<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>Static slicing for pervasive programs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Lu, H; Chan, WK; Tse, TH</td>
</tr>
<tr>
<td><strong>Citation</strong></td>
<td>Proceedings - International Conference On Quality Software, 2006, p. 185-192</td>
</tr>
<tr>
<td><strong>Issued Date</strong></td>
<td>2006</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10722/55043">http://hdl.handle.net/10722/55043</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.</td>
</tr>
</tbody>
</table>
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 [19]. 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, 13, 18, 22, 23]. 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 [21]. 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, 15]. In particular, for control dependence and intra-thread data dependence, our algorithm follows the standard backward traversal [21]. For inter-thread data dependence, however, we show that the constraints of threaded witness [8] and trace witness [15] 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 technique 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) [21]. Ottenstein and Ottenstein [16] 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 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, 24]. One of the additional dependence relationships brought about by concurrency was referred to as interference dependence [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 \( m \rightarrow id \ n \), 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 \( n_2 \rightarrow id n_3 \) and \( n_3 \rightarrow id n_1 \).

Using the above backward traversal algorithm for slicing sequential programs, in Figure 1(a), \( n_2 \rightarrow id n_3 \) and \( n_3 \rightarrow id n_1 \). 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 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 [14], it is undecidable in general [14].

Nanda and Ramesh [15] 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, 15], \( n_1 \) should be included in the slice of \( n_4 \) simply because \( n_1 \rightarrow id n_5 \), \( n_5 \rightarrow id n_4 \), 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 [14], we adopt a simplified model. A context is a data abstraction of environmental attributes of computing entities [4, 20]. 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 \( \langle name, type, value \rangle \), 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:

\[
\text{get}(c) : \begin{cases}
  \text{return } t, & \text{if } \exists t \in T (t.name = c) \\
  \text{null}, & \text{else return null}
\end{cases}
\]

\[
\text{update}(t) : \begin{cases}
  \text{return } t, & \text{if } \exists t' \in T (t'.name = c) \\
  \text{return } T := (T \setminus \{t'\}) \cup \{t\}, & \text{else return null}
\end{cases}
\]

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, 17, 22, 23]. 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 $p(get(C_s)) = true$.

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

\[
\text{evaluate}(s) : \begin{cases}
  p(get(C_s)), & \text{return } p(get(C_s))
\end{cases}
\]

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 $(C, S, A, Ev, act)$, 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 $act : S \rightarrow A$ such that for any $s \in S$, the adaptive action $act(s)$ 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 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 $s_{us}$ 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 $s_{os}$ 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 $Ga = (Na, Ea)$, where each node $n \in Na$ represents a statement and each directed edge $e \in Ea$ 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 $Ga$ contains a unique entry node $entry(Ga)$ and a unique exit node $exit(Ga)$.

**Step 2:** For situation $s$, we treat the $\text{evaluate}(s)$ operation as a special predicate node $n_s$, which is called
Table 1. Example smart delivery system \((C, S, A, Ev, act)\), where \(C = \{g_p, q_p, q_d, q_l, q_v, d\}\), \(S = \{s_d, s_u, s_o\}\), \(A = \{compLedger, replenish, withdraw\}\), \(Ev = \{g_p, q_p, q_d, q_l, q_v, d\}\), \(act(s_d) = compLedger\), \(act(s_u) = replenish\), and \(act(s_o) = withdraw\).

<table>
<thead>
<tr>
<th>Context variables</th>
<th>(s_d)</th>
<th>type of goods stored in pallet</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_d)</td>
<td>amount of goods stored in pallet</td>
<td></td>
</tr>
<tr>
<td>(q_v)</td>
<td>desired amount of goods in pallet</td>
<td></td>
</tr>
<tr>
<td>(q_l)</td>
<td>ledger amount of goods in pallet</td>
<td></td>
</tr>
<tr>
<td>(g_v)</td>
<td>type of goods delivered by van</td>
<td></td>
</tr>
<tr>
<td>(d)</td>
<td>distance between pallet and van</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Situations</th>
<th>(s_d)</th>
<th>(C_d = {g_p, g_v, d}), (p_d = (g_p = g_v) \land (d &lt; 10))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_u)</td>
<td>(C_u = {g_p, g_v, d, q_d, q_l}), (p_u = (g_p = g_v) \land (q_l &lt; q_d))</td>
<td></td>
</tr>
<tr>
<td>(s_o)</td>
<td>(C_o = {g_p, g_v, d, q_d, q_l}), (p_o = (g_p = g_v) \land (q_l &lt; q_d))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adaptive actions</th>
<th>(act(s_d))</th>
<th>compLedger { (q_l := q_l + q_v;) }</th>
</tr>
</thead>
<tbody>
<tr>
<td>(act(s_u))</td>
<td>replenish { (if (q_v &lt; MAX) ), (q_v := q_v + 1;) }</td>
<td></td>
</tr>
<tr>
<td>(act(s_o))</td>
<td>withdraw { (if (q_v &gt; 0) ), (q_v := q_v - 1;) }</td>
<td></td>
</tr>
</tbody>
</table>

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 \(act(s)\) is control dependent on \(n_s\). In line with existing designs of pervasive systems such as [22, 23], we represent the true branch of \(n_s\) as a control flow edge \((n_s, entry(G_{act(s)}))\) while not explicitly showing the false branch. Another control flow edge \((exit(G_{act(s)}), n_s)\) is also needed to signify that variations of context space may re-evaluate \(s\) and invoke \(act(s)\) 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 \(D(n_m) = Ev\) 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 \((C, S, A, Ev, act)\) is a directed graph \(G = (N, E_{ca})\) 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

\[
E_{cf} = \bigcup_{a \in A} \{ins_a \mid s \in S\} \cup \{outs_a \mid s \in S\}
\]

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

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 incoming 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 \(N_{act(s)}\), the nodes of the bound adaptive actions of \(s\). We denote it by \(G_s\), where \(G_s = (N_{act(s)} \cup \{n_s\}, E_{act(s)} \cup \{ins_s, outs_s\})\) for any \(s \in S\). We shall discuss the interactions between the application tier and the middleware tier in the next section.

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, 15]. 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 \( L: E_{icd} \rightarrow C \) 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 \( (n_{ev}, n_{sw}) \) and \( (n_3, n_1) \), so that \( L((n_{ev}, n_{sw})) = g_p \) or \( g_v \) or \( d \) or \( dqd \) and \( L((n_3, n_1)) = g_v \).

### 5.2. Slicing Algorithm

For every strongly connected subgraph of a CaCFG, we use standard algorithms [21] 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 [16] 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 \( (n_1, n_2, \ldots, n_k) \) 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 [15], 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, 15] solve the first condition by constraining that the slicing sequence

\[ E_{icd} \] 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.

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 all possible execution traces. In this case, we say that the criterion node is **propagation dependent** on it. A data dependence or inter-thread context dependence \( (n_i, n_j) \) is said to be killed in an execution trace \( tr \) if \( tr \) contains a node \( n'_i \) such that (a) \( n'_i \) is different from \( n_i \) or \( n_j \), (b) \( D(n'_i) = D(n_i) \), and (c) \( n_i \leftarrow n'_i \rightarrow n_j \). \(^2\)

**Definition 8 (Propagation Dependence)** Given a slicing sequence \( sls(n_1, n_k) = (n_1, n_2, \ldots, n_k) \), \( n_k \) is **propagation dependent** on \( n_i \) if there exists a valid execution trace of which \( sls(n_1, n_k) \) is a witness, such that none of the dependence in the set \( \{ (n_i, n_{i+1}) \mid 1 \leq i < k \} \) is killed in the trace.

In the example shown in Figure 2(b), node \( n_r \) 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

\[ \{ n_i | sls(n, n_c) = (n_1, n_2, \ldots, n_k) \} \]

where \( n_1 = n, n_k = n_c \), and

- for any \( i = 1, 2, \ldots, k - 1 \), if \( (n_i, n_{i+1}) \in E_{icd} \) then \( n_i \) is propagation dependent on \( n_1 \).

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 \( (n_1, n_2, \ldots, n_k) \), 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. Its proof can be found in our technical report [12].

\[^1\]The notation \( E_{icd} \) 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 \( n_1 \Rightarrow n_2 \) denotes that the execution of \( n_1 \) happens before \( n_2 \) [10].
Theorem 1  Given a slicing sequence \( \text{sll}(n_1, n_k) = (n_1, n_2, \ldots, n_k) \) in which

\[
(n_1, n_k) \in E_{cf} \text{ and, for } i = 2, 3, \ldots, k - 1, \ (n_1, n_i) \notin E_{cf}. \tag{1}
\]

\( n_k \) is not propagation dependent on \( n_1 \) if and only if

(i) \( (n_i, n_{i+1}) \in E_{dd} \cup E_{icd} \) for \( i = 2, 3, \ldots, k - 2 \), and

(ii) there exist a set of nodes \( n'_1, n'_2, \ldots, n'_{k-1} \) such that

\[
(n_1, n'_1) \in E_{cf} \text{ and } D(n'_i) = D(n_i) \text{ for } i = 1, 2, \ldots, k - 1,
\]


and \( w = (n_1, n'_1, n'_2, \ldots, n'_{k-1}, n_k) \) is a witness of any valid execution trace.

Whenever a slicing sequence \( \text{sll}(n_1, n_k) \) satisfies both Equation (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 \( \text{PropDep}(n_1, n_k) \) to decide whether \( n_k \) is propagation dependent on \( n_1 \): \( \text{PropDep}(n_1, n_k) \) returns \( \text{true} \) if there is a path from \( n_1 \) to \( n_k \) that does not contain \( w \), and returns \( \text{false} \) otherwise. Optimal static slicing is undecidable [14]. The same limitation applies also to finding a slicing sequence \( \text{sll}(n_1, n_k) \) that satisfies both Equation (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-thrads denoted by \( G_{11}, G_{12}, \text{ and } G_{13} \). In addition, \( n_{s1} \), can update the set of context variables \( E_{c1} = \{c, e\} \). (Inter-thread context dependence edges have been added to the CaCFG for clarity.)

To slice \( n_8 \), we can derive a slicing sequence \( \text{sll}(n_2, n_8) = (n_2, n_{12}, n_{16}, n_8) \). Since both \( n_2 \) and \( n_8 \) are in \( G_{11} \), we find that every path from \( n_2 \) to \( n_8 \) contains the witness \( n_2, n_4, n_5, n_6, n_8 \) that defines the variables \( (x, u, v) \) in the same sequence of \( \text{sll}(n_2, n_8) \). Thus, \( n_2 \) is not included in the slice of \( n_8 \).

Our slicing algorithm adopts an approach similar to those in [8, 15] 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 subgraph \( G_i \). For each node being processed, the algorithm

\[\text{PropDep}(n_1, n_k) = \text{true} \]

if there is a path from \( n_1 \) to \( n_k \) that does not contain \( w \), and returns \( \text{false} \) otherwise. Optimal static slicing is undecidable [14]. The same limitation applies also to finding a slicing sequence \( \text{sll}(n_1, n_k) \) that satisfies both Equation (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-thrads denoted by \( G_{11}, G_{12}, \text{ and } G_{13} \). In addition, \( n_{s1} \), can update the set of context variables \( E_{c1} = \{c, e\} \). (Inter-thread context dependence edges have been added to the CaCFG for clarity.)

To slice \( n_8 \), we can derive a slicing sequence \( \text{sll}(n_2, n_8) = (n_2, n_{12}, n_{16}, n_8) \). Since both \( n_2 \) and \( n_8 \) are in \( G_{11} \), we find that every path from \( n_2 \) to \( n_8 \) contains the witness \( n_2, n_4, n_5, n_6, n_8 \) that defines the variables \( (x, u, v) \) in the same sequence of \( \text{sll}(n_2, n_8) \). Thus, \( n_2 \) is not included in the slice of \( n_8 \).

Our slicing algorithm adopts an approach similar to those in [8, 15] 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 subgraph \( G_i \). For each node being processed, the algorithm

\[\text{PropDep}(n_1, n_k) = \text{true} \]

if there is a path from \( n_1 \) to \( n_k \) that does not contain \( w \), and returns \( \text{false} \) otherwise. Optimal static slicing is undecidable [14]. The same limitation applies also to finding a slicing sequence \( \text{sll}(n_1, n_k) \) that satisfies both Equation (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.
sequence \( sls(n_i, v[k]) \) that satisfies both Equation (1) and condition (i) in Theorem 1, the worst case will also visit \( \Pi_{i=1}^{k} |N_i| \) 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 \( PropDep(n_i, V, v[k]) \) will have a complexity of \( |N_i| \). As a result, the worst case complexity of the slicing algorithm in Figure 5 will be \( O(|N|^{|S|+1}) \times (\Pi_{i=1}^{k} |N_i| + |N_i|) = O(|N|^{2|S|+1}) \).

### 5.3. Examples

We give some examples of worklist records to illustrate the algorithm. Figure 6 shows some of the records 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 \( G_{s1}, G_{s2}, \) and \( G_{s3} \), respectively, and \( v[4] \) represents either \( \perp \) or \( n_{ev} \). Initially, the worklist is set to \( \langle (n_17, (\perp, \perp, n_17, \perp)) \rangle \). 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 \( \{n_17, n_3, n_19, n_11, n_{ev}, n_{s2}, n_{s3}\} \).

Similarly, the slice for node \( n_9 \) in Figure 4 is the set \( \{n_9, n_6, n_{s9}, n_{ev}, n_{s18}, n_{s17}, n_5, n_{s11}, n_{s14}, n_{s5}, n_{ev}, n_{s13}\} \). Note that \( n_{29} \) 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 \( n_1, n_3, \) 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 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