Interface Contracts for TinyOS

Will Archer
School of Computing
University of Utah
warcher@cs.utah.edu

Philip Levis
Department of Computer Science
Stanford University
pal@cs.stanford.edu

John Regehr
School of Computing
University of Utah
regehr@cs.utah.edu

ABSTRACT

TinyOS applications are built with software components that communicate through narrow interfaces. Since components enable fine-grained code reuse, this approach has been successful in creating applications that make very efficient use of the limited code and data memory on sensor network nodes. However, the other important benefit of components—rapid application development through black-box reuse—remains largely unrealized because in many cases interfaces have implied usage constraints that can be the source of frustrating program errors. Developers are commonly forced to read the source code for components, partially defeating the purpose of using components in the first place. Our research helps solve these problems by allowing developers to explicitly specify and enforce *component interface contracts*. Due to the extensive reuse of the most common interfaces, implementing contracts for a small number of frequently reused interfaces permitted us to extensively check a number of applications. We uncovered some subtle and previously unknown bugs in applications that have been in common use for years.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by Contract*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Design, Reliability, Verification

Keywords

Validation, design by contract, TinyOS, sensor networks, automated testing

1. INTRODUCTION

TinyOS has been a successful basis for interrupt-driven sensornet applications. Its component model is designed to minimize application code size by linking in only needed functionality, and to speed application development through component reuse. Unfortunately, creating reliable TinyOS applications by building on existing components, especially those written by others, is notoriously difficult. A principal challenge is that proper use of the TinyOS interfaces has never been carefully specified, giving developers unwanted degrees of freedom. Developers of reusable components are forced to assume that their interfaces will be misused, requiring defensive programming that adds development and resource overhead. Similarly, those reusing existing components are forced to assume that interface calls may fail, even when used properly, necessitating development overhead due to error checking and failure recovery strategies. These problems have been shown to be relevant for the interfaces in TinyOS 1.x, which we checked for this paper, and we believe they are present in TinyOS 2.0 as well.

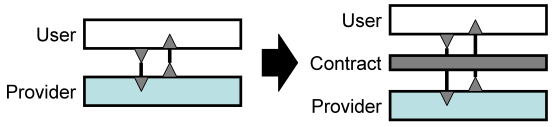

Figure 1: An interface contract enforces correct use of a nesC interface by interposing between the interface's user and provider

As a step towards solving these problems we developed *interface contracts* for TinyOS. An interface contract—depicted in Figure 1—is a checkable, executable specification that codifies the (previously implicit) rules for correctly using an interface. To check component implementations against their contracts, we implemented *dynamic contract checking* via a source-to-source program transformation that adds checks to existing TinyOS applications such that an error is raised any time an interface is misused.

Contracts provide developers with a good value proposition: a contract for a given interface has to be specified just once and then it can be reused for any instance of the interface. Similarly, there is large potential for reusing contracts across multiple applications and for reuse over time: the core TinyOS 1.x interfaces have remained fairly stable for several years.

Introducing an effective and efficient contract checking system into existing TinyOS codes required solving three difficult problems. The first problem is defining the contracts themselves. This requires reviewing the TinyOS code base to learn the expected call patterns, many of which are nonobvious and others are used in contradictory ways in different parts of the code base. Retrofitting contracts into a system that has existed without them for years is fundamentally hard. For example, we have repeatedly found that even

IPSN'07, April 25-27, 2007, Cambridge, Massachusetts, USA.

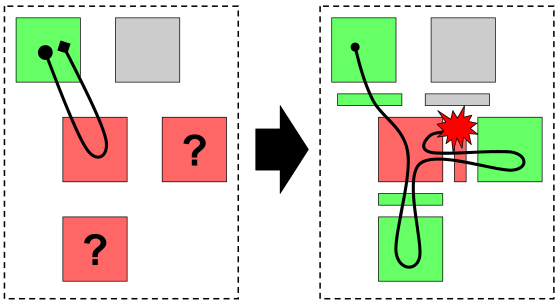

Figure 2: Without interface contracts (left), fault isolation in complex TinyOS applications is difficult. Contracts (right) continuously check interface invariants, supporting rapid and efficient fault detection.

contracts that we believe to be far too weak are routinely violated by applications that, for the most part, work. The second problem is resolving nesC features and idioms with traditional notions of contracts. The nesC language has several advanced features, such as the ability for a function call to "fan-out" to multiple callees, which a contract checker must be able to handle. The third challenge is defining a contract language that can handle the first two challenges, is easy to understand, and does not introduce significant resource overheads on highly constrained sensornet nodes.

Our eventual goal is for every interface to have a contract, including low-level hardware abstractions. This would allow unit tests in a simulation environment, where one component is tested in isolation against arbitrary inputs. Unit tests in simulation will detect some but not all bugs; real nodes should also be able to efficiently enforce contracts while running full applications. Introducing a few hundred cycles of overhead per packet transmission may be feasible, but not on every radio byte interrupt.

Our research has two main benefits. In the short term, as Figure 2 illustrates, contracts serve as checkable, executable documentation that makes it easier for developers to create correct code, and to rapidly locate bugs in incorrect code. In the longer term, we expect that it will be possible to use formal methods to statically check both individual components and entire applications against their interface contracts. Checking individual components is very powerful because it shows that a particular component is correct in any possible instantiation, rather than just in one specific one. Furthermore, we expect that component-level checking will be useful in an assume-guarantee reasoning scheme [5] that can inductively show that an entire application is correct.

2. TINYOS BACKGROUND

TinyOS [7] is a component-based operating system in which components interact through typed interfaces. The OS is written in nesC [4], a dialect of C with support for components, interfaces, concurrency analysis, and network types. Building a TinyOS application involves connecting the interfaces of components together. Interfaces are bidirectional, in that they can describe both the call that a user can make on a service provider (commands) as well as calls a provider can make on a user (events). For example, sending a packet is a command, while receiving a packet is an event. Generally, an interface supports a narrow but complete abstraction such as using a timer, writing to a non-volatile log, or receiving packets.

```
interface SendMsg {
  command result_t send (uint16_t addr,
                         TOS_MsgPtr,
                         uint8_t len);
  event result_t sendDone (TOS_MsgPtr,
                           result_t success);
}
```

Figure 3: The SendMsg interface

```
interface Timer {
  command result_t start (char type,
                          uint32_t interval);
  command result_t stop();
  event result_t fired();
}
```

Figure 4: The Timer interface

TinyOS does not support blocking. Instead, slow operations—especially those that involve hardware latencies—are split-phase. For example, Figure 3 shows the basic packet communication interface, SendMsg. Rather than wait until an operation (e.g., SendMsg.send) completes, the interface command returns immediately, allowing the application to continue processing. When the operation does complete, the interface signals the completion event (e.g., SendMsg.sendDone), at which point the user can reclaim the packet buffer. The split-phase semantics has a number of advantages but it also creates difficulties for developers, who are forced to explicitly maintain component state across multiple invocations. Split-phase operation is the source of quite a few of the interface misuses that we found.

An implementation cannot name another component: components interact solely through interfaces. This explicit separation allows programmers to easily change which implementation is used. For example, a component named AppM that uses SendMsg can be directly connected to a radio-only communication stack, a radio-serial hybrid stack, or to a send queue without changing AppM's code.

The ability to easily switch between different implementations of an interface requires that the implementations are interchangeable. Unfortunately, at present, there is no precise specification of the semantics and call patterns of most interfaces. As there is a good deal of latitude in implementation, a component must be able to handle a wide range of behaviors. This imprecision leads to bloated code, as every component must be programmed defensively.

3. DESIGNING A CONTRACT SYSTEM

This section describes our contract language and how developers can use it to specify an interface's semantics.

3.1 The contract language

Our contracts are specified in a stylized version of C, in order to provide developers with a familiar environment. Figure 4 shows TinyOS's timer interface and Figure 5 illustrates the basic contract syntax by showing the contract for one of the interface's commands. In this example, the Timer.start() command generates a state transition only if the command returns `SUCCESS`. The contract for an interface call contains `PRE` and `POST` sections, where precondition and postcondition code is placed. In Figure 5, the

```
// Global variable which represents
// timer state.
Timer_state_t state = IDLE;

void start (char type, uint32_t interval) {

PRE:
  if(state != IDLE) {
    ERROR ("NON-IDLE TIMER STARTED");
    print_dec_int ((int)ID);
  }

POST:
  if (R_VAL == SUCCESS) {
    if (type == TIMER_ONE_SHOT) {
      state = ONE_SHOT;
    } else if (type == TIMER_REPEAT) {
      state = REPEATING;
    }
  } else {
    state = IDLE;
  }

}
```

Figure 5: Contract for the Timer.start() command

```
// State field to append to each TOS_Msg
typedef struct TOS_Msg {
  TOS_Msg_state_t msg_state;
} TOS_Msg  __attribute__((append));

void send( TOS_Msg * msg, uint16_t length) {

POST:
  if(R_VAL == SUCCESS){
    if(msg->msg_state != USER_OWNED){
      ERROR("SEND ERROR:SEND OS_OWNED");
    }
    msg->msg_state = OS_OWNED;
  }

}
```

Figure 6: Contract for the SendMsg.send() command

safety check is placed in the precondition in order to generate a failure as early as possible.

Contracts are permitted to examine arguments to interface calls, and also to examine data referenced by these arguments. Contracts may access a call's return value using a special `R_VAL` variable. As Figure 5 shows, the return value is often used to make conditional changes to the interface's state based on the success or failure of the call. Contract code may declare state variables; a separate copy of these variables is instantiated for each use of a contract in an application. The special variable `ID` is mapped to the parameterized value of the interface, should one be available. It is provided as a convenience for making error messages more useful.

Contracts are *not* allowed to access component-local information that does not flow across some interface. This would violate our design principle that contracts are about interfaces, not implementations.

Some interface contracts, such as the one in Figure 5, can be expressed in terms of checks on the state of the interface. In other cases, the interface itself is stateless and the contract enforces preconditions and postconditions on data structures in the application.

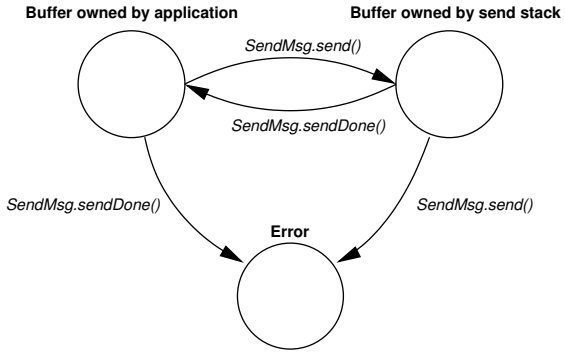

Figure 7: State machine for SendMsg interface

This is the case for the SendMsg interface shown in Figure 3. To support this kind of interface, we permit contracts to augment data structures with new fields, and to access these fields. For example, Figure 6 shows how a state variable is added to the TinyOS packet buffer data structure. In this case, when a buffer is passed to SendMsg.send(), it is assumed to be unavailable for an additional send() request until the corresponding sendDone() event fires. Thus, attempting to send the same buffer twice before the first request completes is a contract violation, although an attempted send() using a different buffer is not.

3.2 Writing contracts

The first, and most important, step in producing a contract is determining the state machine that is implied by the interface. While this process seems obvious, getting the contracts written correctly is non-trivial and required a significant amount of development time on our part. One of the important contributions of our tool is providing a set of contracts for the trickiest and most commonly used interfaces within TinyOS. By providing these we enable relatively thorough checking of existing applications with minimal additional programming. Once a state machine that encapsulates the interface's behavior has been established, the transitions can be translated into executable contracts. The state machine for the SendMsg interface (Figure 3) is shown in Figure 7.

Figure 8 shows the number of interfaces contained in our sample applications and how many we are currently checking. By running checks on only a few interesting interfaces we are able to cover roughly half of the total number of interface instances. For this paper we focused on interfaces that were common enough to apply to a wide variety of applications and interesting enough to have the potential to contain worthwhile bugs. Though there are a significant number of interfaces that we do not cover, the majority of them are not stateful and hence require no contracts, or are specific to individual applications.

3.3 Warnings vs. errors

We came to recognize that contract violations have multiple levels of severity, and we adopted the convention that contract violations are divided into two categories. Warnings indicate usage that is in poor taste but that does not, as far as we know, directly lead to application malfunction. For example, a warning is generated when an application initializes a timer component that already has a timer set to fire. More severe are errors, which indicate behavior that cannot possibly be correct, such as concurrently passing a single packet buffer to the receive subsystem multiple times.

	StdControl	Timer	Send	Receive	Pot	Clock	Leds	ADC	ADCControl	RouteControl	Other
BlinkTask	3	2	0	0	3	2	2	0	0	0	3
CntToLedsAndRfm	15	6	6	7	2	2	3	3	0	0	23
Surge	25	8	12	12	2	2	6	8	4	3	23
Surge_TinySec	26	8	14	14	2	2	6	8	4	3	49
Surge_Reliable	37	10	13	12	2	2	3	4	0	3	61

Figure 8: Number of times each interface is used by some TinyOS applications

4. CHECKING CONTRACTS

We created a source-to-source transformation tool that inputs a collection of contracts and the C code emitted by the nesC compiler, and outputs a new C program that, when run, dynamically checks interface contracts. Our tool is based on CIL [12], a parser, typechecker, and intermediate representation for C.

4.1 Adding contract checks

Our tool, shown in Figure 9, performs the following steps.

Determine interface aliasing. Because interface names are obscured in the nesC compiler's output, our tool requires some additional information before it can add contract checks. The nesC compiler optionally dumps this kind of wiring information as XML. We use this feature to map function names in the emitted C code to their actual interface instances.

Construct callgraph and application wiring from source code. The nesC compiler uses well-defined name mangling schemes, permitting us to recover component wiring information directly from the compiler output. By pulling apart the mangled function names and knowing the interface aliasing for a given application, we know which functions implement a given interface, and which component they belong to. We also build a whole-program callgraph from the application code, supporting an optimization that we describe below.

Add contract checking code. Interfaces in TinyOS contain two different kinds of function calls—commands and events—that require slightly different treatment by our tool. We instrument commands by including precondition code at the beginning of the function and postcondition code before all returns. Instrumenting events, which originate in low-level code and move up, requires a different approach. Because there may be necessary logic before the upcalls, we cannot assume that the state transition occurs at the beginning of the interface function in question. For this reason, we instrument before and after each upcall, plugging in the upcall's return value for R_VAL when needed.

Add contract state variables. Contract state variables come in two flavors: per-contact and per-data-structure. Per-contract variables are straightforward to add, as new global variables. Per-data-structure variables are appended to the end of existing structs.

4.2 Optimizations

Avoiding redundant checking. We found that typically, many uses of TinyOS interfaces simply pass calls through from one component to another without performing any useful computation. Normally these calls introduce no overhead because they are eliminated through function inlining. However, if we instrumented all of these

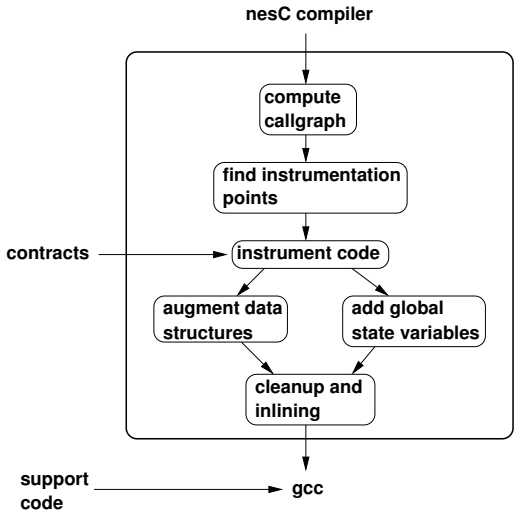

Figure 9: Our infrastructure for adding dynamic contract checks to a TinyOS application

uses of interfaces with contract checks, the overhead would be significant. As an optimization, we avoid performing contract checks in the case where a component passes a call from one instance of an interface to another instance of the same interface without performing any additional processing.

Inlining and cleanup. The nesC compiler attempts to generate small object code by inlining small functions and functions that have one call site. We found that adding contract checking code to the nesC compiler's output invalidated its inlining decisions by making functions larger, resulting in large code size blowup: often by 400% or more. In previous work [3, 14] we developed an inliner and a strong dead code eliminator for C. We use these to reduce the code size overhead of interface contracts. Our inliner makes its decisions based on code size after contracts are added, enabling it to make more appropriate inlining decisions.

4.3 Handling contract violations

When a contract is violated, we have a variety of options for reporting the error or taking corrective action. To support easier testing of applications in Avrora [17], a cycle-accurate sensor network simulator, we developed a simple printf-like function for dumping warnings and errors to the simulator console. When contract-enabled applications are run on real hardware, we suppress warnings and signal errors by using the mote's three LEDs to blink out a failure identification code in octal. This code can be turned into a source code location offline using a separate tool. In a deployed sensor network, a contract violation should be logged over the net-

work using a logging service and the node should be rebooted; we have not implemented this.

5. RESULTS

This section summarizes our results from adding dynamic contract checking to several TinyOS 1.x applications.

5.1 Overhead of dynamic checking

Figure 10 shows the percentage change in data size, code size, and duty cycle due to dynamically checking contracts on the interfaces listed in Figure 11. As described in Section 4, we used our inliner and dead code eliminator to avoid unnecessary resource bloat. To ensure a fair comparison, the numbers in Figure 10 compare the original application, inlined and cleaned up (reducing code and data size by a few percent relative to the default compilation), against the application with contract checks, inlined and cleaned up. Duty cycle—the fraction of time a sensornet application spends with the processor running—was computed using Avrora [17].

The most significant impact on our applications was an increase in data size through the addition of variables to track interface state. In situations where we add fields to data structures this can be especially problematic, because we must conservatively augment all instances of each type of structure that is augmented by any contract. Another kind of bloat comes from parameterized interfaces, necessitating an array of interface state variables.

The increase in application CPU usage as a result of our contract checks—measured by observing the change in duty cycle—is negligible. In spite of instrumenting several widely used interfaces, many of which perform processor-intensive tasks like transmitting packets over the radio, the actual processing overhead for checking the contracts is low. The primary reason for this is that interface calls, which only happen when crossing from one module to another, are not often included inside the main processing loops. Also, a properly designed low-level component will place upcalls to external interfaces (and thus our associated contract checks) in a long-running task, not in an interrupt handler. This convention has been consistently adhered to in the applications we tested.

5.2 Bugs found

Running TinyOS applications compiled with contract checking revealed bugs in several applications. For purposes of this section, a bug is a clear-cut interface contract violation. In many cases, application-specific semantics are such that these bugs are tolerated in one way or another. Indeed, that is what we would expect since the applications we tested are part of the TinyOS distribution and have been in use for several years. Even so, we believe these bugs should be found and fixed: the components in question are intended for reuse and are part of the core TinyOS distribution. Design improvements in TinyOS 2.0 [9, 10] correct some of these problems, attesting to their validity. The fact that such subtle problems can be uncovered by enforcing relatively simple contracts suggests that our approach has merit. Here we describe some of the most interesting bugs.

5.2.1 Split-phase dispatch

Many-to-one wiring, while central to the design of TinyOS, is a source of subtle errors, especially for split-phase operations. If multiple users wire to the service interface, then the completion event of a request from one user is signaled to all users. For example, in the Surge application, the SendMsg interface provided by QueuedSendM component is used by three different components: BCastM, MultiHopEngineM, and MultiHopLEPSM.

In this situation, BCastM, MultiHopEngineM, and MultiHopLEPSM are wired to the same SendMsg interface. The QueuedSendM component has no information to determine which of the three users called SendMsg.send(). Therefore, when it signals SendMsg.sendDone(), the event handler is called on all three users. Because QueuedSendM can have multiple outstanding packets, it is possible that more than one of the users has a packet in the send queue. Therefore, more than one of them might be waiting for a SendMsg.sendDone(). If the user does not check that the buffer passed into the SendMsg.sendDone() event is its own, then it might incorrectly conclude that its transmission has completed when the packet is still in the queue. Both BCastM and MultihopEngineM correctly perform the check, but MultihopLEPSM, which is responsible for link estimation, does not. It can therefore corrupt routing beacons, possibly causing routing failures. Like many of the errors uncovered with our tool, this problem does not manifest itself until the application's components are wired together. Inspection or analysis of the individual nesC components in isolation is insufficient to detect many of the problems discussed here. The example shown above can be seen in the QueuedSendM component's QueueSendMsg.sendDone() call in the C source file for Surge, Surge_reliable, and Surge with TinySec.

This appears to be a fundamental problem with the organization of the TinyOS 1.x communication layers, which create multiple interface layers that all use the same variable within the send data structure to determine event routing. This creates a hole in the idea that a shared component can be treated as solely owned by including a parameter in the interface definition, since it is predicated on global knowledge of which message types have been already defined. This problem has been addressed in TinyOS 2.0 through the use of virtualized sending abstractions [10].

5.2.2 Interface specification ambiguities

If an interface is weakly specified, then some implementations take stronger checking approaches than others. This creates a situation where it is not clear which side of an interface is responsible for checking error conditions. A component tested against a strict implementation may assume the other side of the interface performs the checks. But if that component is wired to a looser implementation that assumes the caller performs the checks, then havoc can ensue.

For example, when the SendMsg.send() call is successful, ownership of the packet buffer is passed to the SendMsg provider. A subsequent SendMsg.sendDone() event transfers of ownership of the buffer back to the SendMsg user. It is an error for a component to access the buffer while the other component owns it. For example, if the user modifies the buffer after a successful call to SendMsg.send(), then it may cause the data payload and the precomputed CRC to become inconsistent, leading to a failed CRC check at the receiver.

Some components that provide SendMsg, such as the AMPromiscuous component that is part of the TinyOS core, implement extra checking: since this component can only transmit one buffer at a time, it rejects multiple sends of the same buffer. Since a repeated send() will automatically fail, with no reads or writes to the buffer, multiple requests are not a source of program errors. Other implementations of SendMsg will tolerate multiple pending send requests, notably QueuedSendM, and in those instances multiple sends of the same buffer *can* compromise its integrity.

In fact, this functionality was discovered when the QueuedSendM module attempted to send a single buffer via AMPromiscuous twice, violating a prior, stricter, version of the SendMsg contract. Because of the unstated checking in AMPromiscuous, this does not result in

	Resource Usage: Without contracts / With contracts (% increase)				
	Data size (bytes)	Code size (bytes)	Duty Cycle (%)	Warnings	Verified errors
BlinkTask	52 / 89 (71.1%)	1540 / 1918 (26.5%)	0.0280 / 0.0283 (1.1%)	0	0
CntToLedsAndRfm	450 / 534 (18.2%)	10572 / 11476 (8.6%)	6.310 / 6.314 (0.063%)	6	0
Surge	1931 / 2242 (16.1%)	16196 / 17306 (6.8%)	7.789 / 7.792 (0.037%)	37	4
Surge_Reliable	2165 / 2490 (15.0%)	21532 / 22818 (6.0%)	9.139 / 9.140 (0.013%)	42	5
Surge with TinySec	2174 / 2476 (13.9%)	24968 / 27398 (4.5%)	8.069 / 8.201 (1.64%)	47	6
HighFrequencySampling	851 / 944 (10.9%)	17082 / 18002 (5.4%)	5.844 / 5.845 (0.017%)	5	0

Figure 10: Results from running applications with dynamic contract checking

	Lines of code in contract	Warnings found	Verified errors found
Send	60	0	7
Timer	57	2	6
Receive	37	0	1
StdControl	66	73	1
ADCControl	47	17	0

Figure 11: Contract statistics. The warning and error counts refer, respectively, to instances of minor and serious contract violations in application source code. For example, a StdControl being started multiple times would generate a warning, whereas a StdControl being started without being initialized is an error.

a program error, but it is interesting to note that the proper operation of the QueuedSendM component relies on AMPromiscuous enforcing sending restrictions that QueuedSendM itself does not. These ambiguities are an obstacle to component reuse, and enforcing contracts will allow developers to make assumptions about the behavior of subcomponents with greater confidence. It should be noted that TinyOS 2.0 has eliminated this inconsistency by disallowing multiple sends from the same component in AMQueue, its version of QueuedSendM [8].

5.2.3 Initialization problems

TinyOS operates on the principle that each module is responsible for initializing and starting any subcomponents that it uses. This design feature results in shared low-level interfaces being repeatedly initialized and started as all the modules that use them come online. This is obviously inefficient, and the many multiple initialization/start warnings for the StdControl interface in Figure 11 illustrate how widespread this behavior is.

In the instance of CC1000RadioIntM, which is the default radio module for the mica platform, the component will begin a one-shot timer every time it starts. Since the start command is repeatedly called, resetting the one-shot timer each time, this contract misuse skews the amount of time before the timer fires. Even though TinyOS is not a hard real-time system, contract misuse *is* responsible for the Timer module not fulfilling its basic functionality, namely failing to trigger an event at the appropriate interval. While this is a benign bug to the best of our knowledge, in a situation where precise timing is required this would result in a difficult to debug error.

Repeated initialization of CC1000RadioIntM is a source of errors in Surge, Surge_Reliable, and Surge with TinySec, because they have similar usage of the underlying radio component. This problem has been recognized, and largely dealt with, in TinyOS 2.0 by separating the initialization and module start features [9].

5.3 Unexpected interface usage

An important result of our work is nailing down the semantics of interfaces. In many situations, we have found the actual usage to be somewhat ad hoc, and by specifying a concrete interface contract we discovered redundancies and hidden requirements for seemingly straightforward interfaces.

For example, the mica2 platform provides an implementation of the ADCControl interface to control its analog to digital converters. To make use of the ADC, the hardware must be initialized, each component has to register itself with a port in the ADC, and then can use the ADC normally. Under the programming conventions given, however, the port registration takes place before initialization. Since using the interface, and underlying hardware, before initializing it is an obvious error, the ADCControl.bindPort() command contains a hidden call to the low-level initialization function, which is duplicated in the subsequent ADCControl.init() call. The hidden call is also contained in a separate call to the implementation's global initialization function, further confusing the situation. The result of this redundancy is that the ADCControl is initialized at least two extra times for each component that uses it, and the actual initialization function must be constructed in such a way as to allow port requests and initialization to be interleaved without corrupting its state.

We cannot in good conscience call this a contract violation, since the established convention is maintained throughout, but it is at least inefficient. Deeper examination of the implied usage requirements of commonly used interfaces often reveal similar inconsistencies and interface quirks that provide ample opportunity for programming difficulties. Imagine developers writing a new ADCControl implementation for different hardware. If they were unaware that they should expect their initialization routine to be called multiple times at random points in the program, it is unlikely they would code it with those constraints in mind. Odd errors will result. We would hope that our contract checking tool would encourage more straightforward programming by shedding light on these previously buried problems.

6. FUTURE WORK

Cross-contract checks. Since our contracts are standalone units, we cannot specify constraints that span multiple interfaces. For example, although it is a serious error for a buffer to be simultaneously used by both a Send and Receive interface, our contract system cannot detect this problem.

Contract requirements beyond command/event instrumentation. Our current contract system can check invariants whenever control flow crosses a component boundary. This is insufficient to check some properties that we would like to check. For example, in addition to verifying that an application respects the

message buffer state machine shown in Figure 7, we would like to check that no component references a buffer that it does not own. This requires instrumenting memory operations: a finer-grained kind of program transformation than our current system supports.

Calls of Death Generation. Explicit contracts will allow us to inductively validate a software component. Using a simulation environment such as TOSSIM or Avrora, we would like to automatically generate unit tests for individual components. These unit tests will execute stub code filling in the other side of all of a component's interfaces. This will allow us to ask whether a component follows its contracts given that all other components follow theirs. This approach is analogous to input-of-death systems that explore the complete set of possible inputs using lazy binding in order to generate an input that causes a crash [1].

Static checking. The modular nature of nesC and TinyOS applications makes the idea of static checking very attractive. Even though most embedded applications are small and relatively straightforward, the degree of concurrency present makes checking the entire program prohibitively expensive. However, checking individual components within the framework of a contract is an interesting possibility, using stub generation techniques similar to those described above.

7. RELATED WORK

Interface contracts. Design by contract [11] is a well-known software engineering technique that is based on ideas from formal specification and verification. Relative to existing work on contracts, our research innovates by applying contracts to the nesC language. Features such as commands, events, fan in, fan out, and parameterized interfaces must all be supported. In addition, we developed a source to source translation tool, that is fairly straightforward to use, that transparently inserts dynamic contract checks into TinyOS applications.

Tool support for reliable sensor networks. Volgyesi et al.'s Gratis tool [18] is the most closely related sensornet work to ours. Gratis is a GUI-based tool for composing sensornet applications with support for verification of component compositions based on *interface automata*. Like our contracts, interface automata encode otherwise implicit rules for interface usage. The primary different between interface automata and our contracts is that interface automata are statically checked against each other (as opposed to being checked against code) using a formally defined notion of *compatibility*. On the other hand, our contracts are dynamically checked against executions of actual component implementations. Chakrabarti et al. [2] previously applied interface automata to TinyOS applications,

Sympathy [13] is a distributed logging, debugging, and fault diagnosis infrastructure for sensornets. Sympathy appears to be almost perfectly complementary to our contract work: it could be used to log contract failure events and relay them to a base station.

t-kernel [6], SoS [15], Virgil [16], and our type-safe version of TinyOS [14] all use language-based protection to avoid memory safety violations in sensornet applications. This kind of protection is also complementary to interface contract checking. We expect that using the two techniques together will make it significantly easier to develop reliable sensornet applications.

8. CONCLUSION

We developed interface contracts for TinyOS components. Contracts are executable specifications of proper interface usage that also serve as documentation, providing developers with an alternative to inferring interface usage by reading code. Contract checking exposes bugs and hidden assumptions, and also permits developers to write fewer lines of defensive error-handling code.

We implemented contracts for a number of commonly-used TinyOS 1.x interfaces, as well as a source-to-source translation tool for adding dynamic contract checks to existing TinyOS applications, with modest resource overheads. We checked a number of applications, uncovering several instances of bugs and unexpected program behavior. The set of contracts that we implemented covers roughly half of the interfaces instances used in our test applications, and a substantially higher percentage of the most interesting and tricky ones.

Interestingly, several problems that our contracts uncovered in TinyOS 1.x applications were precisely those that had motivated the design of TinyOS 2.0. This confirms the TinyOS 2.x developers' intuitions that there were significant quirks and latent bugs in TinyOS 1.x applications.

We believe that interface contracts are highly beneficial to sensornet application developers: they make many aspects of development easier, and allow a modest investment in contract generation to be employed to test a large section of the code base. In the long run, every TinyOS interface should be accompanied by a contract, and contracts should be routinely, or continuously, checked. Furthermore, static checking methods should be used to verify, once and for all, that stable components correctly implement the desired functionality.

9. REFERENCES

[1] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proc. of the 13th ACM Conf. on Computer and Communications Security (CCS)*, Alexandria, VA, October 2006.

[2] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, Marcin Jurdzinski, and Freddy Y. C. Mang. Interface compatibility checking for software modules. In *Proc. of the 14th Intl. Conf. on Computer Aided Verification (CAV)*, pages 428–441, 2002.

[3] Nathan Cooprider and John Regehr. Pluggable abstract domains for analyzing embedded software. In *Proc. of the 2006 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 44–53, Ottawa, Canada, June 2006.

[4] David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

[5] Dimitra Giannakopoulou, Corina S. Pasareanu, and Jamieson M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Proc. of the 26th Intl. Conf. on Software Engineering (ICSE)*, pages 211–220, Edinburgh, Scotland, May 2004.

[6] Lin Gu and John A. Stankovic. t-kernel: Providing reliable OS support to wireless sensor networks. In *Proc. of the 4th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Boulder, CO, November 2006.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, November 2000.

[8] Philip Levis. Amqueue implementation, 2006. `http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/tos/system/AMQueueImplP.nc`.

[9] Philip Levis. TinyOS Extension Proposal (TEP) 107: TinyOS 2.x Boot Sequence, 2006. `http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep107.html`.

[10] Philip Levis. TinyOS Extension Proposal (TEP) 116: Packet Protocols, 2006. `http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep116.html`.

[11] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[12] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, pages 213–228, Grenoble, France, April 2002.

[13] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Proc. of the 3rd ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 255–267, San Diego, CA, November 2005.

[14] John Regehr, Nathan Cooprider, Will Archer, and Eric Eide. Efficient type and memory safety for tiny embedded systems. In *Linguistic Support for Modern Operating Systems (PLOS)*, San Jose, CA, October 2006.

[15] Ram Kumar Rengaswamy, Eddie Kohler, and Mani Srivastava. Software-based memory protection in sensor nodes. In *Proc. of the 3rd Workshop on Embedded Networked Sensors (EmNets)*, Cambridge, MA, May 2006.

[16] Ben L. Titzer. Virgil: Objects on the head of a pin. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, October 2006.

[17] Ben L. Titzer, Daniel Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, Los Angeles, CA, April 2005.

[18] P. Volgyesi, M. Maroti, S. Dora, E. Osses, and A. Ledeczi. Software composition and verification for sensor networks. *Science of Computer Programming*, 56(1–2):191–210, April 2005.