University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

CSE Technical reports | Computer Science and Engineering, Department of

2006

Automated Generation of Context-Aware Tests

Zhimin Wang
zwang@cse.unl.edu

Sebastian Elbaum
University of Nebraska - Lincoln, elbaum@cse.unl.edu

David Rosenblum
University College London, London, England

Follow this and additional works at: http://digitalcommons.unl.edu/csetechreports

Part of the Computer Sciences Commons

Wang, Zhimin; Elbaum, Sebastian; and Rosenblum, David, "Automated Generation of Context-Aware Tests" (2006). *CSE Technical reports*. 23.
http://digitalcommons.unl.edu/csetechreports/23

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Technical reports by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

University of Nebraska–Lincoln, Computer Science and Engineering
Technical Report TR-UNL-CSE-2006-0012
Issued Sept. 11, 2006

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 Rosenblum
Dept. of Computer Science
University College London
London, England
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 community for gaming or socializing. Examples of relevant context include location, time of day, environmental readings (e.g. temperature, humidity), and user preferences (e.g. spoken language, spending limits).

Developing such context-aware applications is full of difficulties unique to the nature of ubiquitous systems. Consider the periodic, asynchronous 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 perceive different things or perceive 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 several efforts to support the specification and to some degree the verification of context-aware applications [14], testing context-aware applications remains primarily confined to the physical emulation of the mobile device and the logical and physical deployment to an affordable subset of the user scenarios [17], and to the tailoring of existing test case generation methodologies (testing context-aware middleware [21]).

The proposed testing techniques, however, 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 drivers of contextual applications. Engineers must identify not only *what* context values to provide, but also *when* variations in context values can impact the behavior of the application, and hence are worth testing. This is an essence of this type of application which differentiates it from the testing of more traditional systems, where the selection of input values can mostly be performed apriori.

Determining when and controlling how to feed a stream of changing contexts values to a context-aware application 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 existing an 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 [22], which capture 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 it 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 pop-ups on the visitors' 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 alerts on presentations or visitors with similar interest 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 will process the data and update the PDA display with the current demo information (e.g., title, duration, current slide,

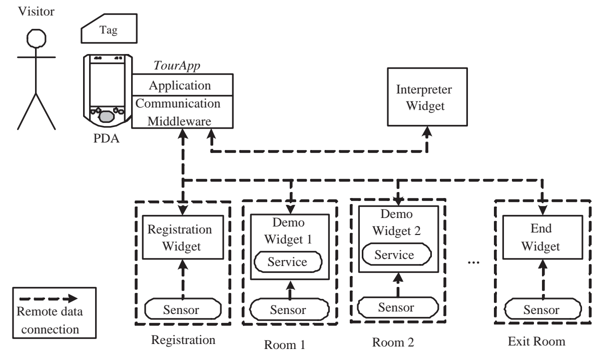


Figure 1. Deployed infrastructure for *TourApp*

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
...	...	...

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 enter 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.

The visitor subscribes through the *TourApp* to the services provided by a group of widgets by calling *subscribeTo*, which will then record within the middleware a reference to the proper application context handler and send the subscription information to the corresponding widget. When context information reaches a subscriber, the middleware will launch an individual thread and call *notifySubscriber*, which will propagate the context data to the proper application context handler.

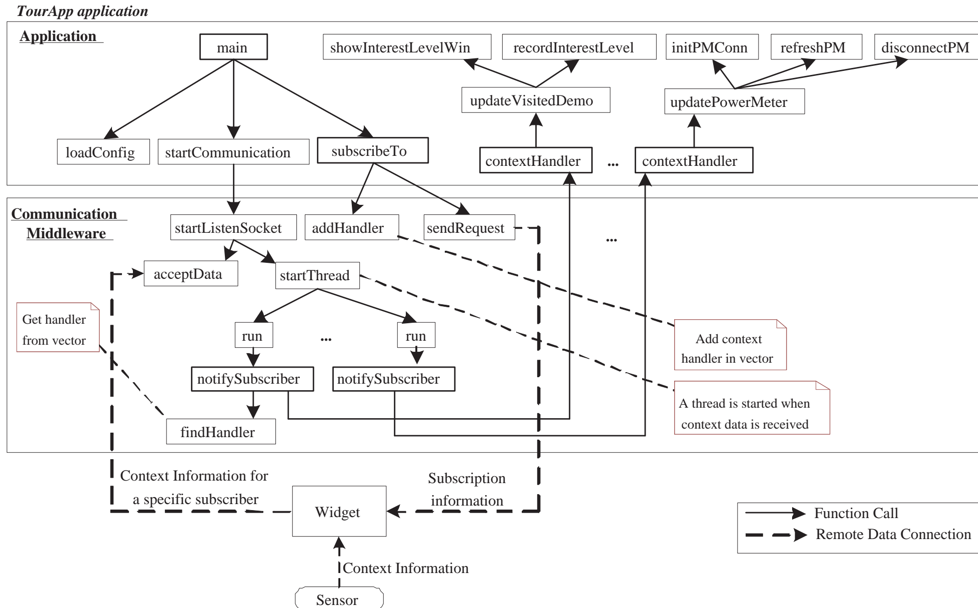


Figure 2. Simplified call graph of *TourApp*

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 contextual inputs such as time, demos available, and location, 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 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 not trivial to anticipate all the relevant contexts changes and when they could matter for 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 *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 extends T by manipulating P during the execution of each test case 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 exploited.

The approach novelty consists in the integrated application of existing analysis techniques to identify and control what contextual scenarios to explore. We will now present the approach within the context 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 the direct context manipulation.
- **Context Manipulator**. This component attempts to expose the application to the enumerated context interleavings through the manipulation of the scheduler.

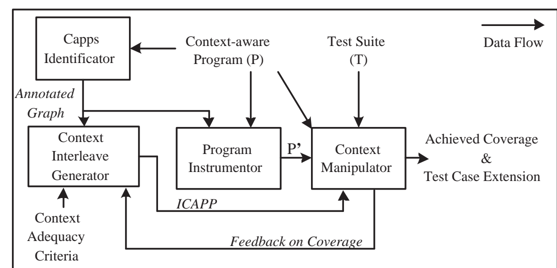


Figure 3. Overview of our testing approach

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 on device */
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 remote 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 received */
   }
   public void notifySubscriber(ContextData ctxData){
      Enumeration e = v.elements(); /* find handlers */
      while (e.hasMoreElements()) { /* call contextHandler in application */
            ((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" strings */
      .....  /* 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*

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 and the signature of 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 signature of the middleware interface indicates that the application must implement the *Subscriber* interface which has a method whose signature is *Subscriber: void contextHandler(ContextData)*.

With these inputs in place we now proceed to identify two types of *capps*: 1) statements depending on reading or writing context data objects, and 2) statements reading or writing interfered objects, that is, objects shared with other content-handling threads. Identifying this latter kind program point is important because the majority of context-aware applications run in a multi-threaded environment to handle asynchronous context change events [16, 19, 23]. 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].

We start by constructing a family of call graphs and interprocedural control flow graph (CFG) from the application. As illustrated in Figure 2, one call graph is rooted in the *main* method while the rest of the call graphs are rooted in the *contextHandler* methods (we used the handler's type to differentiate among them). In addition, we construct interprocedural control flow graphs that are rooted in each type of *contextHandler*. These graphs are meant to assist later phases of the analysis to determine what context changes propagated through the handlers may matter and under which conditions.

To identify *capps* corresponding to statements that depend on reading or writing context data objects we utilize *side-effect analysis*. Side-effect analysis computes the set of objects that could read from/write to each statement and extract dependence relationship between statements; more specifically we have instantiated the approach given by Le et al. [7] to just focus on the contextual data types as follows:

- **Step 1**: For each application statement s generate two sets, $read(s)$ and $write(s)$, containing static fields and the fields of objects (including the ones in parameters and *this*) that could be read and written by statement s.

In *TourApp* example (Figure 4), for method *contextHander*:
$read(s_{11})$=$write(s_{11})$={}; //constant as parameter
$read(s_{12})$={$ctxData.type$}; $write(s_{12})$={};
$read(s_{13})$=$write(s_{13})$={}; // method call
$read(s_{14})$={$ctxData.type$}; $write(s_{13})$={};
$read(s_{15})$=$write(s_{15})$={};

For method *updatePowerMeter*:
$read(s_{19})$=$write(s_{19})$={};
$read(s_{20})$=$write(s_{20})$={};
$read(s_{21})$={$ctxData.value$};
$write(s_{21})$={$this.power$};
$read(s_{22})$={$this.power$}; $write(s_{22})$={}
$read(s_{23})$=$write(s_{23})$={};

- **Step 2**: Aggregate the sets in each method and propagate them to the call sites using the call graphs.
  In *TourApp* example (Figure 4), s_{21} reads the field "*value*" of object "*ctxData*". Using the call graphs, we relate method *contextHander* to method *updatePowerMeter*. Using object references, we know parameter "*ctxData*" in *contextHandler* is the same object with parameter "*ctxData*" in *updatePowerMeter*. These induce $read(s_{12}) \bigcap read(s_{21}) = ctxData$.

- **Step 3**: For each context object construct dependence relations (read-read, read-write, write-read, write-write) between statements.
  In *TourApp* example (Figure 4), from step 2, we know $read(s_{12})$ and $read(s_{21})$ contains a common context data object $ctxData$, s_{12} and s_{21} have read-read dependence on $ctxData$. Overall, in method *contextHander* and method *updatePowerMeter*, the read-read dependence on object *ctxData* is $\{s_{12}, s_{14}, s_{21}\}$. The read-write dependence on object *this* is $\{s_{21}, s_{22}\}$.

- **Step 4**: Mark as *capps* the statements which satisfy those dependence relations on at least one context object.
  In *TourApp* example (Figure 4), the side-effect analysis indicates that the statements that may be affected by *ctxData* (read-read dependence on *ctxData*) are: s_{12}, s_{14}, s_{21}, s_{27}, s_{29}.

To identify *capps* originated from multithreading context handlers, we utilize *escape analysis* starting from each context handler. In this case, escape analysis is to determine whether an object is local to the a thread, i.e. whether other threads can read/write an object. 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] as follows:

- **Step 1**: Associate an escaped object set (esc={}) with each method to record the globally shared objects.

- **Step 2**: Identify the globally shared objects across methods (Java bytecode "getstatic" and "pubstatic" could be used here).
  In *TourApp* example (Figure 4), method *updatePowerMeter* is associated with esc={$power$} because s_{21} writes "*power*" and s_{22} reads "*power*". The same set can be derived from the method *updateVisitDemo* that writes the static field *power*.

- **Step 3**: Union the set of each method in a bottom-top fashion through the call graph and compose an overall set that contains globally shared objects.
  In *TourApp* example (Figure 4), *contextHandler* invokes *updatePowerMeter* and *updateVisitDemo*. Therefore, the esc sets of *updatePowerMeter* and *updateVisitDemo* are combined, which is esc={$power$}. Then tracing back to *contextHandler*, the overall esc={$power$}.

- **Step 4**: Mark as *capps* the statements in the methods rooted in the context handlers that access objects in the overall set.
  In *TourApp* example (Figure 4), method *updateVisitDemo* reads the static field *power* (line 27), so the associated escaped object set for that method is {$power$}. The same set can be derived from the method *updatePowerMeter* that writes the static field *power*. Traversing the call graph in a bottom-top fashion, we union the set associated with each method. For *TourApp*, we get a singleton set {$power$} rooted in the *contextHandler* method and we can mark the following statements as *capps*: s_{21}, s_{22}, s_{27}, and s_{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 is when they are part of an synchronized section of code in which case they would 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 would be even larger. In addition, testers still lack support to exercise the context changes considered worthy. The next components start addressing these issues.

[1] Due to space constrains, we point the reader interested in these techniques to [7, 12]

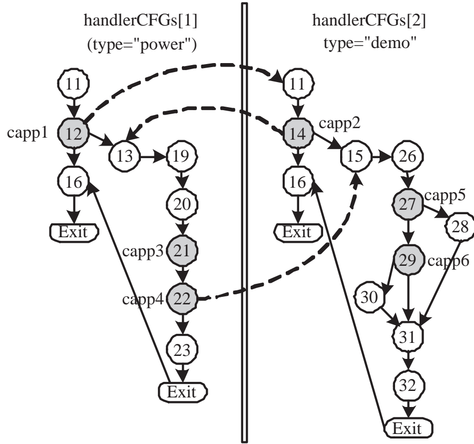


Figure 5. CFGs of two types context handlers

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 in a later phase.

Drivers can be generated through the traversal of a given 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 CFGs because they explore 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* content handler, it switches to the *demo* handler, back to the *power* handler, and again back to the *demo* handler. The generated driver $\{capp1, capp2, capp5, capp3, capp4, capp6\}$ is interesting because it may expose faults such as the display not being adjusted according to the level of power. 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 *capps*, the number of potential 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 two of such potential criteria in this work (see [20, 21] for other potential ones). The baseline criterion we explore is ***CA*: Context-Adequacy**, which requires the existence of drivers that cover at least one *capp* in each type of context handler. For example, a driver that exercises the *power* CFG and a driver that exercises the *demo* CFG would be *CA* adequate. The second criterion we explore, ***StoC*-k: Switch-To-Context-Adequacy**, requires the existence of drivers that cover at least k switches between handlers at any *capp*. For example, a *StoC*-2 adequate set of drivers, D, could include $driver_1$=$\{capp1, capp2, ...\}$, $driver_2$=$\{capp2, capp1, ...\}$, $driver_3$=$\{capp1, capp3, capp1, ...\}$, and $driver_4$= $\{capp2, capp2, ...\}$ (these last two drivers correspond to a context handler being preempted by one of the same type before ending its execution).

Algorithm 1 illustrates how the *StoC*-k driver generator works in producing D. Statement 1 generates the set S with all possible context handler switches required by the chosen criterion. For example, given an application like *TourApp* with the two annotated CFGs from Figure 5, and *StoC-2* as the chosen criterion, the set of required switches to fulfill the criterion is $\{$power to demo, demo to power, power to power, and demo to demo$\}$. The rest of the algorithm traverses the CFGs attempting to generate drivers that exercise the enumerated switches. For each $currSwitch$ in S, we identify the starting *CFG* in $currSwitch$ with the assistance of the $nextCFG$ function with the $init$ as parameter (statement 9), and traverse the *CFG* until a *capp* is found (statement 10). Once a *capp* is found, it is added to the current $driver$, and the algorithm randomly decides (statement 13) whether it is time to switch to the next CFG specified by the switch sequence in the $currSwitch$ or to remain in the current CFG.[2] When a switch is performed, the destination node in $currCFG$ is either its entry node (first time in that context handler) or the node immediately following its last executed *capp* (revisited context handler). This process is repeated until the traversal of the current CFG returns $NULL$ because it has reached an exit node. If the collected $driver$ does not satisfy $currSwitch$, then the process starts again with an empty $driver$. However, if the $driver$ satisfies the sequence in $currSwitch$, then it is added to D (statement 19) and another $currSwitch$ is selected from S. It is worth noting that the traversal in the *CFG* is acyclic, which means we only capture the information of the first loop in a given *CFG* if there exists one. Also note that, if the selected sequence includes switches within the same type of handlers, we duplicate the the corresponding *CFG* (statements 4 to 6). We do this so that all switches are treated in the same way within the repeat loop.

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 sequence of context changes that may also meet that adequacy criteria. Furthermore, the generated driver set may not be feasible 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

[2]Note that, for more stringent coverage criteria, the $timeToSwitch()$ function could be easily replaced without loss of generality.

Algorithm 1 StoC-k-DGenerator($handlerCFGs[]$, k)

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

```
1:  S ← genAllkSwitchSequence(k, handlerCFGs[])
2:  while S is not empty do
3:    currSwitch ← nextSwitchSequence(S)
4:    if currSwitch contains switches among same type then
5:      Duplicate CFGs according to the type
6:    end if
7:    repeat
8:      driver ← empty
9:      currCFG = nextCFG(init, aSwitch)
10:     currCapp=traverseCFGuntilCapp(currCFG)
11:     while currCapp not NULL do
12:       driver = driver + currentCapp
13:       if timeToSwitch() = true then
14:         currCFG=nextCFG(currCFG, currSwitch)
15:       end if
16:       currCapp=traverseCFGuntilCapp(currCFG)
17:     end while
18:   until currSwitch is satisfied
19:   D.add(driver)
20:   S.remove(currSwitch)
21: end while
```

drivers will be later used to guide the program execution toward the target context change sequences.

3.3 Program Instrumentor

The instrumentor takes P and adds to it two new methods, *enterDScheduler* and *exitDScheduler*, and an invocation to each one of those methods before and after each *capp*, to produce P'. Pseudocode 1 illustrates an example of the instrumentation on each *capp* in source code. Before each *capp*, *enterDScheduler* will be invoked at run-time 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 likely execution of each original test case under multiple schedules that expose multiple contextual scenarios.

The manipulator consists of three simple methods contained in Algorithm 2, 3, and 4. Algorithm 2 checks whether a *capp* can be executed by inspecting whether all previous *capps* in a target *driver* have been executed. Algorithm 3, which invokes Algorithm 2, forces the current context handler thread to *wait()* until the execution of the previous *capps* specified in the driver are completed. 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. Since it is possible for some drivers to contain unreachable context interleavings, the manipulator discards an drivers when a parameterizable $timeout$ is reached. (Section 4 provides further details about how to adjust this setting within our infrastructure.)

Clearly, the act of discarding drivers (due to unreachability or any other reason) may stop the coverage criteria from being satisfied. The manipulator should then be smart enough to report the unreachable drivers and coverage information (e.g., the switches currently not covered) to relaunch the context driver generator with a refined set of targets. The generator will then use 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 will stop when a specified condition is met (e.g., coverage percentage is reached, testing resources are exhausted).

3.5 Expected Outcome

Test suites that ignore the contextual input space altogether will miss context scenarios that may reveal unexpected behavior. On the other hand, developing a test suite that incorporates all potential context scenarios is infeasible for applications with many contextual elements (*capps*

Algorithm 2 [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, it is time for the current *capp* to execute.

1: $pos \leftarrow$ locate the position of (tID,$cappID$) in $D[i]$
2: **if** all *capps* before pos are executed **then**
3:     return pos
4: **else**
5:     return -1
6: **end if**

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

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

1: $pos \leftarrow$ checkScheduler($tID, cappID, D[i]$)
2: **while** pos is -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:     $pos \leftarrow$ checkScheduler($tID, cappID, D[i]$)
9: **end while**
10: return pos

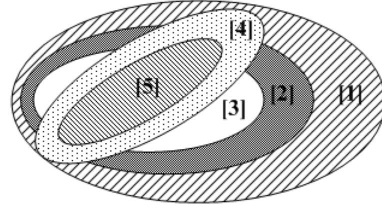
Algorithm 4 [sync]exitScheduler(pos)

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

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

and context handlers). Instead, our approach aims at exposing the target application to a subset of potentially valuable contextual inputs throughout the application's execution.

We use Figure 6 to better illustrate the relationship between the major outcomes of our approach in terms of their exposure to contextual scenarios. Given the whole set of contextual scenarios that could be derived through a purely static program analysis mechanism (set 1), there are only a subset that are reachable (set 2). The original test suite only exposes a region (set 3) of the space of reachable contextual scenarios. The scenarios produced by our driver generator (set 4) are crosscutting in the sense that they are likely to include some scenarios covered by the test suite, some that are reachable but not covered by the suite, and some that are just infeasible. The scenarios explored with the assistance of our instrumentation and manipulator (set 5) are a subset of the ones provided by the generator, and since they were executed, they are clearly within the bounds of reachable scenarios.



[1] All-interleaving
[2] Reachable interleaving
[3] Original interleaving
[4] Interleaving produced by our generator
[5] Interleaving covered by scheduled execution

Figure 6. Space of contextual scenarios

Overall, the value-added of our approach depends on how much the original test suite does in terms of validating the context input space (differences between set 2 and 3), and to what degree variants of the original test cases can expose the generated scenarios.

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 is the middleware). We utilize the application's original test suite enhanced with additional tests derived from scenarios included in the Context Toolkit Installation Guide to serve as the baseline validation. In our extension of the suite, we focused on the 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 13 minutes to execute on a 1.6GHz Pentium CPU and 1GB RAM running Java 1.4 (the same platform was 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-2*, *StoC-3*, and *StoC-4*. We set the *timeout*, used to stop the manipulator when pursuing a given driver, to 50 seconds, which is the maximum time it took for any test case to execute.

We also created two additional versions of *TourApp* called *delayTourApp*. 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()* methods randomly choose delays ranging from 1 to 3 seconds for version *delayShortTourApp*, and from 1 to 10 seconds for version *delayLongTourApp* (the delay values were 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 sequences 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 exception. We used these scenarios to help us 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 execute 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 (e.g., if a test case does not provide coverage for the context of type $power$, then do not pursue any driver that includes $power$). We repeat this process for each coverage criterion while calculating the execution time under each one of them, and 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*: execute repeatedly for as long as it took for the *manipulatedTourApp* to complete execution under the same adequacy criteria. By setting the same time upper bound we can effectively compare the performance across approaches.

3. Compare the contextual coverage achieved and contextual scenarios exposed that lead to failures by *manipulatedTourApp* versus that 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* criteria *manipulatedTourApp* took approximately three times (54m versus 13m) more than 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 2. Timings with *manipulatedTourApp*

Criteria	Generated Drivers	Execution Time (minutes)
CA	21	54
StoC-2	32	66
StoC-3	109	322
StoC-4	545	4916

Table 3. Drivers in *manipulatedTourApp*

	CA	*StoC-2*	*SotC-3*	*SotC-4*
Exposed	12	6	18	111
Timed out	9	26	91	434

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 wasteful exploration of those spaces. Improvements in the characterization of D, such as the addition of flow information, would reduce the number of such drivers from being generated, leading to a shorter manipulation time to reach the coverage criteria.

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 criteria. The same table also reports the percentage of contextual scenarios that may lead to failures were exercised when utilizing each approach.

We note that all approaches get 100% *CA* coverage. However, the coverage percentage decreases as we move to more exhaustive criteria. *originalTourApp* is the one that declines quicker, providing less than 5% coverage at the *StoC-3* level. The approaches utilizing delay provide approximately 15% more coverage on average in *StoC-2* than *originalTourApp*, but the difference is reduced to 4% at *StoC-3*. Note also that there is approximately a 5% difference between *delayShortTourApp* and *delayLongTourApp*, which shows the sensibility of this approach to the chosen delays. *ManipulatedTourApp* performs noticeably better than the rest, specially as the target coverage criterion become more powerful. The exposure to contextual scenarios leading to failure also shows similar patters, with *manipulatedTourApp* detecting all the problematic scenarios at the *StoC-2* level, while the rest of techniques cannot expose more than half of the scenarios.

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.

Table 4. Contextual Coverage and Fault Detection in %

	CA		*StoC-2*		*StoC-3*		*StoC-4*	
	Coverage	Fault	Coverage	Fault	Coverage	Fault	Coverage	Fault
originalTourApp	100%	0%	21%	0%	5%	0%	1%	0%
delayShortTourApp	100%	25%	36%	25%	9%	25%	2%	25%
delayLongTourApp	100%	25%	41%	25%	14%	25%	4%	25%
manipulatedTourApp	100%	75%	88%	100%	82%	100%	81%	100%

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

The appearance of testing approaches for context-aware application is much more recent. Tse et al. have shown the application of a technique for test case generation based on metamorphic testing to a context-aware application [21]. The technique requires the definition of metamorphic properties (e.g, if two visitors have the same preferred illumination, the light should automatically adjust to the same illumination on both visitors' sites) that serve as oracles. If any test cases and their corresponding outputs violate a specific metamorphic property, then it is presumed that program must contain a fault. More recently, the same research group has started to develop a suite of data-flow type coverage criteria that consider some contextual events and their associated actions [9]. These efforts are parallel to ours in they are primarily targeted toward the definition of an initial and the assessment of a context-aware test suite, but it does not consider the problem of automatically identifying $capps$ and manipulating the program execution to expose the interesting 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 faults exposed 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 involve 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 expose interleaving of threads that were not manifested with the original test suite [4]. This approach is relative 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]) utilize deterministic replay of a chosen set of synchronization sequences. This approach generally requires specific tool support and its effectiveness is dependent on the testers selection of sequences. Our approach belongs to this later group that 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 context changes instead of dealing with more standard concurrency primitives such as semaphores and queuing mechanisms or synchronization constructs. Furthermore, our approach provides integrated support for the automatic extension of an existing test suite which requires minimal tester's participation.

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 most constrained space 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. The preliminary assessment of the approach revealed that it can effectively enhance an existing test suite so that it provides exposure to a larger set of interesting context scenarios, even larger than the ones provided by 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 of the number of drivers generated but unhopefully explored. Regarding improvements, we are revisiting some of the sta-

tic analysis tools we use which are quick and somewhat scalable but quite conservative, leading to the identification of extra *capps* or the generation of infeasible drivers that have a negative impact on efficiency. Second, we are incorporating some of the additional adequacy criterion mentioned in Section 5. 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 outcome of the generated test cases.

Acknowledgments

This work was supported in part by NSF CAREER Award 0347518 , by the ARO through DURIP award W911NF-04-1-0104, and by the EPSRC under grant EP/E006191/1. David 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. *TSE*, 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, 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*, 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 *CC*, 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, 2003.

[9] H. Lu, W. Chan, and T. Tse. Testing context-aware middleware-centric programs: a data flow approach and an rfid-based experimentation. In *International Symposium on Foundations of Software Engineering*, page To appear, 2006.

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

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

[12] V. P. Ranganath and J. Hatcliff. Pruning interference and ready dependence for slicing concurrent java programs. In *CC*, 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. *Transactions on Software Engineering and Methodology*, 6(3):250 – 282, July 1997.

[15] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: a middleware platform for active spaces. *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. *TSE*, 29(12):1112–1121, 2003.

[18] A. Schmidt, T. Stuhr, and H.-W. Gellersen. Context-phonebook - extending mobile phone applications with context. In *Mobile HCI Workshop*, Septemper 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*, Oct 2005.

[20] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 18(3):206–215, 1992.

[21] 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, 2004.

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

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