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.

1. INTRODUCTION

TinyOS has been a successful basis for interrupt-driven sensor-net 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. One principal challenge is that proper use of the TinyOS interfaces has never been carefully specified, giving developers too many 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.

As a step towards solving these problems we developed interface contracts for TinyOS. An interface contract 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 implemented just once and then it can be reused many times even inside a single application. 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 requires solving three difficult problems. The first problem is defining the contracts themselves. This requires reviewing the TinyOS codebase to learn the expected call patterns, many of which are contradictory. Retrofitting contracts into a system that has existed without them for years is fundamentally hard. For example, we have repeatedly found that even 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 call patterns and features 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 overheads. Our eventual goal is for every interface to have a contract, including low-level hardware abstractions. Introducing a few hundred cycles of overhead per packet transmission may be feasible, but not on every radio byte interrupt. The very constrained I/O, RAM, and CPU of common mote platforms makes this third challenge particularly critical.

Our research has two main benefits. In the short term, 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 [4] that can inductively show that an entire application is correct.

2. TINYOS BACKGROUND

TinyOS [6] is a component-based operating system in which components interact through typed interfaces. The OS is written in nesC [8], 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, interfaces describe a narrow but complete abstraction, such as writing to a non-volatile log, a timer, or receiving packets.

No operation in TinyOS blocks. Long-lasting operations, especially those that involve hardware latencies, are split-phase. Figure 2 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. This split-phase operation is the source of quite a few of the contract difficulties we encounter in this paper.

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 change implementations assumes that the implementations are equivalent. Unfortunately, there is no precise specification of the semantics and call patterns of many interfaces. As there is a good deal of latitude in implementation, a component must be able to handle a wide range of behaviors. For example, in the case of SendMsg, radio stacks can only have one pending send (subsequent sends return FAIL), while a send queue can have multiple pending sends. This imprecision leads to bloated code, as every component must be programmed defensively.

3. DESIGNING CONTRACTS

This section describes the design of the part of our contract checking system that is visible to contract developers.

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

Figure 2: The SendMsg interface

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

Figure 3: The Timer interface

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 4: Contract for the Timer.start() command

3.1 The contract language

Our contracts are specified in a stylized version of C, providing developers with a familiar environment. As Figure 4 illustrates, we keep the basic syntax of C, but add a few new reserved words that permit relatively easy interfacing with the code being checked. Our contracts separate the state transitions and corresponding correctness assertions into two sections, labeled PRE and POST. This is necessary because certain transitions are expected to take place immediately upon entry into a command, while others only happen upon the command’s return. R_VAL is a special variable available to postconditions that maps to the return value of the function covered by a contract. This permits a contract to make conditional state transitions based on a command’s success or failure. Similarly, ID is mapped to the parametrized value of the interface, should one be available. It is provided as a convenience for debugging contract errors, since the same contract may be tested at many points within the program, and a generic contract error is uninformative.

Figure 4 shows an example contract where a Timer can only be considered to start once the Timer.start() command has successfully completed. The state transition occurs only when the command returns SUCCESS. The safety assertion is placed in the PRE section of the contract, since a request to start a Timer that is already running, or has not been initialized, is an error regardless of its success or failure. In this case we prefer to generate the error and take whatever corrective measures are available before the command executes, potentially corrupting further program state. For this reason, our contracts place the correctness assertions associated with the state in the PRE block whenever possible.

The contract for the Timer interface can be checked by adding some global state to a TinyOS application. On the other hand, some contracts require associating state machines with data structures rather than with interface instances. In these situations our contract checker must add a field into the relevant data structure to
// State field to append to each TOS_Msg
typedef struct TOS_Msg {
    uint8_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 5: Contract for the Send.send() command

Figure 6: Sample state machine for SendMsg interface

represent its contract state. The most common examples of this are
the various send and receive interfaces that handle communication
over the radio. These interfaces track the usage of the send/receive
message buffers and maintain no state regarding the interface as a
whole. In the example given in Figure 5 when a buffer is passed to
SendMsg.send(), it is assumed to be unavailable for an additional
send until the sendDone() event fires. An additional send() call
to the same interface, made with a different buffer, would not be
a contract violation, although attempting to send the same buffer
twice before the first request completes would.

3.2 Writing contracts

The first, and most important, step in producing a contract is
figuring out what patterns of calls are permitted by an interface.
While this step seems totally obvious, getting the contracts written
correctly is non-trivial and required a significant amount of develop-
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 a relatively thorough program test with minimal program-
ing. Once a state machine that encapsulates the interface’s behav-
or has been established, the transitions can be translated into our
contract language and patched into the program source. A sample
state machine for the SendMsg interface is provided in Figure 6.

Figure 7 shows the number of interfaces contained in our sample
applications and how many we are currently checking. As you can
see, by running checks on only a few interesting interfaces we are
able to cover roughly half of the applications. 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.

3.3 Warnings vs. errors

We came to recognize that contract violations have multiple lev-
el of severity, and we adopted the convention that contracts emit
a warning for a less severe violation vs. an error for a more severe
violation. 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 has already been initialized, and a timer set.
More severe are errors, which indicate behavior that cannot possi-
bly be correct, such as concurrently sending a single packet buffer
to the receive stack multiple times.

4. INFRASTRUCTURE FOR CHECKING
CONTRACTS

We created a source-to-source transformation tool that inputs a
collection of contracts and the C code emitted by the nesC com-
piler, and outputs a new C program that dynamically checks inter-
face contracts. Our tool is built on CIL [10], a parser, typechecker,
and intermediate representation for C.

4.1 Adding contract checks

Our tool, shown in Figure 8[11], performs the following steps.

Determine interface aliasing. Because interface names are
obscured in the nesC compiler’s output, our tool requires some ad-
ditional information before it can add contract checks. The nesC
compiler can dump a lot of extra information of this kind as XML
files, we use this feature to map function names in the C code to
interface instances.

Though there are a significant number of interfaces that we do
not cover, the majority of them are relatively uninteresting and ful-
fill initialization functions, or are application-specific and not gen-
erally applicable.
Construct call graph 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 call graph from the application code.

Identify functions to instrument. After parsing all interface contracts that are to be checked, and creating a call graph, it is possible to determine which function calls we need to instrument with contract checks. Many of the function calls that implement an interface are purely connecting functions that take the place of wiring between modules. Since these functions exist only to pass parameters back and forth between the actual implementation and the modules that use it, we prefer to leave them as they are, and only instrument the code that is actually doing the work.

Add contract checking code and global contract state variables. For every instance of every checked interface, our tool adds a new collection global state variables to the program being checked. These variables are specified in the contracts themselves; they keep track of the state of interfaces.

Interfaces in TinyOS contain two different kinds of elements: commands and events. A command is a synchronous request made to the interface, conceptually the same as a function call, only constrained by the interface specification. An event is an spontaneous call back from the interface, commonly in response to a previous request. For example, the Send.send() command requests transmission of a packet across the network. The Send.sendDone() event will then fire once the send has completed, signifying the buffer is no longer in use by the send stack.

Commands and events require slightly different treatment by our tool. We instrument commands in a straightforward fashion by including precondition code at the beginning of the command’s implementation and postcondition code before all returns (performing checks on the return value before updating the state as necessary). Instrumenting events, which originate in low-level code and move checks on the return value before updating the state as necessary). Instrumenting events, which originate in low-level code and move between modules. Since these functions exist only to pass parameters back and forth between the actual implementation and the modules that use it, we prefer to leave them as they are, and only instrument the code that is actually doing the work.

Add contract state fields to data structure definitions. Because global variables to hold contract state are not sufficient to check all interfaces’ contracts, we must also add fields to hold interface state to the pertinent data structures. As before, we must create a separate field for every module that implements the interface. Because we don’t have the capacity to determine which modules will

*Figure 7: Number of times each interface is used by some TinyOS applications*

<table>
<thead>
<tr>
<th>Interface</th>
<th>StdControl</th>
<th>Timer</th>
<th>Send</th>
<th>Receive</th>
<th>Pot</th>
<th>Clock</th>
<th>Leds</th>
<th>ADC</th>
<th>ADCControl</th>
<th>RouteControl</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlinkTask</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3(15)</td>
</tr>
<tr>
<td>CntToLedsAndRfm</td>
<td>15</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>23(77)</td>
</tr>
<tr>
<td>Surge</td>
<td>25</td>
<td>8</td>
<td>12</td>
<td>12</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>23(115)</td>
</tr>
<tr>
<td>Surge_TinySec</td>
<td>26</td>
<td>8</td>
<td>14</td>
<td>14</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>49 (136)</td>
</tr>
<tr>
<td>Surge_Reliable</td>
<td>37</td>
<td>10</td>
<td>13</td>
<td>12</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>61(147)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface</th>
<th>Lines</th>
<th>Warnings</th>
<th>Verified errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send</td>
<td>60</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Timer</td>
<td>57</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Receive</td>
<td>37</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>StdControl</td>
<td>66</td>
<td>73</td>
<td>1</td>
</tr>
<tr>
<td>ADCControl</td>
<td>47</td>
<td>17</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 10: Contract statistics. The warning and error counts refer to instances of bugs in application source code.*

Use a given data structure statically, we must conservatively include all possible contract state fields in the data structure.

Inlining and cleanup. The nesC compiler makes a set of function inlining decisions using a heuristic that is designed to reduce code size. We found that adding contract checking code to the nesC compiler’s output invalidated its inlining decision (by making functions larger), resulting in large code size blowup (often by 400% or more) when the application with contract checks was compiled. In previous work [2, 12] we developed an inliner for C code that is followed up by a strong dead code elimination pass. Our inliner makes its decisions based on code size after contracts are added, enabling it to make good decisions.

4.2 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 [15], a cycle-accurate sensor network simulator, we developed a simple printf-like function for dumping warnings and errors to the simulator console. In deployed systems a contract violation should be logged over the network using a logging service and the node should be rebooted.

5. RESULTS

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

5.1 Overhead of dynamic checking

The table in Figure 9 shows the percentage change in memory size and duty cycle as a result of dynamically checking the contracts from Figure 10. As described in Section 4, we used our inliner and dead code eliminator to avoid unnecessarily bloating applications. To ensure a fair comparison, the numbers in Figure 9 compare the original application, inlined and cleaned up (this invariably reduces code size by a few percent relative to the code size of the default compilation), against the application with contract checks, inlined and cleaned up. Duty cycle—the fraction of time a sensor net application spends with the processor running—was computed using Avrora [15].

The single biggest impact on our applications was an increase in...
data size through the addition of variables to track interface state. This is exacerbated by the necessity of duplicating these variables for each instance of an interface in an application, even though in some cases the duplicate checks add no new error checking power. In situations where we add fields to data structures this is especially problematic. We cannot determine at compile time exactly which modules will access a particular data structure, so we must conservatively add fields to all instances of the structure. Also, when an interface is parametrized, we handle the individual states by changing the single state variable into an array and indexing by each module’s parameter. We also create a modest amount of code bloat by adding interface checks to the application.

The increase in duty cycle as a result of our contract checks 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 comparatively low. We believe the primary reason for this is that interface calls, which only happen when crossing from one module, or one interface, to another, are not often included inside the main processing loops. An interface command or event can certainly be at the exit or entrance, but will not be repeatedly checked while the instrumented module does any heavy computation. In a few cases, duty cycle actually decreases slightly, our guess is that this is due to compiler quirks.

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 [8] [12] 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 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 in 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.

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 [8].

5.2.2 Interface specification ambiguities

If an interface is weakly specified, then some implementations take stronger checking approaches than others. This creates an ambiguity on 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 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 pre-computed CRC to be inconsistent, leading to a failed CRC check at the receiver.

However, 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 AM Promiscuous twice,
violation of a prior, stricter, version of the SendMsg contract. Because of the unstated checking in AMPromiscuous, this does not result in 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.

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 10 illustrate how widespread this behavior is.

In some instances, starting a component will begin a one-shot timer, but each time the start function is called the timer is reset, skewing the amount of time required for the first timer event. 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.

This problem has been recognized, and largely dealt with, in TinyOS 2.0 by separating the initialization and module start features 7.

5.3 Unexpected interface usage

An important result of our work is quantifying the expectations for an interface. In many situations we’ve 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, in case there wasn’t yet enough confusion. The result of all 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 own state.

We cannot in good conscience call this a contract violation, since the established convention is maintained throughout, but it is inefficient to say the least. Deeper examination of the implied usage requirements of commonly used interfaces will, in our opinion, 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 (without corrupting existing state), it is unlikely they would code it with those constraints in mind, which will introduce unexpected errors. We would hope that our contract checking tool would encourage more straightforward programming, but even in the case where the established interface is too firmly entrenched to be done away with we can enforce the proper usage conventions.

6. FUTURE WORK

Cross-contract checks. Our contracts are standalone units, and as such we cannot specify behavior between contracts. For example, should a buffer be simultaneously used by both a Send and Receive interface, which is a serious error, our contract checker will not detect a problem as long as each interface’s usage is correct. Unfortunately, contract behavior between separate interfaces is difficult to create rules about, because the interactions between the interfaces can be extremely difficult to specify. The main focus of the contract associated with Send and Receive is preventing concurrent accesses to a buffer that is being used by another module. Accesses to a buffer while it is ‘owned’ by other modules in the network stack are an error, since no guarantees about the behavior of the underlying modules can safely be made. A read of the buffer will return an undefined result, and a write to the buffer can corrupt data in use by the send subsystem. Because the network send stack operates through multiple layers of the various send and receive interfaces, in many instances the buffer will be owned by more than one module at once. In the Surge application, SurgeM sends messages using the MultiHopEngineM.Send interface. MultiHopEngineM in turn sends via QueuedSendM, and QueuedSendM sends via AMPromiscuous. So, when a message is sent from SurgeM, it will eventually be simultaneously ‘owned’ by all of these interfaces. The is allowable under the current contract, because as long as each module operates on the principle that no accesses can be made while it does not own the buffer, the ownership details of the lower-level send are irrelevant. Buffers are often owned by several Receive modules in the same fashion.

Our contracts are sufficient to enforce proper behavior when a stack of Sends or Receives are operating on the same buffer, but in some applications, notably ones that use the QueuedSendM module, buffers will migrate back and forth between the Send and Receive network stacks. This mixing of interfaces makes a firm specification of which interfaces are allowed to have concurrent ‘ownership’ of the buffer at a given point in the program complicated at best. Even though cross-contract usage of buffers is common in practice, without implementing comprehensive knowledge of the interactions between contracts, these kinds of errors are impossible to account for.

Contract requirements beyond command/event instrumentation. Currently, we make correctness checks by instrumenting the contract state transitions at the interface commands and events. Some contracts have more far-reaching implications than modeling the interface usage can provide. The Send interface implies that the network send stack owns a buffer for the period of time between a successful send() call and the corresponding sendDone() event. During this period any memory access to the send buffer by the top-level sending module is considered to be an error, since reading or writing a buffer it is being concurrently manipulated by an unknown sub-module will result in undefined behavior. To totally satisfy the contract we have to ensure that no access is made to a buffer that is being sent. At this point our contract language and instrument tool have no means of implementing
checks the code between interface calls.

**Optimizing dynamic checks.** When an interface is used in a parametrized fashion, we convert the single contract state variables into an array, and use the given parameter to access the appropriate state. We must make conservative assumptions regarding the values of the parameters used to index into this array, and as a result there is a substantial amount of wasted space. Switching over to a hash-based solution would help reduce our data overhead as the cost of an increase in duty cycle.

**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. A contract provides a set of assertions that is useful for automatically instrumenting a component for use by a static checker, such as BLAST. But what a contract also provides is a constrained usage pattern. If we assume that whatever component eventually makes use of the interface provided by our statically checked component conforms to its side of the contract, we can make assumptions about when and how the component in question is used, thereby placing limits on the state explosions that make static checking so difficult. Then, by statically ensuring that each module both satisfies its obligation both as a provider and user of interfaces, we can check the entire program piece by piece.

7. RELATED WORK

**Interface contracts.** Design by contract [9] 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 [16] is the most closely related sensorsnet 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 difference 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. [4] previously applied interface automata to TinyOS applications.

**Sympathy** [11] 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 [5], SoS [3], Virgil [13], and our type-safe version of TinyOS [12] all use language-based protection to avoid memory safety violations in sensornet applications. This kind of protection is perfectly 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 less 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. 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


