ISSN (e): **2250 – 3005** || Volume, 07 || Issue, 05|| May – 2017 ||
International Journal of Computational Engineering Research (IJCER)

DySy: Dynamic Symbolic Execution for Invariant Inference

[1]debabrata Sahu, [2]rojalin Mallick
Gandhi Institute of Excellent Technocrats, Bhubaneswar,India
Rayagada Institute of Technology and Management, Rayagada, Odisha, India

ABSTRACT
Dynamicallydiscoveringlikelyprograminvariantsfromcon-crete test executions has emerged as a highly promisingsoftware engineering technique.Dynamic invariant infer-ence has the advantage of succinctly summarizing both "ex-pected" program inputs and the subset of program behaviorsthat is normal under those inputs. In this paper, we intro-duceatechniquethatcandrasticallyincreasetherelevanceof inferred invariants, or reduce the size of the test suiterequired to obtain good invariants. Instead of falsifying in-variants produced by pre-set patterns, we determine likelyprogram invariants by combining the concrete execution ofactual test cases with a simultaneous symbolic execution ofthe same tests.The symbolic execution produces abstractconditions over program variables that the concrete testssatisfy during their execution.In this way, we obtain thebenefitsofdynamicinferencetoolslikeDaikon:theinferredinvariants correspond to the observed program behaviors.Atthesametime,however,ourinferredinvariantsaremuchmore suited to the program at hand than Daikon's hard-coded invariant patterns. The symbolic invariants are liter-ally derived from the program text itself, with appropriatevaluesubstitutionsasdictatedbysymbolicexecution.
We implemented our technique in the DySy tool, whichutilizesapowerfulsymbolicexecutionandsimplificationen-gine.The results confirm the benefits of our approach.InDaikon's prime example benchmark, we infer the majorityof the interesting Daikon invariants, while eliminating in-variantsthatahumanuserislikelytoconsiderirrelevant.

I. INTRODUCTION AND MOTIVATION

Dynamic invariant inference was introduced less than adecade ago,pioneered by the Daikon tool [8,9,29], andhas garnered significant attention in the software engineeringcommunity.Withthehelpofatestsuitethatexercisesthe functionality of an application, an invariant inferencesystem observes program properties that hold at pre-selectedprogram points (typically method entries and exits).Theoutcome of the system is a collection of such properties,postulatedasobjectstateinvariants,methodpreconditions,or method postconditions (collectively called "invariants" inthe following).The properties have no formal assurance thatthey are correct, but they do match the observed programexecutions and they are produced only when there is somestatisticalconfidencethattheiroccurrenceisnotaccidental.A crucial aspect of the dynamic invariant inference process isthattheinvariantsproduceddonotreflectonlythebehaviorof the program, but also the assumptions and expectationsof the test suite. This makes the approach doubly usefulforsoftwareengineeringpurposes,byintroducingtheusagecontextofanapplication.
So far, dynamic invariant inference systems have had apre-setcollectionofinvarianttemplates,whichgetinstanti-ated for program variables to produce the candidate invari-antsunderexamination.Theusercanexpandthecollectionby adding more templates, but the number of possible in-stantiationsforallcombinationsofprogramvariablesgrowsprohibitively fast.Therefore, dynamic invariant inferencesystems typically perform best by concentrating on a smallset of simple candidate invariants.Even so, for a tool likeDaikonorDIDUCE[18]toproduceinvariantsthatmatchthe understanding of a human programmer, an extensivetestsuitethatthoroughlyexercisestheapplicationisneces-sary. Furthermore, it is likely that the inference process willalso produce several invariants that are either irrelevant orfalse(i.e.,holdaccidentally).
In this paper we propose a dynamic symbolic executiontechnique to drastically improve the quality of

inferred in-variants (i.e., the percentage of relevant invariants) or theease of obtaining them (i.e., the number of test cases re-quiredto disqualify irrelevant invariants).[1]In dynamic sym-bolicexecution,weexecutetestcases,justlikeatraditionaldynamic invariant inference tool, but simultaneously alsoperformasymbolicexecutionoftheprogram.Thesymbolicexecutionresultsintheprogram'sbranchconditionsbeing

```
inttestme(intx,inty){ intprod=x*y;
if(prod<0)
thrownewArgumentException(); if(x<y){    //swapthem
inttmp=x;x=y;
y=tmp;
}
intsqry=y*y;
returnprod*prod-sqry*sqry;
}
```

Figure 1: An example method whose invariants wewanttoinfer.

collected in an expression, called the path condition in thesymbolic execution literature. The path condition is alwaysexpressed in terms of the program inputs.It gets refinedwhilethetestexecutiontakesplace,andsymbolicvaluesofthe program variables are being updated. At the end of ex-ecution of all tests, the overall path condition correspondsto the precondition of the program entity under examina-tion. Symbolic values of externally observed variables pro-vide the dynamically inferred postconditions, and symbolicconditions that are preconditions and postconditions for allmethodsofaclassbecometheclassstateinvariants.

For a demonstration of our technique, consider the methodofFigure1.(Theexampleisartificialbutisdesignedtoil-lustrateseveralpointsthatwemakethroughoutthepaper.)Appropriateunittestsforthemethodwillprobablyexerciseboththecase"x<y"anditscomplement,butareunlikelytoexercisethecodeproducinganexception,asthisdirectlysignifiesillegalarguments.Considertheoutcomeofexecut-ingthecodeforinputvaluesxsmallerthany(e.g.,x==2,y==5),whilealsoperformingtheexecutioninasym-bolic domain with symbolic values x and y (we overload thevariablenamestoalsodenotetherespectivesymbolicvaluesdesignatingtheoriginalinputs).Thefirstsymboliccondi-tionthatweobserveis"x*y>=0":Thebranchofthefirstifisnottaken,andlocalvariableprodhasthevaluex*yinthesymbolicdomain.Thesymbolicexecutionalsoaccumu-latesthecondition "x<y" fromthesecondifstatement.Attheendofexecutionthesymbolicvalueofthereturnedexpressionis"y*x*y*x –x*x*x*x".Notethatthisexpres-sionintegratestheswappingoftheoriginalxandyvalues.If we repeat this process for more test inputs (also exercis-ingtheothervalidpathofthemethod)andcollecttogether thesymbolicconditions,thenourapproachyields:

- Apreconditionx*y>=0forthemethod.
- Apostconditionresult==(((x<y)–>y*x*y*x –x*x*x*x)else–>(x*y*x*y–y*y*y*y)).

(Our example syntax is a variation of JML [23]:we intro-duce an if-else-like construct for conciseness.Our tool'soutput syntax is different but equivalent.) This capturesthemethod'sbehaviorquiteaccurately,whileensuringthatthe only symbolic conditions considered are those consis-tentwithactualexecutionsofthetestsuite.Thustheapproach is symbolic, but at the same time dynamic:thesymbolicexecutionisguidedbyactualprogrambehavioron test inputs.Note that the inferred invariants are notpostulated externally, but instead discovered directly fromthe program's symbolic execution.This approach directlyaddresses many of the shortcomings of prior dynamic in-variant inference tools (with Daikon used as the foremostreference point). For this example, Daikon-inferred precon-ditionsandpostconditionsareexclusivelyoftheform"var >= 0" or "var== 0", and are often encoding arbitrary ar-tifacts of the test suite, unless a very thorough test planexercises many possible combinations with respect to zero(e.g.,x,ybothnegative,bothpositive,one/bothzero,etc.).

Overall,ourworkmakesthefollowingcontributions:

- We introduce the idea of using dynamic symbolic ex-ecutionforinvariantinference.Webelievethatourapproachrepresentsthefutureofdynamicinvariantinference tools, as it replaces a blind search for possi-ble invariants with a well-founded derivation of suchinvariantsfromtheprogram'sconditionsandside-effects.

We implemented our approach in the invariant infer-ence tool DySy, built on top of the Pex framework forinstrumentation and symbolic execution of .NET pro-grams.We discuss the heuristics used by DySy in orderto simplify symbolic conditions (e.g., our abstractionheuristicsfordealingwithloops).DySyrepresentswellthe benefits of the proposed technique. For instance,DySy'ssymbolicapproachcaninferinvariants,suchaspurity (absence of side-effects), that are too deep fortraditional dynamic tools.In contrast, prior dynamic invariant inference tools (which only observe after-the-fact effects) typically can establish purity only in verylimitedsettings,astheywouldneedtoobservetheen-tirereachableheap.

We evaluate DySy in direct comparison with Daikon, in order to showcase the tradeoffs of the approach. FortheStackArbenchmark(hand-translatedintoC#),which has been thoroughly investigated in the Daikonliterature [10], DySy infers 24 of the 27 interesting in-variants (as independently inferred by a human user),while eliminating Daikon's multiple irrelevant or acci-dentalinvariants.

The rest of this paper begins with a brief discussion ofwhat our work is not (Section 2), and continues with somebackgroundondynamicinvariantinferenceandsymbolicex-ecution (Section 3) before detailing the technical aspects ofourapproachandtool(Section4)andpresentingourevalu-ation (Section 5). Related work (Section 6) and our conclu-sions(Section7)follow.

II. POSITIONING

Ourapproachisacombinationofsymbolicexecutionwithdynamic testing.As such, it has commonalities with mul-tiple other approaches in the research literature. To avoidearly misunderstandings, we next outline a few techniquesthatmayatfirstseemsimilartoourapproachbutaredeeplydifferent.

- Our dynamic symbolic execution is not equivalent toliftin g conditions from the program text (e.g., condi-tionsin**if**statementsorinwh**il**eloops)andpostulating them as likely invariants.(Several prior analy-sis tools do this—e.g., Liblit's statistical bug isolationapproach[25]andDaikon'sCreateSp**i**nfosupporting utility.)For instance, notice how the precondition ofour example in the Introduction (x*y > = 0) does notappear anywhere in the program text.Instead, pro-gramconditionsarechangedduringthecourseofsym-bolic execution:local variable bindings are replacedwith their symbolic values, and assignments updatethe symbolic values held by variables, thus affectingthepathcondition.
- Our approach is not invariant inference through statictechniques (e.g., using abstract interpretation [26], orsymbolic execution [32]). Inferring invariants throughstatic analysis is certainly a related and valuable tech-nique,butitismissingthedynamicaspectofourwork,as it takes into account only the program text and notthe behavior of its test suite. Specifically, our dynamicsymbolic execution uses the test suite as a way to dis-cover properties that users of the code are aware of.This is highly valuable in practice, as invariant infer-ence tools are often used to "read the programmer'smind"anddiscovertheinterestingparameterspaceofamethod(e.g.,fortesting[5]).
- Ourapproachis notconcolic execution (as in toolslikeDart[17],Cute[30],orParasoft's original "dy-namicsymbolicexecution" patent[21]). Although wedoaconcreteexecutionoftestcasesinparallelwithasymbolicone,wedonotusethesymbolicexecutionto produce more values in order to influence the pathtaken by concrete executions.Our technique followsprecisely the concrete program paths that the originaltestsuiteinduces.

III. BACKGROUND

We next present some background on dynamic invariantinference and on symbolic execution, emphasizing the fea-tures of both that are particularly pertinent to our laterdiscussion.

DynamicInvariantInference:Daikon

Dynamicinvariantinferenceisexemplifiedby(andoftenevenidentifiedwith)theDaikontool[8,9,27,29]—thefirstandmostmaturerepresentativeoftheapproach,withthewidest use in further applications (e.g., [2, 6, 7, 11, 22, 36]).Daikontracksa program'svariables during executionandgeneralizestheobservedbehaviortoinvariants—preconditions,postconditions,andclassinvariants.Daikoninstrumentstheprogram,executesit(forexample,onanexistingtestsuiteorduringproductionuse),andanalyzestheproducedexecutiontraces.Ateachmethodentryandexit,Daikoninstantiatessomethreedozeninvarianttem-plates,includingunary,binary,andternaryrelationsoverscalars,andrelationsoverarrays.Examplerelationsin-cludecomparison'sofavariablewithaconstantvalue(x =a, or x> 0), linear relationships (y == a*x + b), order-ing (x <= y), membership and sortedness, etc.Users canextend the invariant templates with application-specific ordomain-specific properties.The number of candidate in-variants grows combinatorially, however. For each invarianttemplate, Daikon tries several combinations of method pa-rameters,methodresults,andobjectstate. Forexample, itmightproposethatsome methodmneverreturnsnull, or that its first argument is always larger than its second.Daikonsubsequentlydisqualifiesinvariantsthatarerefutedbyanexecutiontrace—forexample,itmightprocessasitu-ationwheremreturnednullanditwillthereforeignorethe above invariant.So Daikon summarizes the behavior ob-served in the execution traces as invariants and generalizesit by proposing that the invariants might hold in all otherexecutions as well. Daikon can annotate the testee's sourcecodewiththeinferredinvariantsasJMLannotations[23].

SymbolicExecution

Symbolicexecution[20]isatechniqueforusingapro-gram'scodetoderiveageneralrepresentationofitsbe-havior, by simulating execution with some values being un-known.Specifically,symbolicexecutionreplacesthe con-creteinputsofaprogramunit(typically,amethod)withsymbolicvalues,andsimulatestheexecutionof the pro-gramsothatall variables hold symbolic expressions overtheinput symbols,insteadofvalues. For symbolicexecu-tion to "simulate" regular, concrete execution, its semanticsmustcorrectlygeneralizethat of concrete execution. Thekey property is commutativity:performing symbolic execu-tion and instantiating its output state with concrete valuesmust yield the same result as instantiating the initial sym-bolicstatewiththesameconcretevaluesandperformingconcreteexecution.

A concept of symbolic execution that is particularly im-portant for our work is that of a path condition, defined as"theaccumulatorofpropertieswhichtheinputsmustsatisfyin order for an execution to follow the particular associatedpath" [20].Thus, a path condition can be seen as a precon-ditionforaprogrampath,whichisexactlythewayweuseitinourwork.

Generally,thegreatestchallengeofsymbolicexecutionisto reason about symbolic program properties. For instance,in traditional symbolic execution, when accumulating pred-icates in the path condition, it is important to recognizewhenthepathconditionbecomesunsatisfiablebytheaddi-tionofanextrapredicate—i.e.,whentheexistingpathcon-dition contradicts a program branch.To do so, a symbolicreasoningengine(typicallyanautomatictheoremprover)isemployed.Inourapproach,wedonotneedtorecognizein-feasible program paths, as the concrete execution guaranteesthat the paths we are examining are feasible. Nevertheless,weneedsimilarautomaticreasoningpowerinordertosim-plify path conditions and symbolic expressions and presentthemtotheuserasprograminvariants.

IV. DYNAMICSYMBOLIC EXECUTION FOR INVARIANT INFERENCE

Wenextdiscussthegeneralelementsofourapproach,as well as the technical specifics of our DySy tool, and theabstractionheuristicsweemployforhandlingloops.

OverviewandInsights

As outlined earlier, our dynamic symbolic execution per-forms a symbolic execution of the program simultaneouslywith its concrete execution. For a method under examina-tion, all class instance and static variables, the method'sparameters, and the method's result are treated as sym-bolicvariables.Thepathconditionofthesymbolicexecutionisdeterminedpurelybythepathstakenintheconcreteexecution—no exploration of other paths using symbolic val-ues is performed. When executing a single test case, the pathcondition at the end of the symbolic execution representsthe symbolic condition for the path the program followed.Thus, the path condition corresponds exactly to a precon-ditionfor that particular test case execution. Similarly, thesymbolic values of the method's result and of the objectinstance variables form the method'spostcondition for thespecific test case. Repeating the process for all test cases, weget a collection of preconditions and postconditions, whichall need to hold for the method.Combining the precon-ditions and postconditions for all test runs, we obtain thetotal precondition and postcondition of the method.Theconditionsaresimplifiedthroughsymbolicreasoningbeforebeing presented to the user.Individual conditions (i.e., with-outlogicaldisjunction—seelater)thatconcernonlyinstancevariables (i.e., no parameters) and that hold on entry andexitofallmethodsarereportedasclassinvariants.

This general scheme elides several important elements.Thefirstinterestingpointconcernshowconditionsarecom-bined.ConsiderthefollowingsimplemethodfromtheStackArbenchmark,describedinSection5.

```
publicObjecttop(){ if(Empty)
returnnull;
returntheArray[topOfStack];
}
```

Imagine that we execute this method for two test cases:firstonanemptystackandthen on a non-empty one.Thefirstexecutionproducesapathcondition“Empty== true”.(EmptyisaC#“property”,thereforetakingits valueresultsincallingamethod,whichchecksthevalueoftop0fStack.Nevertheless,thismethodispureso oursystem usesEmptyasalogicalvariableincondi- tions instead of expanding it, as we discuss later in Sec-tion4.2.)Thepathconditionbecomesthepreconditionof the method for this test case.Similarly, the postcon-ditionis“result==null”,againonlyforthisparticu- lar execution. The test case of a non-empty stack pro-ducesaprecondition“Empty==false&&top0fStack>= 0&&top0fStack<theArray.Length”.Thecorresponding postconditionis“result==theArray[top0fStack]”.

Combiningpreconditionsisdonebytakingthedisjunction(logical-or) of the individual test cases’ preconditions.Inthisexample,thecombinedpreconditionbecomes:

Empty==true||
(Empty==false&&top0fStack>=0&&top0fStack<theArray.Length)

Similarly,postconditions are combined by taking theirconjunction but appropriately predicated with the corre-sponding precondition.Following common convention, wereport the conjunction of postconditions as two separatepostconditions. In our example, the inferred postconditionsbecome:

Empty==true==>(\result==null)
and
(Empty==false&&top0fStack>=0&&top0fStack<theArray.Length)
==>(\result==theArray[top0fStack])

Combininginvariantsbydisjunction,conjunction,andim-plication brings out an interesting feature of our approach.Consider method preconditions.The crux of every dy-namic invariant inference system is its abstraction technique.Givenamethodvoidm(inti)andtestinputvaluesfrom1 to1000,themostprecisepreconditionthataninvariantsys-temcaninferisbydisjunction—i.e.,“i==1||i==2|| ... || i == 1000”. Generally, the system can be preciseby inferring one disjunct for every test case executed, andcombining them to form the complete precondition. Never-theless, this precision means that dynamic observations donot generalize to other test inputs that have not been al-readyencountered.Thevalueofaninvariantinferencetoolis exactly in this generalization. Thus, traditional dynamicinvariantinferencetools(suchasDaikonorDIDUCE)oftenavoid combining observations precisely using disjunction andinstead try to generalize and abstract. For instance, a rea-sonable abstract precondition for the above inputs is “i >0”. Once conditions have been abstracted sufficiently, theycan be combined precisely across test cases using disjunc-tion. The tool is overall responsible for heuristically decid-ingwhentousedisjunctionandwhentoabstractawayfromconcrete observations. The typical result is that dynamic in-variant inference tools use disjunction (i.e., multiple cases)sparingly,insteadpreferringtogeneralize,whichoftenleadsto over-generalization.Instead, our approach employs nosuch heuristics.Our observations are already generalized,since they correspond to branch conditions in the programtext, appropriately modified in the course of symbolic exe-cution.Thus, they can freely be combined precisely usingdisjunction. Even if there is a large number of test inputs,thenumberofdisjunctsinouroutputisboundedbythepro-gram paths in the method under examination. (Of course,the number of program paths can be infinite in the case ofloops, and we have to apply special abstraction techniques,discussedlaterinthepaper.)

Another interesting feature of the dynamic symbolic ex-ecution technique is that some relatively “deep” invariantscan be easily established. For the above example, our DySytool easily infers the postconditionpure, indicating that themethodhasnoeffectsvisibletoitsclients.Incontrast,tradi-tionaldynamicinvariantinferencetoolstreatthemethodasa black box, and can only establish shallow properties withobservations at its boundaries. For instance, Daikon infersseveral shallow purity properties for the above example, suchas“theArray==old(theArray)”.Itcannot,however,es-tablishthefullpurityofthemethodrelativetoallreachableheapdata(e.g.,withrespecttotheelementsheldinsid ethearray,andalltheelementsreferencedbythem,etc.).

Finally, the dynamic symbolic execution approach to in-variant inference is heavily dependent on a symbolic rea-soningengine(e.g.,atheoremprover)forproducingoutputthat is close to the expectations of a human user. Withoutsymbolic simplification of conditions, invariants end up tooverbose, with multiple tautologies.For a simple example,consideramethodallIntswiththefollowingstructure:

```
voidallInts(inti){ if(i<O)
{...}//dosomething
elseif(i==0){...}//dosomethingelse i++;
if(i>1){...}//dosomethingelse
}
```

If the program’s regression test suite exercises all paths,then it is natural to expect a precondition of

true ratherthantheunreduced“((i<0)&&!(i+1>1))
||(!(i<0)&&(i==0)&&!(i+1>1))
||(!(i<0)&&!(i==0)&&(i+1>1))”.Thus,sym- bolicreasoningisnecessaryto establish this tautology.It is worth noting that existing dynamic invariant infer-ence tools can also benefit from symbolic reasoning in or-der to simplify their reported invariants.For instance,Daikon produces several extraneous invariants for the ear-liertoproutineofStackAr:apostcondition“topOfStack
\==old(topOfStack)”isreported,butotherpostconditionsincludebothclauses“result==theArray[topOfStack]” and“result==theArray[old(topOfStack)]”.

We next discuss our specific implementation of dynamicsymbolic execution for invariant detection in the DySy tool.DySy benefits from the mature symbolic execution and rea-soningcapabilitiesofthePexframework.

DySy,Pex,andSymbolicReasoning

Pex [33] is a dynamic analysis and test generation frame-work for .NET, developed by the Foundations of SoftwareEngineering group at Microsoft Research. Pex monitors theexecution of a program through code instrumentation. Theinstrumented code drives a “shadow interpreter” in parallel with the actual program execution. For every regular .NETinstruction,there is a callback to Pex,which causes the“shadow interpreter” to execute the operation symbolically.The Pex interpreter is almost complete for the .NET instruc-tion set. It is only missing the logic to perform control-flowdecisions,sinceitispassivelymonitoringtheactualprogramexecution,whichperformsthedecisionactively.

Pex’s main functionality is similar to the Dart tool [17]:Pextestsprogramsexhaustivelyinafeedbackloop,inwhichan automatic constraint solver finds new test inputs thatrepresent execution paths that Pex did not monitor yet.While we do not use this test input generation feature inDySy, we do use Pex’s capability to construct and reasonaboutsymbolicprogramstates.

Background:PexSymbolicStates,Terms

A symbolic program state is a predicate over logical vari-ablestogetherwithanassignmentoftermsoverlogicalvari-ables to locations, just as a concrete program state is anassignment of values to locations. The locations of a statemay be static fields, instance fields, method arguments, lo-cals,andpositionsontheoperandstack.

Pex’s term constructors include primitive constants (inte-gers,floats,objectreferences),andfunctionsoverintegersandfloatsrepresentingparticularmachineinstructions,e.g.,additionandmultiplication.Othertermconstructorsimple-mentcommondatatypessuchastuplesandmaps.Pexusestuplestorepresent.NETvaluetypes(“structs”),and mapstorepresentinstancefieldsandarrays,similartotheheapencodingofESC/Java[13]:Aninstancefieldofanobjectisrepresentedbyasinglemapwhichassociatesobjectref-erenceswithfieldvalues.Constraintsoverthe.NETtypesystemandvirtualmethoddispatchlookupscanbeencodedaswell.Predicatesarerepresentedbyboolean-valuedterms.Peximplementsvarioustechniquestoreducetheoverheadof the symbolic state representation. Before building a newterm,Pexalwaysappliesasetofreductionrulesthatcom-puteanormalform.Asimpleexampleofareductionrule
is constant folding, e.g., 1 + 1 is reduced to 2.All logicalconnectivesaretransformedintoaBDDrepresentationwithif-then-else terms [3]. All terms are hash-consed, i.e., onlyone instance is ever allocated in memory for all structurallyequivalentterms.

Recall the method top given in an earlier example. WhenweexecutethemethodandEmpty==false,thentheresult ofthemethodcall,theArray[topOfStack],willhavethe followingtermrepresentation.

```
select(select(int[]_Map,
select(theArray_Map,this)), select(topOfStack_Map,this))
```

whereselect(m,i)representstheselectionofthevalue storedatindexiinthemapm.theArray_Mapand topOfStack_Maparemapsindexedoverobjectrefer- ences,soselect(theArray_Map,this)correspondsto this.theArrayinthesourcelanguage.int[]_Mapisamap of array references to another map that contains the ele-mentsofthearray,indexedoverintegers.

Astateupdate,e.g.,

```
this.topOfStack=this.topOfStack+1;
```

which method push may perform, is represented using an up-datefunctionupdate(m,i,v),whichrepresentsthemap mafteriswasupdatedatindexiwithnewvaluev.

topOfStack_Map'=update(topOfStack_Map,this,
add(select(topOfStack_Map,this),1))

The interpreter records all conditions that cause the pro-gramtobranch.Inadditiontotheexplicitconditionalbranches performed by the program, Pex's interpreter alsomodels all implicit checks performed by the runtime whichmay induce exceptional behavior—e.g., following a referenceistreatedasanimplicitbranchbasedonwhethertherefer-enceisnull(exceptionalpath)ornot(normalpath).
Basedonthealreadyaccumulatedpathcondition,termsarefurthersimplified.Forexample,ifthepathconditionalreadyestablishedthatx>0,thenx<0reducestofalse.
Pex has a term pretty printer which can translate backreducedandsimplifiedtermsintoreadableC#syntax.

DySyAlgorithm

DySy symbolically monitors the concrete execution of agiven test suite. For the duration of each method call,DySy registers a separate interpreter with Pex's monitor-ing framework.Thus, as soon as there are nested methodcalls, multiple interpreters will be listening to the callbacksof the instrumented code. DySy builds a set of quadruples(method,pathCondition,result,finalState)asitmonitorstheprogram.Each quadruple characterizes an execution pathofamethod.

Step1:Pathconditionandfinalstatediscovery.
When the program initiates a call to a method M(in-cludingtheMainmethodofthetestsuite),DySycreatesa newinterpreterinstancealongwithanewsymbolicstatein-stance. DySy initializes the locations of the symbolic state,including the method's arguments, with logical variables.The interpreter will evolve the symbolic state according toallsubsequentlyexecutedinstructions,includingtransitionsinto and out of other method calls. When the call to M thatspawnedthisinterpreterinstancereturns,DySyrecordsthequadruple(M,pathCondition,result,finalState),andaban-dons the interpreter.The result is the term that M returns.Duringnestedmethodcalls,thestate'slocationsalwaysholdtermsbuiltovertheoriginallogicalvariablesofM,andtheresultofthecallisalsoatermovertheoriginallogicalvariables.Whentheprogramperformsnostateupdatesduringa(nested)call,exceptupdatestothelocalvariablesofnewlycreatedstackframesandupdatestoinstancefieldsofnewlycreatedobjects,DySyconsidersthecallpure.DySyreplacestheresultofapurecallwith atermrepresentingthecall—e.g.,inourearlierexampleitreplacestheexplicit resulttopOfStack>=0withthemethodname Empty.
Also, DySy abstracts all, directly or indirectly, recursivecalls to M in this way, regardless of whether they are purecalls or not.This is a heuristic treatment, which results inrecursive invariants. (This avoids unbounded paths throughrecursivemethods.Theotherinterestingcaseisunboundedpaths through loops, which we discuss separately in Sec-tion4.3.)Forexample,thefactorialfunction

```
intfac(inti){ if(i<=1)
return1;else
returni*fac(i-1);
}
```

iseventuallycharacterizedbyDySyasamethodwithnoprecondition,andthepostcondition
\result==((i<=1)->1)else->i*fac(i-1)tion of path-specific postconditions.A path-specific post-condition is an implication with a path condition on the lefthandsideandaconjunctionofequalities,whereeachequal-ityrelatesalocationtothetermassignedtothatlocationinthefinalstate.(E.g.,recallpostcondition"Empty==true
==>(\result==null)"inourearlierexample.)

AbstractionforLoops

Handling loops is a fundamental challenge for symbolicexecution in general.In our specific context, we discussedearlierhowloopsresultinamethodhavinganinfinitenum-berofpossiblepaths.Sinceoursymbolicexecutionisguidedby a concrete execution, every path we observe has a finitelength, but grows quickly and without bounds. In prac-tice,this means that straightforward symbolic executionproduces enormous path conditions that are overly specificand defeat the purpose of using program conditions as po-tential invariants. We next discuss the heuristics that DySyusesforabstractioninthecaseofloops.

Let us examine the problem with the example of a simplelinearsearchmethod,forwhichwewanttoderiveinvariants.

```
publicintlinSearch(intele,int[]arr){ if(arr==null)
thrownewArgumentException();
for(inti=0;i<arr.Length;i++){ if(ele==arr[i])
returni;
}
return-1;
}
```

Pex does not know the effects of code that it does notmonitor.For example, calls to "native" methods are notmonitored. Here, the user can choose between a conservativeandanoptimistictreatment.

Step2:Classinvariantderivation.

At the end of symbolic execution and before outputtingmethod preconditions and postconditions, DySy first com-putes class invariants, which are used to simplify the meth-ods' invariants.DySy defines the set of "class invariant can-didates" of a class C as the set of conjuncts c of all recordedpath conditions of all methods of C, where c only refers tothethisargumentbutnootherargument.(Forfuturework, onecouldexistentiallyquantifytheotherargumentsymbolsto gain more class invariant candidates.) For each path con-dition and final state of a method of C, DySy then checkswhich candidates are implied by all path conditions in thefinal states of all methods of C.(In fact, in its current im-plementation, DySy does not perform a precise implicationcheckusinganautomatictheoremprover.Instead,itsimplyexecutes the test suite again, and checks the candidates inthe concrete final state of each call to a method of C.) Theimpliedcandidatesarethe"classinvariant"ofC.

Step3:Pre-andpostconditioncomputation.

Finally, DySy further simplifies the method's path con-ditions, assuming the derived class invariant.As a conse-quence of this simplification, some of the quadruples mightcollapsetogether.

Thepreconditionofamethodisthedisjunctionofitspathconditions.Thepostconditionofamethodisthe conjunct Considerrunningtestsforthismethodwithasingleinput array{5,4,3,12,6}andthenumbers0to9ascandi- date element values. Performing symbolic execution alongthepathofconcreteexecutionforeleequalto0willyield a long and too-specific path condition, even after full sim-plification:"arr!=null&&arr.Length==5&&ele!= arr[0]&&ele!=arr[1]&&ele!=arr[2] &&ele!=arr[3]&&ele!=arr[4]".Thepreconditionis not only unwieldy, but also fairly bad for our purposeof inferring invariants because it does not contain generalobservations that may also appear in preconditions derivedfor other test cases: Even after all tests are run, the com-bined preconditions and postconditions will end up havingfewcommonalities(e.g.,"arr!=null&&arr.Length== 5")and5separatecases.(Fourcasescorrespondtothefournumbers from 0 to 9 that appear in the array, and one casecorresponds to the path for numbers that are not found inthe array.) This is exactly the "precise but useless" invariant thatdynamicinvariantinferenceaimstoavoid,asdiscussedin Section 4.1. The problem is that our technique is basedon using program conditions to partition the abstract spaceofpossibilitiesintoafewgeneralbutinterestingcategories.Thesecoarsepartitionscanthenbecombinedt ogetherwithdisjunctions (i.e., case-analysis).When the partitions be-come too fine, there is no abstraction benefit and the in-variantsdescribeexactlythebehaviorofthetestinputsandlittlemore.

Thegeneralapproachtodealingwithsuchover-specificityin program analysis is to force abstraction by forgetting someof the information in the too-precise program paths.Wecan do this by collapsing conditions together (e.g., one con-ditionper-program-point)orbyturningprogramvariablesinto unknowns (i.e., symbolic values) if they get assignedmoretimesthanagiventhreshold.Theidealsolutionwouldbe to produce a concise strongest loop invariant condition.This is generally infeasible, although the rich research resultson automatic techniques for deriving loop invariants (e.g., [4,14,28])areapplicabletotheproblem.DySycurrentlyusesasimpleheuristicthatdoesnotinvolveanattemptf orinvari-ant inference, only local collapsing of conditions.We firstrecognize loop variables by treating specially the commoncode pattern of for loops that introduce explicit variables.Loop variables are then treated as symbolic values, and aloop'sexitconditiondoesnotbecomepartofthepathcon-dition if the loop body is entered at all. Furthermore, sym-bolic conditions inside the body of the loop are collapsedper-program-point with only the latest value remembered:If a certain **if** statement in a loop body evaluates to true inoneiterationandtofalseinanother,thelatestconditionre-places the earlier one in the path condition. This effectivelytreats a loop as if it were an **if**statement with the symbolic conditionsintheloopbodycollapsedper-program-point.

Toillustratetheapproach,our**li**nSearchexampleuses a**for**loopwithloopvariable**i**,declaredexplicitlyinthe for loop's initialization expression.This signals to DySythat variable **i** will likely be assigned multiple values, andwill participate in conditions. DySy then treats **i** as a sym-bolic value and does not keep track of its state updates.By itself, this would be insufficient: executing the loop andthen exiting would produce contradictory symbolic condi-tions.Inourexample,wewouldhave"**i**<**arr.**Length"(forthepartofthepathexecutingtheloopbody)and"!(i
<**arr.**Length**)**"(when thesamepathlaterexitstheloop body).Since both conditions are conjoined (logical-and)togetherinthesamepathcondition,thepathconditionbe-
comesjust**fal**se,whichisclearlyerroneous.Inourheuris- tic, we ignore a loop's exit condition (unless the loop is notenteredatall).Inourexample,thepreconditionbecomes:

```
arr!=null&&
($i<arr.Length&&!(ele==arr[$i])&&$i>=0||
$i<arr.Length&&ele==arr[$i]&&$i>=0)
```

This demonstrates a few interesting points. First, sym-bolicvariable$**i**istreatedasapseudo-input.Essentially,in theabovelogicformula,$**i**isexistentiallyquantified:there existssome$**i**withtheseproperties.Second,nocondition "$**i**>=**arr.**Length"isoutput.Everytestcaseentersthe loop at least once.Third, we can see how path conditionsare collapsed per-program-point inside the loop body: Exe-cutionsthatdofindthesearchedelementproduceboththecondition"**!(**e**l**e==arr**[**$**i])**"(foriterationsoverot herel- ements)andthecomplement,"e**l**e==arr**[**$**i]**"(fortheit- eration that finally finds the element). Yet the former arereplaced when the latter take place. Finally, this precondi-tion contains redundancy. It covers the complementary casesof"e**l**e==arr**[**$**i]**"and"**!(**e**l**e==arr**[**$**i])**",whichcan be simplified away. It is fortunate, however, that the DySysimplifier misses this opportunity because this helps illus-trate how the different cases arise.The separation of thecases does not matter for the method's precondition, butdoesmatterforthepostcondition.There,weobtain(slightlysimplified):

```
!(ele==arr[$i])==>\result==-1|| ele==arr[$i]==>\result==$i
```

This is a quite informative postcondition for the method,andcapturesitsessenceaccurately.

In the immediate future, we plan to refine our heuristicintoaslightlymoresophisticatedversionthathandlesmorethan for loops and also produces useful conditions for exit-ing the loop body. Specifically, we intend to recognize loopvariables by observing program variables that get assignedduring the loop's iterations.Each of these program vari-ableswillgiverisetotwosymbolicvariables—e.g.,$**i**0and
$**i**1.Thefirstsymbolicvariablewillrepresentthevalues of the program variable in the body of the loop, while thesecondwillrepresentthevalueonexitfromtheloop.Thesevariables are again existentially quantified:our conditionswillonlyreflectthatthereissomevalue$**i**0(resp.$**i**1)for which the symbolic conditions derived while executing theloopbody(resp.whenexitingtheloop)hold.

V. EVALUATION

We next discuss and evaluate DySy in comparison withtheDaikondynamicinvariantinferencetool.

Discussion

At a high level, our discussion of the dynamic symbolicapproach should give the reader a qualitative idea of thecomparative advantages of DySy.Every dynamic invariantinference process captures, to some extent, the peculiaritiesof the test suite used. Nevertheless, our symbolic approachhas a smaller risk of being overly specific, since the condi-tions themselves are induced by the program text and refinedthrough symbolic execution.Instead, an approach observ-ing arbitrary, pre-set conditions at method boundaries isboundtobe"fooled"muchmoreeasily.Forthe**li**nSear**c**hmethod of the previous section, with the test cases describedearlier(allnumbers0..9searched inthearray**{**5**,**4**,**3**,** 12**,**6**}**)Daikoninfersalmostnousefulinvariants,buta large number of spurious ones.Example "accidental" in-variantsinclude"s**i**ze**(**arr**[])i**narr**[]**","s**i**ze**(**arr**[])**-1 **i**narr**[]**","arr**[i]!**=**i**",etc.(Theserelatetheindexor size of an array to its contents!) Certainly these spurious in-variantscanbedisqualifiedwithamoreextensivetestsuitethat uses more arrays as inputs.Nevertheless, test suitesencountered in practice tend to exercise as many differentcases in the program logic as possible, but without muchvariety of data. **It** is, thus, very plausible for a programmertounit-testmethod**linSearch**withonlyasinglearray,yet with multiple input search values. A larger test input (e.g.,asystem-test)thatexercises**linSearch**mayalsofailtoin- validatesome ofthefalseinvariants—forinstance,“**arr[i]** !=i”islikelytoholdformanyarrays.

Ontheotherhand, a possible threat for DySy comparedtoDaikonisthatinterestingconditionsarenotreflectedinthe program text. For instance, an interesting concept, suchas ordering, may be implicit or hard to infer from programconditions,yetmaybe inferable by Daikon. Nevertheless,wehavenotfoundthistooftenbethecase.Webelievethatthisisnotsurprising:findinganimplicitinter estingconcept by unguided search over a space of pre-set invarianttemplatesisquiteunlikely.

ACaseStudy

We evaluate DySy by replicating a case study analyzed inthe Daikon literature.The StackAr class was an exampleprogramoriginallybyWeiss[34],whichisincludedastheTable 1:How many of the “ideal” invariants Daikonand DySy infer for StackAr methods and construc-tors exercised by the test suite.(Higher is better.)“Goalinv” isthenumberofourmanuallydeter-mined ideal invariants. “Recognized inv” is the num-ber of these ideal invariants inferred by Daikon andDySy.Foreachtool,wereportastrict and a re-laxedcount(thenumbersinparentheses)becauseof object equality invariants.If the tool does notestablishthedeepequalityof objects (or full pu-rity of a method), but does establish some shallowequality condition (e.g., reference equality, or valueequalityup to level-1) then the “relaxed” numberin parentheses counts this as matching the expectedinvariant.

	Goal inv	Recognizedinv	
		Daikon	DySy
Invariant	5	5	4
Constructor	3	3	2
push	4	2(4)	2(4)
top	3	1(3)	2(3)
topAndPop	4	2(4)	2(4)
isEmpty	3	2(3)	3
isFull	3	2(3)	3
makeEmpty	2	2	2
Total	27	19 (27)	20 (25)

main example in the Daikon distribution. StackAr is a stackalgebraicdatatypeimplementedusinganarray.Ernstetal. [10] examineStackAr in detail and discuss Daikon’s abilityto infer StackAr’s invariants. In order to perform a compar-ison with DySy, we rewrote StackAr in C# (also with thehelp of the Java Conversion Assistant in the Visual StudioIDE).

WeranDaikononthetestsuitessuppliedforStackArby the Daikon authors. To do a comparison of Daikon andDySy, we needed an “ideal” reference set of invariants forStackAr.Before beginning our experimentation, a humanuser hand-produced our reference invariants.Inspection re-veals that this set of invariants is comprehensive and min-imal (in informal terms).It captures the behavior of eachmethod in terms expected by human users.(We discussspecificexampleslater.)

Running DySy on the test suite takes 28 seconds, com-paredto9secondsforDaikon(2.2secondsmonitoringand 6.7 seconds inference reported) on a 2 GHz AMD Athlon 64X2 dual core 3800+ with 4 GB of RAM. Generally, our sym-bolicexecutionaddssignificantoverhead,which,however,isstrictly lower than that of concolic execution [17, 30]. Thisisfastenoughforrealuseonspecificprogramunits.Gener-ally, we believe that the matter of invariant quality is muchmore significant than that of runtime overheads, as there issubstantialpotentialforoptimizationsinthefuture.

The results of the DySy and Daikon inference are summa-rized in Tables 1 and 2. Table 1 shows the number of idealinvariants that were actually detected by Daikon and DySy.As can be seen, the test suite is quite thorough and bothtools detect the vast majority of the target invariants.Aninteresting issue concerns object equality (and method pu-rity), which is often part of the ideal invariant. The meaningofequalityinourhuman-producedinvariantsisdeepequal Table 2:Metrics on all reported invariants (lower isbetter), compared to ideal reference set. “Goal inv”is the number of ideal invariants. “Daikon inv” is thenumber of invariants reported for Daikon. “Uniquesubexpr” are the unique subexpressions produced byDaikon and DySy to present their invariants to theuser.The total expression count is relative to theentire class so here it is less than the sum.(Somesubexpressionsarecommonacrossmethods.)

	Goal inv	Daikon inv	Uniquesubexpr Goal	Daikon	DySy
Invariant	5	8	26	26	16
Constructor	3	7	17	24	17
push	4	21	28	69	43
top	3	22	14	81	25
topAndPop	4	41	21	145	50
isEmpty	3	13	9	53	9
isFull	3	11	13	45	13
makeEmpty	2	15	5	47	22
Total	27	138	89	316	133

ity.This is not always inferred by the tools, but referenceequality is more often inferred (which cannot preclude thatthemembersofanobjectchanged).Thetableoffersastrictandarelaxedcount.Thestrictcountconsidersthe invariantfoundevenifonlyreferenceequalityisestablished.

Although both tools infer the required invariants for thistest suite, the benefit of DySy is demonstrated in its avoidingirrelevantinvariants.Table 2 shows how many total invari-ants Daikon inferred (third column).To detect the 27 idealinvariants,Daikonproduceda total of 138 invariants. WedonotgiveasimilarcountforDySy,sinceitsoutputcon-sistsofcondensedexpressions(e.g.,if-likeconstructsjointogetherinvariantsintoasingletop-levelone)whichmakethe comparison uneven. Instead, we list a more reliable met-ric for both tools' output:The last three columns of Table 2present the number of unique subexpressions in the ideal andinferred invariants.We parse the output for both tools andcountthenumberofuniquesubtrees in the abstract syn-tax tree: if a subtree/subexpression occurs on two branches,it is counted only once.Thus, surface verbosity is ignored:what is measured is the number of truly distinct clauses thateach tool infers. (Measuring the full size of the output wouldbias the numbers in favor of DySy, as its output is simplifiedsymbolically with common subexpressions factored out.)Ascan be seen,DySy infers many fewer total invariants thanDaikon—aboutathirdofthetotalsize.Indeed,theDySyoutputisveryclosetothereferencesetofinvariantsforStackAr.(There areminorinaccuraciesinourcountingofunique subexpressions, due to manual conversions betweenthetools'differingoutputsyntax.Whenindoubt wefa-vored Daikon, by underapproximating the number of uniquesubexpressionsDaikonreports.)

To see an example of the differences, consider methodtopAndPop,whichremovesandreturnsthestack'smostre-cently inserted element, or null if the stack is empty.Thetwo important postconditions for this method concern itseffectontopOfStackanditsreturnvalue.Wehave:

```
\result==((Empty->null)
else->theArray[\old(topOfStack)])
```

and

```
topOfStack==((Empty->\old(topOfStack))
else->\old(topOfStack)-1)
```

BothDySyandDaikoninferthesepostconditions.Onemoreprecondition states that all stack contents below the topelement remain unchanged by the method's execution. Bothtoolsinferthatpreconditionbutonlyundershallowequality.At the same time, to infer these correct invariants, Daikoninfers a total of 41 invariants for this method. Many rangefrom erroneous to irrelevant from the perspective of a humanuser.Oneinvariantis:

```
\old(this.topOfStack)>=0)==>(this.theArray.getClass()!=\result.getClass())
```

The invariant relates the type of the array with the types ofelementsitholds.AnotherDaikoninvariantis:

```
\old(this.topOfStack)>=0)==>((\old(this.topOfStack)>>
stackar.StackAr.DEFAULT_CAPACITY==0))
```

ThisrelatestopOfStackwiththestack'sdefaultcapacity usingabit-shiftoperator!(Wehand-translatedtheaboveinvariantstoJML.TheywereoriginallyonlyoutputinDaikon'sdedicatedinvariantlanguagebecausetheyarenotallowedinJML—e.g.,becauseofreferencestoprivatefields.)EliminatingtheextraneousDaikoninvariantswouldbepossiblewithalargertestsuitethatwouldexercisetheStackArfunctionalityundermanyconditions.Neverthelessthefundamentaltensionremains:IfDaikonistoinferalltrueinvariants,itneedstoexploreagreatnumberofinvarianttemplates,whichincreasestheprobabilityofacci-dentalinvariants.Incontrast,DySyobtainsitscandidateinvariantsdirectlyfromtheprogram'sconditionsand as-signments,thereforetheinvariantsitinfersareverylikely relevant.

VI. RELATED WORK

We have already discussed the most directly related workthroughout the paper.We next present some less directlyrelatedworkthatstillexertedinfluencesonourtechnique,yeteitherusesexclusivelystaticmethodsforinvariantin-ference, or infers program specifications purely dynamically,byexaminingpre-definedpatterns.

For reverse engineering, Gannod and Cheng [15] proposedto infer detailed specifications statically by computing thestrongestpostconditions.Nevertheless,pre/postconditionsobtainedfromanalyzingtheimplementationare usuallytoodetailed to understand and too specific to support programevolution.Gannod and Cheng [16]addressedthisdefi-ciencybygeneralizingtheinferredspecification,forinstanceby deleting conjuncts, or adding disjuncts or implications.Theirapproachrequiresloopboundsandinvariants,bothofwhichmustbeaddedmanually.

Flanagan and Leino [12] propose a lightweight verification-based tool, named Houdini, to statically infer ESC/Java an-notations from unannotated Java programs. Based on pre-set property patterns, Houdini conjectures a large numberofpossibleannotationsandthenusesESC/Javatoverifyor refute each of them. The ability of this approach is lim-ited by the patterns used. In fact, only simple patterns arefeasible, otherwise too many candidate annotations will begenerated, and, consequently, it will take a long time forESC/Javatoverifycomplicatedproperties.Taghdiri[31]usesacounterexample-guidedrefinementprocesstoinferover-approximatespecificationsforproce-durescalledinthefunction being verified. In contrast toourapproach,Taghdiriaimsto approximate the behaviorsfor the procedures within the caller's context instead of in-ferringspecificationsoftheprocedure.

HenkelandDiwan[19]havebuiltatooltodynamicallydiscover algebraic specifications for interfaces of Java classes.Their specifications relate sequences of method invocations.The tool generates many terms as test cases from the classsignature.The results of these tests are generalized to alge-braicspecifications.

Much of the work on specification mining is targeted atinferring API protocols dynamically.Whaley et al.[35]describe a system to extract component interfaces as finitestate machines from execution traces. Other approaches usedata mining techniques. For instance Ammons et al.[1]use a learner to infer nondeterministic state machines fromtraces; similarly, Yang and Evans [37] built Terracotta, atool to generate regular patterns of method invocations fromobserved runs of the program. Li and Zhou [24] apply datamining in the source code to infer programming rules, i.e.,usageofrelatedmethodsandvariables,andthendetectpo-tentialbugs bylocatingtheviolationoftheserules.

VII. CONCLUSIONS

Theexcitementthatfollowedtheoriginalintroductionof dynamic invariant detection in the Software Engineeringworld seems to have been followed by a degree of skepticism.Dynamic invariant inference tools require huge and thoroughregression test suites, and infer properties that are occasion-ally interesting but often too simplistic. Additionally, havingenoughteststoeliminatefalseinvariantsdoesnotprecludeextraneous invariants, which are disappointing to a humanuser.In this paper we presented an approach that holdspromise for the future of dynamic invariant inference: us-ing symbolic execution, simultaneously with concrete testexecution in order to obtain conditions for invariants.Webelievethatthistechniquerepresentsthefutureofdynamicinvariant inference. It combines the advantages of invariantinference through static analysis, with the immediate prac-ticalityofobservinginvariantsbyexecutingtestswrittenby programmers who exercise valid scenarios. Furthermore,thetechniqueisstrictlyanincrementoverpriorapproaches,as it adds an orthogonal dimension: It is certainly possibletocombinedynamicsymbolicexecutionwithobservationofproperties from pre-defined templates, as in other dynamicinvariant detectors.The symbolic simplification approachcan then apply to both symbolically inferred invariants andinvariants instantiated from templates. A complete evalua-tion of such a hybrid is part of future work. We hope thatthiswillbejustoneofmanyavenuesthatthepresentpaperwillopenfordynamicinvariantdetection.

REFERENCES

[1] G.Ammons,R.Bodik,andJ.R.Larus.Miningspecifications.InProc.29thACMSIGPLAN-SIGACTSymposiumonPrinciplesofProgrammingLanguages(POPL),pages4–16.ACM,Jan.2002.M.Boshernitsan,R.Doong,andA.Savoia.FromDaikontoAgitator:lessons and challenges in

building a commercialtoolfordevelopertesting.InProc.ACM/SIGSOFTInternationalSymposiumonSoftwareTestingandAnalysis(ISSTA),pages169–180.ACM,July2006.

[2] K. S.Brace,R.L.Rudell,andR.E. Bryant. Efficientimplementation of abddpackage.In Proc.27thACM/IEEEDesignAutomationConference(DAC),pages40–45.ACM,June1990.

[3] M.Colón,S.Sankaranarayanan,andH.Sipma.Linear invariantgenerationusingnon-linearconstraintsolving.InProc.15thInternationalConferenceonComputer-AidedVerification(CAV),pages420–432.Springer,July2003.

[4] C.CsallnerandY.Smaragdakis.DSD-Crasher:Ahybridanalysistoolforbugfinding.InProc.ACMSIGSOFTInternational Symposium on Software Testing andAnalysis(ISSTA),pages245–254.ACM,July2006.

[5] C.CsallnerandY.Smaragdakis.Dynamicallydiscoveringlikely interface invariants. In Proc. 28th InternationalConference on Software Engineering (ICSE), EmergingResults,pages861–864.ACM,May2006.

[6] S.ElbaumandM.Diep.Profilingdeployedsoftware:Assessingstrategiesandtestingopportunities.IEEETransactions on Software Engineering, 31(4):312–327, Apr.2005.

[7] M.D.Ernst,J.Cockrell,W.G.Griswold,andD.Notkin.Dynamically discovering likely program invariants tosupport program evolution. In Proc. 21st International ConferenceonSoftwareEngineering(ICSE),pages213–224.IEEE,May1999.

[8] M.D.Ernst,J.Cockrell,W.G.Griswold,andD.Notkin.Dynamically discovering likely program invariants tosupportprogramevolution.IEEETransactionsonSoftware Engineering,27(2):99–123,Feb.2001.

[9] M.D.Ernst,J.H.Perkins,P.J.Guo,S.McCamant,C.Pacheco,M.S.Tschantz,andC.Xiao.TheDaikonsystemfordynamicdetectionoflikelyinvariants.ScienceofComputerProgramming,69(1–3):35–45,Dec.2007.

[10] D. EvansandM.Peck.Inculcatinginvariantsinintroductory courses. In Proc. 28th InternationalConferenceonSoftwareEngineering(ICSE),pages673–678.ACM,May2006.

[11] C.FlanaganandK.R.M.Leino.Houdini,anannotationassistantforESC/Java.InProc.InternationalSymposiumofFormal MethodsEurope(FME), pages 500–517.Springer,Mar.2001.

[12] C.Flanagan,K.R.M.Leino,M.Lillibridge,G.Nelson,J. B. Saxe, and R. Stata. Extended static checking forJava. In Proc.ACMSIGPLANConferenceonProgramming Language Design and Implementation(PLDI),pages234–245.ACM,June2002.

[13] C.FlanaganandS.Qadeer.Predicateabstractionforsoftwareverification.InProc.29thACMSIGPLAN-SIGACTSymposiumonPrinciplesofProgrammingLanguages(POPL),pages191–202.ACM,Jan.2002.

[14] G.C.GannodandB.H.C.Cheng.Strongestpostconditionsemantics as the formal basis for reverse engineering. InProc. Second Working Conference on Reverse Engineering(WCRE),pages188–197.IEEE,July1995.

[15] G.C.GannodandB.H.C.Cheng.Aspecificationmatchingbasedapproachtoreverseengineering.InProc.International Conference on Software Engineering, pages389–398.ACM,May1999.

[16] P.Godefroid,N.Klarlund,andK.Sen.DART:Directedautomatedrandomtesting.InProc.ACMSIGPLANConference on Programming Language Design andImplementation(PLDI),pages213–223.ACM,June2005.

[17] S. Hangal and M. S. Lam. Tracking down software bugsusing automatic anomalydetection.InProc.24thInternational Conference on Software Engineering (ICSE),pages291–301.ACM,May2002.J.HenkelandA.Diwan.DiscoveringalgebraicspecificationsfromJavaclasses.InProc.17thEuropeanConferenceonObject-OrientedProgramming(ECOOP),pages431–456.Springer,July2003.

[18] J.C.King.Symbolicexecutionandprogramtesting.Commun.ACM,19(7):385–394, 1976.

[19] A.K.Kolawa.MethodandsystemforgeneratingacomputerprogramtestsuiteusingdynamicsymbolicexecutionofJavaprograms.UnitedStatesPatent5784553,July1998.

[20] N.KuzminaandR.Gamboa.Extendingdynamicconstraintdetectionwithpolymorphicanalysis.InProc.5thInternational WorkshoponDynamicAnalysis(WODA),May2007.

[21] G.T.Leavens,A.L.Baker,andC.Ruby.Preliminarydesign of JML: A behavioral interface specificationlanguageforJava.TechnicalReportTR98-06y,DepartmentofComputerScience,IowaStateUniversity,June1998.

[22] Z.LiandY. Zhou.PR-miner:automaticallyextractingimplicitprogrammingrulesanddetectingviolationsinlarge software code. In Proc. 13th InternationalSymposium on Foundations of Software Engineering(FSE),pages306–315.ACM,Sept.2005.

[23] B. Liblit,M.Naik,A.X.Zheng,A. Aiken,andM.I.Jordan.Scalable statistical bugisolation.In Proc.ACMSIGPLAN Conference on Programming Language DesignandImplementation(PLDI), pages15–26. ACM,June2005.

[24] F.Logozzo.ModularStaticAnalysisofObject-OrientedLanguages.PhDthesis,EcolePolytechnique,June2004.

[25] J.W.NimmerandM.D.Ernst.Invariantinferenceforstatic checking: An empirical evaluation. In Proc. 10thInternationalSymposiumonFoundationsofSoftwareEngineering(FSE),pages11–20.ACM,Nov.2002.

[26] C.S.PasareanuandW.Visser.VerificationofJavaprogramsusingsymbolicexecutionandinvariantgeneration.InProc.11thInternationalSPINWorkshop,pages164–181.Springer,Apr.2004.

[27] J.H.PerkinsandM.D.Ernst.Efficientincrementalalgorithms for dynamic detection of likely invariants. InProc.12thInternationalSymposiumontheFoundationsofSoftwareEngineering (FSE),pages23–32,Nov.2004.

[28] K.Sen,D.Marinov,andG.Agha.CUTE:aconcolicunittestingengineforC.InProc.13thInternationalSymposium on Foundations of Software Engineering(FSE),pages263–272.ACM,Sept.2005.

[29] M.Taghdiri.Inferringspecificationstodetecterrorsincode. In Proc.19thIEEE InternationalConferenceonAutomated Software Engineering (ASE), pages 144–153,Sept.2004.

[30] N. Tillmann, F. Chen, and W. Schulte. Discovering likelymethodspecifications.In Proc.8thInternationalConference on Formal Engineering Methods (ICFEM'06),LNCS.Springer-Verlag,2006.

[31] N.TillmannandJ.deHalleux.Pex–whiteboxtestgenerationfor.NET.InProc.SecondInternationalConferenceonTestsandProofs(TAP).Springer,Apr. 2008.Toappear.

[32] M.A.Weiss.DataStructuresandAlgorithmAnalysisinJava.AddisonWesleyLongman,1999.

[33] J.Whaley,M.C.Martin,andM.S.Lam.Automaticextraction of object-oriented component interfaces. InProc.InternationalSymposiumonSoftwareTestingandAnalysis(ISSTA),pages218–228.ACM,July2002.

[34] T.XieandD.Notkin. Tool-assisted unit test

generationandselectionbasedonoperationalabstractions.AutomatedSoftwareEngineering,13(3):345–371,July2006.
[35] J. Yang and D. Evans. Dynamically inferring temporalproperties.InProc.5thACMSIGPLAN-SIGSOFTWorkshoponProgramAnalysisforSoftwareToolsandEngineering(PASTE), pages 23–28. ACM, June2004.