# Automated Generation of Context-Aware Tests

Zhimin Wang and Sebastian Elbaum
Dept. of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
zwang,elbaum@cse.unl.edu

David S. Rosenblum
Dept. of Computer Science
University College London
London, UK
d.rosenblum@cs.ucl.ac.uk

## Abstract

*The incorporation of context-awareness capabilities into pervasive applications allows them to leverage contextual information to provide additional services while maintaining an acceptable quality of service. These added capabilities, however, introduce a distinct input space that can affect the behavior of these applications at any point during their execution, making their validation quite challenging. In this paper, we introduce an approach to improve the test suite of a context-aware application by identifying context-aware program points where context changes may affect the application's behavior, and by systematically manipulating the context data fed into the application to increase its exposure to potentially valuable context variations. Preliminary results indicate that the approach is more powerful than existing testing approaches used on this type of application.*

## 1 Introduction

Context-aware applications adapt their behavior based on situational data to produce richer services and manage scarce resources. Such applications are becoming more prevalent in the presence of ubiquitous devices such as mobile phones, which can now support services such as shopping guides, transportation booking services, and the formation of ad-hoc communities for gaming or socializing. Examples of relevant context include location, battery level, time of day, environmental readings (e.g. temperature, humidity), and user preferences (e.g. spoken language, spending limits, ringing profile).

Developing such context-aware applications is full of difficulties unique to the nature of ubiquitous systems. Consider the periodic, asynchronous streaming delivery of location-based information to mobile users. The correct and efficient functioning of this application will be influenced quite strongly by changes in the context of the application. These contexts can be highly dynamic (e.g., signal strength), are often approximated (e.g., user location), and may include contradictory data (e.g., sensors perceiving different things or perceiving the same thing but with different timing). That is why the collection and processing requirements imposed by complex contexts have led to specialized software development practices. For example, context-aware applications rely extensively on middleware to support different abstraction paradigms aimed at hiding the complexity of collecting context information, and they often include mechanisms to mask for and adapt to uneven circumstances (e.g., levels of luminosity) [10].

Given the importance of contextual data to enable more powerful services, and the increasing role those services play in our lives, it is vital that we provide the testing mechanisms to help ensure the dependability of such context-aware applications. Although there have been efforts to support the specification and to some degree the verification of context-aware applications [14], specialized testing techniques are lacking. The validation of context-aware applications is primarily confined to the physical emulation of the mobile device, the logical and physical deployment to an affordable subset of the user scenarios [17], and the tailoring of existing test case generation methodologies (such as testing context-aware middleware [22]).

The previously proposed validation techniques have failed to consider a fundamental aspect of context-aware applications: *changes in context can occur and affect the application behavior at any time during the execution*. Although this may happen with other types of inputs, it is particular prevalent with contextual inputs since they are the continuously streaming drivers of contextual applications. Engineers must identify not only *what* context values to provide, but also *when* the stream of variations in context values can impact the behavior of the application, and hence are worth testing. This is an essential difference from the testing of more traditional systems, where the selection of input values can mostly be performed a priori.

Determining when and controlling how to feed a stream of changing context values to a context-aware application

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

is complicated by several factors. First, the often thick layers of middleware that ease the difficulties of developing these context-aware applications also compound the potential scenarios that a tester must consider. Second, engineers must devise control mechanisms to feed such inputs to the application, which often implies interesting tradeoffs (e.g., utilizing the middleware is realistic and requires no additional infrastructure, but it adds propagation noise and non-determinism). Third, contextual events must be handled asynchronously, and such handling must address the possibility of multiple interfering contextual changes.

In this work we start addressing these challenges through an approach that improves the context-awareness of an existing test suite. To achieve such improvement the technique performs the following tasks: 1) it identifies key program points where context information can effectively affect the application's behavior, 2) it generates potential variants for each existing test case that explore the execution of different context sequences, and 3) it attempts to dynamically direct the application execution towards the generated context sequences. Preliminary assessment of the approach shows that it can significantly enhance an existing test suite and outperform alternative approaches.

## 2 Motivating Example

Dey et al. define context as "any information that can be used to characterize the situation of entities (e.g., whether a person, place, or object) that are considered relevant to the interaction between the user and an application..." [3]. Bunningen further characterizes contextual data as continuously changing, temporal, spatial, imperfect and uncertain [23], capturing some of the unique difficulties faced by testers of context-aware applications.

Consider for example *TourApp*, a context-aware application that runs on the mobile devices of visitors attending an exposition to notify them about demos of interest. *TourApp* was originally distributed with the Context Toolkit [16] and has become one of the canonical mobile context-aware applications to demonstrate context-aware middleware. Figure 1 illustrates the deployed infrastructure that supports this application. Each exposition room is equipped with a sensor and a widget (sensor's wrapper that collects and maintains its transient data). At the main entrance, visitors are provided with a PDA running *TourApp* and a tag (e.g., RFID) that serves to sense their location.

Visitors begin the tour at the registration booth. The booth's sensor detects the visitor's tag, and this information is packaged and sent to the visitor's PDA by a *registration widget*. A registration form pops up on the visitor's PDA, where they provide contact information and key words of interest. After the registration is completed, visitors subscribe to the services provided by the exhibition such as

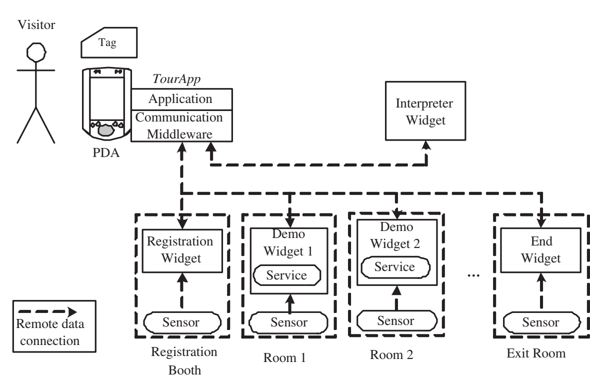


**Figure 1. Deployed infrastructure for** *TourApp*

**Table 1.** *TourApp* **Contextual Information**

| Context Type | Context Value | Action |
|---|---|---|
| Location | Registration | Pop-Up form |
| Location | Demo Room 1 | Display Talk Information |
| ... | ... | ... |
| Power | Low | Minimize Updates |
| ... | ... | ... |
| Time | Talk begins | Display Announcement |
| ... | ... | ... |

alerts on presentations or on visitors with similar interests found in the vicinity. When a visitor enters a room, the room's sensor detects the visitor's tag, and the corresponding *demo widget* notifies the visitor's PDA about the new location. Within the PDA, the communication middleware processes the data and updates the PDA display with the current demo information (e.g., title, duration, current demo slide, a list of colleagues in the room).

Other contextual inputs can affect the application behavior as well. For example, when the PDA's power becomes low, *TourApp* will only display the demo title. There is also a *demo widget* that will provide a *service* to query visitors' interest level on the current demo and notify their PDAs to dynamically adjust future recommendations. A visitor can also explicitly query the *Interpreter widget* about the demos available based on the visitor's interest, interest level for each demo and location. Visitors finish their tour when they reach the exit room, at which point their services are terminated. A sample of the contexts, context values, and actions included in this application are presented in Table 1.

As previously mentioned, middleware plays a large role in this type of application to ease the tasks of accessing and processing contextual information. Examples of supporting middleware infrastructure include the Context Toolkit [16], Context-Phone [18], CARISMA [1], and Gaia [15]. The underlying architecture supporting *TourApp* based on the Context Toolkit and depicted in Figure 2 makes the reliance on such middleware quite evident.

A visitor subscribes through *TourApp* to the services provided by a group of widgets by calling *subscribeTo*, which

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 **IEEE**
IEEE COMPUTER SOCIETY

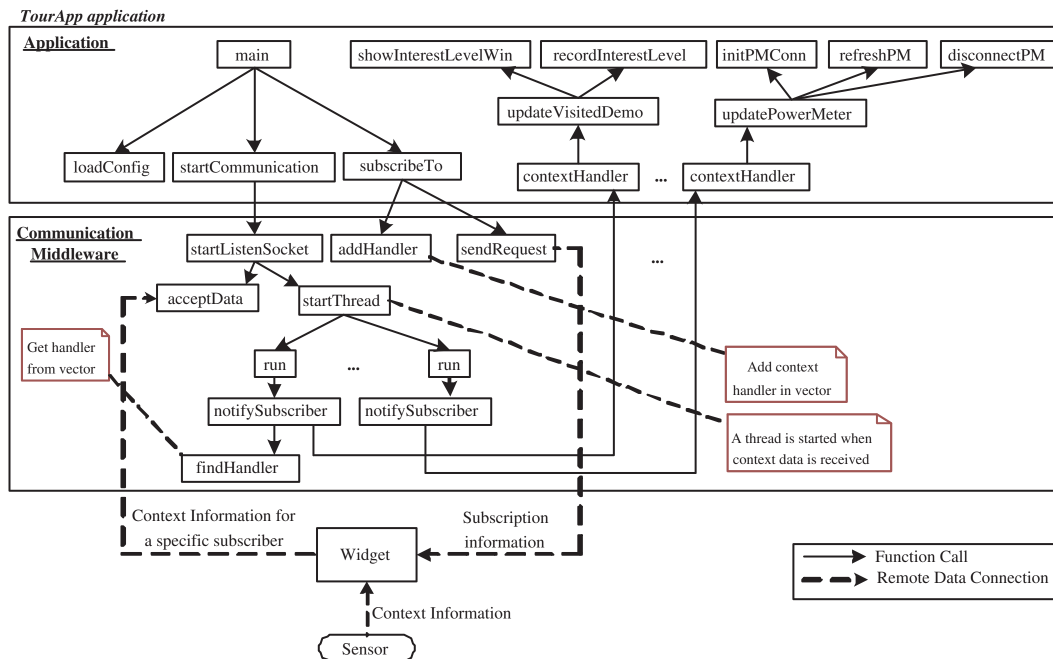


**Figure 2. Simplified call graph of** *TourApp*

then records within the middleware a reference to the proper application context handler and sends the subscription information to the corresponding widget. When context information reaches a subscriber, the middleware launches a thread that calls *notifySubscriber*, which propagates the context data to the proper application context handler.

Observe that there are several attributes distinguishing the context data and necessary infrastructure in this example from more traditional applications and their inputs:

- The behavior of the application depends on streaming and unpredictable contextual inputs such as location and neighbors, whose complexity is abstracted by the middleware.
- Location sensors may feed data to the application at any time and at different rates. Furthermore, sensors may propagate overlapping, incomplete, and contradictory data sets through the system [5].
- Event handlers are likely to operate asynchronously to process the context changes. This often requires multi-threaded support, and may lead to context interactions and races within the same context handler, or between different context handlers.

These observations have strong implications for testing. First, *it is challenging to anticipate all the relevant context changes and when they could impact the behavior of this type of application*. Program-derived testing models such as those based on control- or data-flow must be extended to consider contextual data as well as the concurrency issues introduced by pervasive multi-threaded context handlers. Second, if we are to feed continuous data to a context-aware application, then *we must be able to exercise more control on the executing application*.

## 3 Incorporating Context-Awareness

Given a program $P$ and its test suite $T$, our approach enhances $T$ by manipulating $P$ during the execution of each test case $t \in T$ with the objective of forcing the exploration of potentially interesting contextual scenarios. The underlying assumption is that, given the proper manipulation mechanisms, the potential of $T$ to explore much more of $P$'s behavior can be increased.

The approach's novelty consists in the integrated application of existing analysis techniques to identify and control what contextual scenarios to explore. We now present the approach in terms of its supporting infrastructure depicted in Figure 3. The infrastructure consists of the following components:

- **Context-aware program points (*capps*) Identifier**. This component aims to identify program points where context changes may affect the application's behavior.
- **Context Driver Generator**. This component forms potential context interleavings that may be of value to fulfill a context-coverage criterion.
- **Program Instrumentor**. This component incorporates a scheduler and *capp* controllers into the application to enable direct context manipulation.
- **Context Manipulator**. This component attempts to expose the application to the enumerated context interleavings through the manipulation of the scheduler.

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

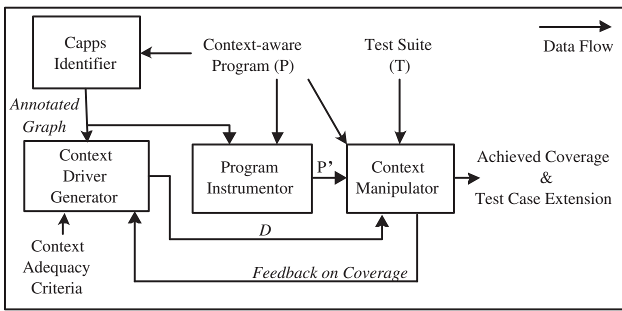


**Figure 3. Overview of our testing approach**

### 3.1 Capps Identifier

The identifier aims at recognizing program locations where context changes may impact the application's behavior. It requires two inputs: the application source and a list of signatures for the context-handling methods defined by the middleware API. Sample code of the *TourApp* application and its supporting middleware is presented in Figure 4, which corresponds to parts of the call graph presented in Figure 2; the middleware code is included for completeness purposes but is not subject to our analysis. In this example, the application must implement the *Subscriber* interface, which has a method whose signature is *Subscriber: void contextHandler(ContextData)*.

With these inputs in place we proceed to identify two types of $capps$: 1) statements dependent on reading or writing context data object fields, and 2) statements reading or writing interfered objects, that is, objects shared with other context-handling threads. Identifying this latter kind of *capp* is important because the majority of context-aware applications run in a multi-threaded environment to handle asynchronous context change events [16, 19, 25]. We use two known program analysis techniques to recognize the *capps*: *side-effect analysis* and *escape analysis*. These techniques have been implemented, with some variations, in freely available toolkits. Our infrastructure utilizes two of those toolkits: Soot [13] and Indus [6] .

To identify *capps* corresponding to statements that depend on reading or writing context data objects, we utilize *side-effect analysis*; more specifically we have instantiated the approach given by Le et al. [7] to just focus on the contextual data types. For our example, this results in all the statements that may be affected by *ctxData*, that is, 12, 14, 21, 27, and 29 ($capps$ #1, 2, 3, 5, 6 respectively).

To identify *capps* originating from multithreading context handlers, we utilize *escape analysis* starting from each context handler. We concentrate on detecting *globally* shared object variables that can be accessed while executing any other event handler thread in the program by instantiating the approach given by Ranganath et al. [12]. For *TourApp*, this process results in the following statements

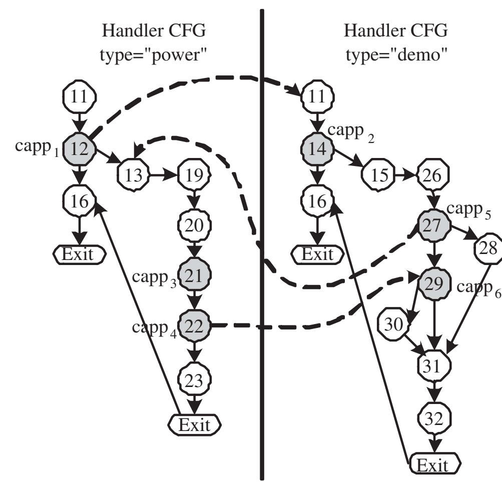


**Figure 5. CFGs of two types context handlers**

identified as $capps$: 21, 22 ($capp$ #4), 27, and 29.[1]

The outcome of this process is a set of interprocedural control flow graphs where certain nodes are annotated with $capps$ *id*s. Figure 5 illustrates the CFGs for the two types of context handlers in the example application, power and demo (room location). The gray nodes in the graph are $capps$, which are generally annotated with unique *id*s (the exception being when they are part of a synchronized section of code, in which case they receive the same *id*).

This information on its own can be valuable for testers to better understand when context changes may matter for the program, which might not be intuitive in the presence of complex flows and interference between context handlers. Still, the number of *capps* can be large, and the potential interleavings of context changes worth testing could be even larger. In addition, testers still lack support to exercise the context changes considered worthy. The next components start addressing these issues.

### 3.2 Context Driver Generator

Once the *capps* have been identified, we would like to explore the context scenarios that are likely to generate different program behavior, and hence are worth exploring. This exploration consists of traversing the CFGs to generate a sequence of nodes that we call *drivers* since they will be used to drive the test execution at a later phase.

Drivers can be generated through the traversal of a single CFG. For example, in Figure 5, a traversal of the CFG corresponding to *power* could generate the driver $\{capp1, capp3, capp4\}$. More interestingly, however, are the traversals across multiple CFGs, because they explore

[1] We point the reader interested in these techniques to the work of Le et al. and Ranganath et al. [7, 12] and for their detailed application to our example refer to our technical report [24].


IEEE COMPUTER SOCIETY

Application

```
   class MySubscriber implements Subscriber {
1    public static Integer  power = new Integer(100); /* global power value, initially 100% */
2     /* some other variable declaration here */
3
4    public static void  main(String[] args){
5      CommMid cm = new CommMid();
6      Widget widget = cm.loadConfig(); /* load info of remote widgets */
7      cm.startCommunication();    /* call startListenSocket() in cm */
8      cm.subscribeTo(this, widget);    /* widget contains address information */
9    }
10   public void  contextHandler (ContextData  ctxData ){
11     System.out.println("new context data is received");
12     if( ctxData .type.equals("power")) /* read ctxData.type */   } capp #1
13          updatePowerMeter(  ctxData );
14     else if(  ctxData .type.equals("demo")) /* read ctxData.type */   } capp #2
15          updateVisitDemo(  ctxData );
16      System.out.println("context has been handled");
17   }
18   public void updatePowerMeter(ContextData     ctxData ){
19     PowerMeter pm = new PowerMeter();  /* the object related to the power meter */
20     pm.initPMConn();  /* initialize power meter connection before update display*/
21     MySubscriber.power = new Integer(    ctxData .value);   /* read ctxData.value */ } capp #3
22     pm.refreshPM(MySubscriber.power);     /* power escaped */  } capp #4
23     pm.disconnectPM(); /* release  power meter connection */
24   }
25   public void updateVisitDemo(ContextData     ctxData ){
26     System.out.println("You are in demo room");
27     if( ctxData .value.equals("demo1") && MySubscriber.power.intValue()>30)  } capp #5
28          System.out.println("A very long description of demo1:    ......");
29     else if(  ctxData .value.equals("demo2") && MySubscriber.power.intValue()>30) } capp #6
30          System.out.println("A very long description of demo2:    ......");
31     showInterestLevelWin();     /* ask for visitor  interest level on demo */
32     recordInterestLevel();    /* record visitor  interest on device */
33   }
34   ...... /* some other methods definition here */
35 }
```

Communication Middleware

```
interface  Subscriber  { /* interface for client application */
  public void contextHandler(ContextData ctxData);
}
class CommMid  {
  private Vector v = new Vector(); /* vector used to store subscriber */
  public void subscribeTo(Subscriber sub, Widget widget){
    addHandler(sub);   /* add sub to vector v */
    sendRequest(sub, widget); /* send subscription to widget */
  }
  public void startListenSocket(){
    ServerSocket s=new ServerSocket(PORT);
    while(true){
          Socket dataSocket = s.accept();     /* receive context info */
          DataThread dt = new DataThread(dataSocket, this);
          dt.start();   /* start a thread to process new context */
  }
  public void notifySubscriber(ContextData ctxData){
    Enumeration e = v.elements(); /* find handlers */
    while (e.hasMoreElements()) { /* call contextHandler in app */
          ((Subscriber)e.nextElement()).contextHandler(ctxData);
    }
  }
  ......  /* some other methods definition here*/
}
class DataThread  extends Thread  {
  public Socket s;
  public CommMid cm;
  public DataThread(Socket s, CommMid cm){
    this.s=s;
    this.cm = cm;
  }
  public void  run()  {  /* ContextData contains "type" and "value" */
    .....  /* retrieve context data from socket (s) here */
    ContextData ctxData = new ContextData();
    ......   /* pack received data into ctxData here*/
    cm.notifySubscriber(ctxData);      } }
```

**Figure 4. Simplified application adapted from** *TourApp*

scenarios that include switches between context handlers, which are more likely to exercise potential interactions between contextual data. For example, the dotted lines in Figure 5 describe a scenario that starts with the *power* context handler, switches to the *demo* handler, returns to the *power* handler, and returns again back to the *demo* handler. Drivers $\{capp1, capp2\}$, $\{capp5, capp3\}$, and $\{capp4, capp6\}$ are interesting because they may expose faults such as the PDA's display not being adjusted according to the power level. These faults are associated with the improper synchronization of context handlers or the poor management of conflicting data reported by the sensors.

Even for programs with a few context handlers and *capp*s, the number of interesting drivers grows quickly. Hence, we utilize a constraining mechanism that can be set to generate drivers that fulfill various coverage adequacy criterion. We explore three such criteria in this work (Taylor et al. and Tse et al. propose other potential ones [20, 22]). The baseline criterion we explore is ***CA*: Context-Adequacy**, which requires the existence of a set of drivers that covers at least one *capp* in each type of context handler. This criterion is valuable because it aims to expose all types of contexts under execution. In our example, having a set of drivers exercising the *power* CFG and the *demo* CFG would make the suite *CA*-adequate.

The second criterion we explore, ***StoC*-$k$: Switch-To-Context-Adequacy**, requires the existence of drivers that cover all possible combinations of $k$ switches between handlers, where each switch can occur at any *capp*. This criterion aims to expose faults that may appear through the interactions between context handlers. In our example, a *StoC*-1 adequate set of drivers is $D$=$\{\{capp1, capp2\}$, $\{capp5, capp3\}$,$\{capp3, capp3\}$, $\{capp5, capp5\}\}$ (with these last two drivers corresponding to a context handler being preempted by one of the same type before ending its execution).

The third criterion, ***StoC*-$k$-*FromCapp*: Switch-With-Preempted-Capp-Adequacy**, requires the existence of drivers that cover all possible combinations of $k$ switches between handlers, and that each switch is exercised at each *capp* in the preempted context handler. This criterion not only exposes the interactions between context handlers, but also aims to explore where such interactions take place. In our example, the demo handler could be preempted by the power handler at *capps*# 1, 3, and 4, while the power handler could be preempted by the demo handler at *capps*# 2, 5, and 6, A *StoC*-1-*FromCapp* adequate set of drivers for our example is $D$=$\{\{capp1, capp2\}$, $\{capp3, capp2\}$, $\{capp4, capp5\}$; $\{capp2, capp3\}$, $\{capp5, capp1\}$, $\{capp6, capp3\}$, $\{capp3, capp3\}$, $\{capp5, capp5\}\}$.



IEEE COMPUTER SOCIETY

**Algorithm 1** StoC-k-FromCapp-DGen($ch_CFGs[], k$)

***Input***: $ch_CFGs[]$: family of interprocedural CFGs (rooted in context handlers) annotated with *capp* id; k: number of switches;
***Output***: $D$: set of *capps* drivers.

1: $S = genAllkSwitchSequence(k, ch_CFGs)$
2: $v[1..k] = storeAllCappsInVectors(k, ch_CFGs)$
3: **while** $hasUncoveredSwitchSeq(S)$ **or** $hasUnmarkedCapps(v[1..k])$ **do**
4: $driver = \{ \}$
5: $curVId = 1$
6: $curSwitchSeq = nextUncoveredSwitchSeq(S)$
7: **if** $curSwitchSeq$ is $NULL$ **then**
8: $curSwitchSeq = getAnySwitchSeq(S)$
9: **end if**
10: $curCFG = nextCFG(init, curSwitchSeq)$
11: $curCapp = nextUnmrkdCapp(curCFG, v[curVId])$
12: **repeat**
13: **if** $curCapp$ not $NULL$ **then**
14: $driver = driver + curCapp$
15: **if** $curVId <= k$ **then**
16: $markCappInV(curCapp, v[curVId])$
17: **end if**
18: $curVId$++
19: $curCFG$=$nextCFG(curCFG, curSwitchSeq)$
20: $curCapp$=$nextUnmrkdCapp(curCFG, v[curVId])$
21: **else**
22: $curCapp = getAnyCapp(curCFG)$
23: **end if**
24: **until** *curSwitchSeq* is satisfied
25: **if** $markedNewCappFrom(driver, v[1..k])$ **or** $coveredNewSwitchSeq(driver, S)$ **then**
26: $D.add(driver)$
27: **end if**
28: **if** $coveredNewSwitchSeq(driver, S)$ **then**
29: $S.markCovered(curSwitchSeq)$
30: **end if**
31: **end while**

Algorithm 1 illustrates how the *StoC-k-FromCapp* driver generator produces $D$.[2] Statement 1 generates the set $S$ of all possible context handler switches required by the chosen criterion. Statement 2 initializes $k$ vectors $v[1..k]$ to keep track of the *capps* preempted by each switch. The rest of the algorithm attempts to generate drivers that exercise the sequences in $S$ and the *capps* in $v[1..k]$; whenever there are uncovered switch sequences or unmarked *capps*, a driver is generated (statements 3 to 31). In statements 4 to 11, the algorithm initializes the driver, selects a target sequence from $S$ called $curSwitchSeq$, gets the CFG of the first handler in $curSwitchSeq$, and selects a *capp* in the CFG. In statements 12 to 24, a driver is generated according to $curSwitchSeq$ by repeatedly concatenating $curCapp$ in $curCFG$ to $driver$. When feasible, the algorithm assigns an unmarked *capp* to $curCapp$. Once the $curSwitchSeq$ is satisfied, statements 25 to 30 check whether the generated driver is unique (either because it covers a new sequence in which case the sequence is marked as covered, or because it covers an existing sequence with the handler preemption occurring at distinct *capps*) and if is, it gets added into the set of drivers $D$.

The outcome of this phase is a set of drivers *D* that, if executed, would achieve a desired context-aware coverage adequacy criterion. Note that the proposed drivers represent one in a set of many potential sequences of context changes that may also meet that adequacy criterion. Furthermore, it may not be feasible to execute a generated driver due to program constrains (e.g., some handlers may be completely synchronized) or test suite limitations (e.g., the original test suite may lack tests that exercise certain contexts). The generated drivers are just guides used by the following phases to explore particular program executions.

## 3.3 Program Instrumentor

The instrumentor takes $P$ and adds to it two new methods, *enterDScheduler* and *exitDScheduler*, and an invocation to each of those methods before and after each *capp*, to produce $P'$. Pseudocode 1 illustrates an example of the instrumentation. Before each *capp*, *enterDScheduler* will be invoked to determine whether it is appropriate (according to a target *D*) to execute the next *capp*. If it is not, then the current thread of execution corresponding to a context handler will wait until its turn comes. Otherwise, the *capp* will be executed and *exitDScheduler* will mark the *capp* as "executed" and notify the waiting handlers.

Pseudocode 1: Instrumentation pseudocode on *capps*

```
/* Added by Instrumentor */
/* Ask scheduler if next capp can be executed */
pos = enterDScheduler(threadId, cappId, D[i]);
/* Original statement */
capp #1;
/* Added by Instrumentor */
exitDScheduler(pos); /*notify other capps to execute */
```

## 3.4 Context Manipulator

The manipulator takes $P'$, $D$ and $T$ as inputs, and runs each test case $t$ on $P'$ while attempting to drive $t$ towards the *capps* corresponding to the scenarios of interest contained in *D*. This is expected to result in the execution of each original test case under multiple schedules that expose multiple contextual scenarios.

The manipulator consists of three methods described by Algorithms 2, 3, and 4. Algorithm 2, which invokes Algorithm 3, forces the current context handler thread to *wait()* until the execution of the previous *capps* specified in the

[2] Due to space constrains we do not discuss how this algorithm deals with several exceptional cases, and do not present the algorithm for *StoC-k* which is a simpler version of Algorithm 1.



IEEE COMPUTER SOCIETY

driver are completed (with $capps$ not in the driver being ignored). Since it is possible for a driver to contain unreachable context interleavings, the manipulator discards such a driver when a parameterizable $timeout$ is reached. (Section 4 provides further details about how to adjust this setting within our infrastructure.) Algorithm 3 checks whether a *capp* can be executed by inspecting whether it is specified in the driver, and if it is, by checking that all previous *capps* in the *driver* have been executed. In Algorithm 4, the "executed" flag of the current executed *capp* is set to true and the other handlers are notified so that they can check whether it is their turn to continue with the execution.

The act of discarding drivers (due to unreachability or any other reason) may stop the coverage criterion from being satisfied. The manipulator then reports the unreachable drivers and coverage information (e.g., the switches currently not covered) to re-launch the context driver generator with a refined set of targets. The generator then uses this feedback to produce an alternative $D$ aiming to address that potential weakness. The new $D$ can then be re-scheduled by the manipulator, leading to an iterative process that stops when a specified condition is met (e.g., coverage percentage is reached, or testing resources are exhausted).

**Algorithm 2** [sync]enterScheduler($tID$,$cappID$,$D[i]$)

***Input***: $tID$: ID of current thread; $cappID$: ID of current *capp*; D[$i$]: a driver;
***Output***: Position of ($tID$,$cappID$) in $D[i]$.

1: *pos* = checkScheduler($tID$, $cappID$, $D[i]$)
2: **while** *pos* == -1 **do**
3: wait(*timeout*)
4: **if** timeout occurs **then**
5: Log "timeout" for feedback to generator
6: exit(); /* exit the program */
7: **end if**
8: *pos* = checkScheduler($tID$, $cappID$, $D[i]$)
9: **end while**
10: return *pos*

**Algorithm 3** [sync]checkScheduler($tID$,$cappID$,$D[i]$)

***Input***: $tID$: ID of current thread; $cappID$: ID of current *capp* in driver $D[i]$.
***Output***: Position of ($tID$,$cappID$) in $D[i]$. If the position is positive or -2, it is time for the current *capp* to execute.

1: *pos* = locate the position of ($tID$,$cappID$) in $D[i]$
2: **if** ($tID$,$cappID$) not in $D[i]$ **then**
3: return -2 /* *capp* will be skipped */
4: **end if**
5: **if** all *capps* before *pos* are executed **then**
6: return *pos*
7: **else**
8: return -1 /* *capp* will wait */
9: **end if**

**Algorithm 4** [sync]exitScheduler(*pos*)

***Input***: *pos*: Position of ($tID$,$cappID$) in driver D[$i$].

1: setExecution(*pos*, true) /* mark an "executed" *capp* */
2: notifyAll()

# 4 Preliminary Assessment

In this section we perform an assessment of the proposed approach in terms of its applicability and its potential to enhance an existing test suite.

## 4.1 Study Design and Settings

We utilize *TourApp*, described previously in the paper, as our object of study [16]. *TourApp* has 11KLoc of Java (about 90% of the application being the middleware). We create test cases derived from scenarios included in the Context Toolkit Installation Guide to serve as the baseline validation. We then extended the test suite with scenarios that include changes of location (registration booth, 2 demo rooms, exit room), visitor's interests (demos, applications, virtual environment) and interest levels (low, middle, high). We developed a total of 36 automated test cases, which take approximately 10 minutes to execute on a 1.6GHz Pentium CPU and 1GB RAM running Java 1.4 (the same platform being used for the whole assessment).

We created an instrumented version of *TourApp* that we call *manipulatedTourApp*, which includes probes to manipulate the *capps* and our context manipulator algorithms. We set our generator to provide drivers aiming to satisfy four context criteria: *CA*, *StoC-1*, *StoC-2*, *StoC-1-FromCapp*, and *StoC-2-FromCapp*. We set the *timeout*, used to stop the manipulator when pursuing a given driver, to 30 seconds, which is the approximate maximum time it took for any test case to execute.

We also created two additional versions of *TourApp* called *delayShortTourApp* and *delayLongTourApp*. These versions were instrumented to implement an approach similar to that performed by the *Contest* tool [4], where a random delay is introduced before shared variables and synchronizing-type structures to add "noise" to the scheduler so that alternative interleavings are explored. The implementation of this approach consisted in inserting *sleep()* at the beginning of each context handler in *TourApp*. The inserted *sleep()* calls randomly choose delays ranging from 1 to 3 seconds for *delayShortTourApp*, and from 1 to 10 seconds for *delayLongTourApp* (the delay values being derived empirically by trial and error until further variations were not observed).

Last, we identified four contextual scenarios that could cause *TourApp* to fail. All failures were related to event se-


0-7695-2828-7/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

quences propagated through the application in the wrong order, due to either lack of proper synchronization primitives or missing handler exceptions. For example, if the visitor is detected by a sensor in a demo room before completing the registration, the demo sensor would trigger a set of messages that would not match the visitor's interest unless the application was ready to deal with such an exception. We used these scenarios to further assess the approaches.

After preparing the test suite and the various versions of *TourApp*, we performed the following steps:

**1.** For *manipulatedTourApp*, each test case in the original test suite was executed on *manipulatedTourApp* under the guidance of each relevant generated driver. We consider a driver to be relevant to a test case when all the types of contexts in the driver are exercised by the original test case. For instance, if a test case does not provide coverage for the context of type $power$, then it is irrelevant for any driver that includes $power$. We repeat this process for each coverage criterion while calculating the execution time under each one of them, keeping track of the number of drivers that could not be completed by the manipulator.

**2.** For *originalTourApp* (unmodified *TourApp* with original test suite), *delayShortTourApp*, and *delayLongTourApp*, each application was executed repeatedly for as long as it took for the *manipulatedTourApp* to complete execution under the same adequacy criterion. By setting the same upper time bound, we can effectively compare performance across approaches.

**3.** We compared the contextual coverage achieved and the exposed contextual scenarios that led to failures by *manipulatedTourApp* versus those from *TourApp*, *delayShortTourApp*, and *delayLongTourApp*.

### 4.2 Results and Analysis

Table 2 summarizes the number of drivers and execution times of *manipulatedTourApp* under the coverage criteria. As expected, execution time increases with more demanding context coverage criteria as more context scenarios are required. It is interesting to note that even for the *CA* criterion, *manipulatedTourApp* took over three times as long (34m versus 10m) as the original test suite , which was also *CA*-adequate. Still, there are many techniques that we have not yet explored that could improve the efficiency of the current prototype.

Table 3 provides evidence of the large number of generated drivers that were not exposable by the application within the set timeouts, which resulted in the manipulator wasting exploration of those spaces. Improvements in the generation of $D$, such as through the use of flow information, would reduce the number of such drivers being generated, leading to a shorter manipulation time to satisfy the coverage criterion.

**Table 2. Timings with** *manipulatedTourApp*

| Criteria | Generated Drivers | Execution Time (minutes) |
|---|---|---|
| *CA* | 4 | 34 |
| *StoC-1* | 16 | 168 |
| *StoC-2* | 28 | 295 |
| *StoC-1-FromCapp* | 46 | 546 |
| *StoC-2-FromCapp* | 57 | 670 |

**Table 3. Drivers in** *manipulatedTourApp*

| | *CA* | *StoC-1* | *StoC-2* | *StoC-1-FromCapp* | *StoC-2-FromCapp* |
|---|---|---|---|---|---|
| Exposed | 4 | 15 | 25 | 45 | 51 |
| Timed out | 0 | 1 | 3 | 1 | 6 |

More important than efficiency, however, is whether the approach is able to expose valuable contextual scenarios. Table 4 provides the coverage achieved by each approach in the time taken by *manipulatedTourApp* to complete execution when using the same criterion. Table 4 also reports on the percentage of failure-leading contextual scenarios exposed through each approach.

We note that all approaches achieve 100% *CA* coverage. However, the coverage percentage decreases as we move to more exhaustive coverage criteria. *OriginalTourApp* is the one that declines quickest, providing less than 20% coverage at the *StoC-2* level. The approaches utilizing delay provide approximately 31% more coverage on average in *StoC-1* than *originalTourApp*, but the difference is reduced to 20% at *StoC-2*. Note also the variability in the performance of *delayShortTourApp* and *delayLongTourApp*, which shows the sensitivity of this approach to the chosen delays. *ManipulatedTourApp* performs noticeably better than the rest, especially as the target coverage criterion becomes more powerful. The exposure to contextual scenarios leading to failure also shows similar patterns, with *manipulatedTourApp* detecting all the problematic scenarios at the *StoC-2-FromCapp* level, while the rest of techniques cannot expose more than half of the them.

## 5 Related Work

The most widely used method for the validation of context-aware applications is simulation [17]. These simulation activities are often supported by various frameworks. For example, Satoh used an agent-based framework to simulate physical mobility to support the exposure of specific trajectories that might impact an application executing in a mobile device [17]. Some aspects relevant to context-aware applications have also received attention within the verification community. For example, Mobile UNITY uses formal specification and proof logic to enable the modeling

Table 4. Contextual Coverage (Cov.) and Fault Detection (F.D) in %

| | *CA* | | *StoC-1* | | *StoC-2* | | *StoC-1-FromCapp* | | | *StoC-2-FromCapp* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Cov. | | | Cov. | | |
| | Cov. | F.D. | Cov. | F.D. | Cov. | F.D. | Switches | *Capps* | F.D. | Switches | *Capps* | F.D. |
| *originalTourApp* | 100 | 0 | 38 | 0 | 18 | 0 | 38 | 7 | 0 | 18 | 7 | 0 |
| *delayShortTourApp* | 100 | 25 | 63 | 25 | 29 | 25 | 81 | 10 | 50 | 46 | 10 | 50 |
| *delayLongTourApp* | 100 | 25 | 75 | 25 | 46 | 25 | 75 | 10 | 50 | 43 | 7 | 25 |
| *manipulatedTourApp* | 100 | 0 | 94 | 75 | 89 | 75 | 94 | 100 | 75 | 88 | 93 | 100 |

and verification of movement, transient data sharing, and transient action synchronization in mobile computing [14].

The appearance of testing approaches for context-aware applications is more recent. Tse et al. applied a technique for test case generation based on metamorphic testing to a context-aware application [22]. The technique requires the definition of metamorphic properties that serve as oracles. If any tests' outputs violate a specific metamorphic property, then it is presumed that the program must contain a fault. More recently, the same group developed an initial suite of data-flow type coverage criteria that considers some contextual events and their associated actions [9]. These efforts are parallel to ours, primarily targeting the definition and assessment of an initial context-aware test suite, without considering the problem of automatically identifying *capps* or the manipulation of the program execution to expose interesting conflicting scenarios.

Our approach is also related to a wide spectrum of validation techniques for concurrent systems (e.g., [2, 4, 8, 20]) aiming at the detection of concurrency faults, in particular execution sequences and schedules. In general, these techniques can be classified in two groups: those that sample over non-deterministic runs, and those that attempt to create specific deterministic runs. The first class of solutions involves executing the program repeatedly over the same inputs in the hope of exercising a reasonable percentage of the possible synchronization events. The *ConTest* tool, for example, inserts random perturbations (e.g., *sleep()*) around concurrency related structures in the program to induce interleavings of threads that were not manifested with the original test suite [4]. This approach is relatively inexpensive to put in place, but it cannot guarantee that even a reasonable subset of the interesting scenarios are exercised.

The second class of solutions (e.g., [2, 20]) deterministically replays a chosen set of synchronization sequences. This approach generally requires specific tool support, and its effectiveness is dependent on the tester's selection of sequences. Our approach belongs to this latter group, which aims to provide a somewhat deterministic program execution. However, our focus is on the specific execution model of context-aware programs as defined by the *capps*, and on how these applications react to arbitrary *external* context changes, instead of dealing with more standard concurrency primitives such as semaphores and queuing mechanisms or synchronization constructs, which typically govern behavior *internally*. Furthermore, our approach provides integrated support for the automatic enhancement of an existing test suite, which requires minimal tester participation.

Our approach also shares elements with model checkers, which systematically explore the scheduling state space of concurrent systems. Both approaches statically analyze the source code to identify program points of interest and then force the execution through some sequence of points to exhibit potential interesting behavior. Our approach, however, trades the soundness of model checkers to verify a set of properties, for the preciseness of testing and the practicality of utilizing an existing test suite to incorporate potentially interesting scheduling variations (as opposed to generating the necessary environment to check a system module [21].)

Last, the initial phase of our approach is also related to several efforts aiming at creating models from which useful test cases can be derived. For example, Memon's techniques for event-based testing targeting *GUI* event handlers utilize event flow graphs to explore the allowable event sequences in the event-based program [11]. However, the rest of our approach to manipulate the execution toward the contextual scenarios of interest requires mechanisms that are not needed for the more constrained domain of GUI events.

## 6 Conclusion and Future Work

We have presented an approach to enhance the test suites of context-aware applications. The approach is novel in that it provides an integrated solution to identify when context changes may be relevant, and a control mechanism to guide the execution of given tests into potentially interesting contextual scenarios as defined by a coverage criterion that is context-cognizant. Preliminary assessment of the approach revealed that it can effectively enhance an existing test suite, providing exposure to a larger set of interesting and valuable context scenarios than alternative testing practices.

We are in the process of addressing several limitations of the approach. First, we are further integrating pieces of the supporting infrastructure and improving their efficiency. For example, we are exploring a closer integration of the driver generator and the manipulator so that they both operate online. We expect that this will result in a reduction



IEEE COMPUTER SOCIETY

of the number of infeasible drivers generated. We are also revisiting the selected static analysis tools, which are quick and somewhat scalable but quite conservative and thus lead to the identification of extra *capps* or the generation of infeasible drivers as well. Second, we are incorporating additional adequacy criteria. Although our focus has been primarily on the adequacy of the generated drivers, we are starting to explore a range of criteria that emphasize the association between elements defined in the middleware and used by the application. Third, we are aware that we need to provide a more comprehensive assessment of the approach. *TourApp* is just a first step, and we must investigate how our approach handles the complexities associated with larger applications that consider other contextual events such as connectivity or proximity. Last, we are extending our approach to consider not just the types and values of context provided by a given test suite, but also to include information from other sources such as the simulation runs that are often used to validate context-aware application models, and that can also serve to provide additional values and to compare the outcomes of the generated test cases.

## Acknowledgments

This work was supported in part by NSF CAREER Award 0347518, by the ARO-DURIP award W911NF-04-1-0104, and by the EPSRC under grants EP/D077273/1 and EP/E006191/1. D. Rosenblum holds a Wolfson Research Merit Award from the Royal Society. We are thankful to A. Dey for providing the Context Toolkit, and to the research groups making Soot and Indus publicly available.

## References

[1] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):929 – 945, Oct 2003.

[2] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, Mar 1991.

[3] A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on the What, Who, Where, When and How of Context-Awareness*, pages 304 – 307, Jun 2000.

[4] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[5] K. Henricksen and J. Indulska. Modelling and using imperfect context information. In *Pervasive Computing and Comm. Workshops*, pages 33 – 37, Sep 2004.

[6] S. Lab. Indus. http://indus.projects.cis.ksu.edu/.

[7] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in jit optimizations. In *Conference on Compiler Construction*, volume 3443, pages 287–304, Apr. 2005.

[8] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering*, 29(6):555–566, Jun 2003.

[9] H. Lu, W. Chan, and T. Tse. Testing context-aware middleware-centric programs: a data flow approach and a rfid-based experimentation. In *Symposium on Foundations of Software Engineering*, pages 242 – 252, Nov 2006.

[10] C. Mascolo, L. Capra, and W. Emmerich. Middleware for mobile computing. In *International Conference of Networking*, pages 20–58, May 2002.

[11] A. M. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Working Conference on Reverse Engineering*, pages 260 – 269, Nov 2003.

[12] V. Ranganath and J. Hatcliff. Pruning interference and ready dependence for slicing concurrent java programs. In *Conference on Compiler Construction*, pages 39–56, Mar 2004.

[13] S. research group. Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[14] G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile unity: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering Methodologies*, 6(3):250 – 282, Jul 1997.

[15] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: a middleware platform for active spaces. *IEEE Mobile Computing and Communications*, 6(4):65 – 67, Oct 2002.

[16] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *Conference on Human Factors in Computing Systems*, pages 434 – 441, May 1999.

[17] I. Satoh. A testing framework for mobile computing software. *IEEE Transactions on Software Engineering*, 29(12):1112–1121, Dec 2003.

[18] A. Schmidt, T. Stuhr, and H.-W. Gellersen. Context-phonebook - extending mobile phone applications with context. In *Mobile HCI Workshop*, Sept 2001.

[19] T. Sivaharan, G. Blair, and G. Coulson. Green: A configurable and re-configurable publish-subscribe middleware for pervasive computing. In *Symposium on Distributed Objects and Applications*, pages 732–749, Oct 2005.

[20] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, Mar 1992.

[21] O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Automated Software Engineering*, pages 116 – 129, Oct 2003.

[22] T. Tse, S. Yau, W. Chan, H. Lu, and T. Chen. Testing context-sensitive middleware-based software applications. In *International Computer Software and Applications Conference*, pages 458–465, Sept 2004.

[23] A. H. van Bunningen, L. Feng, and P. M. Apers. Context for ubiquitous data management. In *Workshop on Ubiquitous Data Management*, pages 17 – 24, Oct 2005.

[24] Z. Wang, S. Elbaum, and D. Rosenblum. Automated Generation of Context-Aware Tests. Technical Report TR-UNL-CSE-2006-0012, University of Nebraska Lincoln, 2006.

[25] S. Yau, F. Karim, Y. Wang, B. Wang, and S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33 – 40, Jul 2002.