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 ideal in a setting where various driver applications are varied and new contexts might require creating new theory constituents appropriate for specific drivers.

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 RESOLVE, programs are composed of several different modules that range from interfaces and realizations, to client (facility) modules. 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

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_1, b_2, ..., 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 rooted in `String_Theory`, a pre-existing RESOLVE math library that provides most of the definitions and theorems used throughout the specifications.

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` upon completion that 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 adds 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.

Even enhancement realizations are neutral since only the operations called are those provided by the base concept – and require no knowledge of how the called operation is implemented.

3.3 Verification

In this section we give a short overview of the verification results of Toggling_Realiz. The first step in proving this particular realization correct is to generate verification conditions (VCs), which, if proven, will establish the correctness of this particular realization[1].

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 may conclude that the VCs generated were of a straightforward variety. Readers interested however in learning more about how the prover goes about transforming and dispatching similar (and other, more complex) VCs should refer to [6].

3.4 Facilities

With our formally specified LED strip component in place – and a verified enhancement on this component – we now present a sample embedded application that iteratively toggles all lights in a single strip.

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

```
Facility LED_Telos_Demo;
    uses Std_Clock_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 must first pair our LED_Template specification with an appropriate realization. This is accomplished via the facility declaration located directly beneath the uses clause which pairs our LED specification, LED_Template, with a "Std_Led_Realiz" realization. In order to give LED strip instances the ability to toggle specific LEDs, we also similarly pair the specification, Toggling_Capability with our Toggling_Realiz realization.

The bulk of logic driving the actual application rests in the non terminating busy loop inside operation Main, where we use the operation provided by Toggling_Capability to turn each light on the strip on then off in succession.

3.4.1 *"External" Realization Support*

Readers might note that we never presented a realization of LED_Template. Indeed, the RESOLVE programmer would ideally, after writing the concept, provide 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 manually interfacing with hardware pins provided by msp430 chipset.

RESOLVE, however, in its current state is far too high level of a language to perform these tasks directly – that is, it currently lacks the driver support to do so. In an effort to address this difficulty, we introduce the notion of *external realizations*, which allows the user to write their own realization of a concept in a language of their choosing. In the case of LED_Template, we chose to provide a hand written Std_Leds_Realiz C realization that matches as closely as possible the conventions and model of translation later detailed in Section #. An example above shows how users communicate to the compiler that a realization is external.

We feel this addition to the language is appropriate and useful for the following reasons:

4. IMPLEMENTATION

Development of the tool presented here can be logically partitioned into three distinct phases:

[1]Note: VCs themselves are typically generated from specific lines of a realization to ensure that the overall content of a realization is consistent with its specification.

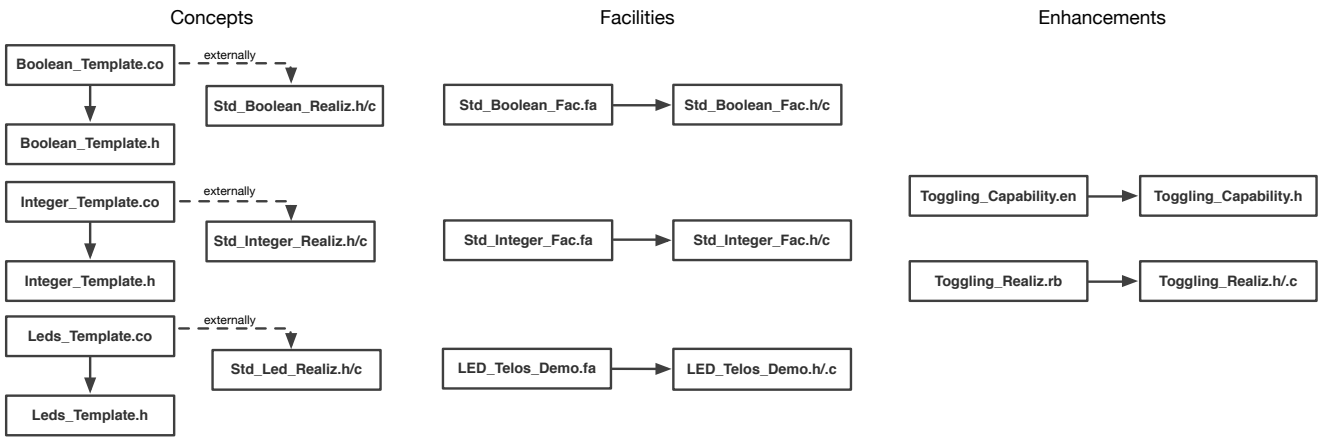

Figure 2: RESOLVE module types and their corresponding C representation

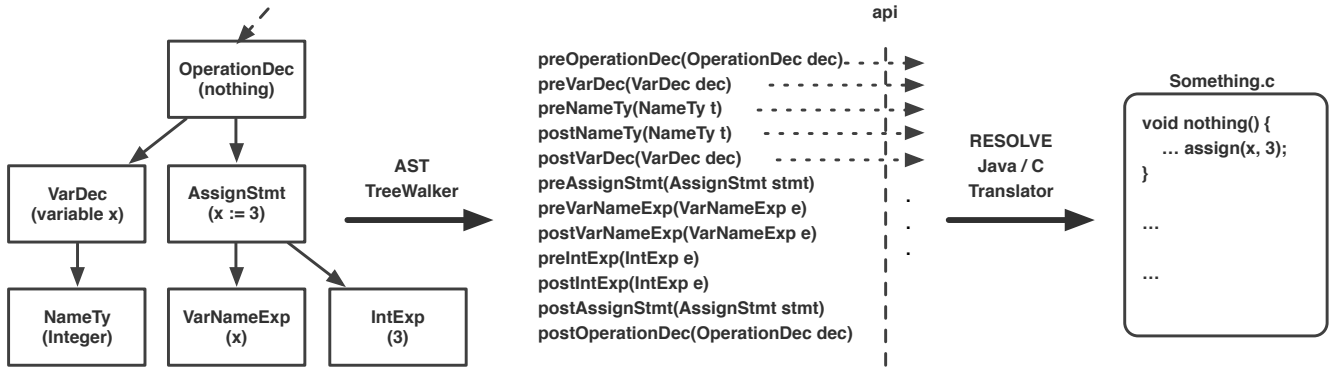

Figure 3: A RESOLVE operation AST, the walk call sequence, and sample translation output.

1. Arriving at a translation model (or, strategy) for an efficient, reliable 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.

In this section we discuss each of these phases, concluding with a demo illustrating each working in tandem to realize the example presented 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 generated makes special considerations for facilities, concepts, and realizations.

The relationship between the RESOLVE module type and c code generated is depicted at a high level in Figure #.

Unsurprisingly, concepts are represented in C by a single header .h listing the various methods

4.2 Translator Implementation

4.2.1 AST Traversal

Translation is performed over the course of a traversal of RESOLVE's abstract syntax tree (AST). The traversal mechanism used is a dervivative of the visitor pattern that provides a SAX-dom style pre-post traversal over all nodes in the tree. Thus, for any given node present, a total of two visits occur: One corresponding to the node being 'hit' during the pre traversal stage, and one for the post.

To make this more concrete, consider the following dummy operation.

```
Operation Nothing(); Procedure
        Var X : Integer;
        X := 3;
end Nothing;
```

Shown in Figure 3.4.1 is the AST representation of operation `Nothing`. Here, language constructs are represented as labeled boxes, while the actual traversal over these constructs – and the order in which it is performed – is communicated on the right via the call stack. Each of these calls

are received in the translator in the order in which they are visited within the tree. It is up to the client (in this case, the author of the C-translator) to decide which of these methods they wish to override and perform custom actions within.

We feel this particular traversal pattern lends itself to task of source to source translation for the following reasons:

- A visit method for a construct provides a convenient encapsulation of all the logic required to translate RESOLVE construct x to appropriate C construct y.
- Overriden visit methods apply to every instance of a construct – meaning RESOLVE's C translator requires few (if any) loops, as the walk itself serves as the method of iteration.
- Finally, only the visit methods for the constructs we currently wish to process need to be overridden. This makes it easy to tweak and optimize the size of the translator's codebase, implementing only visit method for language constructs that explicitly require translation.

4.2.2 *Translation output*

Output of translated code is done using *Stringtemplate* – a third-party tool written in Java that allows users to define parameterizable templates. Like the name suggests, a template is simply "a document with holes" that the user choses when and how to fill.

An example C function definition template is shown below.

```
function_def(modifier, type, name, params,
                                vars, stmts) ::= <<
<modifier> <type> <name> (<params; sep = ", ">) {
    <vars; sep = "\n">
    <stmts; sep = "\n">
}>>
```

User supplied attributes, enclosed in `<..>`, indicates the position of that attribute relative to others. It is entirely up to the user to define which attributes to fill in, and how complex they want them to be. For example, the user might choose to fill the `params` attribute with a simple string, or a separately defined `parameter` template, which in turn might use another separately defined `type` template.

```
parameter(type, name) ::= "<type> <name>"
```

In the context of language translation, these templates, when stored on a stack and manipulated over the course of the aforementioned AST traversal, help simplify the task of producing complicated, structured blocks of C output. For instance, upon visiting `preOperationDec`, a `function_def` template can be instantiated by the client and pushed onto a global translation stack with its `name`, return `type`, and `modifier` attributes filled in. As `preOperationDec`'s children are visited, the `function_def` template currently at the top of the stack receives similarly constructed parameter, variable, and statement templates from the nodes being walked. Upon reaching `postOperationDec`, we can be assured that the function has been completely filled in with the appropriate templates – assuming the user has implemented the children's visit methods.

Hence, the only actual work being performed within visit methods is forwarding appropriate information from tree-node it represents, to an externally defined template. This allows us to exploit (in shameless design pattern parlance) a strict model view controller (MVC) separation in the translator's codebase between the mechanism that does the AST visiting (controller), the tree nodes from which we're adding information to templates (model), and the external file containing all available C language templates (view).

4.3 Memory Allocation

Dynamic memory allocation does not lend itself well with an embedded programming paradigm. With hardware constraints, embedded systems have limited capacities on RAM. The telos motes are limited to 128 bytes of RAM. Many programs, however, require dynamic memory allocation to be used, including RESOLVE, in order to make it extensible. Previous RESOVE translations to C required static memory allocation [1]. In this section a stack based dynamic memory allocator is introduced.

4.3.1 *Allocation using* `salloc`

The `salloc()` is a first fit memory allocator. Rather than allocating on the heap, `salloc()` uses the stack. At compilation, the allocator provisions a fixed size of memory. It requires a small section of meta-data called a block which contains information of about the size of memory allocated, neighboring blocks, as well if the block is free or not. A sample representation of the stack shown in figure 4.3.1, is a typical example of allocated memory on the stack.

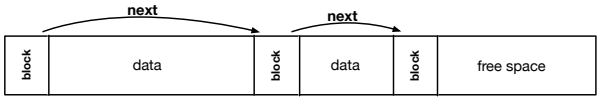

Figure 4: Representation of memory stack

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 for something else. The `sfree()` operation is an analogous implementation to the standard C `free()` function for the stack.

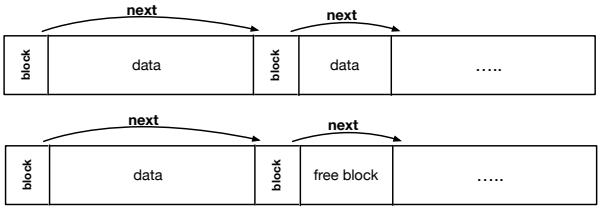

Figure 5: sfree to free memory

4.3.3 *Optimizations*

A common problem that can occur in memory allocation is fragmentation. As shown in figure 4.3.3, fragmentation can lead to inefficient and increased memory usage. This problem is magnified on embedded systems with limited memory capabilities. Simple optimizations can be made however

to reduce fragmentation. Splitting is one optimization that `salloc()` to maximize memory usage. Shown in figure 4.3.3, blocks can be split to the size that is required. Another means of optimizing memory usage is joining together freed blocks. When a call to `sfree()` is made, neighboring blocks are coalesced together, as shown in figure 4.3.3.

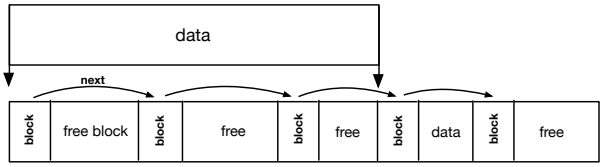

Figure 6: fragmentation

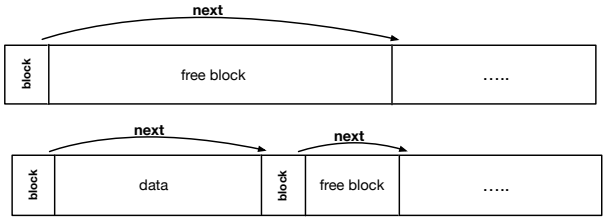

Figure 7: block splitting

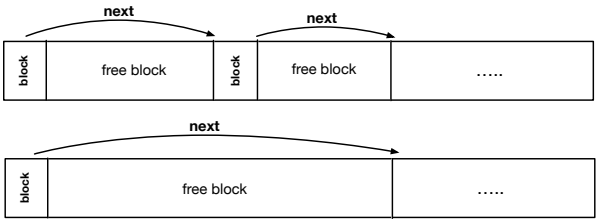

Figure 8: block fusing

5. 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.

6. 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] Hampton Smith. *Engineering Specifications and Mathematics for Verified Software*. PhD thesis, Clemson University, 2013.