To appear in *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), ACM Press, New York, NY, 2008

# Testing Pervasive Software in the Presence of Context Inconsistency Resolution Services *†‡

Heng Lu
The University of Hong Kong
Pokfulam, Hong Kong
hlu@cs.hku.hk

W. K. Chan
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

T. H. Tse
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

## ABSTRACT

Pervasive computing software adapts its behavior according to the changing contexts. Nevertheless, contexts are often noisy. Context inconsistency resolution provides a cleaner pervasive computing environment to context-aware applications. A faulty context-aware application may, however, mistakenly mix up inconsistent contexts and resolved ones, causing incorrect results. This paper studies how such faulty context-aware applications may be affected by these services. We model how programs should handle contexts that are continually checked and resolved by context inconsistency resolution, develop novel sets of data flow equations to analyze the potential impacts, and thus formulate a new family of test adequacy criteria for testing these applications. Experimentation shows that our approach is promising.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.8 [**Software Engineering**]: Metrics—*Product metrics*

## General Terms

Verification, Experimentation, Measurement, Reliability

## Keywords

pervasive computing, context inconsistency resolution, test adequacy



† This research is supported in part by grants of the Research Grants Council of Hong Kong (project nos. 111107, 716507, 717506).

‡ All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2858 4141. Email: thtse@cs.hku.hk.


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


## 1. INTRODUCTION

In pervasive computing, a *context-aware* [2] application widely uses *contexts* [2, 14] to represent and reason about the dynamic computing environment. The application may adapt its behavior accordingly and produce new contexts. Researchers are studying effective techniques to address the identified testing challenges for context-aware applications [13, 14, 20, 21]. Nonetheless, many contexts are inherently noisy and need to be resolved or repaired for improving their data quality [10, 24]. A context-aware application should be implemented carefully.

For instance, Radio Frequency Identification (RFID) is an enabling technology for improving supply chain management, in which RFID tags or other sensors may be attached to individual products, and their RFID data are captured as contexts. Owing to natural variations in radio frequency (RF) signals, RFID tags often induce *false negative reads* or *false positive reads* [10]. The former refers to the case when an RFID reader misses the reading of a tag that is within its detection range, and the latter refers to the case when a reader mistakenly reads a tag that is considered outside its associated range. Over 30% of the tag reads may be lost [10], and contexts obtained from different sources may also be conflicting [1]. Moreover, when applying a context consistency resolution strategy to the *Call Forwarding* application scenario [22] up to 30% error rate, the strategy can achieve around 80% of the optimal effectiveness [24]. Automatic resolution of context inconsistencies is crucial to providing useful services to users of pervasive software [10, 19, 24].

When encountering an erroneous input, an ordinary program may follow a software transaction approach. It raises an exception and skips the "normal" processing, based on the assumption that exceptions rarely occur. In pervasive computing, however, the encountering of a problematic context is a norm rather than an exception. Hence, if the applications merely skip further processing when encountering problematic contexts, intuitively, they can hardly offer adequate service to users.

As a result, a context-aware application must use contexts, even if they may be noisy, to compute other contexts and intermediate results. When some noisy contexts are later resolved, and yet the intermediate results of a faulty program are not adjusted accordingly, the faulty program may result in an inconsistent program state, which may lead to erroneous outputs. Since pervasive software uses context data, and context inconsistency resolution strategies affect the contexts in the computing environment of such software, testing is essential to assure the quality of the latter. To our best knowledge, the topic has not been adequately studied by the software engineering community.

We address the challenges in testing context-aware applications arising from the presence of context inconsistency resolutions

in their computing environment.[1] The resolution mechanism is modeled as a set of *Context Inconsistency Resolution services*, or *CIR services* for short. We generally categorize CIR services into *drop services* and *repair services*, which supports the deletion of context data and the modification of such data, respectively [18].

A context stream is a time series of context instances. When a context instance is detected, it is added to a context stream as the current (and latest) context instance.

A context-aware application will use a context stream and associate it to, say, a context variable. Intuitively, fetching a value from a context variable means fetching the current context instance from the associated context stream, and assigning a value to the context variable means adding a context instance to the context stream as the current context instance.

A drop (repair) service of the computing environment may however drop (revise) the current context instance in the context stream. They generally affect the contents of context variables of the context-aware application, and hence its data flow associations. For instance, a drop CIR service may erase the latest data definition assigned to a variable of the application and thus restore the value of an associated context variable to a previously *killed* data definition; whereas a repair CIR service may produce a new data definition. Both effects essentially reveal implicit data flow associations relevant to the context-aware application.

We may then ask: What is the way to understand and analyze the impact of implicit data flow associations on the application? To answer this question, we study various scenarios and propose a novel set of data flow equations to capture the impact of CIR services on context-aware applications. We further propose a new family of test adequacy criteria based on data flow associations to assure the quality of the applications.

The main contributions of the paper are three-fold: (i) It proposes a data flow equation framework to study the feedback mechanism between CIR services and context-aware applications. It significantly complements our previous work [14], in which any context definition by the application must be reliably accepted by the computing environment. Hence, this paper is a step toward addressing unreliable computing environments from the data flow perspective. (ii) We propose a new family of test adequacy criteria for testing context-aware applications in the presence of CIR services. (iii) We conduct the first set of experiments to evaluate our proposal and its kinds. The experimental result shows that our testing criteria are significantly more effective than random testing.

The rest of the paper is presented as follows. Section 2 will give the background and an example to motivate our work. We then develop our data flow framework in Section 3, followed by a family of test adequacy criteria in Section 4. Section 5 evaluates the proposal empirically. We then review related work in Section 6 and conclude the paper in Section 7.

# 2. MOTIVATIONS

This section gives the background that motivates our work.

## 2.1 Background

### 2.1.1 Context-Aware Computing

A context characterizes an environmental attribute of a computing entity. A context is defined via a data structure consisting of a tuple that record the environmental data [12, 25]. Each context is identified and referenced by a *context variable*, thus:

DEFINITION 1 (CONTEXT VARIABLE). A **context variable** $c$ denotes a tuple of fields $(field_1, field_2, \ldots, field_n)$, where each $field_i$ represents an environmental attribute.

A context instance $ins(c)$ is generated when all the fields in the context variable $c$ are instantiated [23]. Context instances are kept in a global infrastructure, such as a tuple space [15]. In this way, the read and write operations on context variables can be transient and consistent in a pervasive environment (see also [15]).

DEFINITION 2 (CONTEXT INSTANCE). A **context instance** $ins(c)$ of a context variable $c$ is a tuple $(t_1, t_2, \ldots, t_n)$ such that each $t_i$ is of the form $(field_i = value : type)$, where $field_i$ is the name of the corresponding field in $c$, and *value* and *type* are the value and data type, respectively, of the field.

To simplify the presentation, we omit the data types and use the notation $field = value$ to represent the field and value elements in a context instance.

As an example, a context capturing the signal of an RFID tag labeled as tag001 can be modeled as a context variable $c_{\mathsf{tag001}} = (id, category, strength, reader, lifespan, timestamp)$, while a corresponding context instance may be recorded as $ins(c_{\mathsf{tag001}}) = (id = 12345,\ category = \text{tag signal},\ strength = 67,\ reader = \text{reader2},\ lifespan = 3000\text{ms},\ timestamp = 1137615054789)$.

In the above example, *id* uniquely identifies each of the context instances, *category* describes the type or usage of the context, *strength* stores the captured value of the RF signal strength from the tag, *reader* denotes the RFID reader that reads the tag, *lifespan* is the period of time during which every context instance remains effective, and *timestamp* records the machine time at which the context instance is generated.

### 2.1.2 Context Inconsistency Resolution as Middleware Services

To improve the quality of contexts [10], researchers propose to detect inconsistent context data via consistency constraints, and to resolve detected inconsistencies via context inconsistency resolution [18, 23]. Figure 1 describes an overview of typical context processing in the CIR-enabled pervasive computing environment.

DEFINITION 3 (CIR SERVICE). A **CIR service** (or **service** for short), is a couple $\phi = (q, s)$, where $q$ is a constraint that specifies a consistency property over context variables, and $s$ is a resolution strategy that specifies how to resolve the context inconsistency violating $q$.[2]

A CIR service may use a resolution strategy to resolve a context instance $ins(c)$ of a context variable $c$ through one of the following two observable effects [18, 24]: (1) *Drop*: Discard $ins(c)$ and restore the state of $c$ to the one immediately before $ins(c)$ is captured; (2) *Repair*: modify the values of certain fields in $ins(c)$ to fulfill the consistency constraints of CIR services. To ease our discussion, we assume that each CIR service resolves one context variable, and is identified as either a **drop service** or a **repair service** according to whether it applies a "drop" or a "repair"

[1] We should emphasize to the audience that this paper is not targeted for the testing of context inconsistency.

[2] In RCSM [25], for instance, the constraint is implemented as a situation-aware expression; in nesC [6], it can be implemented as an asynchronous event, in which an if-statement may serve the purpose of checking a violation of the constraint.

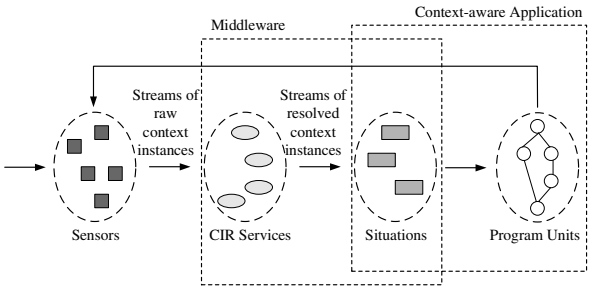


**Figure 1: Overview of context processing in pervasive computing environment in the presence of CIR services**

strategy, respectively. The generalization to resolving multiple context variables can be considered as a series of these context inconsistency resolutions. As we shall present in Section 3, our framework handles the composition of these CIR services to form more sophisticated strategies.

For each context variable $c$, we denote the set of CIR services associated with $c$ as $\Phi_c$. The set $\Phi_c$ is partitioned into two disjoint subsets: the set of drop services $\{\delta_1, \delta_2, \ldots, \delta_n\}$ $\Delta_c$, and the set of repair services $\{\lambda_1, \lambda_2, \ldots, \lambda_m\}$ as $\Lambda_c$, that is, $\Delta_c \cup \Lambda_c = \Phi_c$ and $\Delta_c \cap \Lambda_c = \emptyset$.

## 2.2 Running Example

This section gives a running example that shows an RFID application with a CIR-enabled context-aware scenario.

It is gradually popular to track objects attached with RFID tags, such as products in a supply chain or baggages in an airport. [3] Let us consider a conveyor belt that carries packages passing through an inspection zone (see Figure 2). A line of RFID readers is set up along the conveyor belt and partitions the inspection zone into a series of segments according to readers' respective detection ranges. We label the readers as reader0, reader1, reader2, and reader3 along the moving direction of the conveyor belt, and denote their positions as 0, 1, 2, and 3, respectively. When a package attached with a RFID tag moves inside the inspection zone, its position is sensed and calibrated according to the position of the reader that receives the strongest signal strength [16].

Figure 3 shows a program fragment in pseudo-code for estimating the position of a package, and Figure 4 shows the corresponding CIR services. We use the context variables $r0$, $r1$, $r2$, and $r3$ to represent the reads of the tag by reader0, reader1, reader2, and reader3, respectively. $p$ is the context variable for the package. The program defines the situation *enter_inspection_zone* that detects whether the package has moved into the inspection zone. When the situation is satisfied, the program unit *estimate_position* will be invoked to estimate the package position [14, 25]. The primitive $query(\{c\})$ at $n_1$, $n_6$, or $n_{13}$ retrieves the latest context instances from the corresponding context streams.

As the readers may be close to one another, they may induce false positive reads [10]. For instance, a package may be detected by $r3$ via reader3 even if it is closest to position 0. To tackle the problem, a set of CIR services $\Phi_p$ to resolve inconsistent detections of package positions is defined. $\Phi_p$ includes two drop services $\delta_1$ and $\delta_3$, and one repair service $\lambda_2$; thus, $\Delta_p = \{\delta_1, \delta_3\}$ and $\Lambda_p = \{\lambda_2\}$. They specify the following three consistency constraints: (i) $q_1$: the conveyor belt is moving in unidirection; (ii) $q_2$: the speed of the conveyor belt is restricted so that, in any two consecutive sensing, a package will not move beyond two or more positions; (iii) $q_3$: in a normal working status, at least one reader should receive the RF signal strength above a threshold.

[3] Real-life case studies are available from *RFID Journal* at http://www.rfidjournal.com/article/archive/4/.

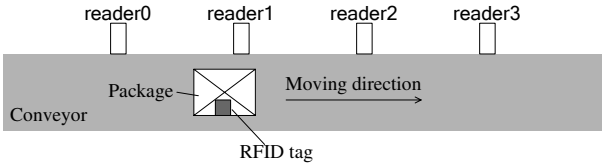


**Figure 2: Conveyor belt example**

```
context_variable {
    p;              // the package
    r0, r1, r2, r3; // reader0, reader1, reader2, and reader3
}

situation enter_inspection_zone {
    triggering_condition:  r0.strength > 0
    adaptive_action:       estimate_position
}

estimate_position {
 n1:  query({r0, r1});
 n2:  if (r0.strength ≥ r1.strength)
 n3:    p.position = 0;
 n4:  else p.position = 1;
 n5:  wait (FIXED_INTERVAL);
 n6:  query({r0, r1, r2});
 n7:  if (r0.strength ≥ r1.strength and r0.strength ≥ r2.strength)
 n8:    p.position = 0;
 n9:  else if (r1.strength > r0.strength and r1.strength ≥ r2.strength)
 n10:   p.position = 1;
 n11: else p.position = 2;
 n12: wait (FIXED_INTERVAL);
 n13: query({r1, r2, r3});
 n14: if (r1.strength ≥ r2.strength and r1.strength ≥ r3.strength)
 n15:   p.position = 1;
 n16: else if (r2.strength ≥ r1.strength and r2.strength > r3.strength)
 n17:   p.position = 2;
 n18: else p.position = 3;
 n19: report p.position;
}
```

**Figure 3: Pseudo code of program fragment to estimate package position in conveyor belt example**

## 2.3 Testing Challenges

In this section, we use the running example in Section 2.2 to illustrate the challenges in testing context-aware programs in the presence of CIR services.

Suppose $n_7$ in Figure 3 is wrongly implemented as follows:

$n_7'$: **if** ($r0.strength \leq r1.strength$ **and** $r0.strength \leq r2.strength$)

The set of test cases

$$T = \{t_1, t_2, t_3, t_4\} \quad (1)$$

where

$t_1 = \langle (30, 20), (20, 30, 10), (25, 35, 20) \rangle$,
$t_2 = \langle (30, 20), (10, 25, 30), (30, 20, 10) \rangle$,
$t_3 = \langle (20, 30), (5, 15, 30), (5, 2, 3) \rangle$, and
$t_4 = \langle (30, 20), (30, 25, 20), (20, 20, 30) \rangle$

**cir_service** $\Phi_p = \{\delta_1, \lambda_2, \delta_3\}$ {
  $\delta_1 = (q_1, s_1)$:
    $q_1$: $\forall\, ins(p)_i \in \{ins(p) \mid ins(p).timestamp < ins(p)_0.timestamp\}.$
      $(ins(p)_0.position \geq ins(p)_i.position)$
    $s_1$: **drop**
  $\lambda_2 = (q_2, s_2)$:
    $q_2$: $ins(p)_0.position < ins(p)_1.position + 2$
    $s_2$: **repair** $\{ins(p)_0.position = ins(p)_1.position + 1;\}$
  $\delta_3 = (q_3, s_3)$:
    $q_3$: $(ins(r0)_0.strength > 5) \vee (ins(r1)_0.strength > 5) \vee (ins(r2)_0.strength > 5) \vee (ins(r3)_0.strength > 5)$
    $s_3$: **drop**
  // $ins(c)_0$ denotes the latest context instance of context variable $c$
  // while $ins(c)_1$ denotes the second latest context instance of $c$
}

**Figure 4: CIR services for example program in Figure 3**

satisfies the all-uses criterion [5] for the program *estimate_position*. In each test case, the two values in the first tuple represent the test inputs of $r0.strength$ and $r1.strength$, respectively, at $n_1$ [13]. Similarly, the three values in the second tuple represent the test inputs of $r0.strength$, $r1.strength$, and $r2.strength$, respectively, at $n_6$, while the three values in the third tuple represent the test inputs of $r1.strength$, $r2.strength$, and $r3.strength$, respectively, at $n_{13}$. We have simplified their presentations to make our subsequent discussion clearer. Other test cases in this presentation format in the paper can be interpreted similarly.

When no CIR service is enabled (say, in the underlying middleware), applying $T$ to *estimate_position* does not distinguish the faulty implementation from the original one because both implementations will compute $p.position$ at $n_{19}$ to be 2, 1, 1, and 3 for the test cases $t_1$, $t_2$, $t_3$, and $t_4$, respectively.

Next, let us illustrate how CIR services may interact with the program, and how the use of this interaction may help expose the fault. To ease our presentation, the notations $ins(p)_0$ and $ins(p)_1$ refer to the latest context instance and the second latest context instance for the context variable $p$ in its context stream.

- In the original version of *estimate_position*, after the program has executed the statement $n_{15}$, $t_2$ will compute $ins(p)_1.position$ to be 2 (via $n_{11}$) and $ins(p)_0.position$ to be 1 (via $n_{15}$). However, the latter breaches $q_1$. The drop strategy $s_1$ will then discard $ins(p)_0$, and $ins(p)_1.position$ will finally be reported at $n_{19}$ to be 2. Thus, the output is 2.

- When applied to the faulty version, $t_2$ will compute $ins(p)'_1.position$ to be 0 (via $n_8$, which is *incorrect*) and $ins(p)'_0.position$ to be 1 via $n_{15}$. Since none of the consistency constraints $q_1$, $q_2$, or $q_3$ is breached, $ins(p)'_0.position$ will be reported at $n_{19}$ to be 1. Hence, the output is 1, and the fault is revealed.

# 3. DATA FLOW FRAMEWORK FOR CIR SERVICES

This section proposes our data flow analysis framework.

## 3.1 Conventional Def-Use Associations

We first review the conventional approach to analyzing def-use associations [5, 8]. A program unit in an application is modeled as a control flow graph (CFG) $G = (N, E)$, where $N$ is a set of nodes representing program statements and $E \subseteq N \times N$ is a set of directed edges representing control flows among statements. Each CFG has a unique entry node and a unique exit node. Figure 5 shows the CFG for the program unit *estimate_position* in Figure 3. The following definitions are taken from [5, 8].

A *complete path* is a path that traverses from the entry node to the exit node along the edges. A variable $x$ is defined or has a *definition* occurrence at node $n$ if the value of $x$ is stored or updated at a statement in $n$. A variable $x$ is used or has a *use* occurrence at $n$ if the value of $x$ is fetched or referenced at a statement in $n$. $Def(n)$ and $Use(n)$ denote the two sets of variables that are defined and used at $n$, respectively.

A subpath of a CFG is *definition-clear* with respect to (w.r.t.) $x$ if none of the nodes in the subpath defines $x$ or makes $x$ undefined. The relation $def\_clear(x, n_i, n_j)$ denotes that there is a definition-clear subpath w.r.t. $x$ from $n_i$ to $n_j$, exclusively. A definition of $x$ at node $n_i$ is a *reaching definition* at node $n_j$ if $def\_clear(x, n_i, n_j)$. The set of reaching definitions of $x$ at node $n$ is denoted by $RD_x(n)$.

$$RD_x(n) = \{n_d \mid x \in Def(n_d) \wedge \exists n_1, n_2, \ldots, n_k \cdot ((n_d, n_1) \in E \wedge n_k = n \wedge \forall 1 \leq i < k \cdot ((n_i, n_{i+1}) \in E \wedge x \notin Def(n_i)))\}$$

A *def-use association* is defined as a triple $(x, n_i, n_j)$, where $x \in Use(n_j)$ and $n_i \in RD_x(n_j)$. $(x, n_i, n_j)$ is covered by a path $\pi$ if both $n_i$ and $n_j$ are in $\pi$ and there is a subpath of $\pi$ from $n_i$ to $n_j$, exclusively, which is definition-clear w.r.t. $x$ [5].

For a set of nodes $M$, we have $RD_x(M) = \bigcup_{n \in M} RD_x(n)$ and $RD^k_x(n) = RD_x(RD^{k-1}_x(n))$, where $k > 2$ and $RD^1_x(n) = RD_x(n)$ [8]. For the CFG in Figure 5, for instance, $RD_p(n_{19}) = RD_p(\{n_{19}\}) = \{n_{15}, n_{17}, n_{18}\}$ and $RD^2_p(n_{19}) = \{n_8, n_{10}, n_{11}\}$. Each node $n$ maintains the following set of data flow equations [8] to compute the reaching definitions of each variable $x$:

$$\begin{array}{lcl} IN_x(n) & = & \bigcup_{m \in Pred(n)} OUT_x(m); \\ KILL_x(n) & = & \begin{cases} IN_x(n), & \text{if } x \in Def(n); \\ \emptyset, & \text{otherwise;} \end{cases} \\ GEN_x(n) & = & \begin{cases} \{n\}, & \text{if } x \in Def(n); \\ \emptyset, & \text{otherwise;} \end{cases} \\ OUT_x(n) & = & GEN_x(n) \cup (IN_x(n) - KILL_x(n)); \end{array} \quad (2)$$

where $IN_x(n)$ is $RD_x(n)$; $OUT_x(n)$ is the set of reaching definitions of $x$ immediately after $n$; $KILL_x(n)$ is the set of reaching definitions of $x$ that are killed by $n$; and $GEN_x(n)$ is the set of reaching definitions of $x$ that are generated at $n$.

The set of predecessors (via control flow edges) of $n$ is denoted by $Pred(n)$. The reaching definitions can be computed by iteratively applying Equation (2) to all the nodes until fixed points have been reached. Following the standard programming style to model an aspect of a context variable as an ordinary program variable, the above computation of reaching definitions also applies to context variables.

## 3.2 Our Data Flow Equations

In a context-aware application, when a CIR service $\phi$ for some context variable $c$ affects the def-use associations of the program (that is, $\phi$ resolves a context instance followed by a use of $c$ in the program), we call such a scenario a *service configuration*. In this scenario, each use (or invocation) of $\phi$ is represented as a *service node* $n_\phi$. The nodes in the original CFG of a program unit are referred to as *program nodes*.

In a service configuration, $n_\phi$ is associated with some node $n_j$ that uses $c$, by diverting an outgoing edge from the node $n_j$, say, $(n_i, n_j)$, to $n_\phi$ before reaching $n_j$. We call $(n_i, n_j)$ a *service point* w.r.t. $\phi$. Intuitively, a service point captures the resolution window of a variable definition of a context variable at node $n_i$ (not due to any environmental update [14]) before the variable use at node $n_j$.

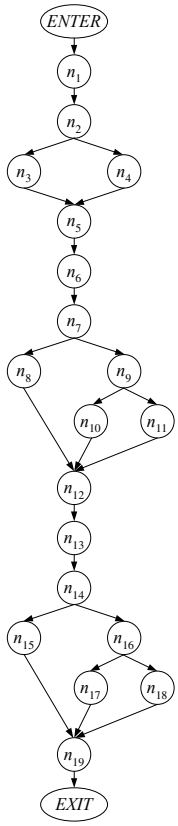


**Figure 5: CFG of program unit** *estimate_position*

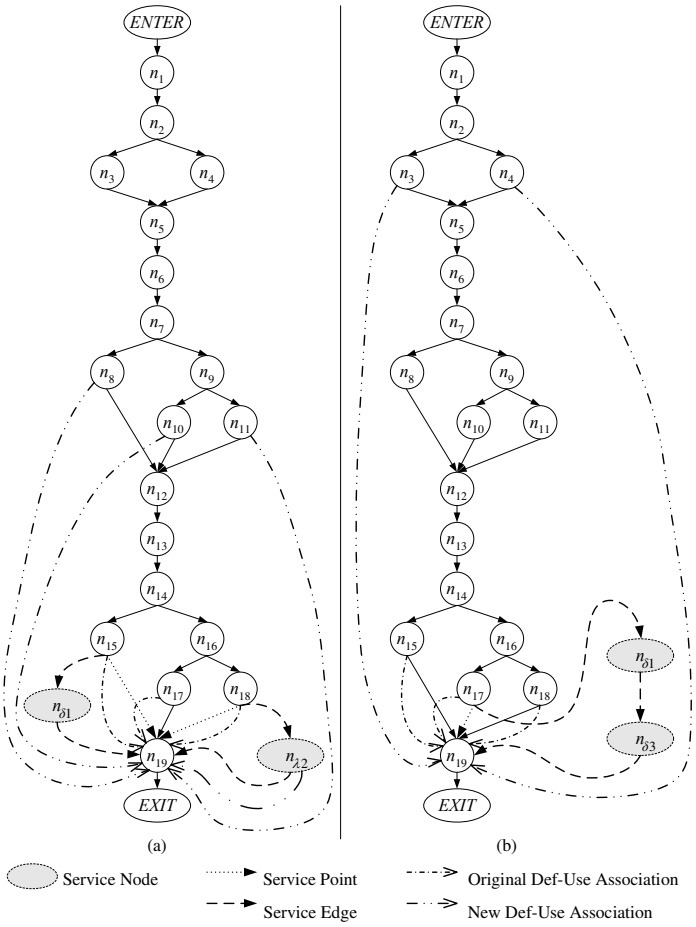


**Figure 6: Example service configurations: (a) single service; (b) composite service**

As we have explained in Section 1, the program may compute a context based on other noisy contexts. Such a window provides a chance to detect context inconsistency [23] followed by one or more context resolution strategies [24] to rectify the detected inconsistency before the next variable use.[4] To construct this service configuration, the CFG for the program should be modified, by deleting edge $(n_i, n_j)$ and adding the *service edges* $(n_i, n_\phi)$ and $(n_\phi, n_j)$. When multiple CIR services are used at the same service point, we use the above scheme to chain up the respective service nodes (to form a composite service) in between service edges according to each interleaving execution order of these services defined by the underlying middleware.

In the CFG shown in Figure 5, for instance, since $n_{19}$ has a use occurrence of the context variable $p$, potential service points w.r.t. $n_{19}$ include $(n_{15}, n_{19})$, $(n_{17}, n_{19})$, and $(n_{18}, n_{19})$. Figure 6 shows two example service configurations on top of Figure 5. In Figure 6(a), CIR services $\delta_1$ and $\lambda_2$ are used at the service points $(n_{15}, n_{19})$ and $(n_{18}, n_{19})$, respectively. In Figure 6(b), CIR services $\delta_1$ and $\delta_3$ are used in turn at the service point $(n_{17}, n_{19})$.

[4] Since *environmental update* [14] is not in the big picture, our previously identified def-use associations [14] do not cover the def-use associations relevant to service points.

A drop service $\delta$ drops a context instance in a context stream and thus restores the reaching definition of a context variable to a previously killed one. To model such an activity, we give the reaching definition equations w.r.t. a context variable $x$ for $n_\delta$ in Equation (3), in which the set of killed definitions restored by the drop service $\delta$ at $n_\delta$ is denoted by $RES_x(n_\delta)$.

$$\begin{array}{lcl} IN_x(n_\delta) & = & \bigcup_{m \in Pred(n_\delta)} OUT_x(m); \\ KILL_x(n_\delta) & = & \begin{cases} IN_x(n_\delta), & \text{if } \delta \in \Phi_x; \\ \emptyset, & \text{otherwise;} \end{cases} \\ RES_x(n_\delta) & = & \begin{cases} \bigcup_{m \in KILL_x(n_\delta)} KILL_x(m), & \text{if } \delta \in \Phi_x; \\ \emptyset, & \text{otherwise;} \end{cases} \\ OUT_x(n_\delta) & = & RES_x(n_\delta) \cup (IN_x(n_\delta) - KILL_x(n_\delta)). \end{array} \quad (3)$$

A service node $n_\lambda$ for the repair service $\lambda \in \Phi_x$ is similar to a redefinition of the variable $x$, but the difference is significant. Unlike any definition of $x$ that always generates a new context instance of $x$, it only modifies the value of the context instance generated by the latest definition of $x$ in the context stream. For the purpose of data flow propagation, its "killed" set, *KILL*, is defined as the union of the *KILL* sets of its reaching definitions. We give the reaching definition equations for $n_\lambda$ in Equation (4).

$$
\begin{array}{lcl}
IN_x(n_\lambda) & = & \bigcup_{m \in Pred(n_\lambda)} OUT_x(m); \\
KILL_x(n_\lambda) & = & \begin{cases} \bigcup_{m \in IN_x(n_\lambda)} KILL_x(m), & \text{if } \lambda \in \Phi_x; \\ \emptyset, & \text{otherwise;} \end{cases} \\
GEN_x(n_\lambda) & = & \begin{cases} \{n_\lambda\}, & \text{if } \lambda \in \Phi_x; \\ \emptyset, & \text{otherwise;} \end{cases} \\
OUT_x(n_\lambda) & = & \begin{cases} GEN_x(n_\lambda), & \text{if } \lambda \in \Phi_x; \\ IN_x(n_\lambda), & \text{otherwise.} \end{cases}
\end{array} \tag{4}
$$

## 3.3 CIR Impact on Def-Use Associations

To compute the def-use associations for context-aware applications having CIR services, our data flow equations presented in the last section reveal the special needs to consider the *KILL* and *OUT* sets, which cannot be effectively captured by conventional approaches (such as Equation (2), or [7], or our previous work [14]). In this section, we first investigate the data flow impact of having one CIR service per service point, which establishes the base case for us to present a more general case, namely a service point having multiple CIR services.

### *3.3.1 Single Services*

For a program node $n_u$ that contains a use occurrence of a context variable $c$, we first analyze the impact on def-use associations for single service nodes on $n_u$. By a "single service", we formally mean any service configuration in which any complete path $\pi$ traverses no more than one service node, such as $n_\phi$, before reaching $n_u$. $n_\phi$ may introduce new reaching definitions (w.r.t. $c$) to $n_u$ only if $\phi \in \Phi_c$ and there is a subpath of $\pi$ from $n_\phi$ to $n_u$ that is definition-clear w.r.t. $c$. To ease our discussion, we denote this set of new def-use associations w.r.t. $c$ as $DU_{CIR}(n_\phi, n_u)_c$, or just $DU_{CIR}(n_\phi, n_u)$ if the context variable is known implicitly and clearly. We also refer to $RD_x(n)$ as the set of reaching definitions that are derived from the original CFG.

According to Equations (2) and (3), a drop service node $n_\delta$ for $c$ will restore all the definitions of $c$ that are killed by the reaching definitions of $n_\delta$. Thus, we have

$$DU_{CIR}(n_\delta, n_u) = \{c\} \times RD_c^2(n_u) \times \{n_u\}. \tag{5}$$

For a repair service $\lambda$ for $c$, according to Equation (4), its service node $n_\lambda$ corresponds to a definition of $c$. As each service node of $\lambda$ refers to an invocation of the same CIR service $\lambda$, we denote the definition occurrence of $c$ at service node $n_\lambda$ as $\lambda$ for short. Thus, we have

$$DU_{CIR}(n_\lambda, n_u) = \{(c, \lambda, n_u)\}. \tag{6}$$

In Figure 6(a), for example, $n_{19}$ has a use occurrence of $p$, and there are two single service nodes $n_{\delta_1}$ and $n_{\lambda_2}$. We have

$$
\begin{aligned}
DU_{CIR}(n_{\delta_1}, n_{19}) &= \{p\} \times RD_p^2(n_{19}) \times \{n_{19}\} \\
&= \{(p, n_8, n_{19}), (p, n_{10}, n_{19}), (p, n_{11}, n_{19})\},
\end{aligned}
$$

and

$$DU_{CIR}(n_{\lambda_2}, n_{19}) = \{(p, \phi_2, n_{19})\}.$$

### *3.3.2 Composite Services*

In this section, we consider the impact of composite services for program node $n_u$ that uses the context variable $c$. By a "composite service", we mean any service configuration in which there is a path from the entry node that traverses multiple service nodes before reaching $n_u$. Since the reaching definitions at $n_u$ w.r.t. $c$ are only affected by the nodes that define $c$ as well as the relevant service nodes for $c$, we model the subpath of a complete path right before reaching $n_u$ as a sequence of definition occurrences and service nodes. We call such a sequence a *definition-service witness*, or simply a *witness*.

DEFINITION 4 (DEFINITION-SERVICE WITNESS). A **definition-service witness** (or simply **witness**) $w(n_u)$ for node $n_u$ w.r.t. context variable $c$ ($\in Use(n_u)$) is a sequence of nodes $\langle n_1, n_2, \ldots, n_k \rangle$, where (a) $\forall 1 \le i \le k$, $n_i$ is either a program node such that $c \in Def(n_i)$, or a service node for some service $\phi \in \Phi_c$; (b) $\forall 1 \le i < k$, $def\_clear(c, n_i, n_{i+1})$; and (c) $def\_clear(c, n_k, n_u)$.

In Figure 6(a), $\langle n_3, n_{10}, n_{15}, n_{\delta_1} \rangle$ is a witness for $n_{19}$. Similarly, in Figure 6(b), $\langle n_3, n_{11}, n_{17}, n_{\delta_1}, n_{\delta_3} \rangle$ is a witness for $n_{19}$. Detailed variable definitions and uses for the program unit *estimate_position* can be found in Figure 3.

As the witness $w(n_u)$ with one or more service nodes may introduce new reaching definitions w.r.t. $c$ to $n_u$, we denote the set of new def-use associations as $DU_{CIR}(w(n_u), n_u)$.

For the ease of representation of $w(n_u)$, in the rest of the section, we use the notation $u$ to represent $n_u$. We classify the nodes in a witness into three categories w.r.t. context variable $c$, namely, program nodes that define $c$, service nodes for drop services of $c$, and service nodes for repair services of $c$. We use the symbols $d$, $\delta$, and $\lambda$ to represent a node in the three categories, respectively. If necessary, we identify different nodes of the same category by subscriptions, such as $d_0$, $d_1$, and so on. We also omit the commas between each two consecutive nodes, and use the symbols "$*$" and "$|$" to represent zero-to-many repetitions of nodes and selection of nodes, respectively. For example, the pattern $(\delta|\lambda)^* d_0$ refers to any sequence in which the definition $d_0$ is preceded by an arbitrary number of drop service nodes and repair service nodes.

In the following, we analyze the impact of witnesses on the introduction of new def-use associations by enumerating all the patterns that may exist in a witness.

**Pattern 1:** The witness is terminated by a definition, that is, $w(u) = (d|\delta|\lambda)^* d_0$. Under this pattern, any definition of $c$ due to a service node appearing before $d_0$ in the witness cannot reach $u$. Hence, $DU_{CIR}(w(u), u) = \emptyset$, which means that witnesses of this pattern will not introduce any new reaching definitions to $u$.

**Pattern 2:** The witness is terminated by a repair service, that is, $w(u) = (d|\delta|\lambda)^* \lambda_0$. Similarly to Pattern 1, any definition of $c$ due to a service node before $\lambda_0$ in the witness cannot reach $u$. Thus, the impact of witnesses with this pattern is the same as that of the single service node $\lambda_0$. Following Equation (6), we have $DU_{CIR}(w(u), u) = \{(c, \lambda_0, u)\}$.

**Pattern 3:** The witness is terminated by a drop service, that is, $w(u) = (d|\delta|\lambda)^* \delta_0$. We first analyze the impact of repair service nodes in this pattern. By exhaustive enumeration, a repair service node $\lambda$ may occur in one of the following three situations:

**Case (a) of Pattern 3:** $\lambda$ is succeeded by (and thus killed by) a definition or by another repair service node. Hence, the repair service $\lambda$ does not have any impact on $u$.

**Case (b) of Pattern 3:** $\lambda$ is preceded by a definition and succeeded by a drop service node, that is, the witness contains a subsequence of the pattern $d\lambda^*\delta$. Since $\lambda$ only modifies the latest context instance generated by $d$, it does not create a new context instance. $\delta$ will then kill the latest definition due to this $d$ as well as all repair service nodes between this $d$ and $\delta$. Therefore, $\lambda$ does not have any impact on $u$.

**Case (c) of Pattern 3:** The definition of $c$ due to a repair service $\lambda$ reaches $u$, because $\delta_0$ (and possibly other drop service nodes) kills the definition(s) succeeding $\lambda$ (as in Case (a) above). Hence, the case gives the same result as a single service $\lambda$.

Based on the above analysis of Pattern 3, we need only consider different combinations of definitions and drop service nodes for context variables.

For witnesses of Pattern 3 with two drop service nodes, there are the following three subpatterns. They correspond to zero, one, and two definitions between two drop service nodes, respectively.

**Subpattern (i):** $w(u) = d^* d_2 d_1 d_0 \delta_1 \delta_0$. According to Equation (3), $\delta_1$ kills $d_0$ and restores $d_1$, while $\delta_0$ kills $d_1$ and restores $d_2$. Hence, $d_2$ reaches $u$ and $DU_{CIR}(w(u), u) = \{(c, d_2, u)\}$.

**Subpattern (ii):** $w(u) = d^* d_2 d_1 \delta_1 d_0 \delta_0$. Similarly to Subpattern (i), $\delta_1$ kills $d_1$ and restores $d_2$, and $\delta_0$ kills $d_0$ and restores $d_2$. Thus, $d_2$ reaches $u$, and we have $DU_{CIR}(w(u), u) = \{(c, d_2, u)\}$.

**Subpattern (iii):** $w(u) = d^* d_2 \delta_1 d_1 d_0 \delta_0$. In this subpattern, $d_1$ reaches $u$ because $\delta_0$ kills $d_0$ and restores $d_1$. $d_1$ is in $RD^2_c(u)$. Hence, according to Equation (5), $DU_{CIR}(w(u), u) = \{(c, d_1, u)\}$, which is a subset of the new def-use associations introduced by the impact of the single service $\delta_0$.

As an example, in Figure 6, the witness $w(n_{19}) = \langle n_3, n_{11}, n_{17}, n_{\delta_1}, n_{\delta_3} \rangle$ follows subpattern (i). Since $n_3 \in RD^3_p(n_{19})$, we have $DU_{CIR}(w(n_{19}), n_{19}) = \{(p, n_3, n_{19})\}$.

We can use mathematical induction to conclude that, for witness $w(u)$ of Pattern 3 with $k$ drop service nodes ($k \geq 2$), $w(u)$ may introduce new def-use associations only if it follows the pattern

$$d^*(\delta_{k-1}|\delta_{k-1}d)\ldots(\delta_2|\delta_2 d)(\delta_1|\delta_1 d)\delta_0. \quad (7)$$

Hence, $DU_{CIR}(w(u), u) = \{(c, d_0, u)\}$, where $d_0 \in RD^{k+1}_c(u)$.

# 4. TEST ADEQUACY CRITERIA

This section proposes our family of test adequacy criteria for context-aware applications in the presence of CIR services.

As mentioned in Section 2.3, an intuitive testing coverage requirement is to use each CIR service at least once. We hereby propose the *all-services* criterion as follows.

DEFINITION 5 (ALL-SERVICES CRITERION). A test suite $T$ satisfies the **all-services** criterion if $T$ satisfies the all-uses criterion and, for each CIR service $\phi = (q, s)$, there exists a test case $t \in T$ which executes a complete path that causes the consistency constraint $q$ to be violated and the resolution strategy $s$ to be applied.

The test suite $T = \{t_1, t_2, t_3, t_4\}$ in Equation (1), for instance, satisfies both the all-uses [5] and the all-services criteria for the program in Figure 3. In Section 3.3, we investigate the impact of CIR services through the introduction of new def-use associations to the original program units. In the rest of the section, we propose adequacy criteria to cover these new def-use associations at different levels.

## 4.1 Covering All Reaching Single Service Nodes

For a stricter criterion over all-services, we propose that, for every program node $n_u$ which uses some context variable $c$, the test suite should also cover all the new def-use associations introduced by all the service nodes in a single-service service configuration that may reach $n_u$. We denote the set of new def-use associations as $DU^1_{CIR}(n_u)_c$.

$$\begin{aligned} DU^1_{CIR}(n_u)_c &= \bigcup_{\phi \in \Phi_c \wedge \exists n_\phi . def\_clear(c, n_\phi, n_u)} DU_{CIR}(n_\phi, n_u) \\ &= \begin{cases} \{c\} \times (RD^2_c(n_u) \cup \Lambda_c) \times \{n_u\}, & \text{if } \Delta_c \neq \emptyset; \\ \{c\} \times \Lambda_c \times \{n_u\}, & \text{otherwise.} \end{cases} \end{aligned} \quad (8)$$

For a CFG $G = (N, E)$ having a set of context variables $C$, we define the union set of all the new def-use associations as:

$$DU^1_{CIR} = \bigcup_{n \in N \wedge c \in (Use(n) \cap C)} DU^1_{CIR}(n)_c \quad (9)$$

Following Equations (8) and (9), we implement an algorithm to compute $DU^1_{CIR}$. We omit the algorithm from the paper owing to page limitation.

We propose an *all-services-uses* criterion that satisfies all-services and at the same time covers all the new def-use associations introduced by the impact of single services.

DEFINITION 6 (ALL-SERVICES-USES CRITERION). A test suite $T$ satisfies the **all-services-uses** criterion if $T$ satisfies the all-services criterion and, for each def-use association $\alpha$ in the set $DU^1_{CIR}$ computed by Equations (8) and (9), there exists a test case $t \in T$ that executes a complete path covering $\alpha$.

For example, for the program in Figure 3, we have

$$DU^1_{CIR} = \{(p, n_8, n_{19}), (p, n_{10}, n_{19}), (p, n_{11}, n_{19}), (p, \lambda_2, n_{19})\}$$

and the following test suite $T_1$ satisfies the all-services-uses criterion:

$$\begin{aligned} T_1 = \{&\langle (30, 20), (20, 30, 10), (25, 35, 20) \rangle, \\ &\langle (20, 30), (10, 25, 30), (30, 20, 10) \rangle, \\ &\langle (20, 30), (5, 30, 15), (2, 5, 3) \rangle, \\ &\langle (30, 20), (30, 25, 20), (20, 20, 30) \rangle, \\ &\langle (30, 20), (5, 2, 1), (1, 2, 3) \rangle\} \end{aligned} \quad (10)$$

## 4.2 Covering All Reaching Composite Service Nodes

In this section, we propose testing criteria to cover new def-use associations introduced by composite services.

Given a program node $n_u$ that uses a context variable $c$, let $DU^2_{CIR}(n_u)_c$ denote the set of new def-use associations introduced by witnesses with *two* service nodes. We have shown in Section 3.3 that only the witnesses of Subpatterns (i) and (ii) (but not Subpattern (iii)) of Pattern 3 may introduce additional def-use associations that are not contained in $DU^1_{CIR}(n_u)_c$. Hence, we have

$$\begin{aligned} DU^2_{CIR}(n_u)_c &= \begin{cases} \{c\} \times RD^3_c(n_u) \times \{n_u\} \cup DU^1_{CIR}(n_u)_c, & \text{if } \Delta_c \neq \emptyset; \\ DU^1_{CIR}(n_u)_c, & \text{otherwise;} \end{cases} \\ &= \begin{cases} \{c\} \times (RD^3_c(n_u) \cup RD^2_c(n_u) \cup \Lambda_c) \times \{n_u\}, & \text{if } \Delta_c \neq \emptyset; \\ \{c\} \times \Lambda_c \times \{n_u\}, & \text{otherwise.} \end{cases} \end{aligned}$$

When considering witnesses with $k$ service nodes, where $k \geq 2$, we denote the set of new def-uses associations thus introduced w.r.t. $c$ as $DU^k_{CIR}(n_u)_c$. Similarly, $DU^k_{CIR}(n_u)_c$ contains additional def-uses associations atop $DU^{k-1}_{CIR}(n_u)_c$ only if the witnesses belong to the pattern of Equation (7). Thus, we have

$$\begin{aligned} DU^k_{CIR}(n_u)_c &= \begin{cases} \{c\} \times RD^{k+1}_c(n_u) \times \{n_u\} \cup DU^{k-1}_{CIR}(n_u)_c, \\ \quad \text{if } \Delta_c \neq \emptyset; \\ DU^{k-1}_{CIR}(n_u)_c, \quad \text{otherwise;} \end{cases} \\ &= \begin{cases} \{c\} \times (RD^{k+1}_c(n_u) \cup \ldots \cup RD^3_c(n_u) \\ \quad \cup RD^2_c(n_u) \cup \Lambda_c) \times \{n_u\}, & \text{if } \Delta_c \neq \emptyset; \\ \{c\} \times \Lambda_c \times \{n_u\}, & \text{otherwise.} \end{cases} \end{aligned} \quad (11)$$

**Algorithm** *Compute_DU$^k_{CIR}$*
**Input:** program unit CFG $G = (N, E)$;
set of context variables $C$;
set of CIR services $\Phi_c$ for each context variable $c \in C$;
**Output:** $DU^k_{CIR}$;

$DU^k_{CIR} = \{\}$;
generate empty table *TRD*;
// *initial step*
**for each** node $n \in N$
  **for each** context variable $c \in (Use(n) \cap C)$
    $TRD_c(n, 0) = RD_c(n)$;
    **for each** node $m \in TRD_c(n, 0)$
      $TRD_c(n, 1) = TRD_c(n, 1) \cup TRD_c(m, 0)$;
    **for each** repair service $\lambda \in \Lambda_c$ // $\Lambda_c$ *is a subset of* $\Phi_c$
      $DU^k_{CIR} = DU^k_{CIR} \cup \{(c, \lambda, n)\}$;
// *inductive steps*
**for each** integer $i$ from 2 to $k$
  **for each** node $n \in N$
    **for each** context variable $c \in (Use(n) \cap C)$
      **for each** node $m \in TRD_c(n, i-1)$
        $TRD_c(n, i) = TRD_c(n, i) \cup RD_c(m)$;
      **for each** node $m' \in TRD_c(n, i)$
        $DU^k_{CIR} = DU^k_{CIR} \cup \{(c, m', n)\}$;

**Figure 7: Algorithm for computing $DU^k_{CIR}$**

For a program unit with CFG $G = (N, E)$ and a set of context variables $C$, we define the union set of new def-use associations by all witnesses with $k$ service nodes as:

$$DU^k_{CIR} = \bigcup_{n \in N \wedge c \in (Use(n) \cap C)} DU^k_{CIR}(n)_c \quad (12)$$

Figure 7 shows the algorithm for computing $DU^k_{CIR}$. In the worst case, the algorithm *Compute_DU$^k_{CIR}$* computes the set $RD_c(n)$ for every node $n \in N$ w.r.t. every context variable $c \in C$, and the calculation of $RD_c(n)$ once requires the visiting of at most $|N|$ nodes; the time complexity of *Compute_DU$^k_{CIR}$* is, therefore, $O(|C| \cdot |N|^2)$.

We propose an *all-k-services-uses* criterion that satisfies all-services and, at the same time, covers all the new def-use association introduced by the impact of all the witnesses with $k$ services.

DEFINITION 7 (ALL-$k$-SERVICES-USES CRITERION). A test suite $T$ satisfies the **all-$k$-services-uses** criterion if $T$ satisfies the all-services criterion and, for each def-use association $\alpha$ in the set $DU^k_{CIR}$ as computed by Equations (11) and (12), there exists a test case $t \in T$ that executes a complete path covering $\alpha$.

As an example, for the program in Figure 3, we have

$$DU^2_{CIR} = \{(p, n_3, n_{19}), (p, n_4, n_{19}), (p, n_8, n_{19}), (p, n_{10}, n_{19}), (p, n_{11}, n_{19}), (p, \lambda_2, n_{19})\}$$

and the following all-2-services-uses-adequate test suite:

$$T_2 = T_1 \cup \{\langle (30, 20), (2, 3, 3), (5, 5, 2)\rangle, \langle (20, 30), (5, 15, 30), (5, 2, 3)\rangle\}$$

where $T_1$ is the test suite shown in Equation (10).

A criterion subsumes another if any test suite satisfying the former also satisfies the latter [5]. As such, all-$k$-services-uses subsumes all-$(k-1)$-services-uses, while all-2-services-uses subsumes all-services-uses. Moreover, all-services-uses subsumes all-services, which in turn subsumes all-uses. Owing to space limitation, other criteria analogous to all-p-uses [5] and the like are not presented here.

# 5. EVALUATION

This section reports the experimental evaluation of our proposal.

## 5.1 Experiment Design

We use *Cabot* [23, 24] as the testbed for our experiment. It includes a middleware that supports context acquisition, reasoning, and triggering of context-aware applications, and an evaluation application that implements the *LANDMARC* RFID-based location sensing algorithm [16], which has also been used to evaluate techniques in our previous work [14, 24]. In version 3, *Cabot* supports CIR services. *WALKPATH* is an application that extends *LANDMARC* [14, 16] and runs on *Cabot*. It tracks a person's walking path in an indoor space equipped with RFID sensors. The person's current locations are obtained via *LANDMARC* by capturing and analyzing the RFID contexts. *WALKPATH* utilizes the location data as incoming contexts and optionally accepts or repairs them through a set of CIR services. Hence, while a person moves, the application senses contexts from its surroundings and reacts accordingly. It includes five CIR services and 20 functions.

Our experiment consists of the following steps:

First, to conduct statistical analysis on testing effectiveness, our tool generates two groups of test suites for the adequacy criteria under evaluation. It maintains a large test pool of 20,000 different test cases and instruments the target program. All the test cases are real-world data captured via RFID readers. In Group 1, we generate 100 independent test suites for each of the all-services, all-services-uses, and all-2-services-uses criteria proposed in Section 4. When generating each test suite, the tool randomly selects a test case from the test pool, executes the instrumented program over the test case, and computes the test coverage w.r.t. the corresponding criterion from execution traces.

A test case is included in a test suite only if it increases the coverage of the test suite. This process continues until either 100% coverage of the criterion has been achieved, or an upper bound of 2,000 trials in selecting test cases have been completed. This test case selection approach is similar to those in [9, 14]. The outstanding def-use associations are deemed infeasible. Table 1 shows the coverage percentages and mean sizes of the Group 1 test suites. It also shows that, on average, the coverage of a test suite is more than 95% for each criterion, which indicates that the problem of infeasible def-use associations is manageable in the experiment.

To compare the effectiveness of different criteria based on comparable cost [3], we expand the test suites in Group 1 to form Group 2 by compensating smaller test suites in Group 1 with additional test cases randomly selected from the test pool so that all test suites, irrespective of their corresponding criteria, have the same fixed size. We set the fixed size to be 62, which is the maximum size of the all-2-services-uses test suites in Group 1. We use the random criterion as the benchmark for the comparison of effectiveness [4]. Thus, Group 2 also contains 100 independent random test suites, each with 62 test cases.

Next, we generate different faulty versions by seeding one fault into each copy of the original target program. These faults simulate the miscomputation of context variables and misuse of CIR resolution strategies. We have invited an experienced programmer to check all the seeded faults and assure their validity.

Finally, our tool executes all faulty versions with all generated test suites to measure the fault detection effectiveness of each adequacy criterion. For each criterion, the tool computes its effectiveness in detecting a specific fault in terms of *fault detection rate* [4, 9], which is defined as the ratio of the number of adequacy test suites that expose the fault to the total number of adequacy test suites generated under the criterion.

**Table 1: Description of Group 1 test suites**

| Criterion | Coverage | | | Mean |
|---|---|---|---|---|
| | Min | Mean | Max | size |
| All-Services | 100% | 100% | 100% | 27 |
| All-Services-Uses | 95.8% | 95.8% | 95.8% | 43 |
| All-2-Services-Uses | 94.8% | 95.3% | 95.7% | 50 |

## 5.2 Data Analysis

We apply the entire test pool to every faulty version to estimate their failure rates [4, 9], and select the faulty version for analysis if its (estimated) failure rate is within the range $(0.000, 0.060)$. In total, 49 versions are selected. Their median failure rate is 0.026; the mean is 0.028.

We first compare the effectiveness of different adequacy criteria. Table 2 shows the overall fault detection rates of both Group 1 and Group 2 test suites. We observe that among the testing criteria, all-2-services-uses is the most effective criterion in detecting faults in either group. In Group 2, the effectiveness of all-services is only comparable to that of random testing. Since all-services subsumes all-uses, it indicates that all-uses may not outperform random testing much, if any.

In the rest of the section, we compare the effectiveness of different criteria using Group 2. We further observe that both all-services-uses and all-2-services-uses outperform random testing — all-services-uses (all-2-services-uses) improves on random testing by 13% (23%) in terms of fault detection rate. According to the "Sd" column in Table 2, these criteria have similar standard deviations of fault detection rates, which indicates that the variations in their fault detection effectiveness are fairly consistent.

To give a clearer comparison of the different criteria, Figure 8 shows the fault detection rate of each criterion for every faulty version in one plot. The horizontal axis is the failure rate and the vertical axis represents the fault detection rate. Each line in the plot represents the variation of mean effectiveness of each criterion on the 49 faulty versions.

We observe that the effectiveness of our criteria gradually improve as the failure rates of the faulty versions increase. When the failure rate is at or above 0.02, the line for all-2-services-uses criterion already exceeds 0.8. On the other hand, random testing will not exceed this threshold until the failure rate is over 0.04, and it cannot catch up with the all-2-services-uses in the entire range. It may indicate that our criteria are more effective in detecting subtle faults than random testing.

In the experiment, we also expand every random test suite by randomly selecting additional test cases until it gives the same fault detection effectiveness as that given by the all-2-services-uses test suites. On average, the size of the former test suites needs 48% more test cases to match the effectiveness of the latter.

We further statistically compare the effectiveness of different criteria using the collected data. In Table 3, $C_{proposed}$ represents one of our proposed criteria under comparison, while $C_{random}$ represents the random criterion. We set the significance level to be 0.01. The condition $C_{proposed} > C_{random}$ means that the effectiveness of $C_{proposed}$ in detecting a particular fault is significantly higher than that of $C_{random}$ with 99% confidence level [4]. The condition $C_{random} > C_{proposed}$ is defined analogously.

Table 3 gives the number of faulty versions that fulfill the two conditions for comparison. For instance, the last row shows that all-2-services-uses is significantly more effective than random testing in detecting 30 out of 49 analyzed faults. For the remaining 19 faults, the effectivenesses of all-2-services-uses and random testing cannot be statistically distinguished. In fact, in Figure 8, when the failure rate is above 0.45, the fault detection rate of either criterion is already close to 1.0.

**Table 2: Overall fault detection rate**

| Test suites | Criterion | Fault detection rate | | | |
|---|---|---|---|---|---|
| | | Min | **Mean** | Max | Sd |
| Group 1 | All-Services | 0.15 | **0.569** | 0.87 | 0.185 |
| | All-Services-Uses | 0.26 | **0.699** | 0.97 | 0.186 |
| | All-2-Services-Uses | 0.33 | **0.757** | 0.97 | 0.164 |
| Group 2 (fixed size) | Random | 0.18 | **0.662** | 0.94 | 0.194 |
| | All-Services | 0.22 | **0.693** | 0.94 | 0.181 |
| | All-Services-Uses | 0.31 | **0.748** | 0.98 | 0.176 |
| | All-2-Services-Uses | 0.44 | **0.812** | 0.98 | 0.154 |

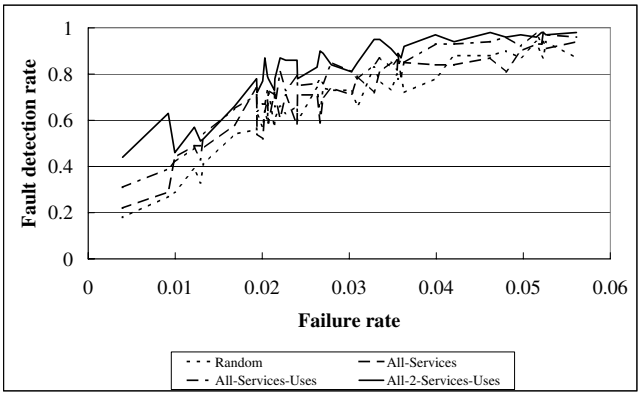


**Figure 8: Variation of fault detection rates for different criteria**

In addition, the rightmost column of Table 3 suggests that, for any of the faulty versions, random testing is never statistically more effective than any of our proposed criterion. This is in line with the above discussion on Table 2.

## 5.3 Discussion

The above experiment shows that our approach can be effective in testing context-aware applications. Among the criteria reported in the experiment, all-2-services-uses is the strongest in detecting faults, followed in turn by all-services-uses and all-services. On the downside, as only around 80% of faults can be detected effectively in the experiment, it may also suggest that faults relevant to context resolution can be intricate.

We are also concerned with the potential threats to validity of the experiment. We have only used one subject program to evaluate our proposal, and plan to conduct more evaluations using nesC programs [6] in the wireless sensor network domain because (i) correction (resolution) of sensory data sent over unreliable wireless channels is common in these applications and (ii) the barebone tinyOS merely captures and transmits contexts, and invokes actions through interrupts to execute applications. Apparently, these characteristics match the model presented in the paper. Owing to the lack of CVS or SVN records in our current subject program, to produce a faulty version, we arbitrarily select a position in the program and inject a fault such that the faulty version can be successfully compiled; we invite an independent programmer to assure whether the fault is realistic and to reject the faulty version if he judges that there is a better position to inject such a fault.

**Table 3: Statistical comparison of fault detection effectiveness**

| Proposed criterion ($C_{proposed}$) | Number of faults (49 in total) | |
|---|---|---|
| | $C_{proposed} > C_{random}$ | $C_{random} > C_{proposed}$ |
| All-Services | 1 | 0 |
| All-Services-Uses | 2 | 0 |
| All-2-Services-Uses | 30 | 0 |

In the future, we plan to use different test pools and use developer faults in additional to seeded faults to evaluate our proposal.

# 6. RELATED WORK

In this section, we review related work on context-aware pervasive computing and program adequacy testing.

Pioneering context-aware frameworks include *Context Toolkit* [2]. Other researchers further find that the middleware-centric architecture successfully leverages the development and functionality of context-aware applications. These models and approaches include, for instance, *CARISMA* [1], *EgoSpaces* [12], *RCSM* [25], and *Cabot* [23].

Handling corrupted or inconsistent context data in a pervasive environment has been investigated. Jeffery et al. [10] and Rao et al. [18] propose techniques to clean up noisy or corrupted context data streams, such as RFID signals, from sensor networks. Researchers find the middleware-centric architecture highly beneficial to the maintenance of context consistency, either by filtering, cleaning, and repairing raw context data [10, 18, 24], or by reasoning and solving context conflicts at a higher level [1].

In [14, 20], we propose to use metamorphic testing to alleviate the test oracle problem, and extend conventional data flow concepts by proposing that the computing environment can update the value of variables We also propose a family of testing criteria [14] to address an orthogonal issue, which measures how well a program is truly context-aware by testing context-aware adaptation. Unlike the present paper, our earlier work does not address the testing issues related to CIR services and the resolved contexts in the computing environments of context-aware applications. By evaluating different concurrent program executions, Wang et al. [21] propose an approach to measuring how well a test suite covers different context-aware situations without altering any context (such as by dropping or repairing). Their efforts are complementary to ours.

Our criteria also differ from conventional data flow testing counterparts. The latter focuses on comprehensive coverage of, say, data flow associations [5, 8], chains [7, 17], subpaths [5], or environmental interaction [11] of program variables. Our criteria focus directly on covering those data flow entities of context-aware applications affected by CIR services. Some concepts in our proposal are styled after those in [5, 8]. Parts of our experimental processes for evaluating our proposed adequacy criteria are similar to the empirical studies reported in [4, 9].

# 7. CONCLUSION

Pervasive computing applications adapt their behavior extensively by using and reasoning about the changing contexts. Nevertheless, context instances may be noisy and inconsistent among themselves. Context inconsistency resolution (CIR) as middleware services is a promising approach in detecting context inconsistencies and resolving them. A faulty application may, however, mishandle resolved contexts and produce incorrect results.

This paper proposes a data flow framework to model context-aware applications in the pervasive computing environment where the middleware supports CIR services. Based on the framework, we further propose a family of novel data flow testing criteria to test context-aware applications. Our criteria focus on the propagation of context variables in context-aware applications, which are potentially affected by CIR services. We have illustrated via examples and empirical evaluation that our approach is promising in detecting faults in context-aware applications.

The work is a step toward understanding the context (feature) interactions of "good" services on emerging pervasive software. Future work includes the generalization of the data flow framework to study other fundamental context-aware computing services, debugging techniques, and cost tradeoff issues.

# 8. REFERENCES


[1] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: context-aware reflective middleware system for mobile applications. *IEEE TSE*, 29 (10): 929–944, 2003.

[2] A. K. Dey, D. Salber, and G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction Journal*, 16 (2–4): 97–166, 2001.

[3] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of SIGSOFT '98/FSE-6*, pages 153–162, 1998.

[4] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE TSE*, 19 (8): 774–787, 1993.

[5] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE TSE*, 14 (10): 1483–1498, 1988.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: a holistic approach to networked embedded systems. In *Proceedings of PLDI 2003*, pages 1–11. 2003.

[7] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM TOSEM*, 16 (2): 175–204, 1994.

[8] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of ICSE '94*, pages 191–200, 1994.

[10] S. R. Jeffery, M. Garofalakis, and M. J. Franklin. Adaptive cleaning for RFID data streams. In *Proceedings of VLDB 2006*, pages 163–174. 2006.

[11] Z. Jin and A. J. Offutt. Coupling-based criteria for integration testing. *Software Testing, Verification and Reliability*, 8 (3): 133–154, 1998.

[12] C. Julien and G.-C. Roman. EgoSpaces: facilitating rapid development of context-aware mobile applications. *IEEE TSE*, 32 (5): 281–298, 2006.

[13] H. Lu. A context-oriented framework for software testing in pervasive environment. In *Doctoral Symposium, Proceedings of ICSE 2007*, pages 77–78. 2007.

[14] H. Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. In *Proceedings of SIGSOFT 2006/FSE-14*, pages 242–252. 2006.

[15] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: a coordination model and middleware supporting mobility of hosts and agents. *ACM TOSEM*, 15 (3): 279–328, 2006.

[16] L. M. Ni, Y. Liu, Y. C. Lau, and A. P. Patil. LANDMARC: indoor location sensing using active RFID. *ACM Wireless Networks*, 10 (6): 701–710, 2004.

[17] S. C. Ntafos. On required element testing. *IEEE TSE*, SE-10 (6): 795–803, 1984.

[18] J. Rao, S. Doraiswamy, H. Thakkar, and L. S. Colby. A deferred cleansing method for RFID data analytics. In *Proceedings of VLDB 2006*, pages 175–186. 2006.

[19] P. Tarr and L. A. Clarke. Consistency management for complex applications. In *Proceedings of ICSE '98*, pages 230 – 239. 1998.

[20] T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. Testing context-sensitive middleware-based software applications. In *Proceedings of COMPSAC 2004*, volume 1, pages 458–465. 2004.

[21] Z. Wang, S. G. Elbaum, and D. S. Rosenblum. Automated generation of context-aware tests. In *Proceedings of ICSE 2007*, pages 406–415. 2007.

[22] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *ACM TOIS*, 10 (1): 91–102, 1992.

[23] C. Xu, S. C. Cheung, and W. K. Chan. Incremental consistency checking for pervasive context. In *Proceedings of ICSE 2006*, pages 292–301. 2006.

[24] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In *Proceedings of ICDCS 2008*. 2008.

[25] S. S. Yau and F. Karim. An adaptive middleware for context-sensitive communications for real-time applications in ubiquitous computing environments. *Journal of Real-Time Systems*, 26 (1): 29–61, 2004.