An Extensible Approach to Verification of Embedded Network Systems*

Daniel Welch
School of Computing, Clemson University
Clemson, South Carolina, 29634
{dtwelch}@clemson.edu

Takumi Bolte
School of Computing, Clemson University
Clemson, South Carolina, 29634
{tbolte}@clemson.edu

ABSTRACT

In this paper we present a flexible means of specifying and verifying the correctness of software for embedded network systems. Our approach uses RESOLVE, an imperative, component based programming and mathematical specification language, to verify the functional correctness of embedded applications. In doing so, we enrich the work originally presented in [1] with the following: A model view controller (MVC) based implementation of a RESOLVE to C translator, a dynamic memory allocation scheme tailored towards embedded systems running the code generated by our tool, and the addition of a new language keyword that enables users to pair custom RESOLVE specifications with 'externally' implemented (non-native RESOLVE) realizations. We demonstrate these additions on an LED (Light emitting diode) driver that showcases recent mathematical developments, as well as formal verification of a toggling capability enhancement that we demonstrate running on a Telos mote.

Categories and Subject Descriptors

D.2.8 [**Software Engineering and Data Communication**]: Verification—*VCs, automated proving, modular software*

General Terms

Reliability, Verification, Languages, Networks

Keywords

automation, components, formal methods, specification, verifying compiler, embedded networks, wireless sensing

1. INTRODUCTION

Within little more than a decade, the area of embedded network systems and wireless sensing has exploded in popularity within industry and academia alike. Tempering however this extremely quick rise in popularity is the inherent difficulties in developing applications that function as intended in low power, event-driven environments. In response to these difficulties, a variety of tools and languages have been put forth to help ease the burden on developers. On one end of this effort are languages such as NesC, (Network embedded sensor C) which strive to minimize concurrency issues and other common sources of error by hiding libraries of pre-written drivers underneath hierarchies of software and interface level abstractions. The other end of this effort is largely comprised of simulation tools such as TOSSIM, Cooja, and Arora that make use of high-fidelity simulations to model networks offline in controlled, repeatable environments. Though these and other tools have indeed proven invaluable in allowing users to test and reason about event-driven code prior to deployment, they remain incapable of providing complete assurance that code will behave as expected when deployed in the field.

We approach this problem by using RESOLVE (Reusable SOftware Language with VErification) as means of authoring, specifying, and ultimately verifying code for embedded network systems. Our decision to use RESOLVE as a language frontend – as opposed to verifying C code directly – ultimately stems from a verification amenability standpoint: Not only does RESOLVE prohibit verification crippling operations such as uncontrolled referencing and aliasing (prevalent in C and many other current languages)[2], but also embodies a number of other characteristics ideal for embedded platforms including:

- **Modularity** RESOLVE enforces a strict separation of concerns between module specifications and client implementations. As a result, for any one particular specification, there can be any number of interchangeable implementations. This separation is ideal considering that many embedded applications happen to fit this pattern nicely: Various drivers oftentimes provide a common set of functionality, but in general have many distinct implementations that vary arbitrarily from platform to platform, vendor to vendor.
- **Mathematical flexibility** RESOLVE offers a rich, mathematical type system that allows users to either draw from a library of preexisting mathematical units when writing specifications, or simply create their own. This is ideal in a setting where drivers might encompass a wide spectrum possible applications, where each might require unique, new mathematical developments.

The paper is organized as follows: First, we open with a brief overview of the Telos mote platform. Next, we present

Clemson 2014 7th Clemson University Mini-Conference on Embedded Network Systems

revised specifications of an LED driver component with a formally verified enhancement. This section is concluded with a review of verification results, and a discussion of any relevant theorems and verification conditions (VCs) used. The remainder of the paper is spent detailing the model of C code generated by the tool, giving a quick overview of generation process itself, and detailing a dynamic, stack-based memory allocation tool utilized by the translated code. We conclude with suggestions for tool improvements, as well as a review of some longer term research goals.

2. THE TELOS PLATFORM

The Telos mote [3] is a programmable, low power, wireless sensing device developed at UC Berkeley. Its hardware includes an msp430 microcontroller containing 128 bytes of RAM and 10kB of programmable flash memory. A cc2420 radio stack provides the mote with broadcast and receiving capabilities, while optional sensing capabilities may be added in the form of light, humidity, and temperature sensors. The mote also contains three onboard LEDs: One blue, one red, and one green.

3. SPECIFIYING LED BEHAVIOR IN RESOLVE

In this section, we provide mathematical and programmatic elements of an LED strip component to give readers both a concrete look at the language, and introduce some new features aimed at making it more amenable to the development of embedded applications. Note that while we provide the level of detail necessary to understand the current example, readers interested in gaining a more complete, in depth knowledge of the language are encouraged to refer to [4, 5].

3.1 Concepts

In RESOLVE, programs are composed of several different modules that range from interfaces and realizations, to client (facility) modules. A *concept* module in RESOLVE defines a specification for a mathematical, abstract type. Similar to an interface in Java, a concept provides a number of operation signatures that implementors are expected to realize. Shown below is a `LED_Template` concept that provides a light strip abstraction.

```
Concept LED_Template(eval Strip_Length: Integer);
    uses Boolean_Theory, String_Theory;
    requires 0 < Strip_Length <= 4;

    Type Family LED is modeled by Str(B);
        exemplar L;
        constraint |L| = Strip_Length;
        initialization ensures
            ...

    Oper Set(updates L : LED; eval b : Boolean;
                              eval i : Integer);
        requires 0 <= i < |L|;
        ensures |L| = |#L| and
                  Element_At(i, L) = b;

    Oper Status(preserves L : LED;
                  eval i : Integer) : Boolean;
        requires 0 <= i < |L|;
        ensures Status = Element_At(i, L);

end LEDs_Template;
```

We model our conceptual LED strip using a mathematical string (finite sequence) of booleans, denoted `Str(B)`, where each boolean within the string indicates the status of that particular LED: On (true) or off (false). The `exemplar` clause located immediately below provides a handle to this abstract model, and is used throughout the remainder of the specification.

It's worth noting that unlike the `LED_Template` presented in [1] which models an LED as the *cartesian product* of booleans b_0, b_1, ..., b_4, the strip model we present here instead uses strings for the following reasons:

- Strings are indexible, and thus do not require separate `Set` and `Status` operations for each individual LED.
- This approach demonstrates the benefits of reusable mathematical theories. The specifications listed here are based (almost) entirely in `String_Theory` and are therefore able to make use of RESOLVE's pre-existing math libraries.

Finally, the concept provides two operations. The first, `Set`, takes as a parameter an instance of an LED strip `L`, a boolean `b`, and an integer `i`. The operation `requires` that `i` falls within the length of the strip, and `ensures` two things upon completion: The length of the outgoing strip `L` is the same as the incoming strip, `#L`, and that the LED in position `i` of `L` is set to boolean `b`. The `Status` operation is specified similarly.

3.2 Enhancements

RESOLVE also allows users to extend the functionality provided by the base concept through *enhancements* – a form of specification inheritance. The enhancement we provide here, `Toggling_Capability`, allows users to flip a specific LED to its complement.

Shown below is a specification for `Toggling_Capability` and one particular realization of it.

```
Enhancement Toggling_Capability for LEDs_Template;

    Oper Toggle(upd L : LED; eval i : Integer);
        requires 0 <= i < |L|;
        ensures Element_At(i, L) =
                        not(Element_At(i, #L));
end Toggling_Capability;

Realization Toggling_Realiz for
            Toggling_Capability of LEDs_Template;

    Proc Toggle(upd L : LED; eval i : Integer);
        Var Content : Boolean;

        Content := Status(L, Replica(i));
        Set(L, Not(Content), Replica(i));
    end Toggle;

end Toggling_Realiz;
```

The enhancement specifies a single operation, `Toggle`, which states that upon termination, the LED located at position `i`

in L is the complement of that same location in the incoming LED, #L.

Note that the enhancement specifications themselves look and function largely the same as a normal concept: Each specifies a purely conceptual module, and hence is implementation neutral.

Enhancement realizations are neutral as well since any method called within the context of an enhancement realization refers to the operation specified in the concept – meaning no knowledge of implementation details is required.

3.3 Verification

We now turn to the task of verifying our small toggling enhancement. The first step in doing so is to generate Verification Conditions (VCs) for Toggle_Realiz, which, if proven, will establish the correctness of this particular implementation. One thing to note about the VCs themselves is that they are generated from specific lines of a realization, and exist to ensure that the content of the realization is consistent with its specification: This entails checking for things such as array access boundary violations, etc.

Condition #	Time (ms)	Steps	Search
VC 0_1	4426	5	0
VC 0_2	5039	5	0
VC 0_3	6324	6	0

Figure 1: Verification results for operation Toggle

As the results summarized in Figure 1 indicate, using RESOLVE's integrated prover, we are able to mechanically and automatically dispatch all VCs for Toggling_Realiz, thus verifying its correctness. In terms of proof difficulty, given the number of steps and time taken to establish each, we conclude that the VCs generated were of a straightforward variety. Readers interested however in learning more about the steps and specific actions the prover takes in transforming givens and dispatching similar (and more complex) VCs should refer to [**?**].

3.4 Facilities

With our formally specified LED strip component in place – and a verified enhancement on this component – we now turn to a small embedded application that combines these elements to iteratively toggle the lights within an LED strip.

Shown below is a RESOLVE facility module that implements the client logic of this embedded application.

```
Facility LED_Telos_Demo;
    uses Std_Clock_Fac, Std_Boolean_Fac,
            Std_Integer_Fac;

    Facility Leds_Fac is LED_Template(3)
        externally realized by Std_Led_Realiz
     enhanced by Toggling_Capability
        realized by Toggling_Realiz;

    Operation Main(); Procedure

        (* Declare LED strip indices *)
        Var I1, I2, I3 : Integer;
        Var Loop : Boolean;

        (* Declare an LED strip *)
        Var L : Led;

        I1 := 1; I2 := 2; I3 := 3;

        Loop := True();
        While(Loop)
            changing Loop;
            maintaining ...
        do
            Leds_Fac.Toggle(L, I1);
            Std_Clock_Fac.Wait_500_Milli_Seconds();

            Leds_Fac.Toggle(L, I2);
            Std_Clock_Fac.Wait_500_Milli_Seconds();

            Leds_Fac.Toggle(L, I3);
            Std_Clock_Fac.Wait_500_Milli_Seconds();
        end;
    end Main;

end LED_Telos_Demo;
```

Prior to using the LED component developed in the previous sections, we first must bind our LED_Template specification with an appropriate realization. This is accomplished via the facility declaration located directly beneath the uses clause, which pairs the specification (LED_Template) with a realization (Std_Led_Realiz). Note that the enhanced ability of toggling lights is added *on top* of this facility declaration in a similar fashion. The remaining bulk of logic driving the application rests in the non terminating busy loop inside operation Main, where we use our enhancement-provided Toggle operation to successively turn each light within the strip on, then off.

3.5 "External" Realization Support

Readers might note that we never presented a realization of LED_Template. Indeed, after having written the concept, the RESOLVE programmer would ideally provide it with a verifiable, native RESOLVE implementation. However, as our target platform is embedded, and our concept aims to provide control for LEDs – a decidedly low level feature on embedded hardware – our realization is forced to operate at similarly low levels by directly manipulating hardware pins provided by msp430's chipset.

RESOLVE, however, in its current state is too high level of a language to perform these tasks directly – meaning it lacks the appropriate driver and language support to do so. In an effort to address this, we introduce the notion of *external realizations*, which allow users to write their own realization of a concept in a language of their choosing.

The LED_Telos_Demo facility above demonstrates these developments through its use of the "externally realized" phrase. This signals to the RESOLVE compiler that the user is providing a non-native realization of the LED_Template, with the expectation that it conforms to the specifications dictated in the concept.

We feel this new keyword is beneficial for the following reasons:

- The language no longer must "hide" the fact that some of the lower level components relied upon are not written in straight-line, native RESOLVE code. The "ex-

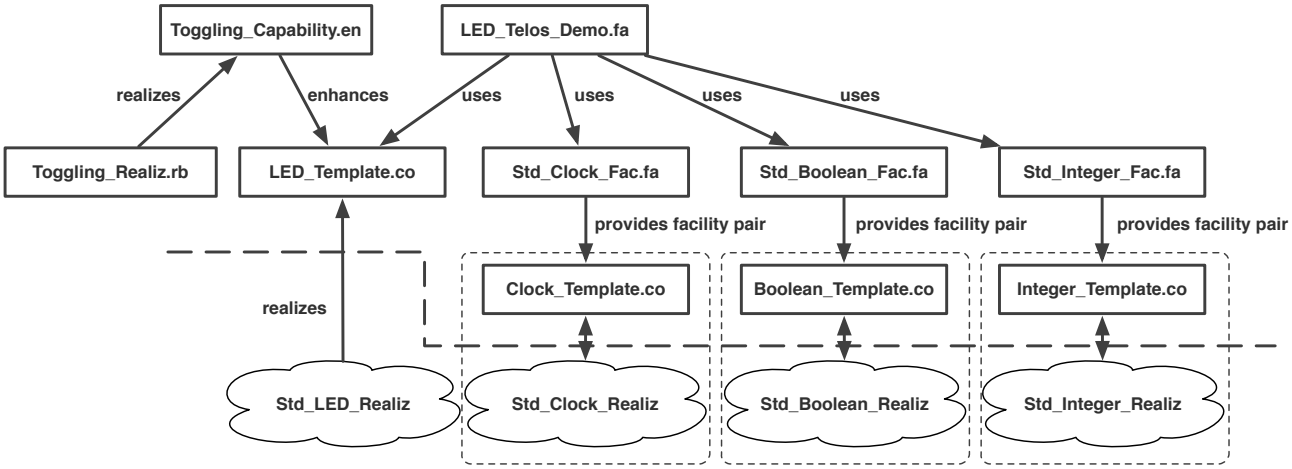

Figure 2: A diagram illustrating inter-component relationships for the example in Section ??

ternally" keyword now transparently indicates this.

- Provides flexibility for those users looking to wrap their low-level programs/drivers with formal RESOLVE interface specifications.

In the terms of embedded systems, these developments are especially as it allows us to write custom low level of the sort required for most embedded applications (e.g. LED strip drivers) while still providing formally specified interfaces that might later form the foundation . It is our hope and intention that new (native) resolve components will be layered on top of these low level, externally realized (yet specified) drivers – eventually reaching a level of abstract where we can concern ourselves exclusively with verified, native RESOLVE components.

4. IMPLEMENTATION

Development of our C translation tool can be logically partitioned into three distinct phases:

1. Arriving at a translation model (or, strategy) for an accurate C representation of RESOLVE.
2. Implementing reusable mechanisms for carrying out the C code generation process.
3. Creation of a memory manager capable of safely allocating and freeing dynamic memory required by the generated code.

We illustrate each of these phases working in tandem on the LED component discussed in Section 3.

4.1 C Translation Model

One of the primary challenges in translating from RESOLVE to C is finding a suitable C analog for each RESOLVE module and the constructs allowable in each. Indeed, since we are dealing with an environment where functional correctness is a primary concern, it is important that the code generated by our tool represents as closely as possible the original RESOLVE source. In an effort to make such considerations, at the highest level, the C code we generate makes special considerations for facilities, concepts, and realizations.

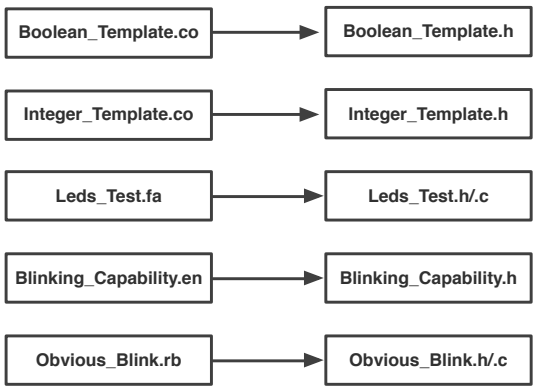

Figure 3: Relationship between RESOLVE module types and the C code generated from each.

4.1.1 Concepts

Concepts produce a single .h that provide function pointers for each operation specified by the original concept, as well as structs for each user defined type.

4.1.2 Facilities

Facilities produce an .h/.c pair: The header .h declares methods that are intended to manage the creation and destruction of all global variables used within the facility, while the .c provides an implementation of these methods. Note that the create and destroy methods are only responsible for freeing *global* variables, all other translated, local functions are responsible for deallocating their own variables.

4.1.3 Realizations

We treat realizations of concepts and enhancements slightly different than facility modules. While a .h/.c pair is still produced, the create method for realizations is designed to create instances of all types specified by the concept, while the destroy method deallocates these types.

4.2 Translator Implementation

Translation itself is performed over the course of a traversal of RESOLVE's abstract syntax tree (AST). The traversal mechanism used is a derivative of the visitor patter that provides a pre post traversal over all nodes in the tree.

4.3 Memory Allocation

Memory limitations on embedded hardware has made dynamic memory allocation a difficult, or impossible practice. Many existing programs, including RESOLVE, however, inherently use heap-based memory. Previous attempts at a RESOLVE to C translation required a constant size for each variable declaration[1]. This approach can provide efficient use of memory for embedded systems, it does not, however, provide a straight forward approach to translation. In this section, we introduce a RESOLVE wrapped dynamic memory allocator implemented on the stack.

4.3.1 Allocation using `salloc`

The function `salloc()`, uses a first fit approach for allocation on the stack, rather than conventional, heap-based allocators. This approach requires that a fixed size of memory is chosen at compilation. In addition, `salloc()`, requires a section of meta-data, which is denoted as a `block`, for each record in the memory pool. A `block` holds referential information of neighboring blocks, the size of the record the block maintains, and if the block is free or is in use. Figure 4 is an example of the stack after several calls to `salloc()`.

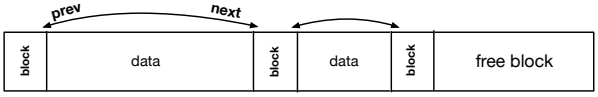

Figure 4: Representation of memory using salloc()
Memory allocated with salloc() allocates a block in front of data allocated. Blocks point to their immediate neighbors and hold their size and usage.

4.3.2 Deallocation using `sfree`

A memory allocator must provide a mechanism to release, or free memory in order to indicate that it is not being used and can be reallocated. The `sfree()` function provides deallocation for memory allocated with `salloc()`. Figure 5 shows an example set of memory before and after calling `sfree()`.

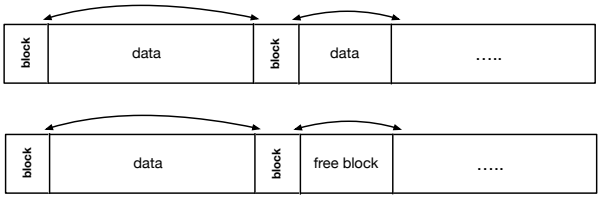

Figure 5: sfree to free memory

4.3.3 Optimizing Memory Usage

A common problem that can occur in memory allocation is fragmentation. During execution, a program can allocate and deallocate an arbitrary number of times. Figure 6 shows problems that result from this. This problem is magnified on embedded systems due to their limited memory capacities. There are simple optimizations that can reduce fragmentation.

Block splitting, as shown in Figure 7, occurs during allocation. It allows blocks of greater size to be partitioned into the size requested by the allocator.

Fusing blocks is another technique used when `sfree()` is called. When memory is deallocated, neighboring free blocks are coalesced to form a single large block, as shown in Figure 8.

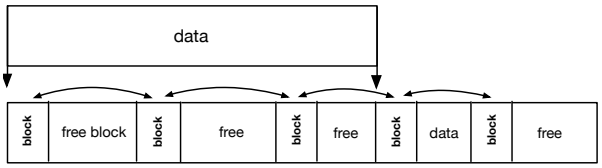

Figure 6: fragmentation
Performing many naive allocation and deallocations, allocatable blocks are restricted to those that match the exact size of memory requested from the allocator.

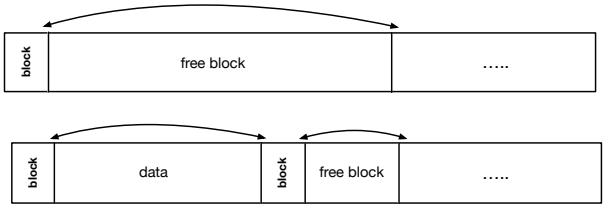

Figure 7: block splitting
Splitting of a free block creates two new blocks: a block that is the size that the allocator requested, and the other contains the remainder of a new free block with the remainder of memory.

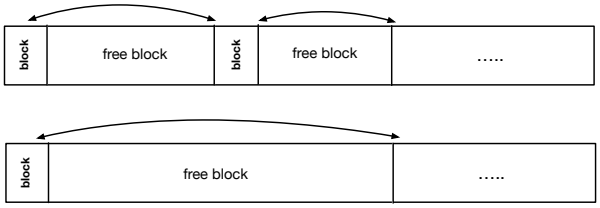

Figure 8: block fusing
An example of fusing blocks together. This creates single larger block, decreasing the number of smaller blocks unable to split.

5. LED DEMO

6. ACKNOWLEDGEMENTS

Special thanks to Mike Kabanni, Mark Todd, and Bruce Weide whose suggestions and initial contributions in constructing the current model of RESOLVE to C translation made this work possible.

```
/*
 * Generated by the RESOLVE to C translator.
 * This file should not be modified.
 */
#ifndef __LEDS_TEST_H
#define __LEDS_TEST_H

#include ".../RESOLVE.h"
#include ".../Facilities/.../Std_Boolean_Fac.h"
#include ".../Facilities/.../Std_Integer_Fac.h"
#include ".../Facilities/.../Std_Clock_Fac.h"
#include ".../Concepts/.../Leds_Template.h"
#include ".../Concepts/.../Std_Leds_Realiz.h"
#include ".../Concepts/.../Toggling_Realiz.h"

typedef struct Led_Facility Led_Facility;
struct Led_Facility {
    Leds_Template* core;
    Toggling_Capability_for_Leds_Template*
                      Toggling_Capability;
};
Led_Facility Led_Facility_Var;
void Leds_Test_create();
void Leds_Test_destroy();

#endif
```

Figure 9: LED_Demo.h

```
/*
 * Generated by the RESOLVE to C translator.
 * This file should not be modified.
 */
#include "Leds_Test.h"
void Leds_Test_create() {
  Std_Boolean_Fac_create();
  Std_Integer_Fac_create();
  Std_Clock_Fac_create();
  r_type_ptr __arg_0 = Std_Integer_Fac_Var.core->
  createFromInteger(4, Std_Integer_Fac_Var.core->Integer);
  Led_Facility_Var.core =
  new_Std_Leds_Realiz_for_Leds_Template(__arg_0);
  Led_Facility_Var.Toggling_Capability =
  new_Toggling_Realiz_for_Toggling_Capability_of_LED_Template(
      Led_Facility_Var.core);
  Std_Integer_Fac_Var.core->Integer->
  destroy(__arg_0, Std_Integer_Fac_Var.core->Integer);
}

void Leds_Test_destroy() {
    free_Std_Leds_Realiz_for_Leds_Template(
         Led_Facility_Var.core);
    Std_Boolean_Fac_destroy();
    Std_Integer_Fac_destroy();
}
```

Figure 10: LED_Test.c

7. REFERENCES

[1] Kalyan Regula. A verifying compiler for embedded networked systems. Master's thesis, Clemson University, 2010.

[2] Gregory W. Kulczycki. *Direct Reasoning*. PhD thesis, Clemson University, 2004.

[3] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 364–369, April 2005.

[4] Murali Sitaraman, Bruce Adcock, Jeremy Avigad, Derek Bronish, et al. Building a push-button RESOLVE verifier: Progress and challenges. *Form. Asp. Comput.*, 23(5):607–626, September 2011.

[5] Gregory Kulczycki, Murali Sitaraman, Kimberly Roche, and Nighat Yasmin. Formal specification. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008.

[6] http://www.ti.com/lit/ds/symlink/msp430f1611.pdf.