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 TinyOS applications. In doing so, we enrich the work originally presented in [?] 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 RESOLVE) realizations. We demonstrate these additions with a revised version of the LED template originally presented in [?] that showcases some of the more recent mathematical developments in "string theory" (resolve's mathematical string library) as well as verification of a simple enhancement for Leds template that includes preliminary generation of performance profile characteristics.

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 is largely comprised of simulation tools such as TOSSIM, Cooja, and Arora that make use of high-fidelity simulations of mote network and hardware behavior 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.

In this paper, we approach this problem by using RESOLVE (Reusable Software Language with Verification) [CITE] [CITE] [CITE] as means of specifying, writing, and ultimately verifying proposed embedded code at a high level, then present a technique of translating this verified code down into lower level, embedded-platform friendly C.

Our choice to use RESOLVE as a language and verification frontend stems from the language's well acknowledged characteristics [CITE] that emphasize the following:

- **Modularity** At the language level, a strict separation of concerns is enforced between module specifications and client implementations. Thus, for any one particular specification, there can be any number of interchangeable implementations. This separation is ideal considering that many embedded applications are composed in similar fashion: Various drivers oftentimes provide a fixed set of operations, but have many distinct implementations that vary from platform to platform, vendor to vendor.
- **Mathematical flexibility** RESOLVE offers a flexible mathematical typing system that allows users to either draw from a library of preexisting mathematical units when writing specifications, or create their own.

This paper aims to leverage RESOLVE's mathematical flexibility of and component based design principles to the challenge of formally verifying an LED driver for the Telos motestack family.

We discuss this challenge in three parts. In the first, we provide both a short overview of the Telos family mote platform – including a description of the LEDs driver which we later specify, verify, and translate – as well as a short overview of the RESOLVE language itself. In the second section, we provide revised specifications for the LED drivers as well as a single enhancement on these drivers. This is followed by a review discussing the results of verification along with a review of some verification conditions generated by the compiler involving preliminary results for performance characteristics of enhancements. In the third and final section of the paper, we discuss the tool itself – providing implementation details as well as a discussion of how this iteration of the tool differs from that originally detailed in [?]. Finally, we conclude by discussing tool improvements and directions for future research.

2. A MODEL FOR C TRANSLATION

One of the primary challenges in translating from RESOLVE to C is finding a suitable C representation for each RESOLVE module and the constructs allowable in each. As we are dealing with an environment where functional correctness is a primary concern, it is especially important that the code generated represents as closely as possible the original RESOLVE source. In an effort to make such considerations, at the highest level, our tool outputs code specifically tailored to one of three distinct RESOLVE module types.

Concepts To model as closely as possible the role of a concept module, our tool produces a single .h file that contains structures for each parameter to a concept, pointers corresponding to the abstract types defined by the concept, as well as set of function pointers for each of the methods provided by the original RESOLVE concept.

Realizations Referring to both enhancement realizations as well as concept realizations, these are translated into a .h/.c pair. The header portion of this pair contains function declarations for the creation and destruction of a "Realization_Specific" facility. Elements of this facility are accessible only through a concept interface. The .c file has 5 sections:

Facilities

The representation of objects make use of a special

3. IMPLEMENTATION

Development of the tool presented here can be logically partitioned into two phases: The tool that performs the actual translation from RESOLVE to C, and a separate tool that is called by the generated code that is responsible for allocating appropriate amounts of memory on the motestack without needing to resort to static allocation.

3.1 Translation AST walking

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 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 well to the this application 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 an 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 especially easy to tweak and optimize the size of the codebase for the C translator.

3.2 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 of a template for a function/operation definition might look like the following:

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

Words enclosed in `<..>` signifies an attribute passable to the template and indicates where it will be positioned. It is entirely up to the user to define which attributes they wish to fill in, and at what level of granularity they wish to compose their templates. For example, the user might opt to fill-in and pass the `parameter(..)` template shown below to the `params` attribute, or, alternatively, a simple string.

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

In the context of language translation, these templates when stored on a stack and manipulated over the course of

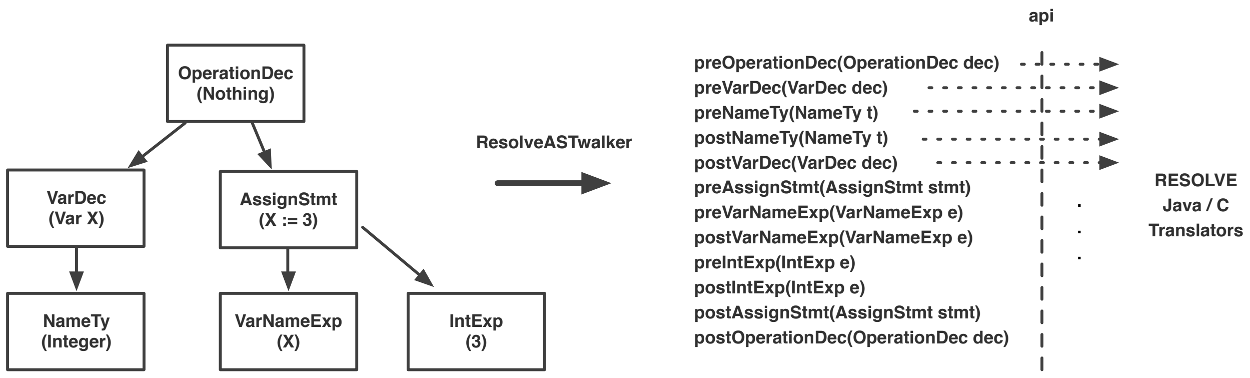

Figure 1: An AST of a RESOLVE operation and its corresponding AST treewalker call sequence.

the AST traversal, simplify the task of producing arbitrarily complicated, nested 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 get visited, the `function_def` template currently at the top of the stack receives similarly constructed parameter, variable, and statement templates from the corresponding child nodes. Upon reaching `postOperationDec`, we can be assured that the function has been completely filled in – assuming the user implemented the visit methods for the children.

Hence, the only work being done within visit methods themselves is forwarding appropriate information from treenodes to an externally defined template. This allows us to exploit (in shameless design pattern parlance) a strict model view controller (MVC) separation in our 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).

3.3 Garbage collection

a source to source translator which, given a resolve component, outputs a C representation of that component.

4. ACKNOWLEDGEMENTS

Special thanks to Mike Kabanni and Mark Todd whose initial contributions and suggestions for the model of C translation paved the way to this paper.