Static Slicing for Pervasive Programs [*][†]

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

W.K. Chan [‡]
City University of Hong Kong
wkchan@cs.cityu.edu.hk

T.H. Tse [§]
The University of Hong Kong
thtse@cs.hku.hk

Abstract

Pervasive programs should be context-aware, which means that program functions should react according to changing environmental conditions. Slicing, as an important class of code analysis techniques, can clarify the dependence between program artifacts and observable system states to facilitate debugging, testing, and other analyses. Existing program slicing techniques, however, do not take the contextual environment into account, resulting in incomplete slices for such kind of program. To tackle this problem, this paper proposes a novel static slicing approach. It develops a graphic representation that captures the context-triggered invocations and the pervasive concurrency features. We have also developed an algorithm to check the propagation dependence in processing inter-thread data dependence. Further optimizations are discussed.

Keywords: Pervasive concurrent program, static slicing.

1. Introduction

Pervasive computing aims at integrating seamlessly computational entities to their environments so that computing can take place anywhere and at any time [18]. Two core features are *context awareness* and *ad hoc communication*. The former enables entities to be aware of and react to their environmental attributes such as temperature, light intensity, and location, known collectively as *contexts*. The latter facilitates instant interactions among entities based on the changing contexts. The middleware-centric approach is a popular type of architecture in many pervasive computing projects [1, 2, 12, 17, 21, 22]. The middleware is responsible for capturing, disseminating, and reasoning about contextual information, for the underlying communication, and for the scheduling of context-aware applications. Context-aware applications only carry out high-level tasks relevant to the end users. In this paper, we concentrate our study on context-aware middleware-centric programs. We shall call them *CM-centric programs*.

Many researches in context-aware computing focus on conceptual models for the representation and reasoning of contexts. The study of the maintenance of context-aware programs has not drawn much attention. To our best knowledge, no work in the literature studies code-level analyses for CM-centric programs. We propose a static slicing approach for this purpose.

Slicing is a code-based technique widely used in software engineering including program analysis, debugging, testing, maintenance, and reverse engineering [20]. A *slice* is a set of statements in a program that may affect the computed values at some program location such as a particular occurrence of a variable. Slicing techniques can be *static* or *dynamic*, differentiated by whether the slice is derived from the code or obtained from dynamic execution traces with specific inputs, respectively.

Our work, like [7], is built on a tuple space model. A novel graph representation, called *context-aware control flow graph* or *CaCFG* for short, is proposed to capture both context-aware invocations and the structure of CM-centric programs. We also describe a basic concurrency model for these programs in terms of *context-aware threads* or *Ca-threads* for short.

Based on CaCFG and the concurrency model, we propose a static slicing algorithm that extends the algorithms for slicing concurrent programs proposed in [8, 14]. In particular, for control dependence and intra-thread data dependence, our algorithm follows the standard backward traversal [20]. For inter-thread data dependence, however, we show that the constraints of threaded witness [8] and trace witness [14] can be relaxed for CM-centric programs because the graph representation of each Ca-thread in CaCFG is strongly connected. We propose a technique to decide on *propagation dependence*, which determines whether a slice includes a given inter-thread data dependence.

The main contributions of the paper are three-fold: (a) To our best knowledge, it is the first slicing technqiue that takes the pervasive environment into account. (b) It develops a context-aware control flow graph specifically for CM-centric programs and convenient for static slicing. (c) A static slicing algorithm for CM-centric programs is proposed.

The rest is organized as follows. Section 2 reviews existing work on static slicing. Section 3 sketches the fundamentals of CM-centric programs and demonstrates the graph representation and concurrency model. The slicing algorithm is given in Section 5, followed by optimization approaches discussed in Section 6. The conclusion and future work are given in Section 7.

2. Related Work on Static Program Slicing

Static program slices are computed from the static information of program code. For a program p, the *slicing criterion* is represented by a tuple $\langle s, V \rangle$ in which s is a statement in p and V is a subset of the variables in p. The static slice of p with respect to the slicing criterion $\langle s, V \rangle$ comprises a subset of statements in p that can affect the value of the variables in V at statement s.

For sequential programs, the effect of variables among statements is propagated through *control dependence* and *data dependence*, which are defined based on the control flow graph (CFG) [20]. Ottenstein and Ottenstein [15] developed a program dependence graph (PDG), in which every node represents a statement of the program and all the nodes are connected by directed edges representing control dependence or data dependence. By translating a CFG to a PDG, the computation of static slice becomes a

Figure 1. Sample interference dependence

(a)　　　　　　　　　(b)
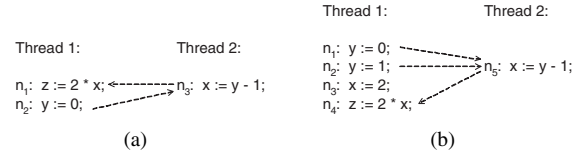
reachability problem in PDG: for a certain slicing criterion represented as a node in the PDG, the slice comprises the transitive closure starting from the criterion node through backward traversal of control dependence and data dependence edges. Horwitz et al. [6] extended the technique to interprocedural slicing. Early work on static slicing of concurrent programs were direct extensions of the approach for sequential programs in terms of reachability analysis on PDG-like graphs [5, 23]. One of the additional dependence relationships brought about by concurrency was referred to as *interference dependence* in [8]. It is induced by the data dependence on shared variables between two concurrent and interleaving threads. A statement n is interference dependent on another statement m, denoted by $m \overset{id}{\rightarrow} n$, if (i) n and m can execute concurrently, (ii) n contains a usage of some variable x, and (iii) m has a definition occurrence of x. In Figure 1(a), for instance, Threads 1 and 2 execute in parallel so that $n_2 \overset{id}{\rightarrow} n_3$ and $n_3 \overset{id}{\rightarrow} n_1$.

Using the above backward traversal algorithm for slicing sequential programs, in Figure 1(a), n_2 should be added to the slice of n_1, which requires n_2 to be executed before n_1. However, the execution path from n_2 to n_1 is infeasible for Thread 1. Krinke [8] highlighted the intransitivity of interference dependence and proposed a more precise algorithm, which checks whether there is a path from each source node of every interference dependence edge in the CFG to the last visited node of the same thread.

In the concurrency model used in [8], threads are nested with the fork-join structure and communicate with one another via shared variables. Threads do not synchronize explicitly. In addition, branches of predicate nodes are treated as non-deterministic branches, as it is well known that the feasible path problem is undecidable; precise slicing based on such treatment is referred to as *optimal slicing*. Although this kind of simplified model is considered fundamental to address a wide range of concurrent applications [13], it is undecidable in general [13].

Nanda and Ramesh [14] extended the work in [8] and proposed a more precise algorithm to deal with loop-carried data dependence. However, the solution for interference dependence in their algorithm is similar to that in [8]. In other words, they consider the reachability of interference source nodes in terms of control flow while

ignoring the influence of variable propagation. Let us consider the example in Figure 1(b). By applying the algorithms in [8, 14], n_1 should be included in the slice of n_4 simply because $n_1 \xrightarrow{id} n_5$, $n_5 \xrightarrow{id} n_4$, and there is a path in Thread 1 from n_1 to n_4. Nevertheless, n_1 cannot affect the value of x and z in n_4 because in any interleaving sequence of Thread 1 and Thread 2, the definition of y in n_1 is killed. Krinke [9] raised this problem with an example, but did not give a solution. The work [9] also extended their previous work [8] to include procedure calls so that recursion can be largely handled efficiently.

3. CM-Centric Programs

To model CM-centric programs, we first revisit contexts and situations. Like [13], we adopt a simplified model. A *context* is a data abstraction of environmental attributes of computing entities [4, 19]. Like [7], we also model context operations based on the tuple space model.

Definition 1 (Context) A **context tuple**, denoted by t, is a triple (*name*, *type*, *value*), where *name* is the unique identifier for some environmental attribute of an entity, *value* is the data abstraction of the environmental attribute, and *type* prescribes the data type of *value*. The set of context tuples is called the **context space** and is denoted by T.

A context space provides consistent views of contexts so that the name component of each context tuple is unique. Formally, for any $t_1, t_2 \in T$, $t_1 \neq t_2 \Rightarrow t_1.name \neq t_2.name$). We, therefore, use a context name to refer uniquely to its belonging **context variable** c. The entire set of context variables for a CM-centric program is denoted by C. That is, $C = \{t.name \mid t \in T\}$.

We define two *atomic* and *non-blocking* operations that address the usage and definition of context tuples in the context space:

$$\begin{array}{ll}
\text{get}(c): & \text{update}(t): \\
\quad \textbf{if } \exists t \in T(t.name = c) & \quad \textbf{if } \exists t' \in T(t'.name = c) \\
\quad\quad \textbf{return } t & \quad\quad T := (T\backslash\{t'\}) \cup \{t\} \\
\quad \textbf{else return } null & \quad \textbf{else } T := T \cup \{t\}
\end{array}$$

The get operation retrieves from the context space the current copy of context tuple with a specified context variable, or returns a constant *null* if the context variable does not exist. The update operation replaces a context tuple with another context tuple that has the same context variable, or inserts a new context tuple if the specified context variable does not previously exist in the context space.

It is also popular to include transparent context reasoning in the middleware tier [1, 2, 16, 21, 22]. Developers may define a set of rules, say, in predicate logic to describe conditions over interesting context variables.

These conditions are matched to a set of applications so that, when current context values satisfy one or more of these conditions during runtime, the relevant applications will be invoked spontaneously by the middleware. Such a process is triggered by context changes and is referred to as a *context-aware adaptation*. The corresponding conditions and applications are called *situations* and *adaptive actions*, respectively.

Definition 2 (Situation) A **situation** s is an ordered pair (C_s, p), where C_s ($\subseteq C$) is a set of context variables and p is a triggering condition with predicate variables in C_s. s is said to be **satisfied** if $p(\text{get}(C_s)) = true$.

In Definition 2, the operation get is directly extended to deal with a set of context variables C_s, where $\text{get}(C_s) = \{\text{get}(c) \mid c \in C_s\}\backslash\{null\}$. We define another atomic and non-blocking operation that returns the evaluation of the triggering condition based current context values in the context space:

$$\begin{array}{l}
\text{evaluate}(s): \\
\quad \textbf{return } p(\text{get}(C_s))
\end{array}$$

An **adaptive action** is a program unit invoked by the middleware when some situation is satisfied. Without loss of generality, we restrict each adaptive action to be bound with one unique situation, and denote it by $\text{act}(s)$, which means that it will be invoked when $\text{evaluate}(s) = true$. The extension to multiple binding is not difficult. We also assume that adaptive actions do not invoke each other explicitly, but only through the middleware.

Contexts may be changed by environmental effects. For example, the middleware may detect and update the temperature context according to thermometer sensors, or refresh the current bandwidth according to a networking monitor. This part of context update differs significantly from that performed by adaptive actions because the environmental effect is outside the scope of the programming logic. A formal definition of CM-centric programs is as follows:

Definition 3 (CM-Centric Program) A **context-aware middleware-centric program**, or **CM-centric program** for short, is a tuple $(C, S, A, Ev, \text{act})$, where C is a set of context variables; S is a set of situations such that $C_s \subseteq C$ for any $s \in S$; A is a set of adaptive actions; Ev ($\subseteq C$) is a set of context variables that can be updated by environmental effects; and act is a bijection $\text{act}: S \to A$ such that for any $s \in S$, the adaptive action $\text{act}(s)$ will be invoked when $\text{evaluate}(s) = true$.

We adapt the application scenario of a *smart delivery system* from [3] for illustration.

> In a supermarket, each pallet stores a type of goods and has a desired quantity level. Each van delivers a type of goods and may move near the pallets. When a pallet

Table 1. Example smart delivery system
$(C, S, A, Ev, \text{act})$, where $\quad C = \{g_p, q_p, q_d, q_l, g_v, q_v, d\}$,
$S = \{s_d, s_u, s_o\}$, $A = \{compLedger, replenish, withdraw\}$,
$Ev = \{g_p, q_p, q_d, g_v, d\}$, $\text{act}(s_d) = compLedger$,
$\text{act}(s_u) = replenish$, and $\text{act}(s_o) = withdraw$.

	g_p	type of goods stored in pallet
	q_p	amount of goods stored in pallet
Context	q_d	desired amount of goods in pallet
variables	q_l	ledger amount of goods in pallet
	g_v	type of goods delivered by van
	q_v	amount of goods delivered by van
	d	distance between pallet and van
	s_d	$(\,C_d = \{g_p, g_v, d\}$, $\quad p_d = (g_p = g_v) \wedge (d < 10)\,)$
Situations	s_u	$(\,C_u = \{g_p, g_v, d, q_d, q_l\}$, $\quad p_u = (g_p = g_v) \wedge (d < 10) \wedge (q_l < q_d)\,)$
	s_o	$(\,C_o = \{g_p, g_v, d, q_d, q_l\}$, $\quad p_o = (g_p = g_v) \wedge (d < 10) \wedge (q_l > q_d)\,)$
	$\text{act}(s_d)$	*compLedger* { $q_l := q_p + q_v;$ }
Adaptive actions	$\text{act}(s_u)$	*replenish* { if $(q_v < MAX)$ $q_v := q_v + 1;$ }
	$\text{act}(s_o)$	*withdraw* { if $(q_v > 0)$ $q_v := q_v - 1;$ }

detects that a nearby van delivering the same type of goods is moving within the effective delivery distance (say, 10 meters), the situation *detected* or s_d is said to be satisfied, and an adaptive action *compLedger* will be invoked to compute the ledger amount of goods in the pallet. Also, if the ledger is lower than the desired quantity, the situation *understock* or s_u is said to be satisfied, and the van will be replenished by invoking the adaptive action *replenish*. On the contrary, if the ledger amount exceeds the desired level, the situation *overstock* or s_o is fulfilled, and the goods will be withdrawn from the van by invoking the adaptive action *withdraw*.

A fragment of the implementation is shown in Table 1.

4. Context-Aware Control Flow Graph

Based on the CM-centric program model, we extend the conventional CFG to construct a graphic representation of a CM-centric program as follows:

Step 1: For each adaptive action a, we construct a CFG $G_a = (N_a, E_a)$, where each node $n \in N_a$ represents a statement and each directed edge $e \in E_a$ represents a

control flow edge. For each node n, we derive two sets $D(n)$ and $U(n)$ representing the set of variables defined and used at n, respectively. We further assume that G_a contains a unique entry node $entry(G_a)$ and a unique exit node $exit(G_a)$.

Step 2: For situation s, we treat the $\text{evaluate}(s)$ operation as a special predicate node n_s, which is called *situation node* and annotated with the triggering condition. Note that $D(n_s) = \emptyset$ and $U(n_s) = C_s$. Clearly, every node in the CFG of $\text{act}(s)$ is *control dependent* on n_s. In line with existing designs of pervasive systems such as [21, 22], we represent the *true* branch of n_s as a control flow edge $(n_s, entry(G_{\text{act}(s)}))$ while not explicitly showing the *false* branch. Another control flow edge $(exit(G_{\text{act}(s)}), n_s)$ is also needed to signify that variations of context space may re-evaluate s and invoke $\text{act}(s)$ iteratively.

Step 3: We create an *environmental node* n_{ev} to represent the *environmental action* that carries out all the environmental influence on the context space. We have $D(n_{ev}) = Ev$ and $U(n_{ev}) = \emptyset$.

We call the resultant graph a *context-aware control flow graph* or CaCFG for short, and give the formal definition as follows. The CaCFG for the example program is shown in Figure 2.

Definition 4 (CaCFG) The **context-aware control flow graph** or **CaCFG** of a CM-centric program $(C, S, A, Ev, \text{act})$ is a directed graph $G = (N, E_{cf})$ such that the set of nodes

$$N = \bigcup_{a \in A} N_a \cup \{n_s \mid s \in S\} \cup \{n_{ev}\}$$

and the set of control flow edges

$$E_{cf} = \bigcup_{a \in A} E_a \cup \{in_s \mid s \in S\} \cup \{out_s \mid s \in S\}$$

where $in_s = (exit(G_{\text{act}(s)}), n_s)$ represents the control flow from the exit node of $\text{act}(s)$ to n_s, and $out_s = (n_s, entry(G_{\text{act}(s)}))$ represents the control flow from n_s to the entry node of $\text{act}(s)$.

Obviously, the transitive closure of E_{cf} defines an equivalent relation on N. Every equivalence class forms a strongly connected subgraph of G. There are a total of $|S| + 1$ strongly connected subgraphs. The environmental node n_{ev} forms a strongly connected subgraph without in-coming or out-going control flow edges. We denote it as a simple graph G_{ev}, where $G_{ev} = (\{n_{ev}\}, \emptyset)$. Each of the remaining strongly connected subgraphs consists of a situation node n_s for some situation s together with $N_{\text{act}(s)}$, the nodes of the bound adaptive actions of s. We denote it by G_s, where $G_s = (N_{\text{act}(s)} \cup \{n_s\}, E_{\text{act}(s)} \cup \{in_s, out_s\})$ for any $s \in S$. We shall discuss the interactions between the application tier and the middleware tier in the next section.

Figure 2. CaCFG of smart delivery system

Figure 2. CaCFG of smart delivery system
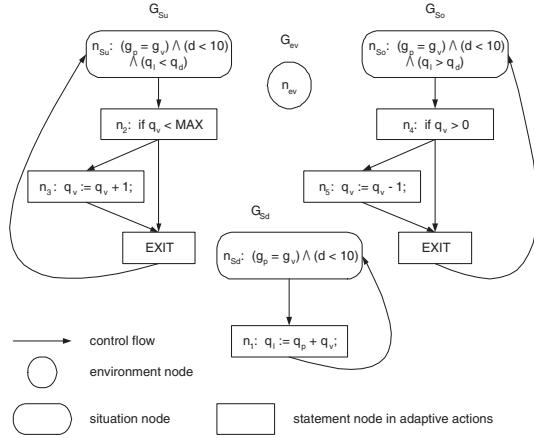
5. Slicing CM-Centric Programs

5.1. Inter-Thread Context Dependence

We have noted in Section 2 that previous work on slicing concurrent programs adopt nested fork-join parallelism and define interference dependence as inter-thread data dependence [8, 9, 14]. They find that interference dependence is not transitive in computing a slice.

In the concurrency model of CM-centric programs, threads (which we call Ca-threads) are not nested and there is no explicit inter-thread control dependence (such as fork and join constructs) or inter-thread synchronization. All inter-thread control flow information is carried out by inter-thread data dependence via context variables, which we call *inter-thread context dependence* and will be explained in this section. Although inter-thread context dependence is similar to interference dependence, the former is more pervasive in CM-centric programs.

We regard each variable that may be shared among different Ca-threads as a context variable, and give the definition of inter-thread context dependence as follows:

Definition 5 (Inter-Thread Context Dependence) A node n_j is said to be **inter-thread context dependent** on another node n_i if they are in different strongly connected subgraphs and a context variable is defined in n_i and used in n_j.

The set of inter-thread context dependence edges for a CaCFG is composed as

$$E_{icd} = \{(n_i, n_j) \mid (n_i, n_j) \notin E_{cf}^* \text{ for some } c \in D(n_i) \cap U(n_j)\}$$

where E_{cf}^* is the transitive closure of E_{cf}.[1] We further define a function $L : E_{icd} \to C$ that labels every inter-thread

context dependence edge by the context variable defined in the source node and used in the destination node. For the CaCFG shown in Figure 2, we have inter-thread context dependence edges such as (n_{ev}, n_{s_u}) and (n_3, n_1), so that $L((n_{ev}, n_{s_d})) = g_p$ or g_v or d or q_d and $L((n_3, n_1)) = q_v$.

5.2. Slicing Algorithm

For every strongly connected subgraph of a CaCFG, we use standard algorithms [20] to compute all the intra-thread control dependence and data dependence. By combining the nodes with control dependence edges E_{cd} and data dependence edges E_{dd}, a program dependence graph [15] is obtained. The slice is computed as the backward transitive closure of $E_{cd} \cup E_{dd}$ from any specific node as the slicing criterion. When slicing is extended to inter-thread level, E_{icd} is taken into account.

Definition 6 (Slicing Sequence) A **slicing sequence** $sls(n, n')$ with respect to two distinct nodes n and n' in a CaCFG is a sequence of nodes $\langle n_1, n_2, \ldots, n_k \rangle$ such that $n_1 = n$, $n_k = n'$, and $(n_i, n_{i+1}) \in E_{cd} \cup E_{dd} \cup E_{icd}$ for $i = 1, 2, \ldots, k-1$.

We adapt the definition of a *trace witness* from [14], thus:

Definition 7 (Trace Witness) A **trace witness** of a CaCFG G is a subsequence of some valid execution trace of the nodes in G.

The source node of an inter-thread context dependence is included in a slice if (i) there is a valid execution trace from the source node to the slicing criterion and (ii) the variable definition at the source node can be propagated to the slicing criterion. Previous work [8, 14] solve the first condition by constraining that the slicing sequence from the source node to the slicing criterion should be a witness of some valid execution trace, which complies with the execution order restricted by the control flow information. This condition is automatically satisfied in CM-centric programs because, as the CaCFG is composed of strongly connected subgraphs, any interleaving of nodes forms a valid witness by default.

The second condition ensures that an inter-thread context dependence edge is counted for slicing only if its labeling context variables can be propagated successfully to the criterion node in some execution trace. In this case, we say that the criterion node is *propagation dependent* on it. A data dependence or inter-thread context dependence edge (n_i, n_j) is said to be *killed* in an execution trace tr if tr contains a node n_i' such that (a) n_i' is distinct from n_i and n_j, (b) $D(n_i') = D(n_i)$, and (c) $n_i \rightharpoonup n_i' \rightharpoonup n_j$.[2]

---

[1] E_{icd} may be a multi-set because more than one variable can be defined in a node and used in another node. For the ease

of presentation, however, we assume that every node except the environmental node defines at most one variable.

[2] The notation $n_1 \rightharpoonup n_2$ denotes that n_1 is executed before n_2 [10].

Definition 8 (Propagation Dependence) Given a slicing sequence $sls(n_1, n_k) = \langle n_1, n_2, \ldots, n_k \rangle$, n_k is **propagation dependent** on n_1 if there exists a valid execution trace, of which $sls(n_1, n_k)$ is a witness, such that none of the data dependence and inter-thread context dependence edges in the set $\{(n_i, n_{i+1}) \in E_{dd} \cup E_{icd} \mid i \in \{1, 2, \ldots, k-1\}\}$ is killed in the trace.

In the example shown in Figure 1(b), node n_4 is not propagation dependent on n_1 because either y or x will be killed in any valid execution trace. As data flow dependence is inherently transitive, the propagation dependence property should only be checked exclusively for inter-thread context dependence. Thus, we formally define slices as follows:

Definition 9 (Slice) The **slice** of a CM-centric program with respect to a criterion node n_c in the CaCFG G is the set of nodes

$$\{n \mid sls(n, n_c) = \langle n_1, n_2, \ldots, n_k \rangle$$
$$\text{where } n_1 = n, n_k = n_c, \text{ and}$$
$$\text{for } i = 1, 2, \ldots, k-1, \text{ if } (n_i, n_{i+1}) \in E_{icd}$$
$$\text{then } n_c \text{ is propagation dependent on } n_i\}$$

Informally, a definition occurrence in a slicing sequence is killed by all possible execution traces only if at least two nodes in the slicing sequence are in the same strongly connected subgraph. Furthermore, each of their intermediate control flow paths will kill the definition occurrence. Thus, whenever an inter-thread context dependence edge (n_1, n_2) is encountered in a slicing sequence $\langle n_1, n_2, \ldots, n_c \rangle$, we start from n_2 and search for the first node in the slicing sequence that is in the same strongly connected subgraph as n_1. If such a node, say n_k, is found, we check whether n_k is propagation dependent on n_1. The soundness of this approach is warranted by the following theorem.

Theorem 1 Given a slicing sequence $sls(n_1, n_k) = \langle n_1, n_2, \ldots, n_k \rangle$ in which

$$(n_1, n_k) \in E_{cf}^* \text{ and, for } i = 2, 3, \ldots, k-1, (n_1, n_i) \notin E_{cf}^*, \quad (1)$$

n_k is not propagation dependent on n_1 if and only if

(i) $(n_i, n_{i+1}) \in E_{dd} \cup E_{icd}$ for $i = 1, 2, \ldots, k-1$, and

(ii) there exists a set of nodes $n'_1, n'_2, \ldots, n'_{k-1}$ such that $(n_1, n'_i) \in E_{cf}^*$ and $D(n'_i) = D(n_i)$ for $i = 1, 2, \ldots, k-1$, and $w = \langle n_1, n'_1, n'_2, \ldots, n'_{k-1}, n_k \rangle$ is a (trace) witness of some valid execution trace.

Because of $sls(n_1, n_k)$ and w, we can find, among others, execution sequences $n_1 \rightharpoonup n_2 \rightharpoonup \cdots \rightharpoonup n_{k-1} \rightharpoonup n_k$ and $n_1 \rightharpoonup n'_1 \rightharpoonup n'_2 \rightharpoonup \cdots \rightharpoonup n'_{k-1} \rightharpoonup n_k$ as illustrated in Figure 3.

Proof of Theorem 1

Sufficiency: Suppose (i) and (ii) hold. We shall prove by contradiction that n_k is not propagation dependent on n_1. Assume

the contrary. By Definition 8, for $i = 1, 2, \ldots, k-1$, $(n_i, n_{i+1}) \in E_{dd} \cup E_{icd}$ cannot be killed by any n'_i such that $n'_i \neq n_i$ and $D(n'_i) = D(n_i)$. Hence, the only possible sequences are $n_i \rightharpoonup n_{i+1} \rightharpoonup n'_i$ and $n'_i \rightharpoonup n_i \rightharpoonup n_{i+1}$. Since the only possible sequence for (n_1, n_2) is $n_1 \rightharpoonup n_2 \rightharpoonup n'_1$, by mathematical induction, we must have $n_i \rightharpoonup n_{i+1} \rightharpoonup n'_i$ for $i = 1, 2, \ldots, k-1$. Thus, we have $n_{k-1} \rightharpoonup n_k \rightharpoonup n'_{k-1}$, which contradicts the execution sequence $n'_{k-1} \rightharpoonup n_k$ witnessed by w.

Necessity: Suppose n_k is not propagation dependent on n_1.

(a) We shall prove by contradiction that (i) will hold. Assume the contrary, that is, there exists some n_p such that $(n_p, n_{p+1}) \in E_{cf}^*$. Since $(n_1, n_k) \in E_{cf}^*$, there exists at least one valid execution trace. Since n_k is not propagation dependent on n_1, for any valid execution trace tr of which $sls(n_1, n_k)$ is a witness, there exists a non-empty subset $\{(n_{i_1}, n_{i_1+1}), (n_{i_2}, n_{i_2+1}), (n_{i_q}, n_{i_q+1})\} \subseteq E_{dd} \cup E_{icd}$ such that, for any (n_{i_j}, n_{i_j+1}) in the subset, there exists $n'_{i_j}(\neq n_{i_j})$ satisfying $D(n'_{i_j}) = D(n_{i_j})$ and $n_{i_j} \rightharpoonup n'_{i_j} \rightharpoonup n_{i_j+1}$. Based on tr, we can construct another valid execution trace tr' as follows:

   • Copy the trace tr to tr'.

   • For any n_{i_j} executed before n_p, if $n_{i_j} \rightharpoonup n'_{i_j} \rightharpoonup n_{i_j+1}$ in tr', replace it by $n_{i_j} \rightharpoonup n_{i_j+1} \rightharpoonup n'_{i_j}$.

   • For any n_{i_j} executed after n_p, if $n_{i_j} \rightharpoonup n'_{i_j} \rightharpoonup n_{i_j+1}$ in tr', replace it by $n'_{i_j} \rightharpoonup n_{i_j} \rightharpoonup n_{i_j+1}$.

   For the new execution trace tr' thus contructed, none of the $(n_i, n_{i+1}) \in E_{dd} \cup E_{icd}$ is killed. This contradicts the fact that n_k is not propagation dependent on n_1.
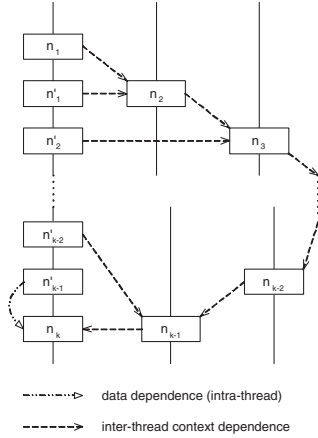
(b) We shall also prove by contradiction that (ii) will hold. Assume the contrary, that is, any witness w of any valid execution trace tr cannot be of the form $\langle n_1, n'_1, n'_2, \ldots, n'_{k-1}, n_k \rangle$ such that $(n_1, n'_i) \in E_{cf}^*$ and $D(n'_i) = D(n_i)$. In other words, there exist some node n_p in $sls(n_1, n_k)$ such that we cannot find a corresponding node n'_p in w satisfying $(n_1, n'_p) \in E_{cf}^*$ and $D(n'_p) = D(n_p)$. For any valid execution trace tr of which w is a witness, we can construct another valid execution trace tr' as follows:

   • Copy the trace tr to tr'.

   • For any n_i executed before n_p, if $n_i \rightharpoonup n'_i \rightharpoonup n_{i+1}$ in tr', replace it by $n_i \rightharpoonup n_{i+1} \rightharpoonup n'_i$.

   • For any n_i executed after n_p, if $n_i \rightharpoonup n'_i \rightharpoonup n_{i+1}$ in tr', replace it by $n'_i \rightharpoonup n_i \rightharpoonup n_{i+1}$.

   For the new execution trace tr' thus contructed, none of the $(n_i, n_{i+1}) \in E_{dd} \cup E_{icd}$ is killed. This contradicts the fact that n_k is not propagation dependent on n_1. ∎

Whenever a slicing sequence $sls(n_1, n_k)$ satisfies both Expression (1) and Condition (i) in Theorem 1, since n_1 and n_k are in the same strongly connected subgraph, we check whether every path from n_1 to n_k contains a witness w. We use a Boolean function $PropDep(n_1, n_k)$ to decide whether n_k is propagation dependent on n_1.

Figure 3. Illustration of Theorem 1

data dependence (intra-thread)

inter-thread context dependence

Figure 4. Example CM-centric program

○ environment node

⬭ situation node → control flow

▭ statement node in adaptive actions - - -▷ inter-thread context dependence

$PropDep(n_1, n_k)$ returns *true* if there is a path from n_1 to n_k that does not contain w, and returns *false* otherwise. Optimal static slicing is undecidable [13]. The same limitation applies also to finding a slicing sequence $sls(n_1, n_k)$ that satisfies both Expression (1) and Condition (i) in Theorem 1. In practice, one may limit the searching of such a slicing sequence by visiting each strongly connected subgraph a finite number of times.

Let us consider the example CM-centric program in Figure 4, which consists of three Ca-threads denoted by G_{s_1}, G_{s_2}, and G_{s_3}. In addition, n_{ev} can update the set of context variables $E_{ev} = \{b, c\}$. (Inter-thread context dependence edges have been added to the CaCFG for clarity.)

To slice n_8, we can derive a slicing sequence $sls(n_2, n_8) = \langle n_2, n_{12}, n_{16}, n_8 \rangle$. Since both n_2 and n_8 are in G_{s_1}, we find that every path from n_2 to n_8 contains the witness $\langle n_2, n_4, n_5, n_6, n_8 \rangle$ that defines the variables $\langle x, u, v \rangle$ in the same sequence of $sls(n_2, n_8)$. Thus, n_2 is not included in the slice of n_8.

Our slicing algorithm adopts an approach similar to those in [8, 14] in recording the last visited node for every Ca-thread. For each node being processed, the algorithm maintains a tuple $V = (v[1], v[2], \ldots, v[|S| + 1])$ in which $|S|$ is the number of situations in the program and each element $v[i]$ represents a node in a strongly connected subgraph G_i.

A slice is computed via a worklist-based algorithm. Each node in the worklist is combined with a corresponding V tuple. Initially, every element in V is set to be $\bot$. When a control dependence or data dependence edge (n_i, n_j) in G_k is encountered, we copy the tuple V of n_j, change $v[k]$ into n_i, and attach the new V to n_i. If more than one element in V is not $\bot$ at a certain node, it indicates that the slicing sequence has traversed inter-thread context
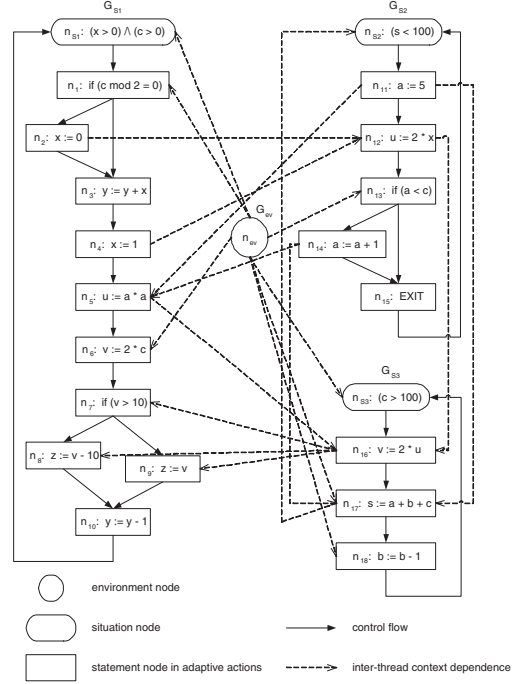
dependence edges. Thus, for each encountered inter-thread context dependence edge (n_i, n_j) in which n_i is in G_k, if the element $v[k]$ in the tuple V of n_j is not $\bot$, there must be a slicing sequence $sls(n_i, v[k])$ that satisfies Expression (1) in Theorem 1. $sls(n_i, v[k])$ can be searched based on V and all the data dependence edges E_{dd}. Consequently, if $PropDep(n_i, v[k]) = true$, n_i is added into the slice and its V tuple will be updated by setting $v[k]$ to be n_j.

The slicing algorithm is shown in Figure 5. We discuss its time complexity here. For each strongly connected subgraphs G_i of the CaCFG G, $i = 1, 2, \ldots, |S| + 1$, suppose the number of nodes is $|N_i|$. In the worst case, the number of elements the worklist to be processed during slicing will be $\prod_{i=1}^{|S|+1} |N_i| = O(|N|^{|S|+1})$ where $|N| = \sum_{i=1}^{|S|+1} |N_i|$ is the number of nodes in G. In searching the possible slicing sequence $sls(n_i, v[k])$ that satisfies both Expression (1) and Condition (i) in Theorem 1, the worst case will also visit $\prod_{i=1}^{|S|} |N_i|$ nodes. If $sls(n_i, v[k])$ is found, since computing an execution path will traverse at most $|N_i|$ nodes, the worst case in computing $PropDep()$ will have a complexity of $|N_i|$. As a result, the worst case complexity of the slicing algorithm in Figure 5 will be $O(|N|^{|S|+1}) \times (\prod_{i=1}^{|S|} |N_i| + |N_i|) = O(|N|^{2|S|+1})$.

5.3. Examples

We give some examples of worklist records to illustrate the algorithm. Figure 6 shows some of the records

Figure 5. Slicing algorithm

```
Input:      slicing criterion n_c,
            the CM-centric program (C, S, A, Ev, act)
                with G = (N, E_cf), E_cd, E_dd, E_icd
Output:     the slice Z
V := (v[1], v[2], ..., v[m]) in which m = |S| + 1, and
            if n_c is in G_i, V.v[i] := n_c;
            else V.v[i] := ⊥;
W := ⟨(n_c, V)⟩;   // the worklist
X := {(n_c, V)};   // the calculated set
Z := {n_c};   // the slice
while W ≠ {}
    remove the first element (n_j, V) from W;
    for all n_i such that (n_i, n_j) ∈ E_cd ∪ E_dd
    // for control dependence and data dependence edges,
    // follow the conventional slicing approach
        find the subgraph G_k containing n_i;
        V' := V;
        V'.v'[k] := n_i;
        if (n_i, V') ∉ X   // has not been calculated
            W := W ∪ {(n_i, V')};
            X := X ∪ {(n_i, V')};
            Z := Z ∪ {n_i};
    for all n_i such that (n_i, n_j) ∈ E_icd
    // for inter-thread context dependence edges
        find the subgraph G_k containing n_i;
        if V.v[k] = ⊥   // no node in G_k has been calculated
            V' := V;
            V'.v'[k] := n_i;
        else if V.v[k] ≠ ⊥
            search for sls(n_i, V.v[k]) that satisfies
                both Expression (1) and Condition (i) in Theorem 1;
            if sls(n_i, V.v[k]) exists and PropDep(n_i, V.v[k]) = false
            // V.v[k] is not propagation dependent on n_i
                continue;   // do not include n_i in the slice
                            // and continue for the next loop
            else
                V' := V;
                V'.v'[k] := n_i;
        if (n_i, V') ∉ X   // has not been calculated
            W := W ∪ {(n_i, V')};
            X := X ∪ {(n_i, V')};
            Z := Z ∪ {n_i};
```

n_3, or n_5 will include all the nodes in the CaCFG.

6. Optimization

A great deal of time of the basic slicing algorithm in Figure 5 spent on the searching of possible slicing sequences in the form of $sls(n_i, v[k])$. If, for every node during slicing, we record not only the last visited node but also the history of traversed intra-thread dependence (namely, control dependence and data dependence edges) for each strongly connected subgraph, then the searching of $sls(n_i, v[k])$ will be reduced to a sorting problem for a sequence of $|S|$ elements and has a worst case complexity of $|S|^2$. Thus, the complexity of the algorithm will be reduced to $O(|N|^{|S|+1}) \times (|S|^2 + |N_i|) = O(|N|^{|S|+2})$. Nevertheless, space consumption will be increased significantly.

One way to improve on the basic slicing algorithm is by treating each strongly connected subgraph as a black-box with all the nodes containing context variable usages as inputs and all the nodes containing context variable definitions as outputs. The intra-thread dependence from every output to every input is pre-computed so that a slice only needs to traverse inter-thread context dependence across different black-boxes. This approach is similar to *coarse-grained* slicing [11], in which the granularity of slicing is adjustable. Although the complexity for the worst case scenario will not change, the improved approach is envisaged to compute slices much faster on average.

We are conducting experiments to evaluate the above approaches and will report our results in the near future. The slicing technique proposed in this paper is based on the pervasive concurrency model in which Ca-threads are scheduled whenever a situation is satisfied and the corresponding adaptive action is activated. We shall also investigate the impact of other schedule approaches in the future.

7. Conclusion

The practice of pervasive computing receives much attention in recent years. However, specific efforts for the maintenance of pervasive software applications such as code-based analysis are understudied. In this paper, we investigate the static slicing of CM-centric programs, as context awareness and middleware-centric architecture are important features of pervasive computing.

The model of CM-centric programs proposed in this paper represents a common design interest in contemporary projects in context-aware pervasive computing. Our context space is built on top of the tuple space model. It provides a unified representation and supports consistent operations on contexts. Our context-aware control flow

of the worklist in slicing node n_{17} of the program shown in Figure 4. The tuple V is configured as $(v[1], v[2], v[3], v[4])$, in which $v[1]$, $v[2]$, and $v[3]$ represent either $\perp$ or a node in G_{s_1}, G_{s_2}, and G_{s_3}, respectively, and $v[4]$ represents either $\perp$ or n_{ev}. Initially, the worklist is set to be $\langle (n_{17}, (\perp, \perp, n_{17}, \perp)) \rangle$. Each subsequent step i shows the resultant worklist obtained by removing and processing the first element in step $i-1$ and adding relevant new elements. The resultant slice of n_{17} is the set $\{n_{17}, n_{s_3}, n_{18}, n_{11}, n_{14}, n_{ev}, n_{s_2}, n_{13}\}$.

Similarly, the slice for node n_8 in Figure 4 is the set $\{n_8, n_7, n_6, n_{16}, n_{s_1}, n_{ev}, n_{18}, n_{s_3}, n_{12}, n_5, n_4, n_{11}, n_{14}, n_{s_2}, n_{13}, n_{17}\}$. Note that n_2 is not included in the slice. (The approaches in related work discussed above would include n_1 and n_2, however.) In the example program of the smart delivery system shown in Figure 2, the slice of any of n_1,

Figure 6. Worklist records in slicing n_{17} of the program in Figure 4

Iteration	Worklist Records
Initial :	$\langle (n_{17}, (\perp, \perp, n_{17}, \perp)) \rangle$
1 :	$\langle (n_{s_3}, (\perp, \perp, n_{s_3}, \perp)), (n_{18}, (\perp, \perp, n_{18}, \perp)), (n_{11}, (\perp, n_{11}, n_{17}, \perp)), (n_{14}, (\perp, n_{14}, n_{17}, \perp)), (n_{ev}, (\perp, \perp, n_{17}, n_{ev})) \rangle$
2 :	$\langle (n_{18}, (\perp, \perp, n_{18}, \perp)), (n_{11}, (\perp, n_11, n_{17}, \perp)), (n_{14}, (\perp, n_{14}, n_{17}, \perp)), (n_{ev}, (\perp, \perp, n_{17}, n_{ev})), (n_{ev}, (\perp, \perp, n_{s_3}, n_{ev})) \rangle$
3 :	$\langle (n_{11}, (\perp, n_11, n_{17}, \perp)), (n_{14}, (\perp, n_{14}, n_{17}, \perp)), (n_{ev}, (\perp, \perp, n_{17}, n_{ev})), (n_{ev}, (\perp, \perp, n_{s_3}, n_{ev})), (n_{ev}, (\perp, \perp, n_{18}, n_{ev})) \rangle$
4 :	$\langle (n_{14}, (\perp, n_{14}, n_{17}, \perp)), (n_{ev}, (\perp, \perp, n_{17}, n_{ev})), (n_{ev}, (\perp, \perp, n_{s_3}, n_{ev})), (n_{ev}, (\perp, \perp, n_{18}, n_{ev})), (n_{s_2}, (\perp, n_{s_2}, n_{17}, \perp)) \rangle$
5 :	$\langle (n_{ev}, (\perp, \perp, n_{17}, n_{ev})), (n_{ev}, (\perp, \perp, n_{s_3}, n_{ev})), (n_{ev}, (\perp, \perp, n_{18}, n_{ev})), (n_{s_2}, (\perp, n_{s_2}, n_{17}, \perp)), (n_{13}, (\perp, n_{13}, n_{17}, \perp)),$ $(n_{11}, (\perp, n_{11}, n_{17}, \perp)) \rangle$
6 :	$\langle (n_{ev}, (\perp, \perp, n_{s_3}, n_{ev})), (n_{ev}, (\perp, \perp, n_{18}, n_{ev})), (n_{s_2}, (\perp, n_{s_2}, n_{17}, \perp)), (n_{13}, (\perp, n_{13}, n_{17}, \perp)), (n_{11}, (\perp, n_{11}, n_{17}, \perp)) \rangle$
7 :	$\langle (n_{ev}, (\perp, \perp, n_{18}, n_{ev})), (n_{s_2}, (\perp, n_{s_2}, n_{17}, \perp)), (n_{13}, (\perp, n_{13}, n_{17}, \perp)), (n_{11}, (\perp, n_{11}, n_{17}, \perp)) \rangle$
8 :	$\langle (n_{s_2}, (\perp, n_{s_2}, n_{17}, \perp)), (n_{13}, (\perp, n_{13}, n_{17}, \perp)), (n_{11}, (\perp, n_{11}, n_{17}, \perp)) \rangle$
9 :	$\langle (n_{13}, (\perp, n_{13}, n_{17}, \perp)), (n_{11}, (\perp, n_{11}, n_{17}, \perp)), (n_{17}, (\perp, n_{s_2}, n_{17}, \perp)) \rangle$
10 :	$\langle (n_{11}, (\perp, n_{11}, n_{17}, \perp)), (n_{17}, (\perp, n_{s_2}, n_{17}, \perp)) \rangle$
11 :	$\langle (n_{17}, (\perp, n_{s_2}, n_{17}, \perp)) \rangle$
12 :	$\langle (n_{s_3}, (\perp, n_{s_2}, n_{s_3}, \perp)), (n_{18}, (\perp, n_{s_2}, n_{18}, \perp)), (n_{ev}, (\perp, n_{s_2}, n_{17}, n_{ev})) \rangle$
13 :	$\langle (n_{18}, (\perp, n_{s_2}, n_{18}, \perp)), (n_{ev}, (\perp, n_{s_2}, n_{17}, n_{ev})), (n_{ev}, (\perp, n_{s_2}, n_{s_3}, n_{ev})) \rangle$
14 :	$\langle (n_{ev}, (\perp, n_{s_2}, n_{17}, n_{ev})), (n_{ev}, (\perp, n_{s_2}, n_{s_3}, n_{ev})), (n_{ev}, (\perp, n_{s_2}, n_{18}, n_{ev})) \rangle$
15 :	$\langle (n_{ev}, (\perp, n_{s_2}, n_{s_3}, n_{ev})), (n_{ev}, (\perp, n_{s_2}, n_{18}, n_{ev})) \rangle$
16 :	$\langle (n_{ev}, (\perp, n_{s_2}, n_{18}, n_{ev})) \rangle$
17 :	$\langle \rangle$

graph (CaCFG) captures context-aware invocations as well as the structures of context-aware applications. We use it as a foundation for slicing.

Specific features of CM-centric programs make our static slicing approach different from that for conventional concurrent programs. In particular, when processing inter-thread data dependence, we check the propagation dependence rather than the witness of a valid trace. We have also demonstrated how our approach produces a more precise slice. Like other static slicing approaches, however, our slicing algorithm has an exponential complexity with respect to the number of execution threads in the worst case. We propose several optimizations and envisage them to improve the slicing efficiency. We shall report our findings in the future.

References

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

[2] 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.

[3] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. A metamorphic approach to integration testing of context-sensitive middleware-based applications. In *Proceedings of the 5th International Conference on Quality Software (QSIC 2005)*, pages 241–249. 2005.

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

[5] J. Cheng. Slicing concurrent programs: a graph-theoretical approach. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging*, volume 749 of Lecture Notes in Computer Science, pages 223–240. 1993.

[6] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12 (1): 26–60, 1990.

[7] C. Julien and G.-C. Roman. Egocentric context-aware programming in ad hoc mobile environments. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE-10)*, pages 21–30. 2002.

[8] J. Krinke. Static slicing of threaded programs. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 1998)*, pages 35–42. 1998.

[9] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of the Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC 2003/FSE-11)*, pages 178–187. 2003.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21 (7): 558–565, 1978.

[11] H. F. Li, J. Rilling, and D. Goswami. Granularity-driven dynamic predicate slicing algorithms for message passing systems. *Journal of Automated Software Engineering*, 11 (1): 63–89, 2004.

[12] 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 the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006)*. 2006.

[13] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *Proceedings of the 33th ACM Symposium on Theory of Computing*, pages 647–656. 2001.

[14] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, pages 180–190. 2000.

[15] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM Symposium on Practical Software Development Environments*, pages 177–184. 1984.

[16] 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.

[17] 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.

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

[19] T. Strang and C. Linnhoff-Popien. A context modeling survey. In *Proceedings of the 1st International Workshop on Advanced Context Modelling, Reasoning and Management*, pages 34–41. 2004.

[20] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3 (3): 121–189, 1995.

[21] 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. 2005.

[22] 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.

[23] J. Zhao. Slicing concurrent java programs. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension*, pages 126–133. 1999.