Static Verification of TinyOS via CQual

Samuel Madden
madden@cs.berkeley.edu

1 Introduction

TinyOS is an operating system for very small, wireless, battery powered sensors with small processors. An example of one of these devices, commonly called a 'mote', is shown in Figure 1. Table 1 gives the characteristics of the current and next generation of these devices: note the very limited amounts of RAM, slow processor speed, and limited radio bandwidth. TinyOS provides a convenient programming model for sending and receiving radio messages, sampling sensors, and accessing hardware such as the EEPROM and LEDs without learning and struggling through embedded assembly code. In addition to these hardware abstractions, it provides a novel event-based programming interface and execution model, which we will discuss in more detail in Section 2.

Despite the conveniences of TinyOS, it suffers from a severe limitation that is characteristic of many embedded programming environments: it is very hard to diagnose and fix bugs in programmed motes. This is due to a number of factors: lack of a display (aside from three LEDs), no debugger, physical inaccessibility of deployed nodes, asynchronous or sporadic nature of many bugs, and only rudimentary simulation facilities. The fact that TinyOS is written in C only aggravates these problems: lack of array bounds checking, a weak type system, pointers, and no synchronization primitives make runtime errors still more likely.

In this paper, we explore the use of the CQual type qualifier system [4] as a tool to reduce the number of runtime errors in TinyOS programs. CQual works by allow programmers to annotate C programs with type qualifiers: enhanced types that constrain the situations in which a variable may be used. The canonical type qualifier is the const keyword in standard C; const parameters may not be modified, and the compiler will issue a warning or error when a program attempts to do so. In the same way, CQual issues warnings when a qualified variable is used in an illegal way.

In the next Section, we present details of the TinyOS and CQual systems. In Section 3, we describe the

<table>
<thead>
<tr>
<th>Property</th>
<th>Current Motes</th>
<th>Next Generation Motes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radio</td>
<td>10 kbit/sec</td>
<td>50 kbit/sec</td>
</tr>
<tr>
<td>Processor</td>
<td>4MHz</td>
<td>4MHz</td>
</tr>
<tr>
<td>RAM</td>
<td>512 B</td>
<td>2 kB</td>
</tr>
<tr>
<td>EEPROM</td>
<td>32 kB</td>
<td>32 kB</td>
</tr>
<tr>
<td>Program ROM</td>
<td>8 kB</td>
<td>16 kB</td>
</tr>
</tbody>
</table>
specific type annotations we added to TinyOS, and in Section 4, we summarize the results of applying those annotations to a number of TinyOS programs. Finally, in Section 5, we offer directions for future work and conclude.

2 Background: TinyOS and CQual

In this section, we first summarize the architecture of TinyOS and the structure of some simple TinyOS programs. We then look at CQual and its facilities for annotating a program with type qualifiers.

2.1 TinyOS

Aside from the hardware abstractions which provide programmers with a convenient way to send and receive messages over the radio and sample data from sensors, the most significant feature of TinyOS is the event-based programming model it provides for developers. More information about TinyOS and the mote platform can be found in the TinyOS ASPLOS paper [3].

In this model, lowest-level events are generated by interrupts which occur when one of the ADC pins on the microprocessor flips from zero to one or one to zero. Interrupts trigger event handlers defined by TinyOS. These event handlers trigger higher level events that are intercepted by components, software abstraction layers similar to objects without inheritance. TinyOS defines some operating system components, and applications typically define a top-level component. Components handle events and, in turn, fire other events to invoke higher level components. In this way, components form a graph which events flow up. Figure 2 illustrates a simple component graph for a program called cnt_to_leds that displays a count from 0 to 7 as a binary number on the mote’s three LEDs. Notice the low level clock interrupts schedule the clock component, which in turn generates events used by the counter component, which forwards counts onto the main application, which lights the LEDs.

LED lighting is an example of control flowing from a higher level component to a lower level component; this is done via commands. Commands are used to invoke various functions of the OS, such as lighting LEDs or sending radio messages. Commands are frequently executed in an asynchronous split-phase model: a request to execute a particular command is passed down, and an event indicating completion of the event is later passed up.

---

1We will not discuss the specific APIs which TinyOS provides, as they are not relevant to this paper.

Figure 1: A TinyOS Sensor Mote
Figure 2: The cnt\_to\_leds application

So far, we have assumed all computation is initiated in response to interrupts. There is, however, another computation facility called tasks. Event handlers can enquire tasks to perform some more extensive computation asynchronously from the main execution path. This facility is necessary because events may be executed with interrupts disabled, precluding the operating system from processing further events – in particular, every bit from the radio is handled via a separate event handler; a 10kbit radio will generate ten thousand events per second, limiting the number of instructions which should be executed in response to any given event handler to about 400.\(^2\) Tasks are placed on a work queue and executed asynchronously, in order, by the operating system. Tasks may be interrupted by event handlers and may execute commands.

We omit any discussion of the programming model for TinyOS, except to say that components are specified in a simple scripting language which gives local names to events and specifies the local events in other components which correspond to this component’s events. These component files are translated into C header files, which are used to statically link together separate component files into a single TinyOS program.

2.2 CQual

We briefly summarize the salient aspects of CQual here. The reader is referred to any of a number of papers on the CQual system and its use in analyzing other software systems [1, 4, 2] for more information.

Data items in programs to be processed by CQual are annotated via type qualifiers. Type qualifiers are distinguished from primary types in C by preceding them with $\$” characters. These qualifiers are stripped out before the program is actually passed to the C compiler. CQual verifies that properly qualified variables are assigned, compared, and passed as parameters throughout a program. In addition to strict equality, qualifiers in CQual can follow a simple inheritance model described by a lattice. As an example of a lattice, consider the const keyword in C: non-const variables may be wherever const variables are used, but const variables may not be used in all places where non-const variables are legal, such as in assignments.

\(^2\)This is the standard rhetoric from the TinyOS project. In practice, very few events are executed with interrupts disabled, and many chains of event handlers span more than 400 instructions.
Each set of type annotations is described by a separate lattice, which CQual uses to determine the legality of type qualified statements.

One particularly interesting aspect of CQual is that it is flow sensitive, meaning that the qualifiers that apply to a particular variable can vary based upon the ordering of statements within a program. For instance, a lock variable may be tagged as $locked by a call to a locking function, and may be later unlocked by some other function. Flow sensitivity should be contrasted to flow insensitivity, where type annotations are static over an entire program, and path sensitivity where type qualifiers over a variable can vary based upon the execution path in which they appear. CQual is not path sensitive, which means that at any given instruction, a variable may have only a single qualifier in each lattice; in a path sensitive analysis, several type qualifiers could apply at each instruction, as different control paths could apply different qualifiers.

CQual is applied to pre-processed C files. It generates a list of places those files where type discipline is violated.

3 Static Checks

In this section, we describe the static checks we perform on TinyOS programs via CQual. Before describing them in detail, it is useful to examine some of the considerations made in the selection and implementation of these checks. First, we sought to reduce the number of annotations that TinyOS programmers would be required to make; although CQual annotations are intuitive additions which most C programmers should easily grasp, empirical evidence suggests that programmers will not generally adopt any tool which requires them to change the code they write. For this reason, each of the checks described below use syntactic information to infer type qualifiers. As we will describe, well-defined coding conventions in most TinyOS programs enable this inference. This inexact checking can lead to both false positives, since a programmer may write correct code that does not strictly follow coding conventions. We will report on situations where this occurred in Section 4 below, although a tool that tells programmers where they are violating coding conventions is also potentially useful.

A second consideration for each of the techniques described below deals with simulating program flow. Because CQual is flow sensitive, a main function which fires program events in some order is required to identify many type-qualifier related problems. For each of the examples in this work, we manually generated sequences of events which represented short, plausible executions of the applications. A more ambitious approach, which we chose not to investigate, would have involved generating random sequences of commands and events; this is more complicated, however, as some execution sequences are not legal. For instance, the INIT and START commands must be called before any other commands in a component.

We now discuss each of the four checks we implemented in detail.

3.1 Pending Flags

As discussed above, many operations in TinyOS are split-phase: requests for data are issued via a command, and the results of those requests are delivered asynchronously at some later time. Reading and writing from the radio and sampling sensors, which are arguably the most important functions of any TinyOS device, all
work in this manner. Because RAM on TinyOS devices is so limited, most split-phase operations allow only one outstanding request at a time: for instance, attempting to send a message over the radio while another message is being sent will result in garbage being transmitted, and should not be done. Unfortunately, this is not enforced by the radio component, so it is up to the application programmer to insure such a situation does not arise. This is typically done by setting a pending flag that indicates that the radio is in use, and unsetting that flag when transmission is complete. This is illustrated in the following simple example:

```c
char TOS_COMMAND.INT_TO_RFM_OUTPUT (int val) {  
    int.tos Led msg* message = (int.tos Led msg*)VAR(data).data;  
    if (!VAR(pending))  
    {  
        VAR(pending) = 1;  
        message->val = val;  
        message->src = TOS_LOCAL_ADDRESS;  
        if (TOS_COMMAND.INT_TO_RFM_SUBSEND_MSG (TOS.Broadcast, AM.Msg (INT_READING), &VAR(data)))  
        {  
            return 1;  
        }  
    }  
    return 0;  
}  
char TOS_EVENT.INT_TO_RFM_SUBSEND_DONE (TOS_MsgPtr sentBuffer) {  
    if (VAR(pending) && sentBuffer == &VAR(data))  
    {  
        VAR(pending) = 0;  
        TOS_SIGNAL_EVENT.INT_TO_RFM_COMPLETE (1);  
        return 1;  
    }  
    return 0;  
}
```

In this example, the variable pending is set to 1 when a message is sent, and set to 0 when a message is received. The use of the VAR(...) wrapper around some variables indicates that the variable is in this component's frame – essentially, a variable accessible to all other functions in this component.

We can use CQual to verify three properties of pending flags. First, we check that when a pending flag is set, it was definitely zero before being set, meaning no request was outstanding. Second, when a pending flag it cleared, we verify it was definitely one before being cleared. Finally, when a message send is imitated, we verify that a pending flag was set at some point before calling the send function – we cannot verify that it happened on the same path as the actual call, as CQual does not offer path sensitive analysis. However, this weaker property will still catch many potential bugs.

These property checks were implemented via a number of techniques. First, we observe that pending flags are almost always named 'xxx_pending' or just 'pending', such that it is easy to infer which variables are pending flags and which are not. We tag such variables with one of three values: $unknown, $inuse, or $free to indicate the current state of the pending flag. We also observe that a value of '0' means a request is not outstanding, while a value of '1' means a request is outstanding. Therefore, if we observe a pending flag being compared to 0 or 1 (or used as a boolean), we know whether or not a request is outstanding in the following statements. Thus, using the same trick used in [2], we can infer that for all statements within the true branch of a statement like if (!VAR(pending)), the pending flag is $free, no request is outstanding, and setting the pending flag is legal. Similarly, when we encounter a statement like VAR(pending) = 1, we can verify that before the statement, the flag had the $free property, and Afterwards we can assert that it is $inuse. Similarly, we can assert the opposite properties of variables when we encounter statements that compare pending flags to 1 and set them to 0.

When a call to initiate a message transfer is encountered, we can verify that, somewhere in this function,
a pending flag is used, and it was definitely marked as $inuse prior to this call being made. While this does not guarantee that a the pending flag actually protects the encountered message call, it does guarantee that a pending flag is $unused in the vicinity of this message call, and the the flag is properly checked and set before the message is transmitted.

### 3.2 Multiple Completion Callbacks

The second major problem we identified in TinyOS has to do with the completion callback for split phase operations. For any particular split phase operator (sending a message, for example), there can be several components which make use of the operation. Since the handler for a particular operation does not know which of those components invoked it, a handler must signal all components which can invoke it every time a completion event occurs. Components can tell whether the completion event belonged to them by comparing a pointer they passed in (usually the data to be sent or the buffer to be read into) to a pointer returned by the handler, as in the `sentBuffer == &VAR(data)` comparison in the code above.

A problem can arise if a component fails to check the value of this pointer: the callback may be for a different component, and thus the current component may take a completion action (e.g. transition to the next phase of an internal state machine) before it should. Some of these problems can be alleviated via static checking with CQual (although these checks involve no type annotations). When we notice a pending flag is being set to zero (indicating a completion event), we can check and see if the current function accepts only a single pointer variable. If it does, we verify that, in an `if` statement enclosing this setting of the pending flag, we compare the pointer parameter to some other pointer value. If the pending flag is not protected by a check of the pointer parameter, a warning is issued via CQual. Of course, the programmer may store the result of the check of the pointer value in a local variable, or check the pointer value via some expression we cannot statically verify, in which case, this approach will falsely detect a program error.

It is interesting to note that this multiple-callback property is a whole-program property, since it only arises in multiple-component situations. Without the whole-program analysis features of CQual, this check would not be feasible.

### 3.3 INIT and START commands

A third major problem has to do with initialization of components. Most components define an `INIT` and a `START` command, both of which must be called in order for the component to perform according to its specification. For example, the `timer`, which manages the hardware timer on the mote, will not start to time if its `INIT` command is not called (several TinyOS programmers reported experiencing this exact problem.)

CQual makes it very easy to check this property. During analysis, when a function called `xxx_START` or `xxx_INIT` is encountered, it is tagged as `$uncalled`. Then, when an invocation of such a function is encountered, the function is tagged as `$called`. Finally, after analysis, the list of all known functions is walked, and those that are `$uncalled` are reported as warnings.

Notice again that this is a whole program analysis: `START` and `INIT` functions are typically called from other components. Also note that the programmer is not required to specifically insert any annotations: TinyOS coding conventions allow us to infer the names of the initialization functions.
3.4 Resource Reservation
The fourth class of error in TinyOS has to do with multiple components reserving the same set of resources. For example, two components may both wish to use the hardware clock to periodically perform some action. Since the clock can only generate interrupts at one frequency, one of the components will not receive events at the frequency it expects, and aberrant behavior with result.

CQual can be used to prevent multiple components from attempting to initialize certain low level hardware resources. CQual is given a list of functions that initialize hardware that should be called only once within a given program. As it is analyzing a program, it tags all such functions as $called when they are invoked for the first time; if a $called function is invoked again, CQual generates a warning. For the purposes of this paper, we have only added support for clock-initialization checks to CQual; if other resources with a single reservable resource are added at a later date, than can easily be checked via this facility.3

4 Results
Having discussed the static checks we implemented in TinyOS with CQual, we now summarize the results we obtained by running CQual against a number of TinyOS applications. Table 2 shows the number of errors observed by type, and shows false positives. Table 3 briefly describes the test programs; they all compile and are a part of the official TinyOS release, meaning that their authors’ claim they function as advertised. In some cases, these programs have been deployed in embedded environments for days, weeks, or months. Because these are all tested programs, it is unlikely that CQual will discover deterministic bugs which always prevent the program from behaving as it should. Using CQual in conjunction with some programs that are in-development remains an area of future work. As we will see, however, CQual is able to diagnose a number of low severity problems with even these deployed programs.

In the next four subsections, we will discuss observations made about each class of error in detail.

4.1 Pending Errors
Pending errors were quite common in the code, and also resulted in the only false positives. There were several places where programs set the pending flag without first checking to be sure it was unset: as discussed above, if these programs were (for some reason) already transmitting when this occurs, Bad Things will result. In the case of MULTIHOPO_COUNT, a command can be issued to tell the network to stop counting. When this occurs, messages are sent without explicitly checking the pending flag; if a message is already in transit at this point, the stop message will not propagate properly. In the case of CHIRP, the message pending flag is checked before a request to sample the light sensor, but the sensor reading is transmitted and the pending flag is set without a check in the sensor reading completion event. This could lead to problems if two requests to sample the light sensor were issued before the first completed.

Several other cases involved the pending flag being set to zero without first checking that its value was one. Although such checks are not required if the completion event merely resets the pending flag, they are

3 It is expected, for instance, that the next generation of motes will support radio speeds of both 10kb/sec and 50kb/sec – a mixture of radio speeds will not be allowed within a single program.
Table 2: TinyOS Program Errors. Programs Analyzed with TinyOS. Number of Errors In Each Program by Category, with (False Positives).

<table>
<thead>
<tr>
<th>Program</th>
<th>Pending</th>
<th>Completion</th>
<th>INIT</th>
<th>Resource</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic_base</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>cnt_toleds</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prog_test</td>
<td>1 (1?)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>calib_test</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>multihop_count</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1 (1)</td>
<td>3</td>
</tr>
<tr>
<td>chirp</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>vibes_logger</td>
<td>1 (1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>bless_test</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>envmon</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>7 (2)</td>
<td>1</td>
<td>8</td>
<td>1 (1)</td>
<td>17 (3)</td>
</tr>
</tbody>
</table>

a: Two pending related errors were identified in generic_base which were false positives. CQual was fixed such that they were no longer reported.
b: A warning is issued in prog_test regarding a pending flag. Because the code is very convoluted, it is not clear if this is actually a bug.

a good idea in practice in case the completion code is made more complex in the future, since if the pending flag is not set, this probably indicates that the reported completion event is spurious (or that the entire program is broken.) In general, executing a spurious completion event handler could incorrectly modify program state.

Finally, there were two false positives. The first, in the PROG_TEST component, may not be a false positive, but the code is so complex that it is very difficult to tell. The variable i2c_pending is used to protect writes to both the i2c serial bus and the radio. A simple state machine keeps track of whether data is being read from the radio, written to the serial bus, or written to the radio, and the pending flag is forced to be one or zero in several places based on a state transitions rather than a send command or completion event. CQual cannot analyze this code properly, as it does not understand the behavior of the state machine. The second false positive (in VIBES_LOGGER) is slightly less complicated, but arises for a similar reason.

Consider the following code:

```c
struct MSG_VALS* VIBES_LOGGER_RX_REQUEST_EVENT (struct MSG_VALS* msgptr) {
    if(TOS_MY_Frame[toe_state.current_node].eprom_pending == 0) {
        TOS_MY_Frame[toe_state.current_node].eprom_pending =1;
        READ_LOG_COMMAND ( ... );
    }
    return msgptr;
}

char VIBES_LOGGER_MSG_SEND_DONE_EVENT (struct MSG_VALS* sent_msgptr)
{
    char *packet;
    if( TOS_MY_Frame[toe_state.current_node].send_pending != 0 & &
        sent_msgptr==& TOS_MY_Frame[toe_state.current_node].msg ) {
        TOS_MY_Frame[toe_state.current_node].send_pending =0;
        if (TOS_MY_Frame[toe_state.current_node].eprom_send_pending != 0 ) {
            TOS_MY_Frame[toe_state.current_node].eprom_send_pending =0;
            if ( AM_SEND_MSG_COMMAND ( ... ) ) {
                ...
                return 1;
            }
        }
        return 0;
    }
    char VIBES_LOGGER_READ_LOG_DONE_EVENT (char* packet, char success) {
```
Table 3: TinyOS Program Descriptions.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic_base</td>
<td>Forwards packets from the radio to the UART and packets from the UART to the radio.</td>
</tr>
<tr>
<td></td>
<td>Allows a PC to participate in a sensor network. Very well tested.</td>
</tr>
<tr>
<td>cnt_to_leds</td>
<td>Extremely simple program to count from zero to eight and back to zero repeatedly,</td>
</tr>
<tr>
<td></td>
<td>setting the LEDs to correspond to the current count.</td>
</tr>
<tr>
<td>prog_test</td>
<td>Program which accepts a new program over the radio, reprograms the mote.</td>
</tr>
<tr>
<td></td>
<td>Not well tested, and very complicated messaging protocol.</td>
</tr>
<tr>
<td>calib_test</td>
<td>Program which calibrates on-board sensors via a stored table in EEPROM.</td>
</tr>
<tr>
<td></td>
<td>Not well tested and algorithmically complicated, but with simple messaging behavior.</td>
</tr>
<tr>
<td>multihop_count</td>
<td>Program to count the number of sensors in a network in a distributed fashion.</td>
</tr>
<tr>
<td>chirp</td>
<td>Periodically transmits light reading over network.</td>
</tr>
<tr>
<td>vibes_logger</td>
<td>Log accelerometer readings to EEPROM, and transmit over radio.</td>
</tr>
<tr>
<td></td>
<td>Uses multiple pending flags.</td>
</tr>
<tr>
<td>bless_test</td>
<td>Beacon-LESS routing. Transmit messages across multiple network hops</td>
</tr>
<tr>
<td></td>
<td>from any node to the root of the network.</td>
</tr>
<tr>
<td>envmon</td>
<td>Periodically transmit light and temperature readings using multi-hop network.</td>
</tr>
<tr>
<td></td>
<td>Deployed for six weeks in Cory Hall.</td>
</tr>
</tbody>
</table>

```c
int i;
if(TOS_MY_Frame[os.state.current_node].send_pending == 0) {
    TOS_MY_Frame[os.state.current_node].send_pending = 1;
    TOS_MY_Frame[os.state.current_node].eprom_send_pending = 1;
    ...
    if(AMSEND_MSG_COMMAND (...)) {
        ...
    }
    return 1;
}
```

This program reads a line from the log, and sends two messages in response. The `send_pending` and `eprom_send_pending` form a state machine. When `send_pending` is zero and `eprom_send_pending` is one, the log is being read. When both are one, the first response message is being sent. When `send_pending` is one and `eprom_send_pending` is zero, the second message is being sent. This code is correct, although the use of the pending flags is somewhat misleading and confuses CQual’s deductions about the meanings of the flags.

4.2 Completion Callback Errors

Only the BLESS_TEST component reported a completion callback error. In this case, the pending flag was set to zero after a message send had completed without verifying that the message buffer actually was the one the program had attempted to send. It is unlikely that this would ever lead to incorrect behavior, as the message completion event does not directly initiate any further computation. However, if this handler were later expanded, this missing check could cause problems.

4.3 INIT/START Errors

Missing calls to INIT/START functions were quite common among the sample programs. In none of these cases did these missing calls actually affect the correctness of the current version of the program, because none of functions who’s calls were missing performed any computation. However, these functions could be
changed in future versions of TinyOS, at which point these programs would fail to function properly. Five of the eight errors were missing INIT calls into the LED component, which sets and unsets the LEDs.

4.4 Resource Reservation Errors
Only a single resource reservation warning was issued across all programs, and it was a false positive. `multihop_count` stops the clock in several places in order to reduce the number of events which are fired and save power; this is perfectly legal, because it is in exclusive control of the clock. This points out a flaw in this test: because the `MULTIHOP` component is the sole user of the clock, it should be able to change clock rate indiscriminately; however, there is no definite way for CQual to effectively determine the component that owns a particular resource.

4.5 Performance
Performance of CQual was excellent for all examples. The largest program, `multihop_count`, weighed in at 15,833 lines of C code, including many copies of header files in the preprocessed code. It took under 3 seconds to process on a 933 Mhz Pentium III with 512MB of RAM. We do not report further performance results because there are no performance concerns in CQual.

4.6 Observations
CQual proved useful in identifying a number of bugs in the test suite of TinyOS programs. Most of these bugs were identified quickly and specifically; there were several cases, however, where CQual failed to clearly or easily locate bugs in programs. In particular, several of the pending errors were extremely unclear, largely because CQual often issues its warnings at the place where calls to incorrectly typed functions occur (since that's where type unification occurs), rather than at the mistyped function. Errors in `chirp`, for example, are stated as occurring in the low level radio component, while they in fact occur in the application specific handler for radio completion events. When used in PAM mode, it is easier to track down such problems, but since CQual does not support PAM mode for multiple file analysis, this was not an option for many of the bugs encountered here.

There were also several cases where the checks discussed in this paper lead to false positives; in some cases, we were able to fix CQual to no longer report bugs, while others will continue to be problems in the future. In general, false positives resulted from our assumptions about conventions in TinyOS which not all programmers followed exactly; in particular, pending flags are not set and checked religiously, especially when there are other ways, such as transitioning into or out of some internal state, to guarantee an event has or has not completed.

Finally, it is worth considering whether any of the problems identified by CQual are 'real' bugs which will lead to incorrect program behavior. As previously mentioned, it is unsurprising that none of the bugs lead to obviously incorrect behavior since these were all deployed, tested programs. Some of the pending bugs, such as the ones identified in `multihop_count`, could lead to real errors when timing conspires to execute an unlikely ordering of events. The remaining pending bugs, and the START/INIT and completion callback errors need to be fixed to improve code quality and reduce the potential for future bugs. Uncalled
INIT functions which currently serve no purpose could later be defined, as could the actions in message completion routines protected by pending and message buffer checks. Even the so-called 'false-positives' discussed above identify places in the code where TinyOS conventions are not followed strictly, leading to confusing, hard to maintain code.

5 Conclusions

We have successfully demonstrated that CQual can be used to statically identify a number of problems in TinyOS programs. All of these problems are identified by inferring type properties of programs via syntactic analysis, rather than requiring programmers to explicitly annotate the types of their programs. We were able to identify 17 errors in 9 programs, two of which were false positives. Although none of the remaining errors were obviously incorrect, several could lead to spurious behavior and all could lead to bugs in future program revisions.

Overall, the application of CQual to TinyOS proved to be straightforward and successful. Our modified CQual tool is currently being integrated as a static checker into the TinyOS build process to improve code quality.

References


