# Incremental State-Space Exploration for Programs with Dynamically Allocated Data

Steven Lauterburg Ahmed Sobeih Darko Marinov Mahesh Viswanathan
Department of Computer Science, University of Illinois, Urbana, IL 61801, USA
{slauter2, sobeih, marinov, vmahesh}@cs.uiuc.edu

## ABSTRACT

We present a novel technique that speeds up state-space exploration (SSE) for evolving programs with dynamically allocated data. SSE is the essence of explicit-state model checking and an increasingly popular method for automating test generation. Traditional, non-incremental SSE takes *one version* of a program and systematically explores the states reachable during the program's executions to find property violations. Incremental SSE considers *several versions* that arise during program evolution: reusing the results of SSE for one version can speed up SSE for the next version, since state spaces of consecutive program versions can have significant similarities. We have implemented our technique in two model checkers: Java PathFinder and the J-Sim state-space explorer. The experimental results on 24 program evolutions and exploration changes show that for *non-initial* runs our technique speeds up SSE in 22 cases from 6.43% to 68.62% (with median of 42.29%) and slows down SSE in only two cases for -4.71% and -4.81%.

**Categories and Subject Descriptors:** D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.4 [*Software Engineering*]: Program Verification

**General Terms:** Performance, Verification

**Keywords:** State-space exploration, incremental computation, model checking, Java PathFinder, JPF, J-Sim

## 1. INTRODUCTION

The widespread use of software in devices and safety critical applications makes software reliability as crucial as software functionality. Testing and model checking are important approaches that help the development of reliable software by automatically identifying potential errors. The main engine of a model checker is the *state-space explorer (SSE)* [5], which operates on the program semantics, modeled as a transition system with vertices being states of the program and edges or transitions being the steps that the program takes. The SSE is a simple graph search algorithm running on the transition system; it starts from the initial state and searches for reachable states that violate the correctness requirements, while pruning the search when a previously visited state is encountered.

Traditional SSE takes a program and systematically explores its state space, so a lot of research has focused on speeding up SSE for *one version* of a given program. However, software evolves over time, due to bug fixes, code optimizations, or functionality extensions. In the context of testing, the usual approach to address evolving software is regression testing that checks whether a newer version of software still passes the tests that a previous version passed. Researchers developed numerous approaches to improve basic regression testing, e.g., (safe) test selection [15, 16, 50], test prioritization [14, 23, 35], impact analysis [2, 22, 33], or building unit tests from system tests [13, 29, 36].

In the context of model checking, the importance of developing algorithms for evolving software was recognized by Sistla more than 10 years ago [37]. Starting from the seminal work of Sokolsky and Smolka [41], there have been some recent techniques proposed for *incremental model checking* [6, 18, 24]. However, the existing incremental techniques mostly focus on control-intensive properties and programs that manipulate dynamically allocated data are not handled well. Our focus is on checking data-intensive properties, especially for object-oriented programs. In this domain, SSE has three characteristics: (1) program states can be large (and checking data properties typically requires the entire concrete state and not an abstraction); (2) operations, i.e., methods, are enabled in almost every state; and (3) execution of operations can take a relatively large time.

We present a technique for *incremental SSE* that can speed up checking of evolving (object-oriented) programs and data-intensive properties. Specifically, this paper makes the following contributions.

**Technique:** We present *ISSE*, a technique for incremental SSE that targets explicit-state model checkers for real code, e.g., BogorVM [34], CMC [27], JPF [47], SpecExplorer [46], or Zing [1]. In these tools, execution of transitions can be expensive [8, 9]: the tool can take a lot of time to execute the code that transitions the program from one state to another (and additionally to check the safety property in the new state). Our specific goal is thus to reduce execution time. We have observed that for many program changes, large portions of the program's state space remain unchanged after the code change. ISSE exploits this observation to speed up SSE by saving the state-space graph from one exploration and examining this graph to determine


*ICSE'08,* May 10–18, 2008, Leipzig, Germany.

whether a certain execution is needed during the next exploration (after a program change). A key design decision of ISSE is to use *re-execution* to restore states; although execution may be slow, it is still faster than saving to disk and loading from disk entire states when the states are large and the exploration needs to restore almost all of them. We analytically model ISSE and analyze the conditions—cost of execution vs. cost of lookup and the frequency of "hits" during lookup—under which ISSE does or does not speed up traditional, non-incremental SSE. Explorations of programs with dynamically allocated data often satisfy the conditions under which ISSE provides speed up, but ISSE can in principle be used for other kinds of explorations.

**Implementation:** We implemented ISSE in two tools: Java PathFinder (JPF) [25, 47], which is a general-purpose model checker for Java programs, and the J-Sim state-space explorer [40], which is specialized for the J-Sim simulation models of network protocols [45]. Our goal is to evaluate ISSE in tools with different design decisions and for different kinds of evolving programs.

**Evaluation:** We evaluated our implementations using 11 subject programs for JPF and 3 network protocols for J-Sim. Our experiments consider various code changes due to program evolution (e.g., fixing bugs or adding functionality) and various exploration changes for the same code (e.g., increasing depth of exploration or increasing number of input values). While it is easy to change exploration for a given code, we had to collect a set of program evolutions to evaluate ISSE for code changes. The existing program evolutions used in research on (regression) testing [11] are not suitable as they have only individual tests that do not involve state-space exploration. We performed experiments on 24 code changes or exploration changes. The experimental results show that for *non-initial* runs, ISSE speeds up non-incremental SSE in 22 cases, from 6.43% to 68.62% (with median of 42.29%). Additionally, ISSE does not have a large overhead, and for the 2 cases where it slows down non-incremental SSE, the slowdowns are -4.71% and -4.81%.

## 2. EXAMPLE

We next show an example usage scenario where our incremental state-space exploration (ISSE) provides a speedup over a traditional, non-incremental state-space exploration (SSE). Our example evolves code that implements a simple file system based on Daisy [32], originally proposed by Qadeer as a sample subject for model checking and testing research. This code was analyzed with various tools such as BogorVM [34], Java PathExplorer [17], JPF [47], and SpecExplorer [46]. We use a simplified version of the code provided by Darga and Boyapati [10]. This implementation, called `filesystem`, has 4 classes with about 170 lines of C-like Java code and uses simple data structures to implement key entities of a file system.

The previous analyses of this subject considered one given version of the code [8, 10]. Our example considers a scenario involving (two) consecutive versions of the code, but before describing this scenario, we present how to perform state-space exploration for one version. Figure 1 shows declarations for four methods used in our exploration of `filesystem`. These methods create and remove directories and files. A traditional test for this code would have a sequence of method calls, e.g., first create an empty file system, then create or remove some directories or files, and finally check

```
class FileSystem {
   int mkdir(FileHandle dir, byte[] name, FileHandle fh) { ... }
   int rmdir(FileHandle dir, byte[] name) { ... }
   int creat(FileHandle dir, byte[] name, FileHandle fh) { ... }
   int unlink(FileHandle dir, byte[] name) { ... }
   ... // other methods, e.g., read/write from/to files
}
```

**Figure 1: Simple Interface for a Part of File System**

```
// L bounds length of method sequences
// M bounds values of method parameters
void main(int L, int M) {
   FileSystem fs = new FileSystem(); // empty file system
   FileHandle[] handles = ... // create M handles
   byte[][] names = ... // create M names
   for (int i = 0; i < L; i++) {
      int m = Verify.getInt(0, MAX_METHOD_ID); // choose method
      FileHandle dir = handles[Verify.getInt(0, M)];
      byte[] name = names[Verify.getInt(0, M)];
      FileHandle fh = handles[Verify.getInt(0, M)];
      int r; // result
      switch (m) {
      case 0: r = mkdir(dir, name, fh); break;
      case 1: r = rmdir(dir, name); break;
      case 2: r = creat(dir, name, fh); break;
      case 3: r = unlink(dir, name); break;
      ... // other methods, e.g., read/write from/to files
      }
      ... // check correctness for state "fs" and result "r"
   }
}
```

**Figure 2: Exploration of Method Sequences in JPF**

that these directories or files indeed do or do not exist. Instead of testing individual sequences, we use a state-space exploration tool to check all sequences within given bounds.

Figure 2 shows driver code suitable for exploring method sequences in JPF [21, 47, 48]. The inputs bound the length of sequences, `L`, and the values for method parameters, `M`, i.e., the number of various file handles and names. (Most of our experiments use only one bound, with `L=M`.) The JPF library method `Verify.getInt(int lo, int hi)` creates a non-deterministic choice and instructs JPF to explore all possible values between `lo` and `hi`. Given such a driver, JPF explores all executions up to the given bounds and reports those executions that violate encoded correctness properties. An important point is that JPF compares states that result from different sequences: if two sequences lead to the same state of the file system, JPF prunes one of those sequences from the exploration.

We obtained several versions of `filesystem` code as follows. We first deleted the method body from `mkdir` (leaving only its specification in the comments) and asked six undergraduate and graduate students to implement the method. This method is non-trivial as it needs to allocate blocks and inodes for a new directory and to perform several checks (the disk should not be full, there should be available inodes, the directory name should not already exist, etc.). We then used our driver to check the code and reported the bugs to the authors so that they can correct their code. We next checked the new versions of the code until either the authors provided a correct version or we used our existing correct version. A representative example consists of two versions: the first version did not check whether a directory name already exists in the given directory (so it could incorrectly create duplicate names), and the second version corrected this error about duplicate names (by adding a check).

We compared the performance of basic JPF using traditional, non-incremental SSE and JPF with our technique

ISSE for incremental state-space exploration. For the example with duplicate names, JPF took the following times (with the bounds L=M=5) for the initial version (with the error): 1728.21s without ISSE and 1752.10s with ISSE. In this run, ISSE was slower (but less than 2%) since it prepared the additional information to speedup the next exploration. Indeed, JPF took the following times (with the same bound of 5) for the second version (without the error): 1047.94s without ISSE and 447.31s with ISSE. In this case, ISSE was more than twice as fast, significantly paying off the small initial slowdown.

# 3. TECHNIQUE

The main idea behind ISSE is to reduce the time necessary for state-space exploration by avoiding the execution of selected transitions (and related computations) that are not necessary for a given exploration. ISSE identifies unnecessary transitions using the information captured during prior exploration, along with a list of code changes made since the prior exploration.

We first present a traditional, non-incremental SSE algorithm to introduce the notation. We then present our modifications that transform the algorithm into incremental state-space exploration, ISSE. We next discuss the correctness of ISSE and finally present a brief analytical discussion of ISSE's performance, focusing on the conditions necessary for ISSE to outperform non-incremental SSE.

## 3.1 Non-Incremental State-Space Exploration

Figure 3 shows the pseudo-code for a non-incremental SSE algorithm [5]. The two major data structures it uses are $ToExplore$ (which stores the states from which no transition has yet been explored) and $Visited$ (which stores the states, more precisely hash codes of the states, that have already been visited). Figure 3 presents a stateful search that avoids visiting a state $s_1$ if another state $s_2$, with the same hash code, has already been visited before.

Initially, $ToExplore$ contains only the initial state (line 2), and $Visited$ contains the hash code of this state (line 3). As long as $ToExplore$ is not empty (line 4), $SSE$ chooses a state from $ToExplore$ and assigns it to the current state $s$ (line 5). For each state $s$ being explored, $SSE$ determines the transitions that are enabled for that state by invoking `genEnabledTransitions` (line 6).

For each transition $t$, $SSE$ generates the successor state ($s'$) by calling the `genNextState` function (line 8) to execute the transition. $SSE$ then *post-processes* $s'$, computing its hash code (line 10) and checking whether it violates any specified correctness property (line 11). If $s'$ violates a property, the exploration prints a message and optionally terminates (line 12). As long as the depth of $s'$ is less than the specified maximum depth `MAX_DEPTH`, and $s'$ has not been visited before (i.e., its hash code is not in $Visited$), $SSE$ adds $s'$ to $ToExplore$ (line 16) for later exploration and adds its hash code to $Visited$ (line 15) to mark it has been visited.

Based on checks for the state $s'$ (lines 13 and 14), we distinguish three disjoint types of transitions: (1) *tree transitions* generate a state $s'$ that has depth less than `MAX_DEPTH` and that has not been already visited; (2) *non-tree transitions* generate a state $s'$ that has depth less than `MAX_DEPTH` but that has been already visited; (3) *deepest transitions* generate a state $s'$ that has depth equal to `MAX_DEPTH` (note that the depth cannot ever be greater than `MAX_DEPTH`).

```
0: procedure SSE()
1:    s_init.depth = 0
2:    ToExplore = {s_init}
3:    Visited = {computeHashCode(s_init)}
4:    while (|ToExplore| > 0) do
5:       s = choose state from ToExplore
6:       transitions = genEnabledTransitions(s)
7:       for each transition t in transitions do
8:          s' = genNextState(s, t)
9:          s'.depth = s.depth + 1
10:         s'.hashcode = computeHashCode(s')
11:         valid = checkProperty(s')
12:         if (not valid) then print(ERROR)
13:         if (s'.depth < MAX_DEPTH &&
14:            s'.hashcode ∉ Visited) then
15:            Visited = Visited ∪ {s'.hashcode}
16:            ToExplore = ToExplore ∪ {s'}
```

**Figure 3: Non-Incremental Exploration**

Depending on the data structure used for $ToExplore$, $SSE$ can use different search strategies, including breadth-first, depth-first, and various heuristic-based search strategies. Our technique for incremental state-space exploration works for all those search strategies.

## 3.2 Incremental State-Space Exploration

As stated above, ISSE aims to avoid the execution of unnecessary transitions (and related computations) based on information from a prior exploration and a specification of what has changed in the code being explored. We have identified three primary sources for reduction: (1) execution for unmodified non-tree transitions, (2) execution for unmodified deepest transitions, and (3) post-processing for unmodified tree transitions.

Figure 4 shows the pseudo-code of $ISSE$, our procedure for incremental state-space exploration. Although similar to $SSE$, $ISSE$ has some key differences that allow select executions and/or their post-processing steps to be avoided.

$ISSE$ first loads a state-space graph from an input file and stores it in the $IN$ data structure (line 1). For each transition $t$, $ISSE$ makes use of two decision variables, *execute* and *postProcess*, to determine what needs to be performed for the transition. If *execute* is `false`, $ISSE$ does not execute the transition nor its post-processing. If *execute* is `true` (line 20), $ISSE$ executes the transition, which generates a new state $s'$ (line 21). Then, if *postProcess* is also `true` (line 23), $ISSE$ performs post processing for the transition: (i) computes the hash code of $s'$ (line 24), and (ii) checks whether $s'$ violates any specified properties (line 27). If *postProcess* is `false`, these two steps are not done.

Regardless of whether a transition is actually executed or post-processed, information regarding that transition is added to the $OUT$ data structure. Essentially, this data structure is a condensed representation of the current state-space graph. $OUT$ stores information about each of the visited states that do *not* violate a property in `checkProperty`. Specifically, for each executed transition $t$ that changes the system state from $s$ to $s'$, $OUT$ stores a triple $\langle h, E, h' \rangle$, where $h$ is the hash code of $s$, $h'$ is the hash code of $s'$, and $E$ encodes the transition $t$ (typically encodes the selected method, called `m` in Figure 2, and the parameters of the method call). Upon completion of the exploration, $ISSE$ writes the $OUT$ data structure to an output file (line 35).

The settings of *execute* and *postProcess* are determined as follows. Initially, both are set to `true` (lines 9 and 10). $ISSE$ then checks whether the transition $t$ represents an

unmodified event (line 11). If not, *execute* and *postProcess* remain `true` since the transition has to be executed and the new state post-processed. If the transition does represent an unmodified event, potential savings may be gained depending on whether or not information about this state-transition pair exists in $IN$. $ISSE$ searches for the hash code of the current state $s$ and the transition $t$ in $IN$ (line 12). If it does not exist, then *execute* and *postProcess* remain `true` since no information is known.

If the information does exist from the previous exploration, $ISSE$ checks if the state depth is about to become equal to `MAX_DEPTH` (line 14). If so (i.e., this is a deepest transition), *execute* is set to `false` because there is no need to execute this (deepest) transition: it would generate a state $s'$ that satisfies the property and $s'$ would not be added to $ToExplore$. If this is not a deepest transition, $ISSE$ checks if the hash code of $s'$ exists in $Visited$ (line 16). If so (i.e., this is a non-tree transition), *execute* is also set to `false` for the same reason as in the former check.

Finally, if the transition is neither deepest nor non-tree, it must be a tree transition. In this case, only *postProcess* is set to `false` (line 19). Since the hash code of $s'$ is in $IN$ (line 13), we know that $s'$ does not violate a property (because $IN$ does not store the hash codes of states that violate a property, as explained above). Hence, we can save the time for post processing this state, both to compute its hash code and to check whether it violates a property. It is important to note that $ISSE$ does not stop exploration for a tree transition (even when it does not need to post-process it) because there can be changed states reachable from $s'$.

```
 0: procedure ISSE()
 1:    IN = readPriorExplorationGraph()
 2:    s_init.depth = 0
 3:    ToExplore = {s_init}
 4:    Visited = {computeHashCode(s_init)}
 5:    while (|ToExplore| > 0) do
 6:       s = choose state from ToExplore
 7:       transitions = genEnabledTransitions(s)
 8:       for each transition t in transitions do
 9:          execute = true // transition modified or not found
10:          postProcess = true
11:          if (isNonModifiedTransition(t)) then
12:             nextHash = get(IN, s.hashcode, t)
13:             if (nextHash ≠ NOTFOUND) then
14:                if (s.depth + 1 = MAX_DEPTH) then
15:                   execute = false; // deepest transition
16:                else if (nextHash ∈ Visited) then
17:                   execute = false; // non-tree trans.
18:                else
19:                   postProcess = false; // tree trans.
20:          if (execute) then
21:             s' = genNextState(s, t)
22:             s'.depth = s.depth + 1
23:             if (postProcess) then
24:                s'.hashcode = computeHashCode(s')
25:                valid = checkProperty(s');
26:                if (not valid) then print(ERROR)
27:             else
28:                s'.hashcode = nextHash
29:             if (s'.depth < MAX_DEPTH &&
30:                s'.hashcode ∉ Visited) then
31:                Visited = Visited ∪ {s'.hashcode}
32:                ToExplore = ToExplore ∪ {s'}
33:             nextHash = s'.hashcode
34:          add(OUT, s.hashcode, t, nextHash)
35:    writeCurrentExplorationGraph(OUT)
```

**Figure 4: Incremental State-Space Exploration**

## 3.3 Correctness of Incremental SSE

We first consider the correctness of a special case of our technique, namely when there is no hashing, or in other words when the function `computeHashCode` (line 10 of Figure 3 and line 24 of Figure 4) is the identity function. In this case, the $SSE$ algorithm (Figure 3) is guaranteed to discover any buggy execution with at most `MAX_DEPTH` number of transitions. The $ISSE$ algorithm (Figure 4) can be easily shown to also discover any incorrect executions of length at most `MAX_DEPTH` and to correctly compute the $OUT$ graph, as long as states and transitions identifiers can be matched across runs.

In the presence of hashing, there are a few subtle issues to consider. First, the traditional $SSE$ is itself only sound but no longer complete: any faulty execution discovered by it is indeed a bug, but it can no longer guarantee the absence of bugs in *all* executions of length at most `MAX_DEPTH`. The reason for this is that the algorithm may visit a state $s_2$ whose hash code is the same as a previously visited state $s_1$ (line 14 of Figure 3), and therefore decide to not explore further any executions from $s_2$. Since the non-incremental algorithm is not complete in the presence of hashing, our incremental algorithm is also not complete for the same reasons. However, $ISSE$ has another potential source of incompleteness. Suppose state $s_1$ is explored in the previous run, and its hash code and transitions are output in the state-space graph. Assume that due to new/modified transitions a state $s_2$ is visited before $s_1$ in the new version, and the hash code of $s_2$ is the same as that of $s_1$. In this case, the incremental algorithm will (incorrectly) deduce that the state $s_2$ was visited in the previous run (conditional in line 13 of Figure 4). The algorithm will use the hash code of the states reached from $s_1$ to store in $Visited$ and output to the new state-space graph. As a consequence, the incremental algorithm may not execute certain paths that would be explored by the non-incremental algorithm. Despite these limitations, the incremental algorithm is *guaranteed to be sound* for safety/reachability properties, i.e., every erroneous trace discovered is indeed a bug in the system. This is because every tree transition is actually executed, even if old hash code is used when computing the graph. Since the discovery of errors is often the goal of state-space exploration (as witnessed by the widespread use of hashing in non-incremental algorithms despite its incompleteness), we expect that these theoretical limitations will not affect the usefulness of our technique in practice.

$ISSE$ has additional requirements on the program states, their hash codes, and transition identifiers between consecutive runs. $ISSE$ requires that the state layout of two program versions does not change, e.g., the new version cannot add or remove fields to the existing classes. Moreover, $ISSE$ requires that the hash code for a program state is the same for repeated runs and that the transition identifiers are the same for repeated runs. The hash code will indeed be the same for state-space explorers that compute the hash code solely from the state, which is the case for both JPF and J-Sim; these tools as well as many others for object-oriented programs actually compute hash code based on state linearization [3, 20, 47] such that the hash code is the same for all states that are isomorphic, i.e., differ only in the identity of the objects but have the same linked structure and the same values for primitive fields. The transition identifiers are encoded with the method identifiers and choices made for method parameters (return values of `Verify.getInt` calls

in drivers such as that shown in Figure 2), and the user instructs the tool on how the new program has changed. (In our current implementations, the user has to select which methods changed, although that can be automatically inferred by comparing the two versions [2, 22, 28, 33].) In summary, while there are cases where $ISSE$ cannot reuse previous graph, the user can always choose to run $SSE$ or $ISSE$ without reading the previous graph.

## 3.4 Performance Analysis

We next analytically derive the conditions under which ISSE does or does not provide speedup over the traditional SSE. The key is to consider various outcomes for the decision variables *execute* and *postProcess* at lines 22 and 25 in $ISSE$, which determine whether ISSE executes a transition and its associated post-processing:

- If both *execute* and *postProcess* are `true`, $ISSE$ executes both the transition and its post-processing. This is a modified transition or a new state with respect to the prior exploration, but the transition can be of any type for the current exploration.
- If *execute* is `true` and *postProcess* is `false`, $ISSE$ executes the transition but not its post-processing. This is a tree transition for the current exploration which was also executed in the prior exploration.
- If *execute* is `false`, $ISSE$ executes neither the transition nor its post-processing. This is either a deepest or non-tree transition for the current exploration which was also executed in the prior exploration.

To estimate the running time of the algorithms, our simplified analysis uses the following six variables: $X$ is the average time to execute a transition, $P$ is the average time of post-processing, $O$ is the overhead of incremental exploration (including file reads and writes, lookups into $IN$, and building of $OUT$), $m$ is the frequency of (enabled) modified transitions (where the condition in line 11 is `false`), $f$ is the frequency of (unmodified) transitions found in $IN$ (where the condition in line 13 is `true`), and $t$ is the frequency of tree transitions in the current exploration among the transitions found in the prior exploration (so line 19 is executed). The approximate running time of each transition in ISSE is $O + (1-m)ftX + (m + (1-m)(1-f))(X+P)$, whereas the approximate running time for non-incremental SSE is $X + P$ as it always both executes the transition and its post-processing. Hence, the expected saving from ISSE is

$$1 - \frac{O + (1-m)ftX + (m + (1-m)(1-f))(X+P)}{X+P}.$$

A more detailed analysis is available in the accompanying technical report [38].

Intuitively, ISSE is faster than non-incremental SSE when (1) the overhead of incremental exploration is small compared to the execution and post-processing ($O < X + P$) and (2) there is reuse from the prior exploration: (2.1) the frequency of modified transitions ($m$) is small, (2.2) the frequency of found transitions is large (so $1 - f$ is small), and (2.3) those found transitions are mostly non-tree and deepest for the current exploration (so $t$ is also small). The worst case for ISSE is when everything changes, i.e., $m = 1$. Our experiments show that the overhead of ISSE even in this case does not significantly slow down ISSE over the non-incremental SSE. The key reason is that *the execution and post-processing times are significant for our subject programs with complex data*. This is why we view our technique suitable for programs with dynamically allocated data: while the ISSE algorithm itself can work correctly in many cases, we do not expect it to provide a speedup when the execution and post-processing times are small (and, in particular, smaller than the lookup into the $IN$ data structure).

# 4. EVALUATION

We performed an experimental evaluation of ISSE. We implemented ISSE in two model checkers: Java PathFinder (JPF) [47] and the J-Sim state-space explorer [39, 40]. Both implementations follow the pseudo-code shown in Figure 4.

We first describe the subject programs that we use in our study. We then present the results using our JPF implementation. We finally present the results using our J-Sim implementation.

The main research question is how our incremental state-space exploration, ISSE, decreases or increases the exploration time compared with the traditional, non-incremental exploration. We also compare the memory requirements for ISSE and the non-incremental exploration.

## 4.1 Subjects

Our experiments use 11 programs for JPF and 3 simulation models of network protocols for J-Sim, taken from a variety of sources. Note that we took *one version* of each subject, but evaluating ISSE requires several versions. For most subject, we had several undergraduate and graduate students implement more versions, and for some subjects, we created versions by seeding errors.

The following nine subjects are simple data structures: `binheap` implements priority queues with binomial heaps [48]; `bst` implements a set using binary search trees [49]; `deque` implements a double-ended queue using doubly-linked lists [8]; `fibheap` is an implementation of priority queues using Fibonacci heaps [48]; `heaparray` is an array-based implementation of priority queues [3, 49]; `queue` is an object queue implemented using two stacks [10]; `stack` is an object stack [10]; `treemap` implements maps using red-black trees based on Java collection 1.4 [3, 48, 49]; `ubstack` is an array-based implementation of a stack bounded in size, storing integers without repetition [7, 30, 42]. The tenth subject is `filesystem`, based on the Daisy code [10, 32] as described in Section 2. These ten subjects are all small, ranging from 1 class (for `heaparray` and `ubstack`) to 4 classes (for `filesystem`) and from 27 (for `stack`) to 301 (for `treemap`) non-comment, non-blank lines of code.

The eleventh subject for JPF is `aodv`, which implements the Ad-Hoc On-Demand Distance Vector (AODV) [31] routing protocol for wireless ad hoc networks, also used for the J-Sim state-space explorer. The remaining two subjects for J-Sim are `dirdiff`, which is the directed diffusion protocol [19] for wireless sensor networks, and `arq`, which is the stop-and-wait Automatic Repeat reQuest (ARQ) [44] protocol. AODV and directed diffusion are reasonably complex network protocols whose J-Sim simulation models (not including the J-Sim library) have about 1,200 and 1,400 lines of code, respectively. ARQ is a simple protocol whose simulation model has about 170 lines of code. We defer the descriptions of the three network protocols to Section 4.3.

For each subject, we use previously written drivers [8] similar to Figure 2. These drivers exercise the main mutator methods. For data structures, the drivers add and remove elements. For `filesystem`, the driver creates and removes directories, creates and removes files, and writes to and reads from files. For the three network protocols, each driver [39, 40] determines the events that are enabled in each state (e.g., packet delivery, packet loss, timeout, node reboot, etc.) and executes them for various parameters.

## 4.2 Java PathFinder

JPF is a general purpose model checker for Java bytecode, implemented as a backtrackable Java Virtual Machine (JVM) running on top of a regular, host JVM. We have implemented our ISSE technique in JPF version 4 [25]. We first present results of a limit study on all 11 subjects. We then present results for several program evolution patterns. We finally present results for several patterns of changing exploration for the same program.

We performed all JPF experiments on a Pentium 4 3.4GHz workstation running under RedHat Enterprise Linux 4. We used Sun's JVM 1.5.0_07, limiting each run to 1.8GB of memory and 1 hour of elapsed time.

### 4.2.1 Limit Study

To better understand the speedups possible using ISSE, we performed a limit study on all 11 subject programs. Figure 5 shows the results. For all experiments in this study, the exploration depth and the number of values for operations are the same (i.e., $L$=$M$ from Figure 2). For each subject, we chose the largest bound for which we could run an exhaustive, breadth-first exploration of the state-space using 1.8GB of memory and 1 hour of elapsed time. We tabulate the exploration time for non-incremental SSE and our incremental ISSE and the speedup (or slowdown when the percentage is negative) obtained using ISSE. The version column indicates the type of exploration.

*Initial* corresponds to the first exploration that always completely explores all transitions. Due to the cost of collecting information about the exploration and saving it to a file, ISSE always takes slightly more time than the traditional non-incremental exploration. The slowdowns range from -0.42% (for `aodv`) to -7.13% (for `stack`.)

*Best* corresponds to the best case, an exploration where no transitions have changed. Although this is not a realistic scenario, it represents the upper limit of performance improvement for the given subject and bound. A best case exploration requires only *tree transitions* to be executed. The savings of ISSE over non-incremental SSE for the best case range from 20.41% to 96.34%, with a median of 74.02%.

*Worst* corresponds to the worst case, a scenario where all of the subjects operations were flagged as modified (though we don't actually modify any code for these experiments). In this scenario, all transitions must once again be executed, so it illustrates the overhead associated with ISSE. The additional cost of ISSE over non-incremental SSE for the worst case ranges from 0.78% to 14.07%, with a median of 5.89%.

We also evaluated the impact that using ISSE has on memory usage. Using Sun's `jstat` monitoring tool [43], we identified the peak usage of garbage-collected heap in the JVM while running our experiments. This measurement represents the most relevant portion of memory used during state-space exploration.

| Experiment | | Time (sec) | | | Memory |
|---|---|---|---|---|---|
| Subject | Ver. | SSE | ISSE | Savings | Savings |
| aodv | initial | 300.47 | 301.72 | -0.42% | -8.54% |
| ($L$=$M$=9) | best | 300.47 | 103.06 | 65.70% | -9.89% |
| | worst | 300.47 | 306.44 | -1.99% | -14.78% |
| binheap | initial | 461.33 | 479.28 | -3.89% | -12.41% |
| ($L$=$M$=8) | best | 461.33 | 47.65 | 89.67% | -49.63% |
| | worst | 461.33 | 480.04 | -4.06% | -50.17% |
| bst | initial | 1081.21 | 1124.62 | -4.01% | -26.05% |
| ($L$=$M$=11) | best | 1081.21 | 185.77 | 82.82% | -62.73% |
| | worst | 1081.21 | 1179.61 | -9.10% | -66.47% |
| deque | initial | 56.73 | 59.24 | -4.42% | -24.67% |
| ($L$=$M$=8) | best | 56.73 | 13.82 | 75.64% | -30.81% |
| | worst | 56.73 | 62.00 | -9.29% | -52.41% |
| fibheap | initial | 416.26 | 435.42 | -4.60% | -13.48% |
| ($L$=$M$=8) | best | 416.26 | 108.15 | 74.02% | -27.74% |
| | worst | 416.26 | 453.56 | -8.96% | -29.70% |
| filesystem | initial | 1024.04 | 1043.75 | -1.92% | -120.72% |
| ($L$=$M$=5) | best | 1024.04 | 37.44 | 96.34% | -246.28% |
| | worst | 1024.04 | 1032.03 | -0.78% | -237.30% |
| heaparray | initial | 105.29 | 108.78 | -3.31% | -14.93% |
| ($L$=$M$=8) | best | 105.29 | 83.80 | 20.41% | -27.25% |
| | worst | 105.29 | 111.49 | -5.89% | -28.83% |
| queue | initial | 88.34 | 93.54 | -5.89% | -1.66% |
| ($L$=$M$=7) | best | 88.34 | 27.58 | 68.78% | -14.33% |
| | worst | 88.34 | 99.68 | -12.84% | -16.06% |
| stack | initial | 65.13 | 69.77 | -7.13% | -0.33% |
| ($L$=$M$=7) | best | 65.13 | 24.71 | 62.06% | -11.83% |
| | worst | 65.13 | 74.29 | -14.07% | -15.31% |
| treemap | initial | 254.41 | 258.20 | -1.49% | -36.21% |
| ($L$=$M$=12) | best | 254.41 | 27.05 | 89.37% | -79.66% |
| | worst | 254.41 | 266.16 | -4.62% | -78.78% |
| ubstack | initial | 149.88 | 154.14 | -2.84% | -14.96% |
| ($L$=$M$=8) | best | 149.88 | 114.72 | 23.46% | -30.77% |
| | worst | 149.88 | 156.70 | -4.55% | -30.09% |

**Figure 5: Limit Study in JPF**

Figure 5 shows that using ISSE in place of non-incremental exploration resulted in increased memory usage (as indicated by negative memory savings percentages). The increase ranges from 0.33% (for the initial version of `stack`) to 246.28% (for the best case version of `filesystem`). These increases are largely due to the space needed for the $IN$ and $OUT$ data structures used by the ISSE algorithm (Figure 3).

### 4.2.2 Code Changes

To evaluate the savings that ISSE can provide in successive explorations, we had to obtain multiple versions for several subjects. Figure 6 shows the results. Each sequence corresponds to versions of the same subject, starting with errors, followed by a corrected (or more correct) version. For all subjects but `aodvA` and `filesystemA`, we obtained the initial versions (Ver. 1) by deleting a method body from a correct version and asking several undergraduate and graduate students to implement the method. If the resulting method contained errors, we asked the student to provide another version with the errors corrected. For `aodvA` and `filesystemA`, we seeded the initial version with errors. As with the limit study, we consider the case where both the exploration depth and the number of values for operations are the same ($L$=$M$). The savings of exploration time with ISSE over non-incremental exploration for *non-initial* runs (Ver. 2 and 3) range from -4.71% to 62.46%, with a median of 56.99%.

Note that ISSE is always slightly slower than SSE in the initial exploration (Ver. 1) as ISSE needs to collect the additional information for the next exploration. ISSE is also slightly slower for Ver. 2 in one case, `heaparrayA`. The reason is that the change in code from Ver. 1 to Ver. 2 makes a big

| Experiment | | | Time (sec) | | |
|---|---|---|---|---|---|
| Subject | $L$=$M$ | Ver. | SSE | ISSE | Savings |
| aodvA | 9 | 1 | 302.24 | 302.46 | -0.07% |
| | | 2 | 302.85 | 113.68 | 62.46% |
| | | 3 | 302.54 | 113.64 | 62.44% |
| binheapA | 8 | 1 | 416.90 | 428.02 | -2.67% |
| | | 2 | 404.78 | 249.13 | 38.45% |
| binheapB | 8 | 1 | 437.29 | 447.60 | -2.36% |
| | | 2 | 407.88 | 251.73 | 38.28% |
| binheapC | 8 | 1 | 537.06 | 543.85 | -1.26% |
| | | 2 | 487.22 | 331.34 | 31.99% |
| bstA | 11 | 1 | 1782.46 | 2238.98 | -25.61% |
| | | 2 | 1140.94 | 807.23 | 29.25% |
| bstB | 11 | 1 | 1094.29 | 1132.47 | -3.49% |
| | | 2 | 1099.22 | 731.57 | 33.45% |
| filesystemA | 5 | 1 | 1083.80 | 1085.16 | -0.13% |
| | | 2 | 1064.53 | 419.03 | 60.64% |
| | | 3 | 1040.02 | 409.41 | 60.63% |
| filesystemB | 5 | 1 | 1073.13 | 1066.63 | 0.61% |
| | | 2 | 1052.57 | 452.68 | 56.99% |
| filesystemC | 5 | 1 | 1728.21 | 1752.10 | -1.38% |
| | | 2 | 1047.94 | 447.31 | 57.32% |
| filesystemD | 5 | 1 | 1053.24 | 1064.40 | -1.06% |
| | | 2 | 1045.59 | 446.91 | 57.26% |
| heaparrayA | 8 | 1 | 67.36 | 70.69 | -4.94% |
| | | 2 | 131.73 | 137.93 | -4.71% |

**Figure 6: Code Change Sequences in JPF**

change in the state-space graph. (A small change in code can result in a large change in the state-space graph, but a large change in code can still result in a small change in the state-space graph.) In Ver. 1, the code has an error that leads to premature expansion of the array used to represent the heap data structure. Correcting this error in Ver. 2 resulted in a great number of "new" states (i.e., states that did not exist in the initial exploration), and all transitions leading from these new states must be executed in their entirety. (In terms of the pseudo-code from Figure 4, it means that in line 12, $nextHash$ gets value `NOTFOUND`.) It is important to point out, however, that the impact associated with ISSE is limited to only -4.71%.

#### 4.2.3 Exploration Changes

In addition to evaluating code changes, we also considered possible changes to the exploration itself. Figure 7 shows the results for three scenarios.

*Adding a new method* – `aodvF` Ver. 1 is an AODV implementation without a reboot operation being exercised in the driver. This operation has been added in `aodvF` Ver. 2. (The code of the protocol itself is not changed, but the driver is changed.) The savings of ISSE over non-incremental exploration for Ver. 2 is 8.53%.

*Adding a new value* – `binheapF` exercises the same code in both runs, but run 1 uses $L$=$M$=7 bounds for exploration depth $L$ and argument values for method calls $M$, whereas run 2 uses $L$=7 and $M$=8, i.e., the number of different argument values has been increased to 8. As a result of reusing the exploration performed with the first 7 values, the savings of ISSE over non-incremental for run 2 is 35.08%.

*Increasing the exploration depth* – `binheapG` exercises the same code in both runs 1 and 2, but run 1 uses $L$=$M$=7, whereas run 2 uses $L$=8 and $M$=7, i.e., the exploration depth is increased to 8. Although a large number of new transitions need to be executed at depth 8, using ISSE instead of non-incremental SSE still reduced the run 2 exploration time by 6.43%. Similarly, `aodvG` exercises the same code in both runs 1 and 2, but run 1 uses $L$=$M$=9, whereas

| Experiment | | | Time (sec) | | |
|---|---|---|---|---|---|
| Subject | L,M | Ver. | SSE | ISSE | Savings |
| aodvF | 9,9 | 1 | 48.05 | 48.66 | -1.27% |
| | | 2 | 301.35 | 275.64 | 8.53% |
| aodvG | 9,9 | 1 | 301.33 | 303.42 | -0.69% |
| | 10,9 | | 1091.20 | 915.39 | 16.11% |
| binheapF | 7,7 | 1 | 25.19 | 25.82 | -2.50% |
| | 7,8 | | 54.85 | 35.61 | 35.08% |
| binheapG | 7,7 | 1 | 25.20 | 25.87 | -2.66% |
| | 8,7 | | 178.79 | 167.30 | 6.43% |

**Figure 7: Exploration Change Sequences in JPF**

run 2 uses $L$=10 and $M$=9, i.e., the exploration depth is increased to 10. Using ISSE instead of non-incremental SSE reduces the exploration time for run 2 by 16.11%.

### 4.3 J-Sim State-Space Explorer

J-Sim [45] is a component-based network simulator written entirely in Java. The J-Sim state-space explorer [39, 40] is a J-Sim component that dynamically checks whether a given J-Sim simulation model of a network protocol satisfies certain assertions (i.e., safety properties). The J-Sim simulation models are themselves also written in Java. A traditional network simulator executes only one path of the model (typically to measure performance). In contrast, the J-Sim state-space explorer takes control of the model execution and explores the (entire) state space created by executing the model along several execution paths, with the goal to find an execution (if any) that violates an assertion.

We implemented ISSE in the J-Sim state-space explorer and evaluated it for the simulation models of three network protocols described in the rest of this text. We consider several versions of simulation models for each protocol, obtained by seeding errors or removing methods. Our J-Sim experiments build the $OUT$ data structure (from Figure 4) and write it to a file only for the first version, and load the information (for unmodified transitions) from the file to the $IN$ data structure for each subsequent version. All experiments in this section use the breadth-first search strategy.

#### 4.3.1 AODV in J-Sim

Ad-Hoc On-Demand Distance Vector (AODV) [31] is a routing protocol for wireless ad hoc networks. In AODV, each network node maintains a routing table. For a node $n$, a routing table entry (RTE) to a destination node $d$ contains several fields, including a next hop address (address of the node to which $n$ forwards data packets destined for $d$), a hop count (number of hops needed to reach $d$ from $n$), and a destination sequence number (a measure of the freshness of the route information). For $n$ to establish a valid RTE to $d$, $n$ has to initiate a route discovery process that involves broadcasting a route request (RREQ) packet and receiving at least one route reply (RREP) packet from either $d$ itself or another node. Each valid RTE is associated with a lifetime. Periodically, an RTE timeout event is triggered invalidating (but not deleting) all the RTEs that have not been used (e.g., to send/forward packets to $d$) for a time interval that is greater than the lifetime. Invalidating an RTE involves incrementing the destination sequence number and setting the hop count to $\infty$. Received RREQ packets are kept, for a specific time duration, in a broadcast ID cache for duplicate detection and suppression. The safety property that we check is the routing *loop-free* property: a node must

| Experiment | | Time (sec) | | | Memory |
|---|---|---|---|---|---|
| Subject | Ver. | SSE | ISSE | Savings | Savings |
| aodv | 1 | 121.74 | 133.24 | -9.45% | -4.67% |
| (MAX_DEPTH=9) | 2 | 123.98 | 38.91 | 68.62% | -14.81% |
| | 3 | 123.65 | 38.94 | 68.51% | -14.81% |
| arq | 1 | 22.35 | 24.30 | -8.72% | -38.15% |
| (MAX_DEPTH=35) | 2 | 30.13 | 31.58 | -4.81% | -30.31% |

**Figure 8: Code Change Sequences in J-Sim**

not occur at two points on a path between two other nodes (as otherwise packets would loop until timeout).

There are six types of events that can be enabled from a state: $T_0$ initiates a route discovery process, $T_1$ is a broadcast ID cache timeout, $T_2$ is an RTE timeout, $T_3$ delivers a packet from the network to a node, $T_4$ loses a packet, and $T_5$ reboots a node. To evaluate ISSE, we consider a hypothetical scenario where a user tries to implement $T_2$ correctly. In Ver. 1, the user implements $T_2$ by deleting an RTE instead of invalidating it, but realizes that this can cause a routing loop [40]. In Ver. 2, the user changes $T_2$ to invalidate RTE but forgets to increment the destination sequence number, which can again cause a routing loop [40]. In Ver. 3, the user figures out the correct implementation, which increments the destination sequence number when invalidating the RTE. While mistakes as in Ver. 1 and 2 have been made while implementing AODV, we actually started from the correct implementation (Ver. 3) and seeded errors.

Figure 8 shows the results. For Ver. 1, ISSE is slower than non-incremental exploration because ISSE adds transitions information to $OUT$ and writes it to a file. The overhead is indicated by a negative number and is only 9.45%. However, for Ver. 2 and 3, ISSE is faster than non-incremental exploration, on average 68.56%. This large speedup is due to a small number of enabled transitions being modified ($m$=0.01) and a large number of non-modified transitions being found in the prior exploration ($f$=0.94). Also, in Ver. 2 and 3, the proportion of tree transitions is small at 6.98% (and it approximates $t$, the frequency of tree transitions found in the prior exploration), while the proportion of non-tree and deepest transitions is 20.55% and 72.47%, respectively. Moreover, we discovered that the costs of executing transitions, computing hash codes, and checking the safety property are considerably high taking together more than 78% of the average total time in the non-incremental exploration of Ver. 3. All these are favorable conditions for ISSE to provide a speedup in state-space exploration time (Section 3.4).

The *number* of transitions executed for Ver. 3 (not shown in Figure 8 but available in the technical report [38]) in non-incremental is 460,150 whereas that in ISSE is 62,074; i.e., ISSE reduces the number of transitions by 86% (or roughly 7x). The reason why ISSE reduces the exploration *time* only 68.51% (or roughly 3x) is that (1) some operations (e.g., inserting or searching a hash code in $Visited$ and inserting a state in $ToExplore$) take almost equal times in both the non-incremental and ISSE techniques, and (2) the average time of executing a tree transition (141.35$\mu s$) is more than the average time of executing a non-tree or deepest transition (109.09$\mu s$ and 94.59$\mu s$, respectively).

Memory usage increased when using ISSE instead of non-incremental exploration, as expected. However, the overhead was less than 15% for each of the three AODV versions.

| Experiment | | Time (sec) | | | Memory |
|---|---|---|---|---|---|
| Subject | Ver. | SSE | ISSE | Savings | Savings |
| dirdiffA | 1 | 20.37 | 21.94 | -7.71% | -3.90% |
| (MAX_DEPTH=10) | 2 | 30.23 | 16.46 | 45.55% | -8.10% |
| dirdiffB | 3 | 18.93 | 20.53 | -8.45% | -4.05% |
| (MAX_DEPTH=10) | 4 | 27.73 | 15.09 | 45.58% | -8.09% |
| dirdiffC | 5 | 18.29 | 19.34 | -5.74% | -2.99% |
| (MAX_DEPTH=10) | 6 | 30.39 | 18.53 | 39.03% | -7.17% |
| dirdiffD | 7 | 16.49 | 17.63 | -6.91% | -3.11% |
| (MAX_DEPTH=10) | 8 | 27.71 | 17.56 | 36.63% | -7.06% |
| *Ver. 1 has events $T_0$, $T_1$, $T_3$, $T_4$, and $T_5$. Ver. 2 adds $T_2$.* | | | | | |
| *Ver. 3 has events $T_0$, $T_1$, $T_4$, and $T_5$. Ver. 4 adds $T_2$.* | | | | | |
| *Ver. 5 has events $T_0$, $T_1$, $T_2$, $T_3$, and $T_4$. Ver. 6 adds $T_5$.* | | | | | |
| *Ver. 7 has events $T_0$, $T_1$, $T_2$, and $T_4$. Ver. 8 adds $T_5$.* | | | | | |

**Figure 9: Exploration Change Sequences in J-Sim**

### 4.3.2 Directed Diffusion in J-Sim

Directed diffusion [19] is a protocol for information dissemination in wireless sensor networks. The implementation details of the J-Sim simulation model of directed diffusion are available elsewhere [38]. Importantly, there are six types of events $T_0, \ldots, T_5$ that can be enabled from a state. These events are conceptually similar to the AODV events.

While for AODV we evaluated ISSE in a scenario where the implementation of an existing event is modified from one version to another (as done in Section 4.2.2), for directed diffusion we evaluated ISSE in four different examples where the implementation of a *new* event is added from one version to another (as done in Section 4.2.3). Figure 9 lists the four cases. The overheads incurred by ISSE in the "initial" versions range from 5.74% to 8.45%. The time savings of ISSE over non-incremental range from 36.63% to 45.58%. The reasons for the savings are similar as for AODV: $m$ is relatively small, ranging from 0.13 to 0.29, and $f$ is relatively large, ranging from 0.75 to 0.76. ISSE increased memory usage for all eight explorations but only up to 8.10%.

### 4.3.3 Stop-and-wait ARQ in J-Sim

Stop-and-wait ARQ is a simple protocol for acknowledging messages: the sender sends a single data packet, sets a retransmission timer, and then waits for a positive acknowledgment (ACK) from the receiver. A 1-bit sequence number is included in the header of each data packet and each ACK packet to enable matching packets and acknowledgments. Upon receiving an ACK, the sender checks the sequence number in the ACK to determine whether to send a new data packet or a retransmission. The safety property is that the receiver does not miss any data packet that the sender believes to have been received by the receiver.

There are five types of events that can be enabled from a state: $T_0$ delivers a data packet, $T_1$ delivers an ACK packet, $T_2$ causes retransmission timer timeout, $T_3$ loses a data packet, $T_4$ loses an ACK packet. As for AODV, we seed faults to create versions. In Ver. 1, the implementation of $T_1$ does not check the sequence number in the ACK before sending a data packet, which can cause a violation of the safety property (as described elsewhere [39]). In Ver. 2, the implementation of $T_1$ is correct.

As shown in Figure 8, ISSE is *not* able to provide a speedup for ARQ. We discovered that this is due to a relatively large value of $m$=0.25 and an extremely small value of $f$=0.02; the latter is caused by a large number of new states that did not appear in the prior exploration and thus information about the transitions from those states is not in

IN (from Figure 4). In fact, more than 98% of the 318,216 transitions executed in non-incremental are also executed in ISSE for Ver. 2. Furthermore, the costs of executing transitions, computing hash codes, and checking the safety property are *not* high taking together less than 51% of the average total time in the non-incremental. Finally, the overall frequency of tree transitions is relatively large at 25%. All these are unfavorable conditions, so ISSE increases exploration time over non-incremental SSE. ISSE also increases memory requirement by 38.15% and 30.31% for Ver. 1 and 2, respectively.

# 5. RELATED WORK

There is a large body of work closely related to the goal of this paper, namely, analyzing the correctness of evolving software. We first review the work done in the context of model checking. We then discuss the work done in the context of software testing.

Starting from the initial work Sokolsky and Smolka [41], several techniques for incremental model checking have been proposed. The techniques address checking rich properties expressed in modal mu-calculus [41] and Monadic Second Order Logic [24] for non-recursive abstract models and checking safety properties of recursive software [6,18]. However, all of these techniques focus on control-intensive properties. The technique proposed in this paper focuses on data intensive properties where large concrete states need to be maintained to check properties, and changes are typically made to methods and functions that are enabled in almost every state.

Another related project on incremental computation for model checking is on incremental heap canonicalization [26]. However, this project considers incremental computation between a pre-state and a post-state of one transition in state-space exploration (SSE) and not incremental computation between two consecutive SSEs. Finally, there is also some recent work on *incremental conformance testing* [12] where the goal is to generate a complete test suite for a new modified version based on the test suite for the previous version. It relies on performing state-space exploration on the specification, and it makes very different assumptions on the software evolution (e.g., that changes in the specification reflect the changes made to the implementation) than the assumptions considered here.

The term *incremental model checking* is also used in the context of bounded model checking [4] that performs checking within a user-specified bound (e.g., bound on the length of the execution). However, the focus in such checking is on incrementally increasing the bound and not incrementally evolving code as in ISSE.

The main approach to address evolving software in the context of testing is regression testing: it effectively checks that a newer version of software still passes the tests that an older version of the same software passed. Researchers have developed numerous methods to improve the basic regression testing.

Regression test selection [15, 16, 50] chooses to run only some of the tests on the new version, thus speeding up the testing process by not running all tests. A key challenge is to have safe selection, i.e., guarantee that tests that are not selected could not reveal errors. To enable safe selection, several techniques for test selection record not only whether a test passed or failed but also some additional information from previous runs (e.g., code coverage). Our work on ISSE is partly motivated by such test selection techniques, but ISSE records the entire state-space graph.

Test prioritization [14,23,35] reorders (all or only selected) tests in order to reveal errors faster, thus reducing the time that a developer has to wait to find failing tests for program changes. We could also consider prioritization in the context of ISSE: if a previous exploration led to a property violation for some execution sequence, then we could first execute that same sequence (and its neighborhood) in the subsequent exploration.

Impact analysis [2,22,33] finds (statically or dynamically) which code changes could affect which tests, thus aiding test selection or debugging by pointing out which changes could (not) lead to failing tests. We could leverage impact analysis to improve how ISSE determines what transitions are modified and need to be executed: ISSE currently uses method-level granularity and re-executes a transition if it involves a changed method. However, even when the code of some method changed, many execution paths may not be affected, and thus the re-execution is not necessary for all possible input states.

Automatically decomposing system tests (often used in regression testing) into unit tests [13, 29, 36] also helps in speeding up regression testing: when a programmer changes some program unit, it is not necessary to rerun an entire, potentially time-consuming system test, but it suffices to rerun a focused, rapid unit test. While this approach is useful in regression testing, it does not appear to have a direct application in ISSE since executions in ISSE are already fairly short and mostly exercise one program unit.

# 6. CONCLUSIONS

We presented ISSE, a technique for incremental state-space exploration that can speed up the traditional, non-incremental state-space exploration for program evolutions or exploration changes. While traditional exploration takes *one program version* and systematically explores the states reachable during program's executions to find property violations, incremental exploration considers *several program versions*: reusing the results of exploration from one version can speed up exploration for the next version, since the state spaces of consecutive program versions can have significant overlap. We implemented ISSE in both JPF and the J-Sim state-space explorer. The experimental results on 11 programs with dynamically allocated data (done in JPF) and 3 network protocols (done in J-Sim) show that ISSE in most cases speeds up the exploration, up to 68.62% (or over 3x), while in a few cases slows it down, up to 14.07%. In the future, we plan to investigate other techniques (e.g., storing dependency between transitions instead of entire state-space graphs or caching computation across different explorations) that could speed up successive explorations.

**Acknowledgments.** The authors would like to thank the late Prof. Jennifer Hou for her encouragement and mentoring over the past few years, including this project. We thank Marcelo d'Amorim for discussions about this work, and Milos Gligoric, Tihomir Gvero, Shan Lu, Aleksandar Milicevic, Sasa Misailovic, and Chen Xu for writing code versions used in the evaluation. This work was partially supported by NSF grants CCF-0448178, CNS-0615372, and CNS-0613665, a Vodafone fellowship, and a gift from Microsoft Research.

# 7. REFERENCES


[1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *CAV 2004*.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE 2005*.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA 2002*.

[4] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS 2004*.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[6] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV 2005*.

[7] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software - Practice and Experience*, 34:1025–1050, 2004.

[8] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *ISSTA 2007*.

[9] M. d'Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *ICFEM 2006*.

[10] P. T. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *OOPSLA 2006*.

[11] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[12] K. El-Fakih, N. Yevtushenko, and G. v. Bochmann. FSM-based incremental conformance testing methods. *IEEE Trans. on Soft. Eng.*, 30(7):425–436, 2004.

[13] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *FSE 2006*.

[14] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. on Soft. Eng.*, 28(2):159–182, 2002.

[15] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[16] M. J. Harrold, J. A. Jones, T. Li, D. Liang, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA 2001*.

[17] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *International Symposium on Verification: Theory and Practice*, 2003.

[19] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCom 2000*.

[20] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE*, page 254, Washington, DC, USA, 2001. IEEE Computer Society.

[21] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS 2003*.

[22] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE 2003*.

[23] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. on Soft. Eng.*, 33(4):225–237, 2007.

[24] J. Makowsky and E. Rawe. Incremental model checking for fixed point properties on decomposable structures. In *MFCS 1995*.

[25] P. C. Mehlitz, W. Visser, and J. Penix. The JPF runtime verification system. Online manual. `http://javapathfinder.sourceforge.net/JPF.pdf`.

[26] M. Musuvathi and D. L. Dill. An incremental heap canonicalization algorithm. In *SPIN 2005*.

[27] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI 2002*.

[28] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE 2003*.

[29] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *WODA 2005*.

[30] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005*.

[31] C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *WMCSA 1999*.

[32] S. Qadeer. Daisy File System. Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software. 2004.

[33] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA 2004*.

[34] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *ESEC/FSE 2003*.

[35] G. Rothermel, R. J. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. on Soft. Eng.*, 27(10):929–948, 2001.

[36] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE 2005*.

[37] P. Sistla. Hybrid and incremental model-checking techniques. *ACM Comput. Surv.*, 28(4es):125, 1996.

[38] A. Sobeih and S. Lauterburg. Incremental state-space exploration in J-Sim. Technical Report UIUCDCS-R-2007-2898, Department of Computer Science, UIUC, Urbana, IL, Sept. 2007.

[39] A. Sobeih, M. Viswanathan, and J. C. Hou. Check and Simulate: A case for incorporating model checking in network simulation. In *MEMOCODE 2004*.

[40] A. Sobeih, M. Viswanathan, D. Marinov, and J. C. Hou. Finding bugs in network protocols using simulation code and protocol-specific heuristics. In *ICFEM 2005*.

[41] O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *CAV 1994*.

[42] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *XP/Agile Universe Conference*, 2002.

[43] Sun Microsystems. *jstat: Java Virtual Machine Statistics Monitoring Tool*. `http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstat.html`.

[44] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall International Inc., 1996.

[45] H.-Y. Tyan. *Design, Realization and Evaluation of a Component-based Compositional Software Architecture for Network Simulation*. Ph.D., Department of Electrical Engineering, The Ohio State University, 2002.

[46] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/FSE 2005*.

[47] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[48] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for Java containers using state matching. In *ISSTA 2006*.

[49] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS 2005*.

[50] J. Zheng, B. Robinson, L. Williams, and K. Smiley. Applying regression test selection for COTS-based applications. In *ICSE 2006*.