PRO 09 Virgil: towards certified sensor nodes

PRO 09.1 Overview

A medical device should not crash or confuse. A device crash can be anything from inconvenient to life threatening, while confusing device behavior can lead a user to draw an incorrect medical conclusion. We envision a certification tool that can meet challenges

related to space bounds, soft-real-time response, life time, and meaningful results. We aim for both fundamental advances in programming language design and static error checking, as well as progress on how to do applications programming for medical monitoring devices. Our goal is to take a major step towards design for certifiability and to bring closer the day when the FDA will use static error checking tools frequently and routinely. A medical monitoring device collects data from the body, carries out local computation, and sends data to an external computer. Together, the device and the external computer form a small sensor network with a few sensors and one base station. We have an NSF-funded collaboration with Majid Sarrafzadeh at UCLA whose group has built software for four monitoring devices that were then tested by doctors and patients in the UCLA Medical School. Majid's devices monitor such things as pressure changes in the upper urinary tract, myotatic stretch reflex, neurological disorder, and diabetic foot ulcer (see Figure 1). The devices are small and the software is typically on the order of a few thousand lines of source code. Eventually we want to be able to certify the software in Majid's four devices.

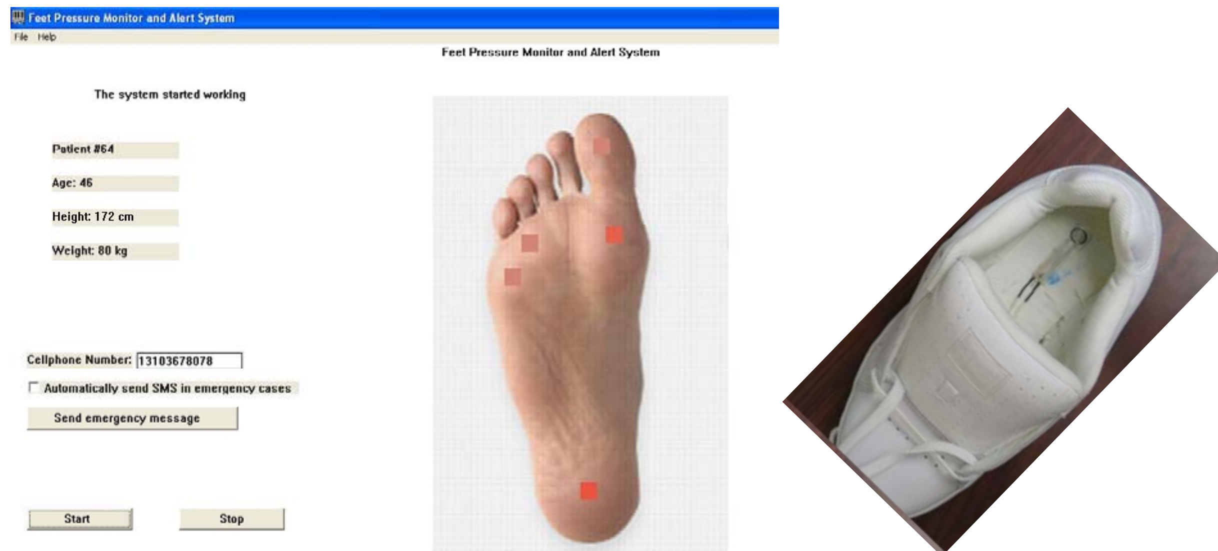

Figure 1. A device for monitoring diabetic foot ulcer.

PRO 09.2 Approach

Our goal is to develop

- a domain-specific language that will encourage design for certifiability and make certification easier, and
- domain-specific tools for certifying the four key properties of space bounds, soft-real-time response, life time, and meaningful results.

In particular, we want the tools to do static error checking, that is, certify the software without running the software. The certification tools will guarantee that certain problems cannot occur; and rigorous testing can then focus on problems that were left unaddressed by the certification tools. The certification tools will increase our confidence in medical devices and help decrease the scope, duration, and cost of the testing effort.

PRO 09.3 System Description and Experiments

Our own language Virgil is our starting point for designing a new domain-specific language. Virgil is a statically-typed, object-oriented language in the tradition of C++, Java, and C#. Virgil is designed specifically for high-level, type-safe systems programming, including programming of device drivers for sensors, radios, timers, analog-to-digital converters, etc., and is targeted to run on tiny devices such as sensor nodes.

PRO 09.4 Accomplishments
We have written drivers in Virgil for all the Mica2 devices. The final driver that we completed was the radio device driver which unsurprisingly turned out to be a major challenge. As a result, we now have the entire base functionality of TinyOS written in Virgil. Software can now control a sensor node using Virgil code alone. Additionally, we have rewritten one of our medical device software applications into Virgil. This has provided us with an excellent benchmark that we use in our day-to-day research. We have also studied approaches to extending Virgil from being a language for programming single nodes to become a language for programming an entire network of sensor nodes. We are focusing on X10, an object-oriented language from IBM. The X10 language contains two key constructs for concurrent programming called async and finish that we believe can be valuable for programming an entire sensor network.

We have made major progress on verification of Virgil programs. We have built a tool that maps a Virgil program to a timed automaton. We then use the UPPAAL real-time model checker to check properties of the timed automaton. Our timed automaton represents all program variables via transitions in the automaton. We represent timings of most operations, except machine level timings of such things as stack manipulation, heap allocation, etc. We don't support recursion and currently we handle procedure calls by duplicating the subautomaton for the callee.

Our benchmark Virgil program has 1094 lines of code, and the derived timed automaton has 628 states.

We have had great success with checking low-level properties

that are usually difficult to reason about, including:

- Atomic access to the ADC converter? We can catch nonatomic accesses, that is, make sure that consecutive calls to the ADC converter don't overlap. If there is access to the ADC converter when the ADC converter is active, then the automaton will move to an error state. The model checker verifies that there is no circumstance under which that could happen. If we change the configuration of the hardware timer such that it starts the ADC converter too often, then we will catch the error.
- Might we drop a sensor reading? In other words, is the timer period for controlling input from the sensor too short? We can catch such problems.

We have also made progress on checking stack overflow and on determining the worst-case time to execute the ADC converter from beginning to end.

PRO 09.5 Future Directions
We will continue our work on porting the software in Majid’s four devices from NesC to Virgil.

We have started work on directed testing for Virgil programs that will enable us to find worst-case event sequences that stress test an application.