaard B. Points-to analysis in almost linear time. *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996, pp. 32–41. doi:10.1145/237721.237727
36. Weiser M. Program slicing. *IEEE Transactions on Software Engineering*, 1984, no. 4, pp. 352–357. doi:10.1109/tse.1984.5010248
37. De Moura L., Bjørner N. Z3: An efficient SMT solver. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24
38. Niemetz A., Preiner M., Biere A. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 2014, no. 1, pp. 53–58. doi:10.3233/sat190101
39. Ganesh V., Dill D. L., A decision procedure for bit-vectors and arrays. *Computer Aided Verification, 19th International Conference*, Berlin, 2007, pp. 519–531. doi:10.1007/978-3-540-73368-3_52
40. Kapus T., Cadar C. A segmented memory model for symbolic execution. *Proceedings of the 2019 27th*

ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 774–784. doi:10.1145/3338906.3338936

41. Jha S., Limaye R., Seshia S. A. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. *International Conference on Computer Aided Verification*, Berlin, 2009, pp. 668–674. doi:10.1007/978-3-642-02658-4_53
42. Rümmer P., Wahl T. An SMT-LIB theory of binary floating-point arithmetic. *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010, p. 151.
43. Stump A., Barrett C. W., Dill D. L., Levitt J. A decision procedure for an extensional theory of arrays. *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, 2001, pp. 29–37. doi:10.1109/lics.2001.932480
44. Guthmann O., Strichman O., Trostanetski A. Minimal unsatisfiable core extraction for SMT. *2016 Formal Methods in Computer-Aided Design (FMCAD)*, 2016, pp. 57–64. doi:10.1109/fmcad.2016.7886661
45. Panichella S., Gambi A., Zampetti F., Riccio V. SBST tool competition 2021. *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021, pp. 20–27. doi:10.1109/sbst52555.2021.00011
46. Abdullin A., Akhin M., Belyaev M. Kex at the 2021 SBST tool competition. *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021, pp. 32–33. doi:10.1109/sbst52555.2021.00014
47. Feofilaktov V., Itsykson V. M. SPIDER: Specification-based integration defect revealer. *International Conference on Tools and Methods for Program Analysis*, Tomsk, 2021. Available at: https://arxiv.org/abs/2202.03943 (accessed 9 February 2021).
48. Itsykson V. Partial specifications of libraries: Applications in software engineering. *International Conference on Tools and Methods for Program Analysis*, Cham, 2019, pp. 3–25. doi:10.1007/978-3-030-71472-7_1

УДК 004.05
doi:10.31799/1684-8853-2022-1-30-43

Kex: платформа для анализа JVM-программ

А. М. Абдуллин[а,б], аспирант, ассистент, orcid.org/0000-0002-9669-2587
В. М. Ицыксон[а,б], канд. техн. наук, доцент, orcid.org/0000-0003-0276-4517, vlad@icc.spbstu.ru
[а]Санкт-Петербургский политехнический университет Петра Великого, Политехническая ул., 19, Санкт-Петербург, 195251, РФ
[б]JetBrains Co. Ltd., Приморский пр., 70, к. 1, Санкт-Петербург, 197374, РФ

Введение: методы статического и динамического анализа программ все чаще используются для проверки качества программного обеспечения. Однако разные виды анализа программ требуют работы с разными моделями представления программ, методами анализа и т. д. Возросла важность платформ для создания инструментов анализа программ, так как они позволяют упростить и ускорить процесс разработки. **Цель:** разработать платформу для анализа JVM-программ. **Результаты:** разработана платформа Kex для построения инструментов анализа программ, компилирующихся в JVM-байткод. Kex предоставляет три уровня абстракции. Первый уровень — библиотека Kfg — реализует граф потока управления в форме статического однократного присваивания для анализа и трансформации JVM-байткода. Второй уровень — символьное представление программы, называемое Predicate State, которое состоит из предикатов логики первого порядка, соответствующих инструкциям программы, контрактам, дополнительным ограничениям и т. д. Третий уровень — интерфейс для создания и работы с SMT-формулами, позволяющий решать задачи выполнимости. Интерфейс в данный момент поддерживает взаимодействие с тремя SMT-решателями. **Практическая значимость:** платформа Kex использовалась при разработке двух инструментов: автоматической генерации тестов для языка Java, который был подан на соревнования SBST 2021, и автоматического поиска ошибок интеграции библиотек. Оба этих прототипа показали, что платформа Kex может быть использована для разработки инструментов автоматического анализа программ.

Ключевые слова — анализ программ, платформа для анализа программ, автоматическая генерация тестов, символьное исполнение.

Для цитирования: Abdullin A. M., Itsykson V. M. Kex: A platform for analysis of JVM programs. *Информационно-управляющие системы*, 2022, № 1, с. 30–43. doi:10.31799/1684-8853-2022-1-30-43
For citation: Abdullin A. M., Itsykson V. M. Kex: A platform for analysis of JVM programs. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2022, no. 1, pp. 30–43. doi:10.31799/1684-8853-2022-1-30-43