

Verifying ANSI-C Context-Aware Applications

Draft

Doina Bucur

January 12, 2009

Abstract

We report on work in progress upon the verification of context-aware applications written in C-based languages. We recognize that context-aware programs are generally either middleware-based and multithreaded, or driven by asynchronous events, and focus on identifying the program points in which the contextual updates impact the application behaviour. Inheriting from related work on the validation of context-aware applications, we implement a technique for detecting context-aware program points over SATABS, a tool for generic verification of specifications of ANSI C/C++ programs. We then briefly review future work with regard to the automatic verification of these programs in SATABS, using the generated context-aware program points.

Contents

1	Introduction	1
2	Related Work	3
3	Context-Aware Program Points in SATABS	5
3.1	SATABS and goto-cc	5
3.2	Identifying Capps	6
3.3	A Formal Static Analysis	8
3.3.1	Context-Aware Program Points	8
3.3.2	Types and Type Judgements	10
3.3.3	Type System	11
4	Verification for Context-Aware Programs	12

1 Introduction

An extreme distribution of services makes ubiquitous computing an application area in which the behaviour of programs changes with both local

and networked events. An added degree of unscheduled device mobility and failure makes this stream of events unpredictable. The *context* of an application is then any such environmental computational information which is relevant to the application behaviour, and an application is *context-aware* if it features actions triggered by context or automatic reconfiguration upon context [17, 5, 1].

Examples of computational context to an application include sensed input from local sensing hardware on a sensor platform, the presence—announced by a hello packet—of another networked device in the near neighbourhood of a router, the contents of a person’s schedule as retrieved from a server in an office environment, or the current human activity undertaken at a location.

Developing context-aware applications to work under such distributed and unpredictable execution environments has specific difficulties. On one hand, the unscheduled nature of contextual updates incoming to a program fueled the development of *asynchronous* languages. Such languages specialize in managing interaction in I/O-heavy systems: an asynchronous function call doesn’t start execution at call time, but is instead appended to a task queue, from which a dispatcher pops one task at a time and runs it to completion. Such deferred execution is the basis for event-driven programming languages, in which tasks are callback code prompted into the dispatcher’s queue by an input event, and whose execution is delayed until the system is idle. Recent languages centered around a support for asynchrony [8, 11] include nesC, a language for writing deeply networked systems and the programming platform for the sensor network operating system TinyOS [18].

On another hand, the opposite approach at tackling the complexity of contextual updates is the development of *middleware*-based applications, in which a specialized middleware software layer takes over the non-trivial process of discovery, acquisition and interpretation of context updates, thus leaving a lesser degree of awareness to the application. A middleware can employ the same asynchronous schemes for managing context updates, with the application then being a standard multithreaded program. A variety of proposed middleware software for adapting to context, of various design concepts and functionality, range from the classic ContextToolkit [7] to refinements such as JCAF [2].

The purpose of this research is to provide verification mechanisms which ensure the dependability of applications adapting to context; these applications can be asynchronous programs, or multithreaded code supported by a middleware layer, as described above. The fundamental difficulty in verifying such adaptive programs is the fact that context updates affect the behaviour of the application involuntarily, at any time during execution: the handlers of asynchronous events preempt the running of the main program or that of any task, or the middleware sprouts threads running context handlers at indetermined times. This can happen for any type of input, but it is

particular for context inputs, which are delivered streamingly (i.e. sensors feed readings to the application periodically, and all new members of the network neighbourhood broadcast hello messages).

The resulting programming difficulty resides in the need for developers to identify when the streams of context updates impact the behaviour of the application. Since contextual events are asynchronous, contextual changes themselves interfere.

Given the current wealth of development and deployment of such context-adaptive systems, and adding these distinctive features of programming for awareness compared to general programming, we note that specialized techniques for the *verification* of context-aware applications are lacking. We build on the few existing specialized techniques for the *validation* of such programs ([20, 19, 14] described in Section 2 on related work) and proceed to improve the context awareness of an existing verification suite (SAT-Based Predicate Abstraction for ANSI-C, SATABS [6]), by identifying the program points in which the data flow of a certain source of context updates influences application behaviour (called context-aware program points, or *capps*).

We then show the directions for future work, which include a refinement upon the techniques for identifying *capps*, with analyses borrowed from the field of secure information flow [3]. We then state that the final goal is the verification of context-aware programs; for this, we are looking into writing specifications of such programs and automatically generating assertions to be verified with the counterexample-based abstraction and refinement (CEGAR) loop employed by SATABS.

For all purposes stated above, we settled on programs written in C-like languages; our case studies include nesC applications for sensor networks and C++ industrial programs. Since the SATABS suite takes as input ANSI-C programs, we employ translators for flattening nesC and C++ to C, and techniques for handling the remaining features of these languages which are not present in C (e.g. the modeling of nesC’s scheduling scheme).

2 Related Work

Given the problem of analysing the influence that an external contextual input (e.g. variable `ctx` given as an argument to the context’s handler, as in Fig. 1) has upon a multithreaded or event-based program, we settle on identifying the set of context-aware program points. This set is composed of program statements which (i) have the side-effect of leaking a measure of the current value of `ctx` to another variable, such as an assignment `power=ctx` where `power` is a global variable, or (ii) have an effect which varies with the current value of `ctx` or a leaked-upon `power`, such as expressions `if(power) E1 else E2, printf("power),` or `if(power > MAX) while(1).`

The process of identifying (ii) statements, only for the case in which the

leaked-upon variable influencing the statement is strictly a global, is related to the problem of identifying data racing points. However, given the fact that the external contextual input starts always as a variable local to a function (be it an event handler or a thread's main function), all other program points can only be identified by more sophisticated techniques related to information flow.

In the remaining of this section we review related research, either on the verification of our typical case studies for race conditions, or on the validation of context awareness. To start with, data races are a source of errors in concurrent programs, in a similar way to that of data interference upon program behaviour. Race detection is a safety verification problem for concurrent programs: a race occurs when two threads can access (i.e. read or write) a global variable simultaneously, and at least one of the accesses is a write.

Henzinger et al. [9] recognizes that, while existing race checkers fall into two categories (dynamic, lockset-based tools and static, type-based tools), programmers write code using less obvious synchronization schemes such as those locking access based on the value of a variable. Such synchronization schemes cause false positives with standard static analysis tools, and the authors develop a more precise analysis of access, tracking the values of variables.

Their tool is implemented in the C model checker BLAST [10], and was run over nesC applications. NesC employs specific synchronization idioms: (i) tasks are nonpreemptible, and as such there can be no races on variables accessed only in tasks, but only between events and events and tasks (ii) there are `atomic` sections. The nesC compiler itself implements a flow-based alias static analysis to catch race conditions on shared variables from event handlers, and reports false positives at certain synchronization methods, as described above. The tool models away the system hosting a nesC application as an arbitrary number of threads, each executing a while-loop triggering hardware interrupts (if interrupts are enabled by a certain global variable) or tasks (if the execution goes idle) nondeterministically, as well as the nesC's task queue and split-phase synchronization.

Moving closer to the point of dealing with contextual input instead of generic concurrent programs, Wang, Elbaum and Rosenblum's [20] work on automated generation of tests of context awareness for Java programs sets the tone for specialized techniques for context awareness. It improves the test suite of a context-aware Java application by identifying program points where context changes affect program behaviour, for then to construct sequences of such points and only select the ones which satisfy certain coverage adequacy criteria.

Their capp identifier takes as inputs the program source and the context object, passed to the application from the middleware as the argument to an instance of the context handler function, running in a newly sprouted thread.

The capps are detected by side-effect analysis [13] and escape analysis [15] over Java code, and the outcome is a set of multi-function control flow graphs in which nodes are annotated with capp identifiers.

All subsequent related work moves away from real application code, towards modeling. Other validation techniques [19, 14] stem from either metamorphic testing or data-flow analysis, also trying to measure the comprehensiveness of capp sequences as test sets. They formalize the notion of *context* as a pair of context variable and value, that of *situation* (residing at the middleware level) as a triggering condition over some contexts, which starts an action (residing at the application level) when the trigger holds.

In their middleware-centric model program, context variables are standard data variables enhanced with the ability of being updated from the environment, context-aware program points are computed as those expressions in which a context variable is read or written, either in a situation or in an action. Based on the above, the program is modeled as a context-aware flow graph: essentially, the standard control flow graphs of situations and actions are enhanced with "phantom" nodes modeling the potential environmental update of context variables. Then, a set of test-suite adequacy criteria are settled, based on the relationship between the context-aware program points and their (situation or action) location.

Other verification schemes for context awareness are based on logic [16, 4]; as expected, they trade practicability for the ability of verifying strong specifications (such as the fulfilling of distributed security firewalls upon types of contexts [4]) over networks of mobile, context-aware entities and mobile code.

3 Context-Aware Program Points in SATABS

We proceed to add the capp identifier techniques (in Subsection 3.2), after first reviewing the existing program analysis and verification features in SATABS (Subsection 3.1).

3.1 SATABS and goto-cc

SATABS is a generic model checking tool targeted at the software industry. Its practicality stems from the fact that, unlike usual model checkers, which take as input programs written in model languages, SATABS analyzes ANSIC programs, and supports a large subset of the language's features (e.g. complex data types, pointer arithmetics, function pointers, dynamic memory and, partially, the `pthread`-library concurrency [12]). Consequently, support for C++ is an expected (and work in progress) extension, while support for C-like languages such as nesC [8] depends only on the existence of a

translator from nesC to C¹.

While this effectively removes the need for developing models of programs as input for the model checker, given the size of typical input applications, the problem of the verification scalability becomes central. For this, SATABS computes an *abstract model* [6] of the input code (now denoted as *concrete model*) automatically: such an abstraction preserves the code’s original control flow, but replaces a subset of the original program variables with chosen boolean variables and leaves all others nondetermined. This is a conservative process (i.e., any properties upon variables which are true in the original program remain true in the abstracted program), hence properties of the input code can be proven true based on a small abstract program, by a standard model checker. On the other hand, if the property is false over the abstract program, the counterexample returned by the model checker is checked upon the concrete model; if not spurious, the property is proven false, but otherwise the counterexample-guided abstraction refinement (or CEGAR) technique [6] drives the construction of an increasingly complex abstraction.

Crucially, in the preverification stage, SATABS manipulates the input code for ease of analysis: the rich ANSI C language is translated into a simplified C-like language called *goto-cc*, in which all control flow statements are rewritten as guarded `gotos`, function calls are inlined, and complex statements are broken into simple ones. Over these, SATABS implements preliminary static analyses, such as pointer analysis (i.e. the memory field a pointer points to at a given program statement). The capp identifier itself, presented next, is also a collection of static analyses extending SATABS at this preverification stage.

3.2 Identifying Capps²

To identify the context-aware program points (capps) in a program taking unscheduled environmental events, Wang, Elbaum and Rosenblum’s [20] Java capp identifier employs two known static analysis techniques (side-effect and escape analyses, as in [13, 15]), each giving out a subset of their capp set. We borrowed this idea, and as an initial step extended SATABS [6] with these capp-identifier techniques.

The Java *side-effect analysis* procedure from Le et al. [13]—translated for use over ANSI-C—computes, for every statement s in the program, sets $read(s)$ and $write(s)$ containing all variables (or fields of variables of type

¹`ncc`, the nesC compiler used in building TinyOS, does create a platform-dependent, inlined, non-ANSI `.c` program out of the given nesC application plus the set of TinyOS components the application is wired with. This is then compiled into machine code and deployed on sensors. The effort required to translate this application code into ANSI C is relatively little; nonetheless, in the long term we are aiming towards a fully-automatic, component-based translator from nesC to ANSI C.

²This is basically a description of the old implementation, from 2008. Can be skipped.

structure) read and written by statement s . This is based on a preliminary pointer analysis, which retrieves the variable a pointer points to, in any given program state; also, the read and write effects of function calls need to be propagated back to their call statement. All variable names are uniquely identified by SATABS based on their scope: `update_demo::ctx` and `update_power::ctx` are a pair of identically named variables `ctx`, arguments of different functions; this extends to same-name functions in different source files.

After the read and write sets are computed for all statements, the adapted escape analysis in [15] defines an approximated interference dependence in a concurrent environment as in Definition 1:

Definition 1. *An interference dependence in a concurrent program is a pair of program points m and n in distinct threads t_m and t_n such that a (field of a) variable is written at m and read at n .*

In what follows, we keep Wang, Elbaum and Rosenblum’s [20] case study on TourApp, a middleware-based application running demos on mobile devices at an exhibition, translated into C using the `pthread` library (Fig. 1), as a base example.

When applied to the capp identification problem, the techniques above translate into a static analysis which proceeds as follows:

- It starts by identifying the particular program variables which hold the values of incoming environmental contexts; it then adds these to an empty set of variables “tainted” by contextual input, $tainted_0$; in the example code in Fig. 1, the middleware passes a pointer to `ctx` of type `ctx_t` to the `context_handler(void *arg)` function, which will run as a new thread at application level. Hence, $tainted_0 = \text{middleware} :: 1 :: \text{ctx}$ and a $tainted(s)$ is also calculated for each program statement.
- The side effects (i.e. read and write sets) and tainted set of program statements are calculated for all statements; for all initial thread statements s , $tainted(s) = tainted_0$, and:
 - if at any statement s , $read(s)$ intersects $tainted(s)$, then s is a context-aware program point, or capp;
 - for all non-initial statements s , $tainted(s)$ is inherited from the preceding statement in the program flow;
 - if the above happens, and additionally $write(s)$ is not empty, then $write(s)$ is added to $tainted(s')$, for all statements s' flowing directly after s .

```

unsigned char power;
unsigned char demo;

void update_power(ctxt *power_ctx)
{
    power = power_ctx->value;
}

void update_demo(ctxt *demo_ctx)
{
    if(demo_ctx->value == 1 && power > 30)
        demo = 1;
    else if(demo_ctx->value == 2 && power > 30)
        demo = 2;
}

void* context_handler(void *arg)
{
    if((ctxt *)arg->type == TYPE_POWER)
        update_power(arg);
    else if((ctxt *)arg->type == TYPE_DEMO)
        update_demo(arg);
    return NULL;
}

```

(a) application

(b) middleware

Figure 1: The [20] case study on TourApp, translated into C using the `pthread` library

The capps thus identified are program points at which the program statement accesses the contextual variable `ctx`, or of any other program variable which was assigned an expression containing `ctx`, even indirectly, such as `power` in `power = power_ctx->value` from Fig. 1(a). As in [20], for the sample code in Fig. 1, the capps identified are depicted in Fig. 2, in which the `goto-cc` statements highlighted in gray are the context-aware program points. These capps serve as the reading program points (at location n) in Definition 1 for interference dependences.

3.3 A Formal Static Analysis

3.3.1 Context-Aware Program Points

We aim to mark `goto-cc` expressions with *context-aware program points*, or *capps*, a similar concept with *security levels* when securing information flow; Boudol’s [3] comprehensively calculates the flow of information from a confidential location of memory (in our case, a contextual variable) in a multithreaded ML-like toy language. Technically, the analysis we use to generate the capps is an extension of the side-effect and escape analyses,

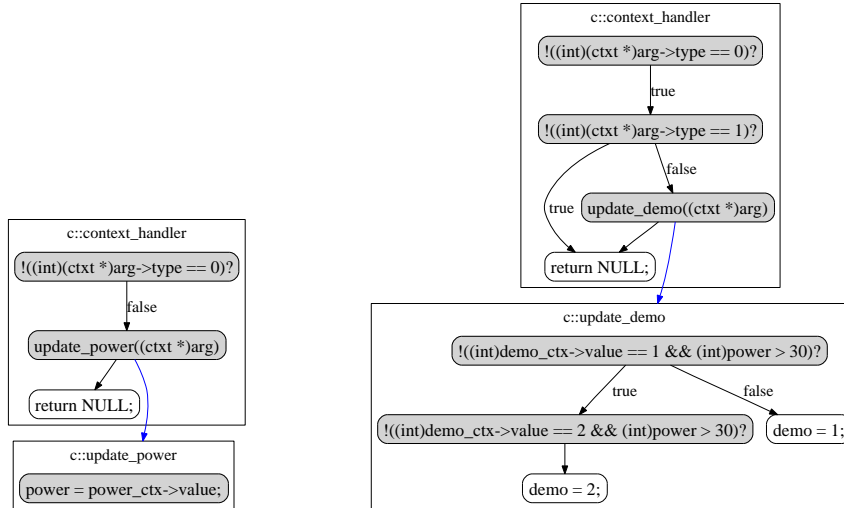


Figure 2: Context-aware program points (goto-cc statements in gray) as detected by side-effect analysis in SATABS

covering termination effects and user effects, in addition to read and write effects.

The capps will mark those program expressions whose *behaviour* is affected by the value of the program’s *contextual variables*. Contextual variables are standard variables, i.e. memory objects pointed to by variable names, with the added twist that their value is received from agents external to the program (e.g., `ctx` in Fig. 1).

Informally, the behaviour of an expression is a set of effects on the lines of:

```
printf("%d", ctx);
```

(i.e. visible effects to the user; this we call the *user effect*). Also, leaks of the contextual input’s value into the program’s other local or global variables:

```
int x = ctx;
```

could be followed by a user effect `printf("%d", x);`; such leaks we call *write effects*. A *read effect* is e.g. a control-flow expression whose condition is `ctx`-dependant, such as

```
if(x > MAX) f(); else g();
```

Finally, an expression has a *termination effect* if its termination depends on `ctx`, such as the goto-cc variant of the C expression:

```
if(x > MAX) while (1);
```

which could be followed by a write effect $y = 1$, with the same semantics as $y = \text{ctx} \leq \text{MAX}$.

A capp is a tuple of four read, write, termination, and user effects, as above; we define the syntax of effects and capps as follows.

Definition 2 (Expressions' effects). *Let \mathcal{V} be the countable set of all references (e.g. variable names or pointers), with $\mathcal{C} \subseteq \mathcal{V}$ the set of all contextual references. A read effect r , write effect w , termination effect t , and user effect u are defined as:*

$$r, w, t, u \subseteq \mathcal{V}$$

Definition 3 (Context-aware program points). *A context-aware program point (capp) γ is a 4-tuple of expression effects:*

$$\gamma ::= (r, w, t, u).$$

We retrieve the individual effects out of a capp γ by writing $r(\gamma), w(\gamma), t(\gamma), u(\gamma)$.

We say that a program expression is a read-capp if its capp γ has $r(\gamma) \neq \emptyset$ (and similarly for the other three effects).

The individual effects can be thought of as in the following:

- An expression E 's read effect r is the set of those references which (i) E reads, and (ii) influence its resulting *value*.
- An expression E 's write effect w is the set of all the references which E writes.
- An expression E 's termination effect t is the set of those references which influence E 's *termination*.

TODO. Define effect lattice, lower/upper bounds, meet/join.

We denote by \perp the 4-tuple of empty effects $\perp ::= (\emptyset, \emptyset, \emptyset, \emptyset)$, and generally abuse the notations for set operators (e.g. \cup) for operations over capps.

3.3.2 Types and Type Judgements

We now introduce the types of goto-cc expressions, based on our notion of capps.

Definition 4 (Expression's type). *Let β denote a boolean whose **true** value signals that the expression always terminates, and whose **false** value denotes that the expression might not terminate. Also, let Λ be the set of goto-cc labels, with subsets $\lambda \subseteq \Lambda$. $l \in \Lambda$ is then a termination condition such that **true**(l) denotes an expression's termination iff the expression comprises no statement labeled with l .*

We then attach the following types τ to goto-cc expressions:

$$\tau ::= (\gamma, \lambda, \beta(l))$$

This way, an expression's type records the 4-tuple of the expression's effects, its list of labels, plus conservative knowledge about whether the expression surely terminates or not.

Type judgements for expressions S are then of the form:

$$\Gamma \vdash S : \tau$$

where Γ is a typing context which assigns variables to variables, $\Gamma \subseteq \mathcal{V} \times \mathcal{V}$; intuitively, Γ will include the pair (\mathbf{x}, ctx) when type-checking an expression S if an expression of the form $\mathbf{x} = \text{ctx}$; is part of S , and the same for (\mathbf{y}, \mathbf{x}) .

An update of Γ with adding (or replacing an existing) $(\mathbf{x}, *)$ is written $\Gamma[(\mathbf{x}, *)]$, and one with removing any existing $(\mathbf{x}, *)$ is written $\Gamma[\setminus \mathbf{y}]$.

We define the (ordered) composition of two typing contexts Γ and Γ' by $\Gamma \circ \Gamma'$, so that the most recent \mathbf{x} -pair appears in the result, with the exception of the case in which $(\mathbf{y}, \mathbf{x}) \in \Gamma'$ and $(\mathbf{x}, \mathbf{z}) \in \Gamma$ —in this case, the \mathbf{y} -pair is replaced in the result with (\mathbf{y}, \mathbf{z}) .

3.3.3 Type System

The type system for goto-cc expressions is as follows:

$$\begin{array}{l} \text{CONST} \quad \Gamma \vdash c : \perp, \text{true} \quad \text{VAR} \quad \frac{x \in \mathcal{V} \setminus \mathcal{C}}{\Gamma \vdash x : \perp, \text{true}} \quad \frac{x \in \mathcal{C}}{\Gamma \vdash x : (x, \emptyset, \emptyset, \emptyset), \text{true}} \\ \\ \text{OP} \quad \frac{\Gamma \vdash E : \gamma, \beta \quad \Gamma \vdash E' : \gamma', \beta'}{\Gamma \vdash (E \text{ op } E') : \gamma \cup \gamma', \beta \wedge \beta'} \\ \\ \text{ASSIGN} \quad \frac{\Gamma \vdash E : (r, w, t, u), \beta \quad E \text{ points to } x \quad x \in \mathcal{V} \setminus \mathcal{C} \quad \Gamma \vdash S : \gamma', \beta'}{\Gamma[\alpha] \vdash (E = S) : (\emptyset, w, t, u) \cup \gamma', \beta \wedge \beta'} \\ \text{with } \alpha = (x, r(\gamma')) \text{ if } r(\gamma') \neq \emptyset \text{ and } () \text{ otherwise} \\ \\ \text{SEQ} \quad \frac{\Gamma \vdash S : (r, w, t, u), \beta \quad \Gamma \vdash S' : (r', w', t', u'), \beta'}{\Gamma \vdash (S; S') : (\emptyset, w, t, u) \cup (\emptyset, w', t', u'), \beta \wedge \beta'} \\ \\ \text{IF} \quad \frac{\Gamma \vdash S : \gamma, \beta \quad \Gamma \vdash S' : \gamma', \beta' \quad \Gamma \vdash S'' : \gamma'', \beta''}{\Gamma \vdash (\text{if } S \text{ } S' \text{ else } S'') : \gamma \cup \gamma' \cup \gamma'', \beta \wedge \beta' \wedge \beta''} \end{array}$$

4 Verification for Context-Aware Programs

Also as future work, we are looking into (i) writing specifications for context-aware applications, and (ii) automatically generating SATABS-style assertions from these specifications, to be verified by the tool's generic counterexample-based abstraction and refinement procedure.

References

- [1] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggle. Towards a Better Understanding of Context and Context-Awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307. Springer-Verlag, 1999.
- [2] Jakob E. Bardram. The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. In Hans Gellersen, Roy Want, and Albrecht Schmidt, editors, *Proceedings of the 3rd International Conference on Pervasive Computing (Pervasive 2005)*, volume 3468 of *Lecture Notes in Computer Science*, pages 98–115. Springer Verlag, May 2005.
- [3] Gerard Boudol. On Typing Information Flow. *Lecture Notes in Computer Science (LNCS)*, 3722:366–380, 2005.
- [4] Doina Bucur and Mogens Nielsen. Secure Data Flow in a Calculus for Context Awareness. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 439–456. Springer, 2008.
- [5] Guanling Chen and David Kotz. A Survey of Context-Aware Mobile Computing Research. Dartmouth Computer Science Technical Report TR2000-381, 2000.
- [6] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.
- [7] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2–4):97–166, 2001.
- [8] David Gay, Philip Levis, and Robert von Behren. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings*

of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI), 2003.

- [9] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI*, pages 1–13, 2004.
- [10] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
- [11] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 179–188. ACM, 2007.
- [12] Daniel Kroening. The CPROVER User Manual. SATABS – Predicate Abstraction with SAT. CBMC – Bounded Model Checking. <http://www.verify.ethz.ch/satabs/download/manual.pdf>; accessed August 28, 2008.
- [13] Anatole Le, Ondrej Lhoták, and Laurie J. Hendren. Using Inter-Procedural Side-Effect Information in JIT Optimizations. In *CC*, pages 287–304, 2005.
- [14] Heng Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 242–252. ACM, 2006.
- [15] Venkatesh Prasad Ranganath and John Hatcliff. Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs. In *CC*, pages 39–56, 2004.
- [16] Gruia-Catalin Roman, Christine Julien, and Jamie Payton. A Formal Treatment of Context-Awareness. In *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE 2004)*, pages 12–36. Lecture Notes in Computer Science 2984, Springer, 2004.
- [17] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, 1994.
- [18] The Open TinyOS Community. TinyOS. <http://www.tinyos.net/>; accessed August 8, 2008.

- [19] T. H. Tse, Stephen S. Yau, W. K. Chan, Heng Lu, and T. Y. Chen. Testing Context-Sensitive Middleware-Based Software Applications. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 458–466. IEEE Computer Society, 2004.
- [20] Zhimin Wang, Sebastian Elbaum, and David S. Rosenblum. Automated Generation of Context-Aware Tests. *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 406–415, May 2007.