HKU CS Tech Report TR-2006-03

Testing Context-Aware Middleware-Centric Programs: a Data Flow Approach and an RFID-Based Experimentation[†] [*]

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 context-aware software is an emerging kind of application. Smart personal digital assistants and RFID-based location sensing software are two examples. Many of these systems register parts of their context-aware logic in the middleware. On the other hand, most conventional testing techniques do not consider such kind of application logic. This paper proposes a novel family of testing criteria to measure the comprehensiveness of their test sets. It stems from context-aware data flow information. Firstly, it studies the evolution of contexts, which are environmental information relevant to an application program. It then proposes context-aware data flow associations and testing criteria. Corresponding algorithms are given. It uses a prototype testing tool to conduct experimentation on an RFID-based location sensing software running on top of context-aware middleware. The experimental results show that our approach is applicable, effective, and promising.

1 Introduction

Radio Frequency Identification or RFID is widely considered as an enabling technology ranging from Internet payment systems to supply chain management. For example, Wal-Mart[1] in USA, Metro[2] in

[*] © ACM, 2006. This is the authors' version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2006/FSE-14*)), pages 242–252. ACM Press, New York, 2006. http://doi.acm.org/10.1145/1181775.1181805.

[†] This work is supported in part by a grant of the Research Grants Council of Hong Kong (project no. HKU 7175/06E) and a grant of The University of Hong Kong.

[‡] Part of the work was done when Chan was with The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.

[§] All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Email: thtse@cs.hku.hk.

[1] Source: "Report Shows How Wal-Mart Did It", *RFID Journal* (November 14, 2005), available at http://www.rfidjournal.com/article/articleview/1983/1/1/.

[2] Source: J. Collins, "Metro Readies RFID Rollout", *RFID Journal* (January 13, 2004), available at http://www.rfidjournal.com/article/articleview/734/1/1/.

Europe, and Hutchison Ports[3] in Asia are implementing their RFID solutions. "These companies were attracted to RFID because it held out the potential of offering perfect supply chain visibility — the ability to know the precise location of any product anywhere in the supply chain at any time."[4] Because of the emerging importance of RFID technology, the testing of their software applications is essential.[5] Since precise location tracking of products (such as within an area of one square meter) is desirable and yet unattainable [19], it is difficult to determine whether the imprecision is caused by a fault in the software, hardware, or both. Moreover, although researchers [29] are tackling the test oracle problem for the software part, little progress has been reported in the literature on software test adequacy, "a criterion that defines what constitutes an adequate test" and one of the most important research problems in software testing [35].

In this paper, we shall focus on the study of the measurement problem in software test adequacy for context-aware applications. We propose a novel family of context-aware data flow test adequacy measurement criteria for context-aware middleware-centric applications. We apply and evaluate our proposal in an RFID-based location sensing program on a context-aware middleware [32, 33] with a published estimation algorithm [19]. The experimentation shows that our approach is very promising.

Pervasive computing [6, 17, 21, 29, 34] has two core properties, namely context awareness and ad hoc communication. The context-aware property enables an application to acquire and reason about environmental attributes known as *contexts*. Based on the contextual information, the application can react impromptu to changes in the environment. The ad hoc communication property facilitates mobile interactions among components of the application with respect to the changing contexts. For example, a "polite" phone for car drivers by Motorola Labs will transfer an ongoing call to the speaker phone when entering a car park, route calls to a voice mailbox in complex driving situations, and call a pre-defined emergence number if an airbag has been deployed[6].

Many existing proposals for pervasive computing, such as [6, 34], are middleware-centric. One of the major reasons is that the middleware-centric multi-tier architecture favors the development and configuration of pervasive computing applications in terms of the *separation of concern* in a highly dynamic and compositional environment, in which the middleware transparently acquires, disseminates, and infers the contexts on behalf of the applications over ad hoc networks. For the purpose of brevity, we shall limit ourselves in this paper to the study of context-aware middleware-centric systems. We shall refer to a context-aware middleware-centric program simply as a *CM-centric program*.

A component of an application hence interacts with the middleware or other components of the application via a clear, loosely coupled and context-aware interface residing in the middleware. Although context-aware middleware can handle the ad hoc communication property, the other core property — the context-aware property — requires the collaboration of application components; otherwise, an application may not be context-aware. In the rest of the paper, an application function directly invoked by a middleware will be referred to as an *adaptive action*.

Because of such a common design in context-aware systems, the program logic of a CM-centric program normally spans over the application tier and the middleware tier. Firstly, a context-aware interface at the middleware tier may invoke an adaptive action whenever the middleware detects the interesting contexts registered in the interfaces by the CM-centric program. Secondly, the invoked adaptive action would serve as an entry of the CM-centric program at the application tier to react to the interesting contexts. Thirdly, other

[3] Source: M. Roberti, "RFID Container Seals Deliver Security, Value", *RFID Journal* (October 31, 2005), available at http://www.rfidjournal.com/article/articleview/1965/1/1/.

[4] Source: "What is RFID", *RFID Journal*, available at http://www.rfidjournal.com/article/articleview/1339/1/129/.

[5] See, for example, the scope of Sun's RFID Test Center in Dallas, Texas available at http://www.sun.com/software/solutions/rfid/testcenter/.

[6] Source: R. M. Gardner, "Technology solutions toward improved driver focus", Panel on Technology and Distracted Driving, International Conference on Distracted Driving, Toronto, Canada, October 2–5, 2005.

actions of the CM-centric program will utilize the results of the adaptive actions to provide tailored services to its users. Since black box testing techniques do not consider the structural organization of program units in a context-aware system with middleware support, they are intuitively less effective than their white box counterparts in detecting faults specific to context-aware systems designed by the above approach. We shall, therefore, restrict ourselves in this paper to white-box testing techniques.

The structural organization of program logic poses challenges to white-box testing of CM-centric programs, since it would be inadequate to consider the program structure of the components in the application tier alone. The program logic resided in the context-aware interface would be overlooked, and similarly for the interactions between the middleware and the CM-centric program. Our previous work [29] has succinctly shown that conventional structural testing techniques are not effective to detect faults for context-aware software for this reason. Thus, it is desirable to enhance existing test adequacy criteria for the purpose.

The main contributions of the paper are as follows: (*i*) This paper is among the earliest ones: To our best knowledge, there is no published work in investigating the test adequacy problem for pervasive context-aware middleware-centric programs. (*ii*) New types of data flow associations are formalized to capture the context-aware dependencies specific to pervasive CM-centric programs. (*iii*) A novel family of data flow test adequacy criteria to measure test set quality and corresponding algorithms are proposed accordingly. (*iv*) The proposed family of adequacy criteria is evaluated on the *Cabot* platform, a pervasive context-aware middleware system [32, 33]. Our prototype testing tool automatically generates adequacy test sets according to our family of adequacy criteria. Evaluation of the fault-detection rates of these test sets indicates that our approach is effective and promising.

The rest of the paper is organized as follows: Sections 2 and 3 outline the technical preliminaries and testing challenges of CM-centric programs, respectively. Next, Section 4 will present our novel data flow associations followed by our testing criteria to measure the test sets comprehensiveness in Section 5. Section 6 evaluates our proposal by an RFID-based experimentation. This is followed by discussions, a review of related work, and the conclusion in Sections 7, 8, and 9.

2 CM-Centric Programs

2.1 Fundamentals

In this section, we formally explain the fundamentals on which our testing proposal relies. We formalize our model based on common designs in contemporary CM-centric projects and technologies [6, 17, 21, 29, 34].

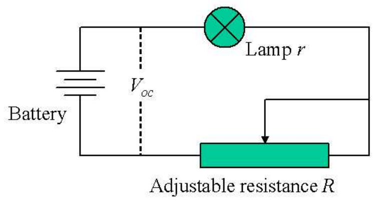
A *context* of an entity characterizes its environmental attributes. An entity can be of diverse granularities such as a composite component, an object, or a pattern. Without loss of generality, we assume that C denotes the set of all context variables and V_c denotes the domain of values applicable to the context variable c. The commonly agreed baseline of a context is a key-value pair [7]. Its formal definition is as follows:

Definition 1 (Context) *A* **context** *is an ordered couple $\langle c, v \rangle$, where c is a context variable and v is a value in V_c.*

The middleware continually detects all the context changes by assigning an instance value v to a context variable c [33]. Such detections and instantiations are transparent to applications atop the middleware. Since every context variable is unique in a CM-centric program, any occurrence of a context variable in the program will consistently give the same context evolution. Moreover, as we shall show later, the key-value model is sufficient to capture the interesting data flow information among contexts. To maintain generality, therefore, we shall refrain from using a more specific model.

Next, we formalize the notion of context-aware interaction between a middleware and an application. It is a common design that the middleware tier contains a context reasoning component, whose applications

Figure 1: Luminance control circuit example



can define a set of rules, known as *situations*, to describe the interesting conditions over context variables [6, 7, 21, 32, 34]. When an interesting condition is satisfied for a particular combination of values of context variables, an adaptive action in the application will be invoked mechanically by the middleware. We assume that the middleware uses the standard event-condition-action (ECA) approach (as in [1, 18], for instance) to invoke the adaptive actions. In the rest of the paper, we refer to the above process as a *context-aware adaptation*.

Based on this understanding of context-aware adaptation, we formulate the concept of a situation as follows:

Definition 2 (Situation) *A* **situation** *is a triple $\langle C, p, Act\rangle$, where C is a set of context variables that the situation subscribes, p is a triggering condition whose variables are in C, and Act is an adaptive action to be invoked by the middleware if and only if p is evaluated to be true, denoted by $p(C) \equiv true$. A situation is said to be* **satisfied** *if $p(C) \equiv true$; otherwise it is said to be* **outstanding**.

To enable context awareness, a context reasoning component at the middleware will receive the updates of the subscribed context variables from the context detection component [32, 33, 34]. When a situation is satisfied, the middleware will invoke an adaptive action. We assume that such a mechanism is spontaneous. We further assume an interleaving semantics for concurrent executions of multiple actions, as in the case of *RCSM* [34] and the *Cabot* middleware [32, 33]. Without loss of generality, each situation will be bound to at most one adaptive action, as in Definition 2. [7] Before we introduce the motivation example, we formally define CM-centric programs as follows:

Definition 3 (CM-Centric Programs) *A* **Context-Aware Middleware-Centric Program***, or* **CM-centric program** *for short, is a triple $\langle C, U, S\rangle$, where C is a set of context variables, U is a set of adaptive actions, and S is a set of situations such that, for every situation $s = \langle C_s, p, Act\rangle \in S$, $Act \in U$ and $C_s \subseteq C$.*

When a standard program is seen as a set of program units, a CM-centric program extends it by taking also its context-aware interfaces into account. Hence, we assume that all interactions of a CM-centric program with its environment are made through the context-aware interfaces.

2.2 A Motivation Example

In this section, we outline a fragment of a sample CM-centric program, which is a smart streetlight application [29] implemented on the *Cabot* middleware platform, to be used as a running example. The application scenario is as follows:

> *Consider a system of smart streetlights that collaborate to illuminate a city zone. It includes two features. (i) Every visitor can personalize their favorite level of illumination irrespectively*

[7] A situation involving multiple actions can be considered as a set of situations, each of which is associated with one adaptive action.

of their location within the zone. (ii) At the same time, the system maximizes energy savings by dimming unnecessary streetlights. When there is no visitor nearby, a streetlight will turn itself off. When a visitor walks toward a particular streetlight, the light detects the visitor and brightens itself. A streetlight nearby may dim itself if the closest light has provided sufficient illumination. Another streetlight may not dim, however, if there are other visitors requiring illumination. Because of the interference from other light sources and the presence of other visitors nearby, the resulting illumination for the visitor may differ from the favorite level. Finally, the system assumes that the effective distance for any streetlight to serve a visitor is at most 5 meters.

The basic feature of the smart streetlight application is to adjust the power supply of a streetlight. It is supported by a luminance control circuit as illustrated in Figure 1. The current I of the circuit can be adjusted and computed according to Ohm's Law $I = V_{oc}/(r+R)$ from the open-circuit voltage V_{oc} as well as the fixed resistance r and the variable resistance R of the streetlight. The output power of the streetlight can then be computed using the formula $P = kI^2r$, where k is a constant determined by the physical characteristics of the streetlight.

Apart from adjusting the power, the application also implements a few context-aware adaptive actions, which depend on situations. Let us concentrate on three particular situations: when the system detects a visitor nearby ($s_{visitor\_nearby}$), when the illuminance is lower than the favorite level ($s_{low\_illuminance}$), and when the illuminance is higher than the favorite level ($s_{high\_illuminance}$). Their implementations are shown in Tables 1(c), 1(d), and 1(e), respectively, in a tabular format for the ease of understanding.

Take the situation $low\_illuminance$ in Table 1(d) as an example. The middleware defines $s_{low\_illuminance}$ as a triple $\langle C_{low}, p_{low}, incCurrent\rangle$. C_{low} contains all the context variables that will be referenced or updated in the triggering condition and/or the adaptive action. The set of context variables used in p_{low}, namely, $\{E_f, E_v, d\}$, is a subset of C_{low}. Whenever the values of E_f, E_v, and d satisfy p_{low}, the middleware will invoke the adaptive action *incCurrent*. The situations $s_{visitor\_nearby}$ and $s_{high\_illuminance}$ are defined in a similar way in Tables 1(c) and 1(e), respectively.

Thus, the set of situations $S_{streetlight}$ for the application is $\{s_{low\_illuminance}, s_{high\_illuminance}, s_{visitor\_nearby}\}$. Similarly, the set of context variables used in the application scenario, as listed in Table 1(a), is given by $C_{streetlight} = \{V_{oc}, I, I_{max}, R, E_v, E_f, d\}$. The set of adaptive actions $U_{streetlight}$ is $\{incCurrent, decCurrent, computIllum\}$. In this way, the CM-centric program is $\langle C_{streetlight}, U_{streetlight}, S_{streetlight}\rangle$. *Cabot* distinguishes a context variable, say I_{max}, from the local variable in an adaptive action by adding a prefix "\$" to the latter, as in $\$I_{max}$.

The system works as follows:

(1) When a streetlight detects a visitor within a distance of 5 meters, as specified as the triggering condition in Table 1(c), the situation $s_{visitor\_nearby}$ is satisfied. The corresponding adaptive action *computIllum* in Table 1(c) will be invoked to compute the following in sequence: (*i*) the current of the circuit for the given situation based on Ohm's Law, (*ii*) the output power of the streetlight, and (*iii*) the illuminance of the incident light at the visitor's location.

(2) Similarly, the situations $s_{low\_illuminance}$ and $s_{high\_illuminance}$ will both determine whether the actual illuminance satisfies the visitor's favorite level. A satisfaction of $s_{low\_illuminance}$ means insufficient illuminance; it causes the middleware to invoke the adaptive action *incCurrent*, which increases the current by reducing the adjustable resistance.

(3) In the same way, a satisfaction of $s_{high\_illuminance}$ causes the middleware to invoke *decCurrent*, which leads to the opposite effect. The adjustment will end if the corresponding triggering condition

is satisfied. Because of the instability of location sensing [19, 33], however, the sensed distance d may vary even if the locations of both the visitor and the streetlight are fixed. Thus, the context values of E_v, R, I, and P may oscillate.

3 Testing Challenges

In this section, we report on our study in identifying three kinds of obstacle that hinder the effective application of standard data flow testing criteria [35] to CM-centric programs because of context-awareness.

Context-Aware Faults: Our previous study [29] uses a program fragment to show that conventional data flow adequacy criteria are inadequate to expose a context-aware-related fault in CM-centric programs, resulting in the overlooking of a context-aware subcondition. In brief, the context-aware condition with a missing subcondition may fail to invoke an adaptive action and may, in turn, force a test case to miss out a def-use association. As a result, a superfluous test case may need to be added to a "non-redundant test suite with respect to a specific data flow adequacy criterion.

Environmental Interplay: CM-centric programs use sensors, which may be physical or software-based, to interact with the dynamic surroundings and capture raw data for contexts. For instance, in the smart streetlight example, the context variable d, which denotes the distance of a visitor from a streetlight, is collected via the sensed location of the visitor. The sensed location is in turn computed from certain raw data captured by the location sensing subsystem. In the RFID case, the raw context is the signal strength [19] received from different RFID tags attached to the visitor. The possible walking trails of the visitor are, however, unpredictable. They are not predefined in programming logic. In addition, different sensors may oversee the same surrounding environment. The surrounding environment may change as a result of the invoked adaptive actions, such as *incCurrent*( ) to brighten a streetlight. Some raw data of contexts are, therefore, correlated in an unforeseeable manner. As a result, an adaptive action may invoke follow-up adaptive actions through their interactions with the environment. The conventional def-use association model does not cover this aspect.

Context-Aware Control Flow: A context-aware middleware triggers an adaptive action when a situation is satisfied. It involves a situation-related control flow relationship between the middleware and the program units. Although it is clearly the responsibility of the middleware that invokes the adaptive action, it is hard to identify the exact cause of the invocation, such as specific program statements or specific sensor signals that trigger a change of contexts resulting in the situation being satisfied. This is best illustrated by an example. Consider a scenario where a visitor walks near a smart streetlight. The favorite level of illuminance E_f and the distance d are continuously sensed by the middleware. At the same time, *computIllum* is continuously invoked to compute the illuminance E_v at the visitor's location. As long as their values satisfy the triggering condition $(E_f - E_v > \varepsilon) \wedge (d \leq 5)$, *incCurrent* will be invoked by the middleware. It is very difficult to enumerate all possible control flow traces for the invocation of an action and to compute the full set of traditional data flow associations. As a tradeoff, our approach captures def-use associations between pairwise invoked adaptive actions. The relevant criterion is defined in Section 5. The evaluation in Section 6 indicates that this approach is still very effective.

4 Data Flow Associations

4.1 Conventional Def-Use Associations

The terminology of our proposed data flow criteria closely resembles that of conventional approaches in [11, 14, 22]. We shall first summarize the latter in this section. We model a standard program in an application as

Table 1: Excerpts of program details for streetlight example

(a) Contexts used in sample application:

Context	**Description**
V_{oc}	voltage of open circuit
I	current
I_{max}	maximum current allowed
R	adjustable resistance
E_v	illuminance of incident light at visitor's location
E_f	visitor's favorite level of illuminance
d	distance between visitor and streetlight

(b) Constants used in sample application:

Constant	**Description**
ε	maximum tolerance between E_v and E_f
k	physical constant for output power
r	fixed resistance of streetlight
R_{max}	upper bound of adjustable resistance

(c) Code fragment for situation *visitor_nearby*:

Situation $visitor\_nearby = \langle C_{in}, p_{in}, computIllum \rangle$	
Contexts	$C_{in} = \{d, V_{oc}, R, I, I_{max}, E_v\}$
Triggering condition	$p_{in} \equiv (d \leq 5)$

Adaptive action	$computIllum$ {
	   $\$V_{oc} = V_{oc}$;
	   $\$R = R$;
	   $I = \$V_{oc}/(r + \$R)$;
	   $\$P = k * \$I * \$I * r$;
	   $\$d = d$;
	   $E_v = \$P/(\$d * \$d)$;
	 }

(d) Code fragment for situation *low_illuminance*:

Situation $low\_illuminance = \langle C_{low}, p_{low}, incCurrent \rangle$	
Contexts	$C_{low} = \{E_f, E_v, d, I, I_{max}, R\}$
Triggering condition	$p_{low} \equiv (E_f - E_v > \varepsilon) \wedge (d \leq 5)$

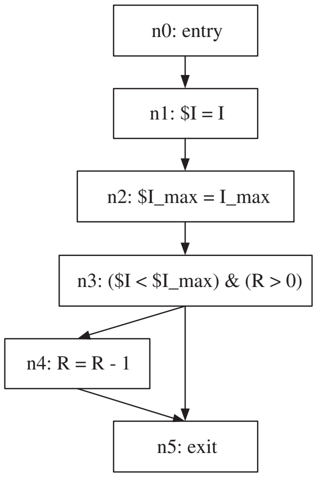
Adaptive action	$incCurrent$ {
	   $\$I = I$;
	   $\$I_{max} = I_{max}$;
	   if $(\$I < \$I_{max})$ & $(R > 0)$
	     $R = R - 1$;
	 }

(e) Code fragment for situation *high_illuminance*:

Situation $high\_illuminance = \langle C_{high}, p_{high}, decCurrent \rangle$	
Contexts	$C_{high} = \{E_f, E_v, d, R\}$
Triggering condition	$p_{high} \equiv (E_v - E_f > \varepsilon) \wedge (d \leq 5)$

Adaptive action	$decCurrent$ {
	   if $(R < R_{max})$
	     $R = R + 1$;
	 }

Figure 2: Control flow graph of program unit *incCurrent*



a control flow graph (CFG) $G = (N, E)$, where N is a set of nodes and E is a set of edges. For instance, the CFG of the program unit *incCurrent* (in Table 1(d)) is shown in Figure 2. We assume that every CFG starts with a unique entry node. A *complete path* is a path in the CFG starting from the entry node and ending at an exit node [11]. The storage of the value to a variable v, such as the occurrence of v on the left-hand side of an assignment statement, is a *definition* (or *def*) of v. A *usage* (or *use*) of a variable v refers to the fetching of a value of v, such as an occurrence of v on the right-hand side of an assignment statement or in a predicate of a statement [14].

A sub-path $\langle n_i, \ldots, n_j \rangle$ in a CFG is said to be *definition-clear* with respect to the variable x when none of $n_i, \ldots, n_j$ defines or undefines x [11]. A *def-use association* is defined as a triple $\langle x, n_d, n_u \rangle$ such that the variable x is defined at node n_d and used at node n_u, and there is a definition-clear sub-path with respect to x from n_d to n_u, exclusively [11]. For the ease of presentation, we define a predicate $def\_clear(\langle x, n_d, n_u \rangle)$ to be true if and only if $\langle x, n_d, n_u \rangle$ is a def-use association.

A complete path π_x is said to *cover* a def-use association $\langle x, n_d, n_u \rangle$ if a sub-path of π_x from node n_d to node n_u, exclusively, is a definition-clear path with respect to x [11]. Thus, if any of the definitions can reach a use in a CFG from the entry node of CFG via a def_clear path, a def-use association is identified. The set of all *reaching definitions* at node n in CFG_i is denoted by $RD_i(n)$ The techniques and algorithms for computing reaching definitions have been studied extensively in the literature [12, 13]. We shall not repeat them in this technical report.

4.2 Context-Aware Definitions and Uses

For a standard program, the conventional def-use associations presented in Section 4.1 aim at relating a definition of a variable to a use of the variable via a definition-clear sub-path. A CM-centric program extends a standard program with a set of context variables and a set of situations. The data flow associations need to be extended accordingly.

A *context variable* is a special type of variable. It serves as an ordinary variable in a standard program and, at the same time, is used for exchanging contexts with the pervasive environment. In this section, we examine the definitions and usages of variables and, in particular, context variables.

4.2.1 Definitions of Variables

According to existing proposals of context-aware systems [34, 32], a context variable can be defined and updated via either an *assignment statement* or *sensing the environmental contexts*. They are described as follows:

Type 1: An **update via an assignment statement** refers to the occurrence of a variable in a statement that stores the value of the variable. It coincides with the definition of a variable in the conventional def-use association [11]. Consider the statement "$n12 : E_v = \$P/(\$d * \$d)$" of the program unit *computIllum* in Table 1(c), where E_v is a context variable. The occurrence of E_v is said to be a definition of the context variable E_v.

Type 2: An **environmental update**, or an update by sensing environmental contexts, is achieved through the environmental context acquisition module of the context-aware middleware. It is analogous to updating the content of a memory location (or a variable) of a program through an external means. It is not attained by an explicit programming construct in a program unit, such as an assignment operator. The "definition" of a variable in conventional def-use association cannot capture this type of update.

We further observe that every context variable is unique in a CM-centric program. Hence, every occurrence of such a variable anywhere in the program refers to the same value. When an update of a context variable v is made, any reference to v in any statement will immediately be changed. We therefore extend the definition of a variable for conventional def-use associations to take the above observations on context variables into account as follows:

Definition 4 (Definitions (def) of Variables) *A* **definition** *or* **def** *of a variable v is an occurrence of v (i) in a statement where a value is assigned to v, or (ii) in a statement where v is a context variable that can be instantiated by the sensing environmental contexts.*

For simplicity, we refer to a definition of a context variable c as a *context definition*, denoted by def_c. We further use Def_c to represent the set of all the context definitions of the context variable c in the program.

4.2.2 Usages of Variables

We continue to examine the usages of variables in a CM-centric program. For ordinary variables, the conventional definition of usage of a variable strictly applies. We do not observe any difference. For context variables, a usage occurrence of a context variable can occur in either a program unit or a situation. We call them action use and situation use, respectively. They correspond to the only two types of position for developers to code the reference of a variable in a CM-centric program. They are further elaborated as follows:

An *action use* coincides with the conventional definition of a use of a variable. It refers to an occurrence of a context variable in a statement that fetches the value of the variable, such as on the right-hand side of an assignment statement.

A *situation use* is the usage of a context variable in the triggering condition of a situation. For instance, in the situation $s_{low\_illuminance}$ in Table 1(d), all the occurrences of the context variables E_f, E_v, and d are situation use. Note that a situation use can be recorded only if the corresponding situation is satisfied; otherwise the usage is not observable.

Definition 5 (Usages (or Uses) of Variables) *A* **usage** *or* **use** *of a variable v is an occurrence of v (i) in a statement that fetches the value of v $($known as* **action use**$)$*, or (ii) in the triggering condition of a situation if v is a context variable $($known as* **situation use**$)$.

We denote a situation use of a context variable c by $suse_c$ and an action use by $ause_c$. The set of uses, the set of situation uses, and the set of action uses of c are denoted by Use_c, $SUse_c$, and $AUse_c$, respectively.

4.2.3 Update-Uses of Variables

It is apparent from Definitions 4 and 5 that a context definition and a usage of the same context variable may occur in the same statement. Consider, for instance, the context variable d in the code in Table 1(c). Suppose the current value of d sensed by the middleware is d_0, where $d_0 \leq 5$. Thus, the triggering condition is satisfied and *computIllum* is invoked. Let us focus on the statement $\$d = d$. Suppose there is an environmental update of the variable d immediately before the execution of this statement. The value of the variable d in the statement no longer depends on the previous value d_0. Instead, it will depend on the environmental update.

Such a scenario of an environmental update followed by a usage occurrence of the same variable is possible if a context variable appears in the triggering condition of a situation, in a statement that fetches the value of the variable, or in the predicate of a statement. As a result, an occurrence of an environmental update can be determined from the source code by determining a usage occurrence of a context variable. We refer to such an occurrence of the environmental update as an update-use occurrence and define it as follows:

Definition 6 (Update-Use Occurrences of Variables) *An* **update-use** *occurrence of a context variable c is an occurrence containing a context definition def_c and a context use use_c of c, where def_c refers to the instantiation of c due to the sensing of environmental contexts.*

For simplicity, the set of context definitions of a context variable c involving update-use occurrences will be denoted by $UDef_c$. The set of other context definitions of c will be denoted by $ODef_c$. Thus, $Def_c = UDef_c \cup ODef_c$, and $UDef_c \cap ODef_c = \emptyset$.

4.2.4 Context-Aware Flow Graph

We would like to adapt the concepts in data flow testing to test CM-centric programs. Based on the observations in Sections 4.2.1 to 4.2.3, we define a *Context-aware Flow Graph* (*CaFG*) to describe the control and data flow information in a CM-centric program. The procedure for constructing a CaFG as well as its components for a CM-centric program $\langle C, U, S\rangle$ is as follows:

Step 1: Every program unit $Act \in U$ is modeled as a CFG. [8] We denote the CFG as $G_{Act} = (N_{Act}, E_{Act})$. A sample CFG is shown in Figure 2.

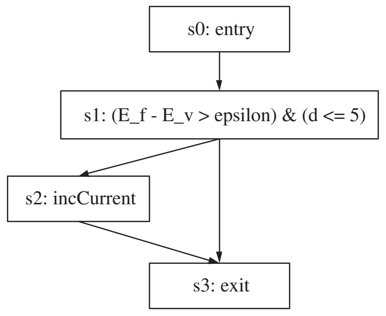
Step 2: Each situation $s = \langle C_s, p, Act\rangle \in S$ is modeled as a CFG referred to as a *situation graph* and denoted by $G_s = (N_s, E_s)$. It consists of the entry node, an exit node, a predicate node representing the triggering condition p, and a node that follows the true branch of the predicate node and invokes the adaptive action Act. A sample situation graph for the situation $s_{low\_illuminance}$ is shown in Figure 3.

Step 3: For every node n in a given CFG or situation graph that contains at least one update-use occurrence, n is marked as an *update-use node*. We denote the set of all update-use nodes as $NAct$. If the context definitions are represented at the node level, then $N_{uu} = \bigcup_{c\in C} UDef_c$. Each update-use node $n \in N_{uu}$ is the entry point to a subgraph called an *update-use subgraph* and denoted by $G_n = (N_n, E_n)$.

The update-use subgraph consists of the entry node, an exit node, nodes that we shall call *use nodes*, and phantom nodes. A sample update-use subgraph for the update-use node $n2$ of Figure 2 is shown in Figure 4. A use node, depicted as a standard rectangle, is annotated with the original statement of the update-use node but does not represent an update-use occurrence. For each update-use occurrence, a *phantom node* representing an environmental update is introduced. A control flow edge, which we call a *phantom control flow*, will connect the entry node via a phantom node to the corresponding use node. Another phantom

[8] Other researchers model a standard program as a CFG [11, 22].

Figure 3: Situation graph of $s_{low\_illuminance}$



control flow edge will directly connect the entry node to the use node. When entering the subgraph, one may choose whether or not to visit a phantom node, that is, whether or not to have an environment update.

Step 4: We also add an edge from every node containing a context definition to the predicate note of the corresponding situation graph of the context definition. We refer to this edge as a *cross-boundary context-aware data flow*. We denote the set of such edges as E_{CC}. Examples of cross-boundary context-aware data flows will be given in Section 4.3.2.

Now we can have the following definition of CaFG.

Definition 7 (Context-Aware Flow Graph) *A* **context-aware flow graph** (**CaFG**) *with respect to a CM-centric program* $\langle C, U, S\rangle$ *is a directed graph* $G = (N, E)$ *such that*

$$N = \left(\bigcup_{Act \in U} N_{Act}\right) \cup \left(\bigcup_{s \in S} N_s\right) \cup \left(\bigcup_{n \in N_{UU}} N_n\right)$$

and

$$E = \left(\bigcup_{Act \in U} E_{Act}\right) \cup \left(\bigcup_{s \in S} E_s\right) \cup \left(\bigcup_{n \in N_{UU}} E_n \cup E_{CC}\right)$$

where N_{UU} *is the set of all update-use nodes.*

Figure 5 shows the CaFG of the streetlight application program. Note that situation graphs are created for $s_{low\_illuminance}$ and $s_{visitor\_nearby}$. Update-use subgraphs are also created for the update-use nodes $n2$, $n11$, $n16$, $s1$, $s5$, and $s9$ involving the context variables E_f, V_{OC}, d, and I_{max}. Directed edges with hollow arrows represent the invocation relationships among graphs.

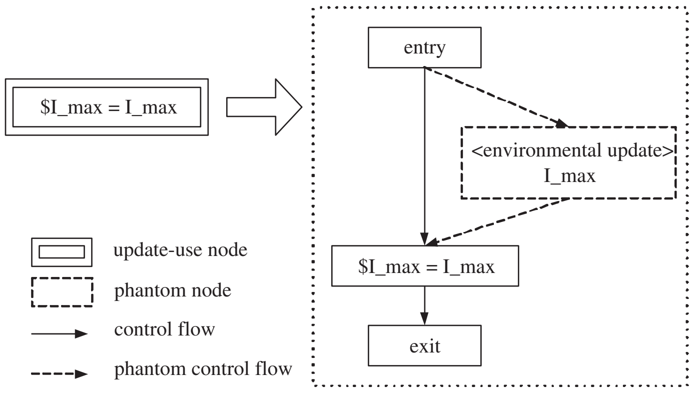
4.3 Def-Use Associations for CM-Centric Programs

We are interested in determining the definition-usage relationships in a CM-centric program with a view to conducting data flow testing. In this section, we discuss the data flow associations that we find to be special to context variables in CM-centric programs. For ordinary variables, we adopt the conventional approach to compute data flow associations (see Section 4.1).

Variables in a CM-centric program can appear in program units, in situations, or both. There are, by exhaustion, three types[9] of def-use association to link up a definition and a usage of a variable:

[9] The type of def-use association such that the definition occurrence appears in a situation and the usage occurrence of the same variable appears in a program unit is analogous to type (DU1), because definition occurrences appearing in situations can only be update-use occurrences.

Figure 4: Update-use subgraph of node $n2$ in Figure 5



Type (DU1): Both a definition occurrence and a usage occurrence of a variable appear in a situation. Type (DU2): A definition occurrence of a variable appears in a program unit and a usage occurrence appears in a situation. Type (DU3): Both occurrences are within program units. We elaborate on each of these types below.

4.3.1 Def-Use Associations of Type (DU1)

We recall that the only way to update a context variable in a situation is through an environmental update, and the only way to specify a reference of a context variable in a situation is via a usage occurrence of the variable. Hence, type (DU1) refers to an update-use of a context variable c in a node n_s followed by a situation use of c in a node n_e such that there is a definition-clear path with respect to c from n_s to n_d. We note that every context variable in a CM-centric program is unique. An environmental update of c will affect all occurrences of this variable, which should include its occurrence in node n_e. The only possible definition-clear path is, therefore, $\langle n_e, n_e \rangle$. Thus, the definition-usage association is $\langle c, n_e, n_e \rangle$, where n_e is a triggering condition of a situation. In Figure 3, for example, the def-use associations of type (DU1) include: $\langle E_f, s1, s1 \rangle$ and $\langle d, s1, s1 \rangle$. The corresponding situation graphs are included as components of the CaFG in Figure 5. For the update-use subgraph of the predicate node $s1$, for instance, the above two def-use associations are $\langle E_f, p13, p15 \rangle$ and $\langle d, p14, p15 \rangle$, respectively.

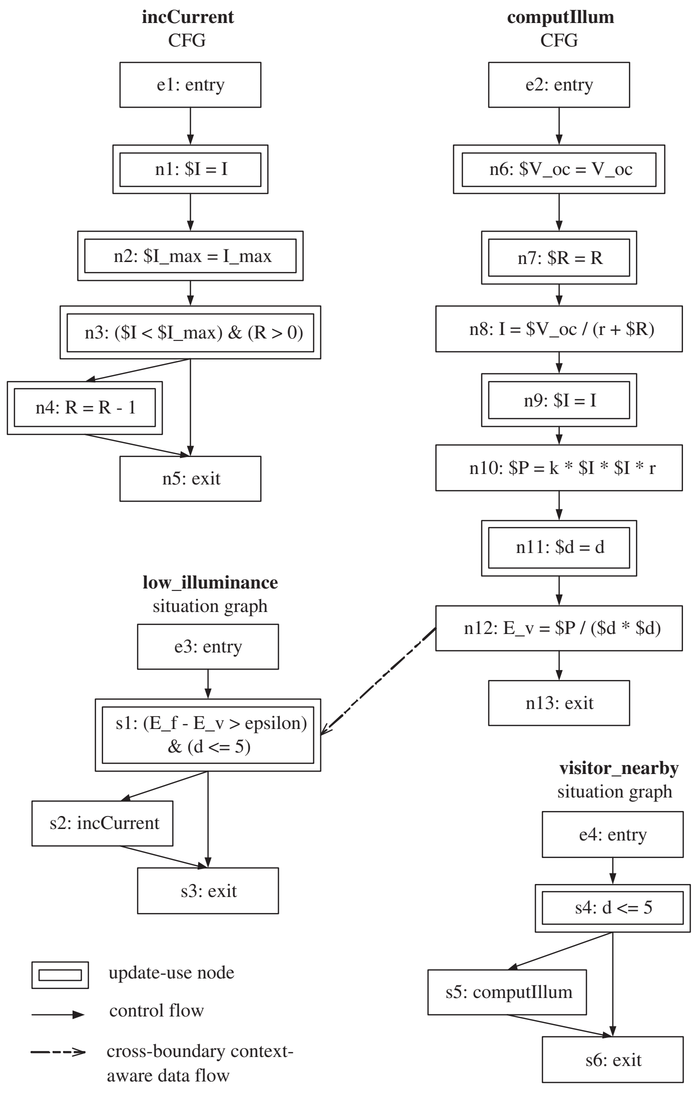
4.3.2 Def-Use Associations of Type (DU2)

Type (DU2) refers to the case when a definition occurrence of a variable appears in a program unit and a usage occurrence of the same variable appears in a situation. In this type of def-use association, the data flow goes out of the boundary of the program unit in which the definition occurrence appears. We refer to this kind of data flow as cross-boundary context-aware data flow, which was introduced in Section 4.2. As adaptive actions are autonomously invoked by their respective situations, we cannot derive explicit control dependencies between different program units. The cross-boundary context-aware data flow provides a means for us to capture the def-use associations of variables that have definition and usage occurrences in different program units.

In Figure 5, there are two cross-boundary context-aware data flow edges $(n17, s1)$ and $(n17, s5)$, which indicate two def-use associations of this type, namely, $\langle E_v, n17, s1 \rangle$ and $\langle E_v, n17, s5 \rangle$.

We note that, for a def-use association involving a definition and a situation use of a variable c, the types (DU1) and (DU2) correspond to the definitions of c by $UDef_c$ and $ODef_c$, respectively. We combine the two

Figure 5: Context-aware flow graph of streetlight program



types and define def-situ associations as follows:

Definition 8 (Def-Situ Associations) *A* **def-situ association** α *for a context variable c is a triple* $\langle c, def_c, suse_c\rangle$ *satisfying the following four conditions:* (i) $suse_c = \langle C, p, Act\rangle$, (ii) $c \in C$, (iii) $p(C) \equiv true$, *and* (iv) $def\_clear(\langle c, def_c, suse_c\rangle) = true$.

For the type (DU1), the condition $def\_clear(\langle c, def_c, suse_c\rangle)$ in Definition 8 is always true because, in an update-use subgraph of a predicate node in a situation graph, it is apparent that there is a definition-clear sub-path from every phantom node (representing environmental update) to the corresponding use node. For the type (DU2), if the value of a context variable *c* is synchronized in a CM-centric program immediately after def_c, the condition $def\_clear(\langle c, def_c, suse_c\rangle) = true$ always holds as well, since the predicate $def\_clear(\langle c, def_c, suse_c\rangle)$ is equivalent to whether there exists a definition-clear sub-path from def_c to where *c* is synchronized.

Figure 6: The def-situ associations algorithm

```
Algorithm   COMPUTE DEF-SITU
  Input     The CaFG for a CM-centric program $\langle C, U, S \rangle$;
  Output    The set DEF-SITU of all def-situ associations;

  Associate every node in the CaFG of every program unit
     with the set of context definitions;
  Associate every node in each situation graph with the
     set of update-use occurrences and situation uses;
  for each $c \in C$
     Collect $UDef_c$, $ODef_c$ and $SUse_c$;
     if $ODef_c = \emptyset$
        for each $def_c \in UDef_c$ and each $suse_c \in SUse_c$
           if $def\_clear(\langle c, def_c, suse_c \rangle) = true$
              Add $\langle c, def_c, suse_c \rangle$ to DEF-SITU;
     else
        for each $def_c \in ODef_c$ and each $suse_c \in SUse_c$
           if $def\_clear(\langle c, def_c, suse_c \rangle) = true$
              Add $\langle c, def_c, suse_c \rangle$ to DEF-SITU;
```

The algorithm to compute the set of def-situ associations for a CM-centric program $\langle C, U, S \rangle$ is given in Figure 6. Suppose that the number of lexical tokens of the program is n and each evaluation of *def_clear* takes time m. It takes $O(n)$ time to identify and collect all the context definitions, update-use occurrences, and situation uses by scanning the CaFG. By keeping the sets of context definitions and situation uses in a tree, it will take $O(nlog(n))$ time to construct the list or retrieve a record. In addition, the algorithm requires that every context variable in the set C will be enumerated. Thus the algorithm will have a time complexity of $O(n + |C|mnlog(n))$, or simply $O(|C|mnlog(n))$.

4.3.3 Def-Use Associations of Type (DU3)

Type (DU3) corresponds to the circumstance where both the definition and the usage in a def-use association occur within the same program unit. We divide this type into two subtypes:

Let **subtype** (a) represent the case where a context definition in a program unit (that is, an adaptive action) Act_i is referred by another program unit (another adaptive action) Act_j via a definition-clear path. We recall that a def-situ association α defines a cross-boundary context-aware data flow edge from a context definition of a variable c in action Act_i to a situation use of the variable in a situation. The association requires the corresponding situation to be satisfied. Once the situation is satisfied, the associated adaptive action, Act_j should be invoked. We use the notation $Act_i \overset{\alpha}{:\Rightarrow} Act_j$ to indicate that Act_i carries out the context definition of the triple α and Act_j is invoked by the situation that receives the situation use of α.

In general, an adaptive action or an environment may continue to enable the situation to be satisfied. Hence, the middleware will continue to activate functions. Ideally, the middleware may invoke many, possibly infinite, number of actions one after another, such as

$$Act_i \overset{\alpha}{:\Rightarrow} Act_j \overset{\beta}{:\Rightarrow} Act_k \overset{\gamma}{:\Rightarrow} \cdots$$

while the values of the variables propagate from one action to another.

Since it is undecidable to determine whether a path is feasible or whether an execution is terminating, we compute in a pairwise manner the related data flow associations between two consecutive calls of adaptive actions, that is, between two program units with an $:\overset{\alpha}{\Rightarrow}$ relationship. For every relationship $Act_i :\overset{\alpha}{\Rightarrow} Act_j$, the execution order that can be statically identified is that Act_j is invoked only if def_c (the context definition of α) is executed. Hence, we first compute the set of all reaching definitions at def_c in the CFG of Act_i, and denote the set by $RD_i(def_c)$. Then, if any of the definitions can reach a use in Act_j from the entry node of Act_j via a def_clear path, a def-use association is identified.

In Figure 5, $\langle I, n13, n1\rangle$ is an example of subtype (a). It is derived from the relationship $visitor\_nearby \overset{\langle E_v, n17, s1\rangle}{:\Longrightarrow} low\_illuminance$, in which I is a reaching definition at $n17$ and can reach $n1$ via a def_clear sub-path $(n17, s1, s2, n1, n1)$.

Subtype $(\boldsymbol{b})$ corresponds to the case where a definition occurrence in a program unit is being referred in the same program unit. We would like to argue that the set of def-use associations derived using conventional approaches may not be sufficient to reflect the context-aware nature of data flows even within a program unit. Consider the variable $\$I$ in the program unit *incCurrent*. By intraprocedural data flow analysis, such as computing the set of reaching definitions at each node of the CFG) [11, 13, 35], we can easily derive the def-use associations $du_1 = \langle \$I, n1, (n3, n4)\rangle$ and $du_2 = \langle \$I, n1, (n3, n5)\rangle$. [10] As discussed above, since the definition of $\$I$ in $n1$ depends on the value of I, which has a context definition in the program unit *computIllum*, we can use backward tracing to find two sub-paths $\pi_1 = (n17, s1, s2, n0, n1, n2, n3, n4)$ and $\pi_2 = (n17, s1, s2, n0, n1, n2, n3, n5)$ that covers du_1 and du_2, respectively.

However, if there is a certain program unit *Act* that contains a context definition def_I of I whose set of reaching definitions contains $n1$, the executions of π_1 and π_2 are not sufficient to cover all the data dependency cases of $\$I$, because *computIllum* and *Act* impose different contexts to the execution of *incCurrent*. Consequently, the sub-paths $\pi'_1 = (def_I, s1, s2, n0, n1, n2, n3, n4)$ and $\pi'_2 = (def_I, s1, s2, n0, n1, n2, n3, n5)$ are also needed to track the definition of I inside *Act*.

As a result, we find it necessary to derive def-use associations of subtype (b) with respect to each def-situ association. In other words, if $Act_i :\overset{\alpha}{\Rightarrow} Act_j$, each def-use association du within the program unit Act_j is annotated with α, which means that there exists a sub-path π from Act_i to Act_j containing α such that π covers du. If there are two def-situ associations α and β, both covering du but each involving a different sub-path, then du should be split into two def-use associations of subtype (b), one annotated with α and the other annotated with β.

In Figure 5, for the relationship $visitor\_nearby :\overset{\alpha}{\Rightarrow} low\_illuminance$, where $\alpha = \langle E_v, n17, s1\rangle$, the def-use associations of subtype (b) include $\langle \$I, n1, (n3, n4)\rangle$ and $\langle \$I_{max}, n2, (n3, n4)\rangle$, to name a few.

Combining the two subtypes in (DU3), we formally define pairwise context-aware def-use association as follows:

Definition 9 (Pairwise DU Associations) *A* **pairwise context-aware def-use association** (*or simply a* **pairwise du association**) *du with respect to* (i) *two program units* Act_i *and* Act_j *and* (ii) *a def-situ association* $\alpha = \langle c, def_c, suse_c\rangle$ *such that the relationship* $Act_i :\overset{\alpha}{\Rightarrow} Act_j$ *holds, is of one of the following two types:*

(a) $du = \langle c', def_{c'}, au_{c'}\rangle$ *in which* $def_{c'}$ *is from* Act_i, $au_{c'}$ *is from* Act_j, $def_{c'} \in RD_i(def_c)$, *and* $def\_clear(c', e_j, au_{c'}) = true$, *where* e_j *refers to the entry node of* Act_j.

(b) $du = \langle x, def_x, use_x\rangle$ *in which both* def_x *and* use_x *occur in* Act_j, *and* $def_x \in RD_j(use_x)$.

According to Definition 9, since every pairwise du association is strictly related to a def-situ association, the computation of pairwise du associations can be founded on Algorithm COMPUTE *DEF-SITU* of Figure 6. Figure 7 shows the algorithm to compute the set of pairwise du associations with respect to a specific

[10] For predicate uses inside a program unit, we adopt the standard representation of predicate use or p-use [11, 35].

Figure 7: The pairwise du associations algorithm

```
Algorithm  COMPUTE PAIRWISE-DU
  Input    A def-situ association $\alpha = \langle c, def_c, suse_c \rangle$ such that
               $Act_i :\overset{\alpha}{\Rightarrow} Act_j$;
  Input    $G_i$ and $G_j$, respective CFGs for $Act_i$ and $Act_j$;
  Output   The set $PAIRWISE\text{-}DU(\alpha)$ of all pairwise du
               associations with respect to $\alpha$;

$RD_i(def_c)$ = the set of reaching definitions at $def_c$ in $G_i$;
for each variable $c'$ such that there exists $def_{c'} \in RD_i(def_c)$
    for each node $nd$ in $G_j$
        if $nd$ has a use of $c'$ and $def\_clear(c', e_j, nd) = true$
        // Note: $e_j$ is the entry node of $Act_j$
            Add $\langle c', def_{c'}, nd \rangle$ to $PAIRWISE\text{-}DU(\alpha)$;
for each node $nd$ in $G_j$
    $RD_j(nd)$ = the set of reaching definitions at $nd$ in $G_j$;
    for each variable $x$ that has a use at $nd$
        if $x$ has a definition $def_x$ in $G_j$ such that $def_x \in RD_j(nd)$
            Add $\langle x, def_x, nd \rangle$ to $PAIRWISE\text{-}DU(\alpha)$;
```

def-situ association. We use n' and $|C'|$ to denote, respectively, the number of lexical tokens and the number of involved context variables in a program unit. The evaluation of *def_clear* in this algorithm corresponds to the computation of reaching definitions at a node in a program unit and has time complexity of $O(n')$. If we assume that the set of reaching definitions for every node of every CFG is already available, and each evaluation of *def_clear* takes time m, then the algorithm takes at most $O(|C'|mn'^2)$ time to compute the set of pairwise du associations for each def-situ association. In general, to obtain all the pairwise du associations of a CM-centric program, all the reaching definitions can be computed in advance in one go, with a time complexity of $O(n^2)$, where n is the total number of lexical tokens in the program. By considering the complexity of the algorithm COMPUTE *DEF-SITU*, the time complexity for obtaining all the pairwise du associations will be $O(n^2 + |C|mnlog(n) + |C|mn^2)$, or simply $O(|C|mn^2)$.

5 Test Adequacy Criteria

To measure the quality of a test set for a CM-centric program, we define in this section our test adequacy criteria using the data flow associations described in Section 4. First of all, we recall that a test adequacy criterion C_1 is said to *subsume* a test adequacy criterion C_2 if every test set that satisfies C_1 will also satisfy C_2 [35]. This term will be used in the rest of the paper.

According to Definition 3, a CM-centric program differs from a standard program by having a set of context variables and a set of situations. Our first test adequacy criterion is to exercise every situation at least once. Whenever a situation is satisfied, the middleware will invoke the corresponding adaptive action. On the other hand, an adaptive action may be invoked by another adaptive action. Hence, a test set that covers all adaptive actions does not necessarily cover all situations.

Criterion 1 (All Situations) *A test set T satisfies the* **all-situations criterion** *for a CM-centric program $\langle C, U, S \rangle$ if and only if, for each situation $s \equiv \langle C', p, Act \rangle$ in S, the complete path of at least one test case t in T includes the predicate node of s such that $p(C') \equiv true$.*

We refine the all-situations criterion to cover all def-situ associations, whose definition is as follows:

Figure 8: Subsumption hierarchy of context-aware test adequacy criteria

$$\begin{tikzcd}
\text{all-pairwise-du-associations} \arrow[d] \\
\text{all-def-situ-associations} \arrow[d] \\
\text{all-situations}
\end{tikzcd}$$

Criterion 2 (All Def-Situ Associations) *A test set T satisfies the* **all-def-situ-associations criterion** *for a CM-centric program $\langle C, U, S\rangle$ if and only if, for every def-situ association α obtained from the algorithm COMPUTE DEF-SITU, there is at least one test case t in T such that $def\_clear(\alpha) = true$.*

We note that a usage occurrence in a situation should either be an update-use or have a corresponding context definition in a program unit. Moreover, according to condition (*iii*) of Definition 8, the corresponding situation should be satisfied. Hence, a test set satisfying the all-def-situ-associations criterion should also satisfy the all-situations criterion. In this connection, the all-def-situ-associations criterion subsumes the all-situations criterion.

We further refine the above criterion by linking up a definition occurrence and a usage occurrence in two adaptive actions through a situation. If the above criterion is seen to address type (DU2), then we want our third criterion to address type (DU3). In brief, for every def-situ association α, this criterion requires an adequate test set to exercise all pairwise du associations that are relevant to α. It is simple to observe from condition (*i*) of the following definition that the criterion subsumes the all-def-situ-associations criterion.

Criterion 3 (All Pairwise DU Associations) *A test set T satisfies the* **all-pairwise-du-associations criterion** *for a CM-centric program $\langle C, U, S\rangle$ if and only if the following two conditions are satisfied: (i) T satisfies the all-def-situ-associations criterion; and (ii) for every pairwise du association du obtained from the algorithm COMPUTE PAIRWISE-DU(α), there is at least one test case in T such that $def\_clear(du) = true$.*

We note that the definition of α can be generalized to be a series of adaptive actions. By so doing, more general form of the above two criteria can be developed. The extension is not difficult.

For completeness, we also include the all-outstanding-situations criterion that requires all situations of a CM-centric program to be outstanding. While intuitively simple, it can nevertheless detect the context-aware faults discussed in Section 3 (Testing Challenges).

Criterion 4 (All Outstanding Situations) *A test set T satisfies the* **all-outstanding-situations criterion** *for a CM-centric program $\langle C, U, S\rangle$ if and only if, for each situation $s \equiv \langle C', p, Act\rangle$ in S, the complete path of at least one test case t in T includes the predicate node of s such that $p(C') \equiv false$.*

Figure 8 shows the subsumption hierarchy for the family of test adequacy criteria defined in this section. The individual relationships have been discussed above.

6 Evaluation

This section reports on the experimentation results of our test adequacy criteria, namely, all-situations, all-def-situ-associations and all-pairwise-du-associations. We also benchmark our proposal against Frankl and Weyuker's all-uses criterion [11]. Furthermore, the study is also a novel attempt to test RFID-based systems. Our prototype testing tool automates the test set generation and evaluation processes.

6.1 Experimental Design

We use *Cabot* [32] and its evaluation application as our testbed. The testbed consists of a middleware supporting context-aware reasoning and triggering, and an application implementing the *LANDMARC* RFID-based location sensing algorithm [19]. Datagram-based communications are adopted for transmission of contexts at runtime. *Cabot* continually collects radio frequency strength signals (context values) from various RFID tags. Based on the different radio frequency strength signals and other environmental contexts, the middleware triggers different adaptive actions to allow the application to estimate the location of a tracking tag.

Our experiment focuses on the testing of the *LANDMARC* application. We use the original implementation as the golden version to check for failures of the faulty versions. The experiment consists of three steps.

Firstly, our tool generates 40 test sets for each adequacy criterion given in Section 5. For comparison purpose, we also generate 40 test sets based on the standard all-uses criterion [10, 11, 14]. The test cases are randomly selected from a test pool containing 8 000 radio frequency strength signal data collected from different locations of tracking tags from online RFID sensor networks. Since full coverage of testing criteria is infeasible in practice, an upper bound on the number trials in selecting test cases is set for each criterion. A test set generation process thus stops when full coverage is achieved or the upper bound has been reached. A test case that increases the coverage of the criterion concerned will be added to the adequacy test set. It also reports the percentage of coverage. Our automated method to select test cases is similar to that described in [14].

Figure 9 shows the algorithm that our tool uses to generate test sets for the all-pairwise-du-associations coverage. The algorithm accepts the test pool and a hashtable *dua* as inputs and produces an adequacy test set as output. *dua* is a hierarchical hashtable having an entry for each def-situ association, as well as an entry for each pairwise du association related to the respective def-situ associations. For a sequence of test cases, *dua* is covered in a cumulative way; the test cases will achieve full coverage when all the pairwise du associations in *dua* are marked as covered.

In the algorithm, *trace* is a sequence of variable definition and usage occurrences that are recorded when a test case is selected and forced to execute. For any individual pairwise du association covered, the algorithm carries out two rounds of forward and backward searches to find out the corresponding definition-clear path.[11]

It is worth noting that the construction of test sets for all-situations and all-def-situ-associations criteria are byproducts of the algorithm EVALUATE *PAIRWISE-DU*.

Secondly, we create 50 faulty versions, each having one fault. There are 18, 11, 13, and 8 faults in the context repairing logic, context-aware rules, situational application configuration, and estimation logic, respectively. Simple faults such as data conversion format mismatches are excluded. All versions are assured by an experienced software developer.

Finally, our tool applies all the generated test sets to all versions to evaluate their fault-detection capabilities. For each adequacy criterion, the *fault-detection rate* [10] is defined as the ratio of the number of corresponding adequacy test sets (each containing at least one test case that exposes the fault) to the total number of corresponding adequacy test sets with respect to the criterion. For the purpose of comparison, we use all 8 000 test cases from the test pool to test every faulty version. The corresponding percentage of failed test cases denotes the failure rate of the version. Our tool also compares automatically the execution results of faulty versions and that of the golden version. It computes the fault-detection rate for each criterion.

[11] Suppose the average length of an execution trace is L and the number of context variables is $|C|$. Then, for each test case t selected, the time cost to evaluate the coverage of t will be $O(|C|L^4)$. When the upper bound of the number of selected test cases is N, the total time cost to generate a test set for the all-pairwise-du-associations criterion will be $O(N \cdot |C|L^4)$.

Figure 9: Algorithm for generating all-pairwise-du-associations adequacy test set

```
Algorithm   EVALUATE PAIRWISE-DU
  Input     TestPool;
  Input     dua, the set of pairwise du associations;
  Output    TestSet for the all-pairwise-du-associations
               criterion;
  int $i = 0$;
  while dua is not fully covered and $i < UPPER\_BOUND$
      $i++$;
      Randomly select a non-redundant t from TestPool;
      Execute t to obtain trace;
      for each element $e_{f_1}$ in forward search of trace
         if $e_{f_1}$ is a situation use of context variable c
            for each element $e_{b_1}$ in backward search from $e_{f_1}$
               if $e_{b_1}$ is a definition of c
                  if dua contains an entry $\langle c, e_{b_1}, e_{f_1} \rangle$
                        subdua = the set of pairwise du
                               associations of $\langle c, e_{b_1}, e_{f_1} \rangle$;
                        SEARCH-DU($e_{f_1}$, subdua);
      if the coverage of dua increases
         Add t to TestSet;

Procedure   SEARCH-DU($e_{f_1}$, subdua)
  for each element $e_{f_2}$ in forward search from $e_{f_1}$
     if $e_{f_2}$ is a usage occurrence of variable v
        for each element $e_{b_2}$ in backward search from $e_{f_2}$
           if $e_{b_1}$ is a definition of c
              if subdua contains an entry $\langle v, e_{b_2}, e_{f_2} \rangle$
                    Mark $\langle v, e_{b_2}, e_{f_2} \rangle$ in dua as covered;
```

Table 2: Statistics of test sets generated according to different adequacy criteria

Criterion	Coverage (in %) of generated test sets (40 sets per criterion)		
	min.	avg.	max.
all-situations	96.3	99.4	100
all-def-situ-associations	93.8	96.6	99.2
all-pairwise-du-associations	86.7	87.1	87.4
all-uses	71.2	72.6	72.9

6.2 Data Analysis

Through the analysis by Algorithms COMPUTE *DEF-SITU* and COMPUTE *PAIRWISE-DU*, the *LANDMARC* testbed program has a total of 81 situations, 129 def-situ associations, 7 682 pairwise du associations, and 177 def-use associations with reference to the standard all-uses criterion. Our tool generates 40 test sets for every test adequacy criterion. For the all-situations, all-def-situ-associations, all-pairwise-du-associations, and all-uses adequacy criteria, we set the upper bound of number of selected test cases to be 300, 1 000, 2 000, and 2 000. The generated test sets are summarized in Table 2. We note that the coverage rate of all-pairwise-du-associations and all-uses are relatively low even if their test sets are selected from 2 000 test cases. We infer that this is due to infeasible paths in the program units [10, 11].

Table 3: Overall fault-detection rates

Type of faulty versions	Fault-detection rate of adequacy test sets			
	Criterion	min.	avg.	max.
Situations (10 faulty versions)	all-situations	0.075	**0.548**	0.875
	all-def-situ-associations	0.15	**0.723**	1.0
	all-pairwise-du-associations	0.175	**0.878**	1.0
	all-uses	0.025	**0.730**	1.0
Actions (25 faulty versions)	all-situations	0.0	**0.326**	1.0
	all-def-situ-associations	0.0	**0.496**	1.0
	all-pairwise-du-associations	0.05	**0.617**	1.0
	all-uses	0.0	**0.472**	1.0
Total (35 faulty versions)	all-situations	0.0	**0.389**	1.0
	all-def-situ-associations	0.0	**0.544**	1.0
	all-pairwise-du-associations	0.05	**0.691**	1.0
	all-uses	0.0	**0.546**	1.0

After applying the entire test pool to all the 50 faulty versions, we find 8 faults which cannot be exposed by any test case. In addition, there are 6 faulty versions that have failure rates higher than 0.95 and another version with a failure rate of 0.34. A deeper investigation of these 7 highly detectable faults shows that all of them are located on the key paths of the module to compute the final estimated tracking location, so that they have high chances to be exposed. Since these 15 faulty versions may not be suitable for determining the fault-detection capabilities of the adequacy test sets, they are discarded in the sequel. The remaining 35 faulty versions have failure rates within a range of 0.0003 to 0.06, with an average of 0.011.

We first compare the criteria irrespective of the failure rates of faulty versions. The overall fault-detection rates of the three adequacy criteria are given in Table 3. All the faulty versions are further categorized according to whether a fault is related to a situation or an action. Out of the 35 faulty versions, 10 and 25 belong to the respective category. For each category, the minimum, average, and maximum fault-detection rates with respect to each criterion are computed. On average, the all-pairwise-du-associations criterion gives the highest fault-detection rates.

In order to study the change of fault-detection rates with respect to failure rates, we present line plots of the fault-detection rates of various adequacy criteria against the faulty version, as shown in Figure 10. The 35 faulty versions are sorted along the horizontal axis in ascending order of their failure rates. The vertical axis scales the values of the fault-detection rates. The values for the same adequacy criterion are connected to demonstrate the trend of fault-detection with the increase of failure rates. We find that the fault-detection rates of the three criteria proposed in Section 5 generally increase with the increase of failure rates. On the other hand, the fault-detection rate of the all-uses criterion fluctuates greatly, ranging from less than 0.1 to nearly 1.0, along the middle segment of the failure rate.

A finer inspection of Figure 10 shows that the greatly fluctuating segment of the all-uses line starts from the 12th faulty version and ends at the 26th faulty version, representing failure rates from 0.002 to 0.012. We therefore classify the 35 faulty versions into three intervals known as A, B, and C, partitioned by the failure rates of 0.002 and 0.012. Figure 11 shows the fault-detection rates of the four criteria in each of the intervals. We find that all the criteria are promising in interval C (with our three criteria almost reaching full detection). In interval B, only all-pairwise-du-associations have a detection rate of more than 0.8, outperforming all-uses by 37%. In interval A, the fault-detection rate of every criterion is below 0.2 as a result of the very low failure rate (average 0.0007). In particular, the effectiveness of all-uses drops most rapidly in interval A to become the lowest among all criteria.

Figure 10: Fault-detection rates of adequacy criteria with respect to failure rates

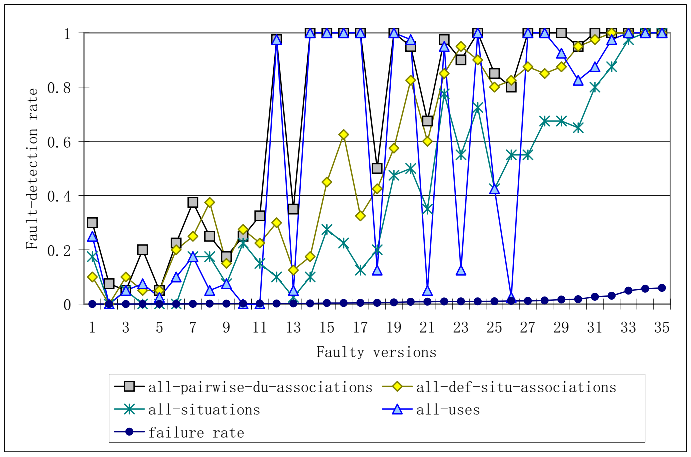
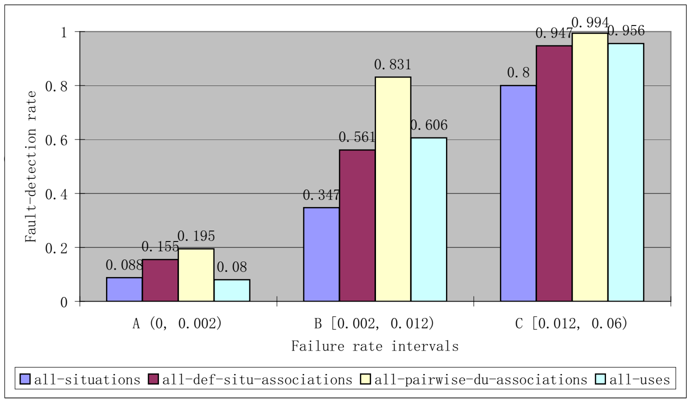


Figure 11: Fault detection rates with respect to different failure rate intervals



For a better comparison of fault-detection effectiveness of different criteria, we carry out statistical hypothesis testing [25] based on the collected data. The method is similar to that in [10]. When applying a criterion to test a faulty version program, the experiments on different test sets are mutually independent. While the testing results of different test sets follow a binomial distribution, the 40 test sets constitute a sufficiently large sample to support the assumption that the average differences of fault-detection rates between any two criteria are approximately normally distributed [25]. For any two criteria C_1 and C_2 under comparison, let r_1 and r_2 be their respective fault-detection rates. In the following, we set the null hypothesis as $H_0: \; r_1 \leq r_2$ and the alternate hypothesis as $H_1: \; r_1 > r_2$ with a significance level of $\alpha = 0.05$. We use the notation $C_1 > C_2$ to denote the result that the null hypothesis $r_1 \leq r_2$ can be rejected, meaning that r_1 is significantly higher than r_2. In addition, we also use the notation $C_1 = C_2$ to represent the result that another null hypothesis of neither $r_1 \leq r_2$ nor $r_2 \leq r_1$ can be rejected, meaning a comparable effectiveness of C_1 and C_2 in detecting a certain fault.

Table 4 compares the effectiveness of different categories of criteria. The all-situations, all-def-situ-associations, all-pairwise-du-associations, and all-uses criteria are denoted by C_s, C_{ds}, C_{pdu}, and C_u, respectively. The "Description" column lists the hypothesis tests that the category of faulty versions have passed. For instance, the first row illustrates that the category $C_{pdu} > C_u$ contains a total of 15 faulty versions, in which the fault-detection rate of all-def-situ-associations is significantly higher than that of all-uses. Moreover, the 15 faulty versions are classified into the three failure rate intervals shown in Figure 11.

Table 4: Effectiveness comparison of different criteria

	Description	Number of faulty versions			
		Total	Failure rate intervals		
			(A)	(B)	(C)
1	$C_{pdu} > C_u$	15	5	7	3
2	$C_{pdu} = C_u$	20	5	9	6
3	$C_{ds} > C_u$	10	2	6	2
4	$C_u > C_{ds}$	11	1	8	2
5	$(C_{pdu} = C_u) \wedge (C_{pdu} = C_{ds}) \wedge (C_{ds} = C_u)$	8	4	0	4
6	$(C_{pdu} > C_{ds}) \wedge (C_{pdu} > C_s)$	15	2	10	3

It is worth noting that any two of the categories given in Table 4 are not necessarily mutually exclusive.

7 Discussions

Our preliminary experimentation shows that our adequacy criteria are promising in measuring the quality of test sets for CM-centric programs. Firstly, our technique has been demonstrated to effectively handle faults related to contexts. As shown in Table 3, the all-pairwise-du-associations criterion has an average fault-detection rate of 0.878 for detecting faults related to situations.

Secondly, among our three adequacy criteria, the all-pairwise-du-associations criterion demonstrates a fault-detection rate which is higher than or equal to the standard all-uses criterion. The hypothesis test in Table 4 supports that the former is statistically more effective than the latter in detecting 15 out of all the 35 faults, while the latter never outperforms the former. On the other hand, the all-def-situ-associations and all-uses criteria have comparable fault-detection rates in general, with averages of 0.544 and 0.546, respectively, as shown in Table 3. The 3rd and 4th rows in Table 4 also reflect the comparability.

The fluctuating behavior of the all-uses criterion shown in Figure 10 is interesting. Figure 11 shows that, for faulty versions above the upper bound of interval B (with failure rates a little higher than 0.01), almost every criterion looks promising and is close to full detection, while for faults with failure rates below the lower bound of interval B (lower than 0.002), none of the four criteria seems effective. This observation is further supported by hypothesis testing in the 5th row of Table 4, which shows that the three most effective criteria demonstrate statistically comparable behaviors in almost half of the total faulty versions (8 out of 19) in intervals A and B. Furthermore, in interval B, the fault-detection rate of the all-uses criterion is by no means steady. We have computed the standard deviations of the fault-detection rates for all the data flow criteria reported. The one for the all-uses criterion is 0.443, which is almost twice as that of any of the other three criteria. This phenomenon warrants further investigation.

We would also like to highlight potential threats to validity of the experiment. For internal validity, we note that the experiment uses one program and fault types relevant to the program to evaluate our criteria. Its test pool is based on 8 000 real RFID data only. The 8 faults that could not be exposed by any test case might actually be detected by further testing if a larger test pool were available. In addition, our prototype testing tool may contain faults, even though it has been "thoroughly" tested. More empirical studies are warranted to further evaluate the proposal.

8 Related Work

In this section, we review related literature on the testing of context-aware software systems. To do this, we first brief review the background of *Cabot* [32, 33], the middleware on which our evaluation has been conducted.

Context-aware computing is an important research topic of pervasive computing [24]. According to the survey in [7], the *Active Badge* system [30] is the pioneer study. Since then, there are many proposals such as *Gaia*, [23], *RCSM* [34], *CARISMA* [5], *CARMEN* [4], *MobiPADS* [6], and *Cabot* [32, 33]. Among these, *RCSM* is the first general-purpose context-aware middleware proposal and *Cabot* is the first proposal to support (in)consistency constraint checking in its middleware. Since *Cabot* is designed with the quality assurance mind-set, it is a good platform for us to evaluate our testing proposal.

The prototype application of *Cabot* implements the RFID-based location sensing algorithm *LANDMARC* [19] and uses the middleware to context-sensitively adjust detected context information so as to reduce the average errors of the sensed locations. This has been evaluated in previous work [32, 33]. In the present work, *Cabot* with its *LANDMARC* implementation is used as the testbed. Details of the empirical study have been described in Section 6.

In the rest of this section, we review the testing research for pervasive computing. To our best knowledge, it is not plentiful. Axelsen et al. [3] propose a specification-based approach to test reflective software in an open environment, and use a random selection strategy to produce test inputs. When the execution sequence of any test input violates the specifications, it detects a failure. Flores et al. [8] apply temporal logic to define contextual expressions in context-aware software. They then use a kind of category partitioning to break up the temporal logic expressions. Intuitively, they require each partition to be covered. However, their work does not provide test case generation guidelines. Our previous work [29] basically generates multiple context tuples as test cases to check whether the outcomes satisfy isotropic properties of context relations. It is a black-box approach with a focus on the test oracle problem. The proposal presented in this paper is a program-based approach that does not rely on any specifications. It focuses on the test set adequacy problem. It complements our previous work.

We also review constraint-based approaches as the context-aware interface can be regarded as a set of constraints. Tai [27], among others such as [2, 16, 28, 31], investigate a rule-based approach to construct predicate-based test cases. He focuses on specifications of programs consisting only of predicate rules. These rules cannot trigger adaptive actions that may produce instant side effects to other parts of the same expression. Jin and Offutt [15] propose a set of constraint-based test adequacy criteria for software systems that need to interact with the environment. However, their target programs have explicit "system calls" to identify the locations where the environment can affect the program or be affected. As we have explained in Section 3, a context-aware middleware-centric application interacts with the environment in a much more complicated way.

When compared with researches in test adequacy criteria of standard programs, our notions are in line with them [11, 22]. Various empirical studies in conventional data flow testing are performed and evaluated in [9, 10, 14]. Investigations on data flow testing at the integration level can be found in [12, 20].

9 Conclusion

Pervasive context-aware software is a novel kind of applications that challenges existing testing techniques. In this paper, we have investigated the testing of context-aware middleware-centric pervasive applications. We have formalized the notion of context-aware data flow entities. Based on them, we have proposed a novel family of test adequacy criteria to measure the quality of test sets. Corresponding algorithms are given. We have also reported the experimentation results of an RFID-based location-sensing system, in which we applied a testing tool to automate the construction of adequate test sets and the evaluation of test results. The empirical results are promising.

Our approach is applicable and effective to pervasive computing. The context-aware program logic normally spans over the application tier and the middleware tier. The part residing in the middleware tier is ignored by conventional testing methods. Failures specific to context-aware features relevant to the middleware is, therefore, difficult to be exposed by conventional testing techniques.

We plan to conduct more case studies on different applications on other platforms to further evaluate our approach. For programs with very low failure rates (< 0.002), our experimentation shows that the results are not very satisfactory. We plan to propose further testing techniques for programs with very low failure rates.

10 Acknowledgement

We are grateful to S. C. Cheung and Chang Xu for their experimental platform as well as their discussions leading to improvements in the paper.

References

[1] A. Adi and O. Etzion. Amit: the situation manager. *The VLDB Journal*, 13 (2): 177–203, 2004.

[2] A. Avritzer, J. P. Ros, and E. J. Weyuker. Reliability testing of rule-based systems. *IEEE Software*, 13 (5): 76–82, 1996.

[3] E. W. Axelsen, E. B. Johnsen, and O. Owe. Toward reflective application testing in open environments. In *Proceedings of the Norwegian Informatics Conference* (*NIK 2004*), pages 192–203. Tapir, Trondheim, Norway, 2004.

[4] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Context-aware middleware for resource management in the wireless Internet. *IEEE Transactions on Software Engineering*, 29 (12): 1086–1099, 2003.

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

[6] A. T. S. Chan and S.-N. Chuang. MobiPADS: a reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29 (12): 1072–1085, 2003.

[7] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381. Dartmouth College, Hanover, New Hampshire, 2000.

[8] A. Flores, J. C. Augusto, M. Polo, and M. Varea. Towards context-aware testing for semantic interoperability on PvC environments. In *Proceedings of the 2004 IEEE International Conference on Systems, Man, and Cybernetics* (*SMC 2004*), volume 2, pages 1136–1141. IEEE Computer Society Press, Los Alamitos, California, 2004.

[9] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT International Symposium on the Foundations of Software Engineering* (*FSE-6*), pages 153–162. ACM Press, New York, 1998.

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

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

[12] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16 (2): 175–204, 1994.

[13] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, New York, 1977.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering* (*ICSE '94*), pages 191–200. IEEE Computer Society Press, Los Alamitos, California, 1994.

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

[16] S. H. Kirani, I. A. Zualkernan, and W.-T. Tsai. Evaluation of expert system testing methods. *Communications of the ACM*, 37 (11): 71–81, 1994.

[17] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37 (7): 56–64, 2004.

[18] A. K. Mok, P. Konana, G. Liu, C.-G. Lee, and H. Woo. Specifying timing constraints and composite events: an application in the design of electronic brokerages. *IEEE Transactions on Software Engineering*, 20 (12): 841–858, 2004.

[19] 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.

[20] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20 (5): 385–403, 1994.

[21] A. Ranganathan and R. H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, 7 (6): 353–364, 2003.

[22] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11 (4): 367–375, 1985.

[23] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1 (4): 74–83, 2002.

[24] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8 (4): 10–17, 2001.

[25] A. F. Siegel and C. J. Morgan. *Statistics and Data Analysis: an Introduction*. Wiley, New York, 1998.

[26] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19 (2): 239–252, 1997.

[27] K.-C. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22 (8): 552–562, 1996.

[28] W.-T. Tsai, R. Vishnuvajjala, and D. Zhang. Verification and validation of knowledge-based systems. *IEEE Transactions on Knowledge and Data Engineering*, 11 (1): 202–212, 1999.

[29] 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 the 28th Annual International Computer Software and Applications Conference* (*COMPSAC 2004*), volume 1, pages 458–465. IEEE Computer Society Press, Los Alamitos, California, 2004.

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

[31] E. J. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20 (5): 353–363, 1994.

[32] C. Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundation of Software Engineering* (*ESEC 2005/FSE-13*), pages 336–345. ACM Press, New York, 2005.

[33] C. Xu, S. C. Cheung, and W. K. Chan. Incremental consistency checking for pervasive context. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), pages 292–301. ACM Press, New York, 2006.

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

[35] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29 (4): 366–427, 1997.