Clemson University
TigerPrints

All Theses | Theses

8-2010

A Verifying Compiler for Embedded Networked Systems

Kalyan chakradhar Regula
Clemson University, kalyan.chakradhar@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses

Part of the Computer Sciences Commons

Recommended Citation

Regula, Kalyan chakradhar, "A Verifying Compiler for Embedded Networked Systems" (2010). *All Theses*. 899.
https://tigerprints.clemson.edu/all_theses/899

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

A Verifying Compiler for Embedded Networked Systems

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master Of Science
Computer Science

---

by
Kalyan Chakradhar Regula
August 2010

Accepted by:
Dr. Jason O. Hallstrom, Committee Chair
Dr. Murali Sitaraman
Dr. Brain Malloy

Abstract

Embedded networked devices are required to produce dependable outputs and communicate with peer devices given limited computing resources. These devices monitor and control processes within the physical world. They are used in applications related to environmental monitoring, telecommunications, social networking, and also life-critical applications in domains such as health care, aeronautics, and automotive manufacturing. For such applications, software errors can be costly - both in terms of financial and human costs. Therefore, software programs installed on these devices must meet the appropriate requirements. To guarantee this, one must *verify* that the implemented code meets the corresponding specifications. Manual trial-and-error validation of such applications, especially life-critical software programs, is not a feasible option.

This work presents a verifying compiler developed for embedded network programs by extending the RESOLVE verifying compiler with a software module that translates RESOLVE code to equivalent C code. Specifications and implementations for embedded networked applications are written in the RESOLVE language. The compiler supports automated verification, automatically generating mathematical assertions, which, if satisfied, ensure that the code is correct. These assertions are proved using the mathematical theorems and lemmas provided by the RESOLVE mathematical library. The verified code is then translated to C and installed on the embedded target.

The contributions described in this thesis are: (i) We explore the use of RESOLVE in specifying pin-level drivers for components of an embedded device. (ii) We describe the translation strategies implemented to generate correct-by-construction C source code from verified RESOLVE code, with examples of basic and reusable operations such as *sense data*, *broadcast data*, and *receive data*. (iii) We provide techniques used to optimize the generated code in terms of memory usage and runtime efficiency.

Dedication

To my family.

Acknowledgments

I would like to express my deepest gratitude and respect to my advisor Dr. Jason O. Hallstrom, for his continuous support and faith in me. I would like to thank the RESOLVE research group at Clemson University for helping me in understanding the RESOLVE compiler. I also thank the faculty of the School of Computing at Clemson University who helped in creating a wealth of required knowledge.

A special thanks to Hao Jiang, Sravanthi Dandamudi, Sally Wahba and Yvon Feaster for their support, suggestions and encouragement. Finally, I would like to thank my family for their love and standing by me in all my aspirations.

Table of Contents

Title Page . . . . **i**

Abstract . . . . **ii**

Dedication . . . . **iii**

Acknowledgments . . . . **iv**

List of Tables . . . . **vii**

List of Figures . . . . **viii**

List of Listings . . . . **ix**

1 Introduction **1**
- 1.1 Embedded Networked Systems 1
- 1.2 Problem Statement 2
- 1.3 Solution Approach 2
- 1.4 Contributions 3
- 1.5 Organization of Chapters 4

2 Related Work **5**
- 2.1 Testing Embedded Software 6
- 2.2 Verification of Embedded Software 7
- 2.3 Verification of C Programs 8
- 2.4 RESOLVE 11

3 RESOLVE **12**
- 3.1 Overview 12
- 3.2 Operators 12
- 3.3 Keywords 13
- 3.4 Concepts 13
- 3.5 Realizations 16
- 3.6 Enhancements 18
- 3.7 Facilities 21

4 Compiler Translation Strategies **22**
- 4.1 Overview 22
- 4.2 Datatypes 23
- 4.3 Swap Operation 23
- 4.4 Arrays 25
- 4.5 Concept Instantiation 29

4.6 Operations as Parameters 36
4.7 Translator Optimizations 39

5 Validation and Evaluation 42
5.1 Broadcast Data Application 45
5.2 Receive Data Application 50
5.3 Optimization Results 54

6 Conclusions 57

Bibliography 59

List of Tables

3.1 RESOLVE Operators 13
3.2 RESOLVE Keywords 14
3.3 RESOLVE Parameter modes 14

5.1 Memory Usage of Applications Compiled for the ATMEGA644 Processor 56

List of Figures

4.1 Overview of Compiler Translation Process 23
4.2 Variable Declaration in RESOLVE and C 24
4.3 Swap Operation 26
4.4 Arrays Representation in Memory 27
4.5 Steps Involved in Concept Instantiation 30
4.6 Representation of Nested Components - Stack of Stacks of Integers 34

5.1 Execution Times for Basic and Lazy Array Initialization Strategies 56

List of Listings

3.1 Declaration of Queue Concept in RESOLVE [48] 14
3.2 Queue Type Family Declaration [48] 15
3.3 Queue Operations [48] 15
3.4 Queue Realization declaration [48] 16
3.5 Implementation of Queue Operations [48] 17
3.6 Stack Concept [48] 18
3.7 Copying Capability Stack Enhancement [48] 19
3.8 Copying capability realization [48] 20
3.9 Stack Facility 21
3.10 Stack Facility with Enhancement 21
4.1 RESOLVE Generic Datatypes Declared in C 24
4.2 Translated C Code for Integer Variable Declaration in RESOLVE 24
4.3 Swap Operation in C Source Code 25
4.4 Integer Array Declaration in RESOLVE 27
4.5 Translated Integer Array Declaration in C Source Code 27
4.6 Using Swap on an Integer Array in RESOLVE 28
4.7 Translated C code for Swap Operation on Integer Array 28
4.8 RESOLVE Concept Declaration 29
4.9 Record Declaration in RESOLVE 29
4.10 Facility Declaration in RESOLVE 29
4.11 Facility Declaration in RESOLVE 30
4.12 Representation Structure in Translated C Code 32
4.13 Int_Stack Variable Declaration in RESOLVE 32
4.14 Int_Stack Variable Declaration in Translated C Code 32
4.15 Generated Initialization Function for Int_Stack in C 33
4.16 Generic Stack Concept Declaration [48] 33
4.17 Facility Declaration in RESOLVE 33
4.18 Representation Structure for Nested Stack Components in C 35
4.19 Generated Initialization Functions for Nested Stack Components in C 35
4.20 Use of an Operation as a Parameter to a Realization in RESOLVE [48] 36
4.21 Facility Using Copy Enhancement of Stack in RESOLVE 37
4.22 Generated C Source Code for Obvious_CC_Realiz 37
4.23 Generated C Source Code for Copy_Stack_Fac 38
4.24 Translated C Code for Integer Variable Declaration in RESOLVE 39
4.25 Optimized Translated C code for Swap Operation on Integer Array 40
4.26 Array Initialization 40
4.27 Array Declaration and Access in RESOLVE 41
4.28 Optimized Translated Code for Array Declaration and Access in C 41
5.1 Leds_Template Specification in RESOLVE 43
5.2 ADC_Template Specification in RESOLVE 44
5.3 UART_Template Specification in RESOLVE 46

5.4 Facility Declarations in the Broadcast_Data Application 46
5.5 Translated Facility Declarations in Broadcast_Data Application in C 47
5.6 Variable Declaration and Component Initialization in the Broadcast_Data Application 47
5.7 Translated C code for Listing 5.6 48
5.8 Sensing and Broadcasting Logic in the Broadcast_Data Application 49
5.9 Translated Sensing and Broadcasting Logic in the Broadcast_Data Application in C 50
5.10 Facility and Variable Declarations in the Receive_Data Application 51
5.11 Translated Facility and Variable Declarations in the Receive_Data Application in C 52
5.12 Data Receiving Program Logic in the Receive_Data Application 53
5.13 Translated Data Receiving Program Logic in the Receive_Data Application in C . 54

Chapter 1

Introduction

1.1 Embedded Networked Systems

Embedded networked systems are composed of small computing devices capable of processing data and communicating with peer devices within a network. These systems are closely coupled with physical processes and are generally used to monitor and control them. For example, a *wireless sensor network* contains sensing devices that collect sensed data and communicate that data over a network. The resources on such devices are limited; for example, a Tmote Sky [14] relies on an 8MHz microcontroller with 10K RAM and 48K Flash .

Practical applications of embedded networked systems can be found in the areas of environmental monitoring, telecommunications, and social networking. Embedded networked systems are also widely used in the fields of health care, aeronautics, and automotive engineering, where applications are life-critical. The information produced by these physical processes is unlimited, therefore embedded devices are deployed in large numbers and are programmed to function over long periods of time. The limited computing resources on these devices constraints their behavior, including their ability to collaborate with other devices, causing developers to face significant system design challenges.

1.2 Problem Statement

Embedded networked systems are built targeting quality attributes such as accuracy, reliability, security, and availability in spite of their limited computing and power resources. Though each of the attribute is equally important, their priority changes depending on the domain and applications. For example, the aeronautics and medical fields demand more accuracy and reliability, whereas environmental monitoring applications require more availability. In the health care field, embedded software correctness is more important because of two reasons: First, decision support systems are developed based on the data sampled by embedded devices in the medical instruments; Second, it involves risk to patient's life. Similarly, the criticality of software written is higher in the fields of aeronautics and automotive manufacturing because software errors in embedded devices used in such systems can lead to loss of life.

In addition, incorrect software may affect the power consumption of an embedded device leading to reduced lifetime. For example, an embedded networked system deployed for environmental monitoring uses devices that have limited power (usually from batteries). The software interacts with and controls various hardware components, such as radios, sensors, and storage devices. If an error exists in the software, these components may consume more power than required. In such scenarios, batteries may have to be replaced often, which is time-consuming and expensive. There has been extensive research on maximizing the lifetime of embedded networked systems and minimizing the maintenance costs incurred due to power consumption [50, 2, 32]. In the case of software maintenance, it is difficult to manually program each device in the network; instead, the operating system should support a remote maintenance interface for upgrades, such as dynamic reprogramming [19]. All these techniques introduce more complexity in the software that is being installed on the devices, which in turn increases software maintenance costs [5].

In summary, embedded networked systems need software correctness to provide reliable data for applications and to reduce maintenance costs. The following section introduces the approach taken in this work.

1.3 Solution Approach

To guarantee software correctness, one must *verify* that the implemented code meets the corresponding specifications. *Formal* verification is the approach taken in this work. This process can

be done either by model checking or logical inference. In a logical inference approach, mathematical assertions that specify behavior of the system are used. However, the complexity of these assertions increases with the complexity of the code, making verification more difficult. Therefore, manual verification, which uses human-written mathematical assertions, is not always feasible, especially in the case of a life-critical software. In a model checking approach, finite state concurrent systems are verified. The requirements are provided as a model (written in formal language) and tested with specifications (written as temporal logics). A good description of model checking technique is provided in [36]. Our work is based on logical inference approach because model checking approach works only for finite state systems and increases confidence in software correctness by finding bugs that are not found using testing and simulation [15].

The verification process involves specifying software components, implementing those components, generating mathematical assertions for the implementations, and proving their correctness based on mathematical theorems. The specification of embedded hardware drivers, their implementation and application logic are written in the RESOLVE Language [20]. These specifications can be implemented in numerous ways [8]. The RESOLVE compiler supports automatic verification of component implementations. This work extends the functionality of the RESOLVE compiler with a translator module that generates C code that can be installed on an embedded networked device.

1.4 Contributions

The following are the contributions described in this thesis: (i) We explore the use of RESOLVE in specifying pin-level drivers for components of an embedded device. (ii) We describe the translation strategies implemented to generate correct-by-construction C source code from verified RESOLVE code, with examples of basic and reusable operations such as *sense data*, *broadcast data*, and *receive data*. (iii) We provide techniques used to optimize the generated code in terms of memory usage and runtime efficiency.

1.5 Organization of Chapters

Chapter 2 presents related work in the area of programming verification. Chapter 3 provides an overview of the basic RESOLVE language constructs, definitions of the basic RESOLVE keywords and modules. Chapter 4 details the translation strategies and optimizations applied in this work. Chapter 5 presents application examples specified, implemented, and verified using the RESOLVE language, and the resulting C code. Chapter 6 concludes the work with summary of contributions.

Chapter 2

Related Work

Hoare in [30] discusses the criteria to be considered for "grand challenge" computing research problems. As an example, he presents the verifying compiler problem which lists attributes to be considered for solving the verification problem. The logical inference approach is one approach to verifying programs, based on the mathematical theorems to model and prove program correctness. Ireland in [34] presents the practical issues involved in proving the associated theorems and proof planning techniques that are used to search proofs in automated theorem proving. He explains issues associated with verifying tools that depend on factors such as use of annotations, code generation techniques, and targeted programming languages, which helps in identifying the designing issues of the verifying compiler. He also describes the importance of feedback provided by a verifying compiler that helps in program debugging. Holzmann in [31] explains the economic factors that should be considered to understand the importance of formal verification, especially in case of finding defects that are rare and catastrophic. He mentions that traditional testing practices may not be helpful and can be expensive to find such defects.

This chapter introduces some of the related work focused on improving the software correctness using verification. In the following sections we discuss the prominent approaches to software testing and verification in embedded networked systems and prior work done on C verification. We briefly discuss the RESOLVE verification system, deferring a complete treatment to Chapter 3.

2.1 Testing Embedded Software

Beutel et al. in [6] argue that a complete test methodology is required to develop a robust wireless sensor network. They present a distributed unit testing framework that decreases software testing time and provides set of features that help in achieving software correctness. The framework relies on wireless sensor network simulators and testbeds. They claim that test cases developed for the framework can be reused across different testing platforms (simulator or testbed), ensuring software correctness. As this work is based on testing, though it helps in finding defects, it will not help in checking software correctness as per the specifications.

Network simulators imitate the behavior of networked devices without the use of a network. They are widely used to test code prior to deployment. They offer two advantages: First, it is less expensive. Second, testing can be performed with simulated hardware configurations, which saves time in building hardware for testing. Titzer et al. in [49] present a "cycle-accurate" instruction level sensor network simulator called *Avrora*. The objective of the work is to enable scalable, cycle-accurate simulation. This allows testing time-dependent properties of large networks. Since *Avrora* supports instruction-level emulation, the sensor network installations will be at a higher confidence level before actual deployment. Levis et al. in [42] present another simulator called *TOSSIM* for wireless sensor networks developed using *TinyOS*. It focuses on providing network scalability, complete coverage of possible system interactions, accuracy in capturing network behavior, and enabling developers to test their implementation code. Since both these tools are based on simulation, which is an abstraction from reality, and helps finding defects, they cannot verify the software correctness.

Network reprogramming can be used to test new or updated features of an embedded networked system. Dunkels et al. in [19], describe a run-time dynamic linker and loader using *Executable and Linkable* file format for reprogramming a sensor network. Energy cost and execution time are used as metrics to evaluate the dynamic linking mechanism. Hui and Culler in [33] explain a data dissemination protocol called Deluge. This protocol implements a multi-hop dissemination service used to propagate data (divided into fixed-size pages) from one device to another. In general, these techniques are preferred after the software is installed in the real network in order to reduce the maintenance costs. Our work avoids such maintenance costs by verifying the software well before the installation in the real network.

Another testing approach is to use a network testbed, a network of embedded devices created

for experimenting with application code. Arora et al. in [3] present *Kansei*, a heterogeneous testbed including stationary, mobile and portable sensor nodes. Using these hardware resources, a time-accurate hybrid simulation engine is built into the testbed. *Kansei* is more efficient for experiments that run for long durations. Werner-Allen et al. in [52] present *Motelab*, a web-based testbed that provides web interface. It supports data logging for debugging programs, network reprogramming and maintaining testbed jobs. Dalton et al. in [18] present the first sensor network testbed that supports visualization for application developed using *Tiny OS 2.0*. A *UML*- based sequence diagram is provided to understand the node-level behavior. It provides developers with program analysis and instrumentation features to manage the non-determinism in the execution order for distributed wireless sensor networks, device messages and asynchronous event-based semantics. Though testbeds provide a good architecture to find defects in embedded software, they do not guarantee the software correctness.

Khan et al. in [37] present *Dustminer*, a tool to identifying errors in wireless sensor networks. It logs different event types occurring in a sensor network. The event log contains information about the node, event type, attribute values associated with each event and a timestamp. It performs data mining on the collected data to identify execution sequences that lead to network failure. This is done in two steps: identifying repeated patterns that lead to failure and correlating the patterns with less frequently occurring events to find the actual bug. They describe a configurable software architecture that provides an interface for plug-in modules. This tool also addresses non-determinism in distributed network interactions (by classifying them into "good" and "bad") and the complexity of interactions. To avoid redundant and false patterns, *Apriori* algorithm is implemented in the tool. This work is based on data mining on the logged information, which is fundamentally different from our approach of verifying embedded software.

2.2 Verification of Embedded Software

Cousot and Cousot in [15] present problems and perspectives in the verification of embedded software. One of the problems they mentioned is handling data structures that use pointer aliasing in embedded software programs. Such data structures are generally ignored in the model checking approach because of the complexity. Our work handles such problems by using the RESOLVE [20] language for writing specifications and implementations. RESOLVE minimizes aliasing by use of

swap operator.

Hanna et al. in [26] present *Slede*, a framework that focuses on automatically verifying sensor network security protocols. It uses a formal verification process based on model-checking techniques. It generates models from *nesC* language implementations for security protocols using supplemental program information such as message structure, topology, and protocol properties to be verified. A separate annotation language is used to provide this information. It verifies a protocol with an intruder model that introduces malicious code in the message structure information. Their work includes an evaluation of security protocols such as "one-way key chain based one-hop broadcast" authentication protocol and μ-tesla protocol. Model checking verification technique faces scalability issues and our work avoids such issues by using logical inference technique.

Kim et al. in [38] propose an automated protocol verification framework for wireless sensor networks that uses XML-based test procedure description language (TPD) to describe the test cases, test conditions and test procedures. The framework contains test application block installed on the user computer and is responsible for executing the test cases as described in TPD and generates message required by the test driver block installed on sensors. This framework is based on testing and does not provide a verification procedure for software correctness.

2.3 Verification of C Programs

Most embedded networked systems are implemented using the C programming language. Accordingly, our work focuses on generating verified C code from embedded networked system specifications and implementations. Other authors considered verifying C programs using model checking [23] and automated reasoning [17] based on low-level memory models for C [51] and models specific to VLSI designs [24].

Schulte et al. in [45] present *Verifying C Compiler* that performs formal verification based on a logical inference technique. It generates verification conditions from annotated C programs, which are proved using an automatic theorem prover. It is specifically designed to verify operating systems; as a result it supports type safe, pointer arithmetic and volatile data access. By adding additional assertions in the generated verification conditions, it also checks for null pointer references, dangling pointers, double frees, division by zero, a over/underflow. Leinenbach et al. in [41] in the context of pervasive systems verification, present a dialect of C programming language that is compiled and

verified to check the correctness of program implementations . The work mainly focuses on proving logical blocks that involve dynamic memory allocation, address alignments and function calls based on *Haore's* partial correctness logic. Tuch in [51], presents the research work on verifying system C code based on its low-level memory model improved techniques to prove correctness of code, especially programs with pointer address arithmetic and structure types such as *structs* . The input source is annotated with pre-/post-conditions and invariants for each functional program block. The verification framework uses this information and translates source to *higher order logic* required by the prover. Filliâtre and Marché in [23] present a similar research on verifying C programs with pointers and prototype implementation based on *Burstall's model* for structures. The work inserts annotated pre and post conditions, global invariants, and loop variants in to C programs and uses *Why* [22] tool for generating verification conditions. It also supports pointer aliasing. Crocker and Carlton in [17] present research work that uses *Perfect developer* [16] tool to reason about requirements and specifications using an automatic theorem prover, along with the ability to generate code based on a formal specification language called *Perfect* [1]. They extended this work to verify annotated C programs to support developers interested in writing code by hand for embedded software. Many of these works supports verification of annotated C, however by using annotations everywhere these approaches makes the code complex and difficult to maintain. Additionally errors in these annotations may introduce complexities of their own.

Blazy in [7] presents an optimizing and verifying compiler, *CompCert* that uses the *Coq* proof assistant. The language is based on C, except that it does not support `goto` and `longjmp` instructions. The compiler optimizations include constant propagation, common subexpression elimination, and instruction scheduling. *Compcert* generates six intermediate languages, therefore Blazy defined formal semantics for all languages of *CompCert* sharing a common memory model. Gallardo et al. in [25] discuss about construction of a model checker using *OPEN/CSAR* for distributed applications that use C source code. Ivanicic et al. in [36] present a procedure to generate a model from C semantics that can be used for model checking C programs. A C program is transformed into smaller subsets until the program state is represented as scalar variables and boolean representations. This is achieved by presenting the model as a finite state system with the use of abstractions provided by F-Soft tool [35]. They modeled pointers as integers, heap as a finite array, stack as a global array. All functions are moved in the main function, variables are declared globally, and return statements are replaced with goto statements. A control flow graph is generated from labels

and goto statements. F-Soft tool is developed targeting sequential C programs, specifically verifying if all label statements are reachable within the program scope using control and data flow of the program. It also verifies NULL pointer references, array bound violations, etc. Chaki et al. in [9] present Modular Analysis of Programs in C (MAGIC) that verifies a C program. This work also supports compositional verification. MAGIC extracts a finite model from a program using predicate abstractions and theorem proving. It uses label transition systems (LTS) to generate a specification state machine and uses actions as state transitions. For procedures that perform multiple tasks depending on the parameters, multiple LTS are created and are selected using guards. This information is provided as predicate abstraction for procedures. MAGIC creates a model from a control flow graph generated for the program, verifies the program and refines the model when it fails. These works adapted the model checking approach and as mentioned earlier they face scalability issues which is not a problem in our work.

Coen-Porisini et al. in [10] describe the usage of symbolic execution for building models useful for verifying safety-critical systems software programs. Symbolic execution uses symbolic expressions (that can be formalized) to represent values of program variables. They use Safer-C [28], a subset of C language, that can be analyzed by symbolic execution. In other words, this work does not support complex programs involving dynamic memory allocation, recursion, file operations, etc. They use Path Description Language (PDL) to represent the safety-related properties of the system as predicates in the model. PDL is a formal language used to express structural properties of a program. They also developed the Symbolic execution Aided Verification Environment tool to validate their approach. Sharma et al. in [46] describe Assertion Checking Environment (ACE) for compositional verification of programs written in MISRA C [4], a subset of C language used for safety-critical systems. This compositional verification reduces the complexity of verification by slicing the program into smaller units to be verified. ACE uses Haore logic [29] to define pre and post conditions of a program. ACE generates a call graph of the program and translates the annotated C program to Simple Programming Language and specifications containing axioms and properties (expressed in temporal logic), are used as input to Stanford Temporal Prover [43]. The leaf nodes in the call graph are verified initially so that a compositional verification can be performed on function that calls another functions in a program. The pre-conditions of verified leaf nodes (functions) are used to build prefunction annotations for their caller function. All these research works use subset of C language and does not support complex programs. Our work do not have such restrictions since

we use RESOLVE language to write implementation and generate verified C source code.

2.4 RESOLVE

Kirschenbaum et al. in [39] present a case study that explains the importance of automating the verification process and the challenges faced using a sorting example. The first challenge was that the generated verification conditions could not be proved because of the human error in writing specifications and loop invariants. The second challenge was lack of supported theories and lemmas. So theories from the RESOLVE mathematical theory library were imported to *Isabella* theory library as *Isabella* theories. They have also tagged the new lemmas imported into *Isabella* theory, to help the tool in constructing proofs. Finally they have eliminated universal quantifications in annotations to ease reuse of theories and create general versions.

Smith et al. in [47] explain the importance of verifying reusable components. As an example, they created specification of a List component, implemented and extended its features using RESOLVE. The RESOLVE verifier module is used to generate verification conditions for List component specifications and implementations. These verification conditions are proved using RESOLVE prover and translated to Java language using RESOLVE Translator.

Chapter 3

RESOLVE

3.1 Overview

RESOLVE is a specification and implementation language developed to support the design of reusable software components and formal reasoning about software correctness. In this context, a software component is defined as a unit of a software system that hides its internal implementation and provides a well-defined interface for interaction. The framework supports verification of components and automatic proofs of software correctness. RESOLVE adopts logical inference approach to formal verification. In this approach, objects provided by a component are modeled as mathematical entities. For example, a `Stack` object is modeled as a mathematical *String*. The reasoning is based on a set of mathematical assertions generated (from formally specified pre-conditions and post-conditions), and proved using a set of relevant mathematical theorems and lemmas. The following sections provide an overview of RESOLVE. We describe component specifications, termed *"concepts"*, implementations, termed *"realizations"*, extensions, termed *"enhancements"* and instantiations, termed *"facilities"*.

3.2 Operators

As a programming language, RESOLVE provides a set of primitive operators; Table 3.1 lists some of the most basic. As explained in [40], object aliasing introduced by references causes difficulties for program reasoning. To avoid aliasing of object references, RESOLVE uses a *swap*

Arithmetic Operators	***Name***
+	Add
-	Subtract
/	Divide
%	Modulus
*	Multiply
^	Exponential
REM	Remainder
Logical Operators	***Name***
=	Equal to
/=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
~	Not
AND	Logical and
OR	Logical or
Data Movement Operators	***Name***
:=	Function Assignment
:=:	Swap

Table 3.1: RESOLVE Operators

operator for basic data movement instead of reference copying (Java) or deep copying (C++).

3.3 Keywords

We now introduce some of the RESOLVE keywords that will be used in the remaining chapters. Table 3.2 serves as a keyword reference.

In RESOLVE, operation parameters are declared with a *mode* that provides additional information about the effect of the operation on the value of the parameter passed.Table 3.3 explains the different modes and their usage. In the table, if **E** is a parameter to an operation, its value prior to the operation call is denoted **#E**, and its post value is denoted as **E**.

3.4 Concepts

Software component specifications are expressed as *concepts*, which use mathematical models to represent component state and behavior. We present a `Queue` specification as an example to make these concepts more concrete. To support queue entries of different data types, the `Queue` concept is

Keywords	***Description***
`Integer`	Data type representing whole numbers
`Character`	Data type representing character symbols
`Boolean`	Data type representing *true* or *false* values
`type`	Introduces a generic type
`Type Family`	Introduces a concept as an object type
`Array`	Static array data structure
`Record`	Similar to a *struct* in C language
`exemplar`	Used within a Type Family declaration to represent the concept
`constraint`	Defines conditions that needs to be true before and after calling a public method
`requires`	Defines pre-conditions for an operation
`ensures`	Defines post-conditions for an operation
`uses`	Declares the list of facilities, theories, and concepts used by the current module

Table 3.2: RESOLVE Keywords

Keywords	***Description***
`restores` E	operation uses #**E**, potentially changes its value during the operation, and restores **E** to #**E** before returning
`replaces` E	operation ignores #**E** and replaces its value
`alters` E	operation uses #**E**, potentially changes its value to a random value because **E** will not be used later in the program
`updates` E	operation uses #**E** and potentially modifies its value that will be used later in the program
`preserves` E	operation uses #**E** and maintains **E** value with #**E** value
`clears` E	operation removes the value of #**E** and sets it to the initial value for its type
`evaluates` E	operation evaluates that **E** contains a constant data value

Table 3.3: RESOLVE Parameter modes

defined as a generic component, parameterized by the type of entry it contains. Listing 3.1 presents a generic *Queue* declaration in RESOLVE

```
1 Concept Queue_Template(type Entry; evaluates Max_Length: Integer);
2     uses Std_Integer_Fac, String_Theory;
3     requires Max_Length > 0;
```

Listing 3.1: Declaration of `Queue` Concept in RESOLVE [48]

Here, the concept has two parameters: *Entry* of generic type `type` and `Max_Length` of type **Integer**. The **uses** clause specifies concepts, facilities, and theories used by the concept. The `Queue` concept uses `Std_Integer_Fac` (explained in section 3.7) and `String_Theory`. One of the advantages of RESOLVE is that it provides reusable theories. For example, `String_Theory` can also be used to model a stack or a list. The **requires** clause defines a pre-condition that requires

the length value passed as argument be greater than zero (for creating an instance of this concept). String theory is used to express the component model, as shown in Listing 3.2.

```
1 Type Family Queue is modeled by Str(Entry);
2 exemplar Q;
3 constraint |Q| <= Max_Length;
4 initialization ensures Q = empty_string;
```

Listing 3.2: `Queue` Type Family Declaration [48]

Since the concept is generic, it defines the model's type as a `Family`; it must be instantiated with actual parameter values before it can be used [20]. The **exemplar** keyword is used to define a prototypical instance of the concept, and the **constraint** clause asserts a condition on the queue's length. The **initialization** clause asserts that the queue be empty at the point of declaration. Listing 3.3 shows the queue operations defined in the concept.

```
1    Operation Enqueue(alters E: Entry; restores Q: Queue);
2        requires |Q| < Max_Length;
3        ensures Q = #Q o <#E>;
4
5    Operation Dequeue(replaces R: Entry; updates Q: Queue);
6        requires |Q| > 0;
7        ensures #Q = <R> o Q;
8
9    Operation Swap_First_Entry(updates E: Entry; updates Q: Queue);
10       requires |Q| > 0;
11       ensures there exists Rem: Str(Entry) such that
12               #Q = <E> o Rem and Q = <#E> o Rem;
13
14   Operation Length(restores Q: Queue): Integer;
15       ensures Length = (|Q|);
16
17   Operation Rem_Capacity(restores Q: Queue): Integer;
18       ensures Rem_Capacity = (Max_Length - |Q|);
19
20   Operation Clear(clears Q: Queue);
```

Listing 3.3: `Queue` Operations [48]

All parameters are passed with a *mode* that imposes conditions on their post-conditional values. The # symbol, as noted earlier, denotes the value of a variable prior to the operation call, and the symbol $\circ$ denotes the string concatenation operator. Since `Queue` is modeled as a *String*, the `Enqueue` operation appends the new entry variable at the end of the string representing the queue. $|\mathbf{Q}|$ denotes the length of the string (i.e., the number of entries in the queue).

3.5 Realizations

A *realization* module defines implementations for the operations defined in a given concept. Each concept may have many realizations. For example, a `Queue` concept can be implemented with a static array or a `Stack` component. Listing 3.4 presents the header of a circular array-based realization of a `Queue` concept.

```
1 Realization Circular_Array_Realiz for Queue_Template;
2     Type Queue = Record
3         Contents: Array 0..Max_Length of Entry;
4         Front, Length: Integer;
5     end;
```

Listing 3.4: `Queue` Realization declaration [48]

Here, the `Type Queue` is represented as a `Record` consisting of an `Array` of unknown type (`Contents`) and two integer variables (`Front`, `Length`). The operation implementations are declared as *procedures* in the realization module unlike operations in the concepts that specify the required behavior. An example is shown in Listing 3.5.

Note that the pre-conditions specified in the concept are not checked in the implementation code. The conditions must be satisfied by modules that use the realization.

```
1   Procedure Enqueue(alters E: Entry; updates Q: Queue);
2        Var I: Integer;
3        I := (Q.Front + Q.Length) mod Max_Length;
4        Q.Contents(I) :=: E;
5        Q.Length := Q.Length + 1;
6    end Enqueue;
7
8    Procedure Dequeue(replaces R: Entry; updates Q: Queue);
9         R :=: Q.Contents(Q.Front);
10       Q.Front := (Q.Front + 1) mod Max_Length;
11       Q.Length := Q.Length -1;
12   end Dequeue;
13
14   Procedure Swap_First_Element(updates E: Entry; updates Q: Queue);
15     Q.Contents(Q.Front) :=: E;
16   end Swap_First_Element;
17
18   Procedure Length(restores Q: Queue): Integer;
19       Length := Q.Length;
20   end Length;
21
22   Procedure Rem_Capacity(restores Q: Queue): Integer;
23       Rem_Capacity := Max_Length - Q.Length;
24   end Rem_Capacity;
25
26   Procedure Clear(clears Q: Queue);
27      Q.Front := 0; Q.Length := 0;
28   end Clear;
```

Listing 3.5: Implementation of Queue Operations [48]

3.6 Enhancements

A RESOLVE concept is intended to provide basic operations. Any extensions to a concept can be defined using *enhancement* modules. These extensions are specified and implemented in a similar fashion as *concepts* and *realizations*. For an enhancement, an implementation may only use public methods declared in the corresponding concept. For example, consider the `Stack` concept in Listing 3.6.

```
1  Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
2      uses Std_Integer_Fac, Modified_String_Theory;
3      requires Max_Depth > 0;
4
5      Type Family Stack is modeled by Str(Entry);
6          exemplar S;
7          constraint |S| <= Max_Depth;
8          initialization ensures S = empty_string;
9
10     Operation Push(alters E: Entry; updates S: Stack);
11         requires |S| < Max_Depth;
12         ensures  S = <#E> o #S;
13
14     Operation Pop(replaces R: Entry; updates S: Stack);
15         requires |S| /= 0;
16         ensures #S = <R> o S;
17
18     Operation Depth(restores S: Stack): Integer;
19         ensures Depth = (|S|);
20
21     Operation Rem_Capacity(restores S: Stack): Integer;
22         ensures Rem_Capacity = (Max_Depth - |S|);
23
24     Operation Clear(clears S: Stack);
25
26 end Stack_Template;
```

Listing 3.6: Stack Concept [48]

To enhance Stack with a copy operation, the specification in Listing 3.7 would be provided.

```
1 Enhancement Copy_Capability for Stack_Template;
2     Operation Copy_Stack(replaces S_Copy: Stack; restores S_Orig: Stack);
3         ensures S_Copy = S_Orig;
4 end Copy_Capability;
```

Listing 3.7: Copying Capability Stack Enhancement [48]

Here, Stack_Template is the name of the concept being enhanced. (Recall that Stack is the type family defined in the concept). The operation Copy_Capability copies all of the elements from S_Orig to S_Copy without modifying the contents of *S_Orig*. The implementation of the above specification is shown in Listing 3.8.

On line 3, the implementation declares an operation parameter used to copy stack entries. The specification provided in the operation parameter helps in verifying the program to make sure it copies the stack entries. A temporary Stack is used to store the entries of S_Orig in reverse order. The entries in the temporary stack are then popped, copied and pushed to S_Orig and S_Copy. This satisfies the conditions imposed by the parameter modes (replaces, restores) noted in the operation signature. The *while* loops declare *loop invariants* and *variant functions* used in verification. The invariant (**maintaining**) and progress metric (**decreasing**) keywords specify the conditions that have to be satisfied for each iteration of the loop.

```
1 Realization Obvious_CC_Realiz
2     (
3         Operation Copy_Entry(replaces Copy: Entry; restores Orig: Entry);
4         ensures Copy = Orig;
5     )
6 for Copy_Capability of Stack_Template;
7     uses Std_Boolean_Fac;
8     Procedure Copy_Stack(replaces S_Copy: Stack; restores S_Orig: Stack);
9         Var Next_Entry, Entry_Copy: Entry;
10        Var S_Reversed: Stack;
11
12        While ( Depth(S_Orig) > 0 )
13            maintaining #S_Orig = Reverse(S_Reversed) o S_Orig;
14            decreasing |S_Orig|;
15        do
16            Pop(Next_Entry, S_Orig);
17            Push(Next_Entry, S_Reversed);
18        end;
19        Clear(S_Copy);
20
21        While ( Depth(S_Reversed) > 0 )
22            maintaining S_Copy = S_Orig and #S_Orig = Reverse(S_Reverse) o S_Orig;
23            decreasing |S_Reversed|;
24        do
25            Pop(Next_Entry, S_Reversed);
26            Copy_Entry(Entry_Copy, Next_Entry);
27            Push(Next_Entry, S_Orig);
28            Push(Entry_Copy, S_Copy);
29            Depth(S_Reversed);
30        end;
31    end Copy_Stack;
32 end Obvious_CC_Realiz;
```

Listing 3.8: Copying capability realization [48]

3.7 Facilities

A *Facility* provides an instance of a concept implemented by a particular realization. In the case of a standard concept, such as **Integer, Boolean,** etc., RESOLVE provides instances as "standard" facilities. A user-defined facility declaration includes actual values for both the concept parameters and the realization parameters. For example, consider the `Stack` concept with two parameters provided in Listing 3.6; its corresponding *Facility* declaration is shown in Listing 3.9.

```
1 Facility ISF is Stack_Template(Integer, 5)
2         realized by Array_Realiz;
```

Listing 3.9: `Stack` Facility

The code in Listing 3.10 creates a facility for the `Stack` concept with the copying capability enhancement. The operation parameter is implemented in the calling module and passed as argument to the enhancement realization. This facility can also be used to write the "main" function of the application.

```
1     Operation Copy_Integer(replaces Copy: Integer; restores Orig: Integer);
2         ensures Copy = Orig;
3     Procedure
4         Copy := Orig;
5     end;
6     Facility SF is Stack_Template(Integer, 5)
7         realized by Array_Realiz
8               enhanced by Copy_Capability
9                 realized by Obvious_CC_Realiz (Copy_Integer);
10    Operation Main();
11    Procedure
12       ...
13    end Main;
```

Listing 3.10: `Stack` Facility with Enhancement

Chapter 4

Compiler Translation Strategies

4.1 Overview

On successful verification of a RESOLVE module, the compiler translates the RESOLVE code into functionally equivalent C code. This verification process is carried out in a hierarchical manner, verifying all included modules first, followed by modules instantiated using facilities, and finally the top-most RESOLVE module itself. The translation process follows the same structure. *Concepts* are translated to C header files; they specify the interface of a component. *Realizations* may represent a concept in several ways. For example, a `Queue` can be represented as an array of `Record` type or a `Stack` component. Each realization is translated to a C file and a header file. *Enhancements* extend the behavior of a component by specifying additional operations. Each enhancement is translated into a header file, and each corresponding realization is translated into a C file. A *Facility* is created to instantiate a concept with actual parameters and to write the application logic that uses instantiated concepts and their enhancements. For example, a typical sense and broadcast application could be created as a facility that uses a `Queue` component to store the sensor readings prior to transmission over the radio. Each facility is translated into a C file, and if a concept is instantiated in the facility, corresponding header files and C files are generated for that particular instance of that concept. Figure 4.1 provides an overview of the translation process used for each type of RESOLVE module. A common header file is created to include libraries shared by multiple RESOLVE modules.

This chapter introduces the strategies used to translate RESOLVE to C Code. We addi-

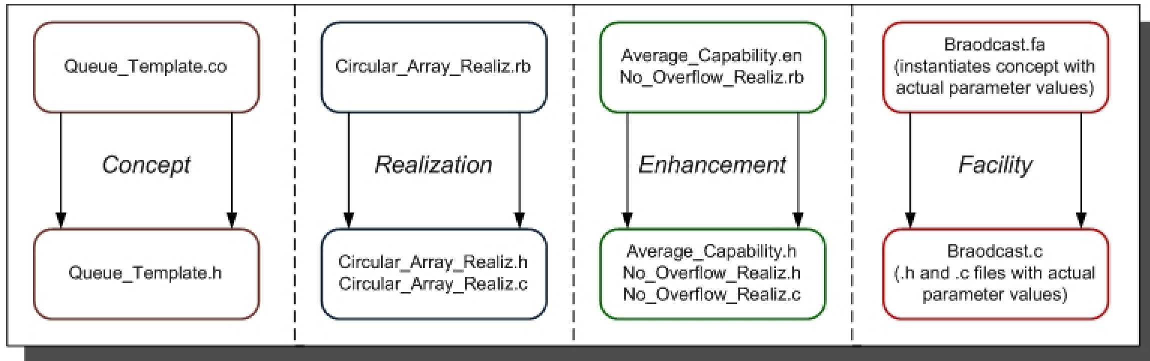


Figure 4.1: Overview of Compiler Translation Process

tionally discuss the optimization techniques used to handle scalar variables, scalar constants, and array initialization.

4.2 Datatypes

The compiler supports three built-in datatypes: `Integer`, `Character`, and `Boolean`. There is no RESOLVE realization code available for these concepts; the realizations are provided in C. `Integer` variables declared in RESOLVE are translated to *long int*s; `Characters` are translated to *char*s; `Booleans` are translated to *uint8_t*. In case of `Booleans`, symbolic constants (TRUE/FALSE) are created using preprocessing directives in C.

4.3 Swap Operation

The basic data movement operator in RESOLVE is the *swap* operator (:=:). The operator swaps the values of two objects in constant time. The chief advantage of using swap is that it does not create object aliases; i.e., it does not create multiple references to a single object. Aliasing is a well known obstacle in verifying software correctness. This is not a problem in our compiler because all RESOLVE programs control aliasing using swapping [27]. To provide the same operator implementation for swapping object of various types in C, we define a new data type, `r_type` (*"resolve type"*), as shown in Listing 4.1, realized as a void pointer. We also define a pointer to `r_type`.

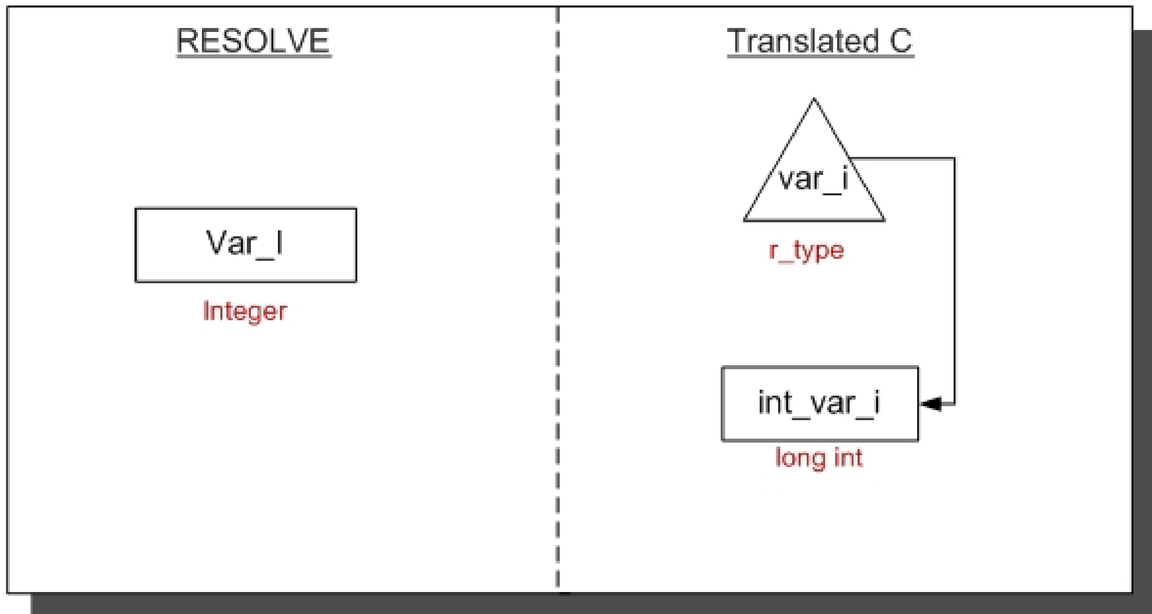


Figure 4.2: Variable Declaration in RESOLVE and C

```
1 typedef void* r_type;
2 typedef r_type* r_type_ptr;
```

Listing 4.1: RESOLVE Generic Datatypes Declared in C

A variable declared (both local and global) in RESOLVE is translated to a global variable in the generated C code to make sure the object value is persistent across the stack frames. More precisely, each variable declaration is associated with a storage location, and a pointer (`r_type`) to that location as shown in Figure 4.2. The variable value is accessed only using its pointer in the generated code. For example, if a variable `Var_1` is declared as an `Integer`, the statement will be translated to the C code shown in Listing 4.2

```
1 long int int_var_1 = 0 ;
2 r_type var_1 = &int_var_1 ;
```

Listing 4.2: Translated C Code for Integer Variable Declaration in RESOLVE

Here the storage location for `Var_1` is declared as `int_var_1`, and the translated C code uses `var_1` to read and modify its value. All variable declarations are translated to `r_type` variables including the operation parameters. Further, all constants used within the input source are wrapped in temporary variables and accessed using pointers in the translated code.

This translation strategy enables constant time swap implementation based on the pointer (r_type) reassignment. The swap implementation accepts two r_type_ptr (pointer to r_type) variables as parameters and performs a "shallow" swap. Since swap operates on pointers to objects, a shallow swap is both sufficient and efficient. For a more detailed treatment, the behavior of the swap operation is illustrated in Figure 4.3. The figure illustrates a case where swap operates on two Integer variables, Var_1 and Var_2. The operation is defined in the common C file as shown in Listing 4.3. In Figure 4.3, the program state that is illustrated in the numbered blocks corresponds to the line numbers in Listing 4.3

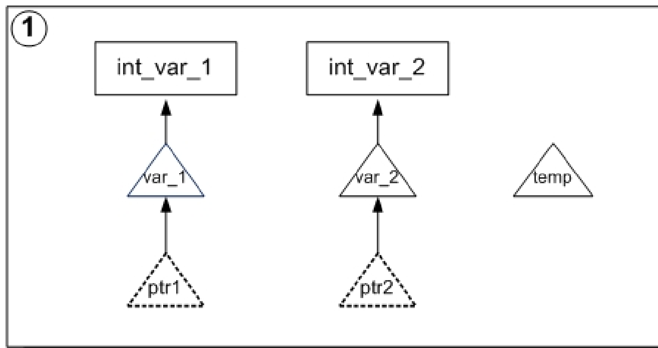
```
1 void swap(r_type_ptr ptr1, r_type_ptr ptr2){
2     r_type temp = *ptr1;
3     *ptr1 = *ptr2;
4     *ptr2 = temp;
5 }
```

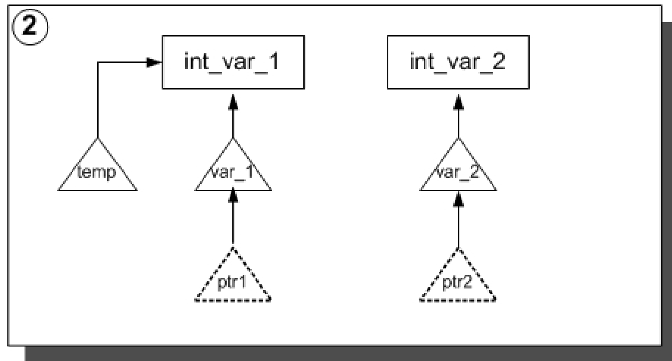
Listing 4.3: Swap Operation in C Source Code

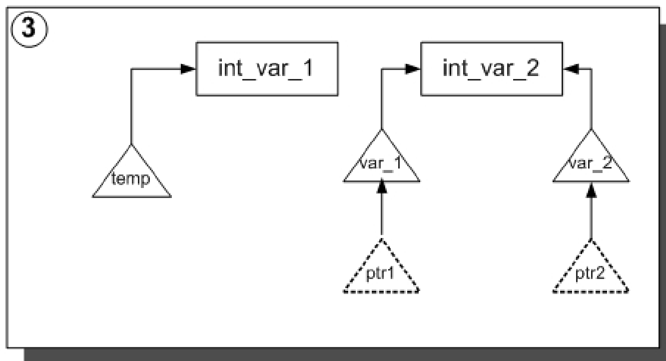
This strategy enables a single generic swap implementation for all data types. We have optimized the translation for scalar variables and scalar constants, as explained in Section 4.5.

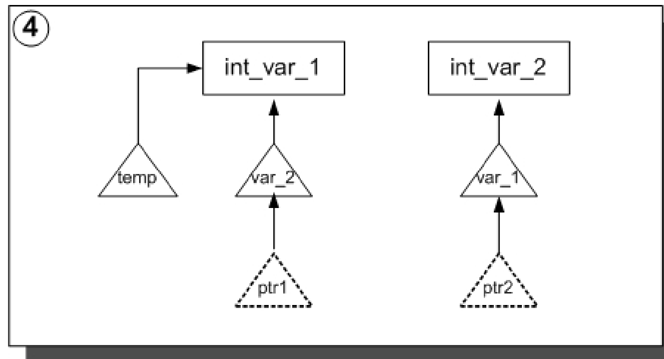
4.4 Arrays

We now describe the translation process for arrays and their initialization. In RESOLVE, every variable is initialized to the default value for its type prior to the point of first access. This is done by initializing the storage location of the translated variable to a default value, as shown in Listing 4.2. Here variable int_var_1 is initialized to 0 because Var_1 is declared as Integer type in RESOLVE. In the case of arrays, initialization depends on the type of its contents. As is the case for all objects, each array variable in RESOLVE is declared as an r_type variable in the translated code. An associated array of the same dimension and actual contained type is also declared, similar to the storage strategy for scalar variables. Listing 4.4 shows an example declaration for an array containing Integers.









LEGEND:

long int

r_type

r_type_ptr

Figure 4.3: Swap Operation

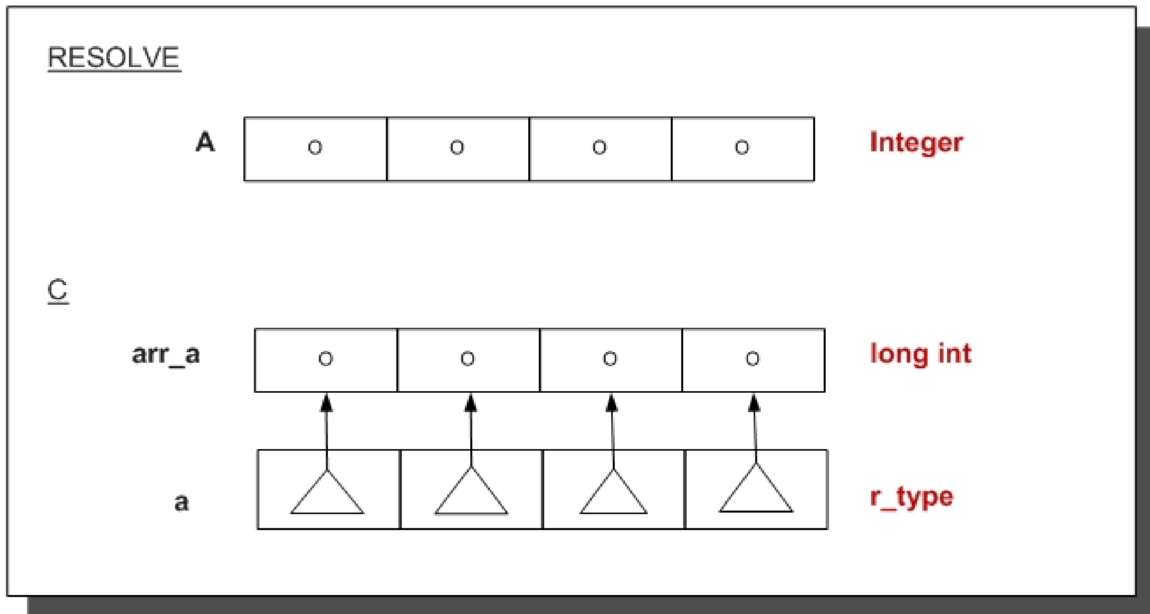


Figure 4.4: Arrays Representation in Memory

```
1 ...
2 Type Arr = Array 1..4 of Integer;
3 ...
4 Var A : Arr;
```

Listing 4.4: `Integer` Array Declaration in RESOLVE

First note that the array size is declared as 1..4, which means that the minimum index that can be accessed is 1, and the maximum index that can be accessed is 4. To generate equivalent C code, the corresponding array size is declared as [`maximum index - minimum index + 1`]. When array index '`i`' is accessed, the translated code will access index [`i-minimum index`]. The translated code for the array declaration example in Listing 4.4 is shown in Listing 4.5. A pictorial representation is shown in Figure 4.4.

```
1 long int arr_a[4] = { 0 };
2 r_type a[4-1+1] = { &arr_a[0],&arr_a[1],&arr_a[2],&arr_a[3] };
```

Listing 4.5: Translated `Integer` Array Declaration in C Source Code

Since the type of `a` is `Array of Integer`, the generated code defines an associated array `arr_a` of type `long int`, initialized to 0. The variable `A` in RESOLVE is declared as `r_type`

in C and initialized with the addresses of the `arr_a` content slots. This initialization strategy is implemented to enable constant time swapping over the array elements. The `swap` operation on `Integer` array in RESOLVE is illustrated in Listing 4.6.

```
1 Type Arr = Array 1..4 of Integer;
2 Operation Main();
3    Procedure
4    Var I : Integer;
5    Var A : Arr;
6    I := 6;
7    A(2) :=: I;
8 end Main;
```

Listing 4.6: Using Swap on an `Integer` Array in RESOLVE

Here the RESOLVE code contains a procedure "`Main`" that declares an `Integer` variable `I` and a variable `A` of type `Array of Integer`. On line 7, a swap operation is performed on `A(2)` and `I`. Now consider the translated C code in Listing 4.7. Lines 1 to 4 contain variable declarations and initialization logic as explain earlier. The RESOLVE procedure `Main` is translated to the `main` function in C. Line 6 shows the translated code for accessing an `Integer` variable using its r_type pointer.

```
 1 long int arr_a[4] = { 0 };
 2 long int int_i= 0;
 3 r_type i = &int_i;
 4 r_type a[4-1+1] = { &arr_a[0],&arr_a[1], &arr_a[2],&arr_a[3] };
 5 void facility_main(){
 6     *(long int*)i = 6;
 7     swap(&a[2-1], &i);
 8 }
 9 int main(){
10     facility_main();
11 }
```

Listing 4.7: Translated C code for Swap Operation on `Integer` Array

On line 7, the translated code contains a call to the `swap` function with parameters containing the addresses of variables on which the swap is operated. This ensures that `a[2-1]` contains the value 6, and `i` contains 0, the default value for `Integer`, after the swap function completes.

4.5 Concept Instantiation

The translation strategy for instantiating a *concept* involves several steps. In this section, we explain the generic instantiation strategy using an example concept, *Int_Stack*. A concept can be realized in several ways, and each of its *realizations* may have a unique representation. So the type of a concept is always declared as `r_type` in C, and the representation defined in each realization is declared as a *struct*. In RESOLVE, the `Int_Stack` concept is declared as shown in Listing 4.8.

```
1 Concept Int_Stack(evaluates Max_Length: Integer);
```

Listing 4.8: RESOLVE Concept Declaration

A concept may take parameters, such as `Max_Length` for `Int_Stack`. These parameters are accessible in concepts, realizations, and enhancements. For example, consider the representation declared in Listing 4.9. `Int_Stack` is represented as a `Record` type containing an `Array of Integers` having maximum length `Max_Length`.

```
1 Type Int_Stack = Record
2     Contents: Array 1..Max_Length of Integer;
3     Top: Integer;
4 end;
```

Listing 4.9: Record Declaration in RESOLVE

Actual parameter values are provided at the time of facility creation, which is required for concept instantiation. The facility type declaration will ensure that a value for `Max_Length` is provided to `Int_Stack`, as shown in Listing 4.10.

```
1 Facility IS_Fac is Int_Stack(5)
2         realized by Array_Realiz;
```

Listing 4.10: Facility Declaration in RESOLVE

In the case of enhancements, the facility declaration must include the enhancement and its realization details. For example, if `Int_Stack` has an associated copying capability enhancement (explained in Section 3.6) which is realized by `Integer_CC_Realiz`, the facility declaration would be as shown in Listing 4.11.

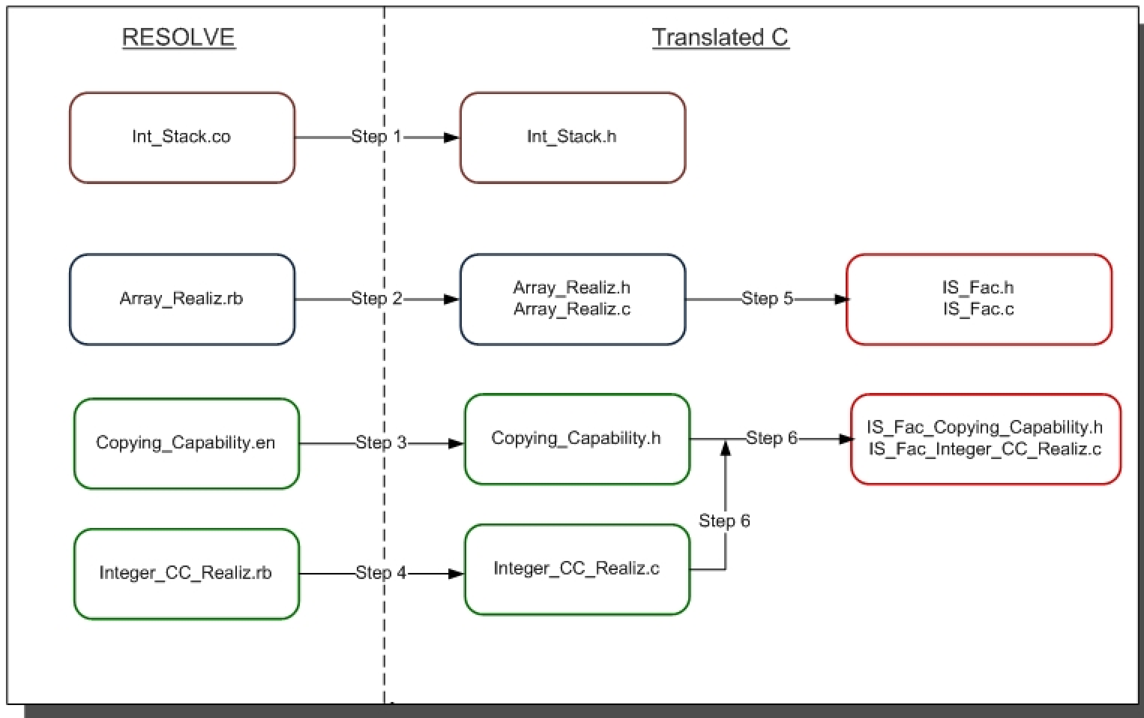


Figure 4.5: Steps Involved in Concept Instantiation

```
1 Facility IStack;
2 ...
3 Facility IS_Fac is Int_Stack(5)
4         realized by Array_Realiz
5             enhanced by Copying_Capability
6                 realized by Integer_CC_Realiz (Copy_Integer);
7 ...
8 end IStack;
```

Listing 4.11: Facility Declaration in RESOLVE

When the compiler parses the above facility declaration, it instantiates the concept with actual parameter values. The instantiation process is shown in Figure 4.5. It involves the following steps.

- Step 1: `Int_Stack` is translated into C header file
- Step 2: `Array_Realiz` is translated into C file and associated C header file containing the representation as a C structure

- Step 3: `Copying_Capability` is translated into C header file
- Step 4: `Integer_CC_Realiz` is translated into C file
- Step 5: `IS_Fac.h` and `IS_Fac.c` files are generated as a copy of C header file and C file from Step 2 with actual parameter values
- Step 6: `IS_Fac_Copying_Capability.h` and `IS_Fac_Integer_CC_Realiz.c` are generated as a copy of C header file and C file from Step 3 and Step 4 respectively, with actual parameter values

As explained in Section 4.1, Step 1 corresponds to *Concept* translation, Step 2 corresponds to *Realization* translation, Step 3 and 4 correspond to *Enhancement* translation, and Steps 5 and 6 correspond to *Facility* translation. The main operation defined in a facility will be generated as "main" function in C source code.

The translated files in Step 2 and 4 are non-functional intermediate files without actual concept parameter values. Precisely, these intermediated files contain an unique string pattern for each parameter variable. This string pattern is replaced with actual parameter value in Steps 5 and 6. The file names for files generated in Step 5 and 6 contain the facility name provided for concept instantiation. For example, in Listing 4.11, the facility name is declared as `IS_Fac`, therefore the files generated have `IS_Fac` in their file names.

The RESOLVE language allows to create multiple instances of a single concept with different parameter values. This is done by creating separate facilities for each set of parameter values. To support multiple instances of a concept in C, the facility name (for example, `IS_Fac` in Listing 4.11) is used as unique key and prefixed to the function names and representation names in the files generated in Step 5 and 6. For example, the translated representation from concept realization is shown in Listing 4.12. Note that the original array shown in Listing 4.9 is translated to two arrays (as explained in Section 4.4).

```
1 /* From file IS_Fac.h */
2 typedef struct is_fac_rep{
3     r_type contents[5-1+1];
4     long int contents_store[5-1+1];
5     r_type top;
6     long int int_top;
7 }is_fac_rep;
```

Listing 4.12: Representation Structure in Translated C Code

Now the variable of type `Int_Stack` is declared as shown in Listing 4.13. Here the facility name `IS_Fac` is used as the qualifier to uniquely identify the object for the `Int_Stack` type.

```
1 Operation Main();
2     Procedure
3         Var IS_Var: IS_Fac.Int_Stack;
4 end Main;
```

Listing 4.13: `Int_Stack` Variable Declaration in RESOLVE

The translated C code declares `IS_Var` along with its representation structure in global scope. The initialization is performed inside the main function. Listing 4.14 shows the translated code in C.

```
1 /* From file IStack.c */
2 int_stack is_var;
3 is_fac_rep is_var_rep;
4 void facility_main(){
5     is_fac_array_realiz_init(&is_var_rep);
6     is_var = &is_var_rep;
7 }
8 int main() {
9     facility_main();
10 }
```

Listing 4.14: `Int_Stack` Variable Declaration in Translated C Code

Here `is_var` is the concept variable and `is_var_rep` is its representation. Similar to other arrays, these objects need to be initialized before they are accessed. The initialization of the `Int_Stack` variable requires initializing the representation structure. On line 5, the `is_fac_array`

_realiz_init function initializes the representation structure. This function is generated during Step 5 of the concept instantiation process; its implementation is shown in Listing 4.15.

```
1 void is_fac_array_realiz_init(is_fac_rep* s){
2     long int al_1;
3     memset(s->contents_store,0, sizeof(s->contents_store));
4     for(al_1=0;al_1<(5-1+1);al_1++)
5         s->contents[al_1] = &s->contents_store[al_1];
6     s->int_top = 0;
7     s->top = &s->int_top;
8     }
```

Listing 4.15: Generated Initialization Function for Int_Stack in C

The RESOLVE language supports nested components, such as *stack of stacks of integers*, *stack of queues of integers*, etc. The compiler supports nested components using a compositional translation strategy. For example, consider a generic stack concept, Stack_Template [48] that has two parameters, Entry (a generic data type) and Max_Depth (maximum number of stack elements). Listing 4.16 shows the Stack_Template declaration.

```
1 Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
```

Listing 4.16: Generic Stack Concept Declaration [48]

To instantiate a stack component containing stacks of integers, the facility declaration would be written as shown in Listing 4.17. Here IS_Fac instantiates Stack_Template with the Integer data type and maximum depth of 5. SS_Fac instantiates a nested stack component with IS_Fac.Stack as the data type and maximum depth of 5. The pictorial representation of *stack of stacks of integers* is shown in Figure 4.6

```
1 Facility IS_Fac is Stack_Template(Integer, 5)
2         realized by Array_Realiz;
3 Facility SS_Fac is Stack_Template(IS_Fac.Stack, 5)
4         realized by Array_Realiz;
```

Listing 4.17: Facility Declaration in RESOLVE

The translated representation structures for IS_Fac is shown in Listing 4.12 and for SS_Fac are shown in Listing 4.18. The stack datatype in ss_fac_rep is a generic Stack type that stores elements of is_fac_rep type.

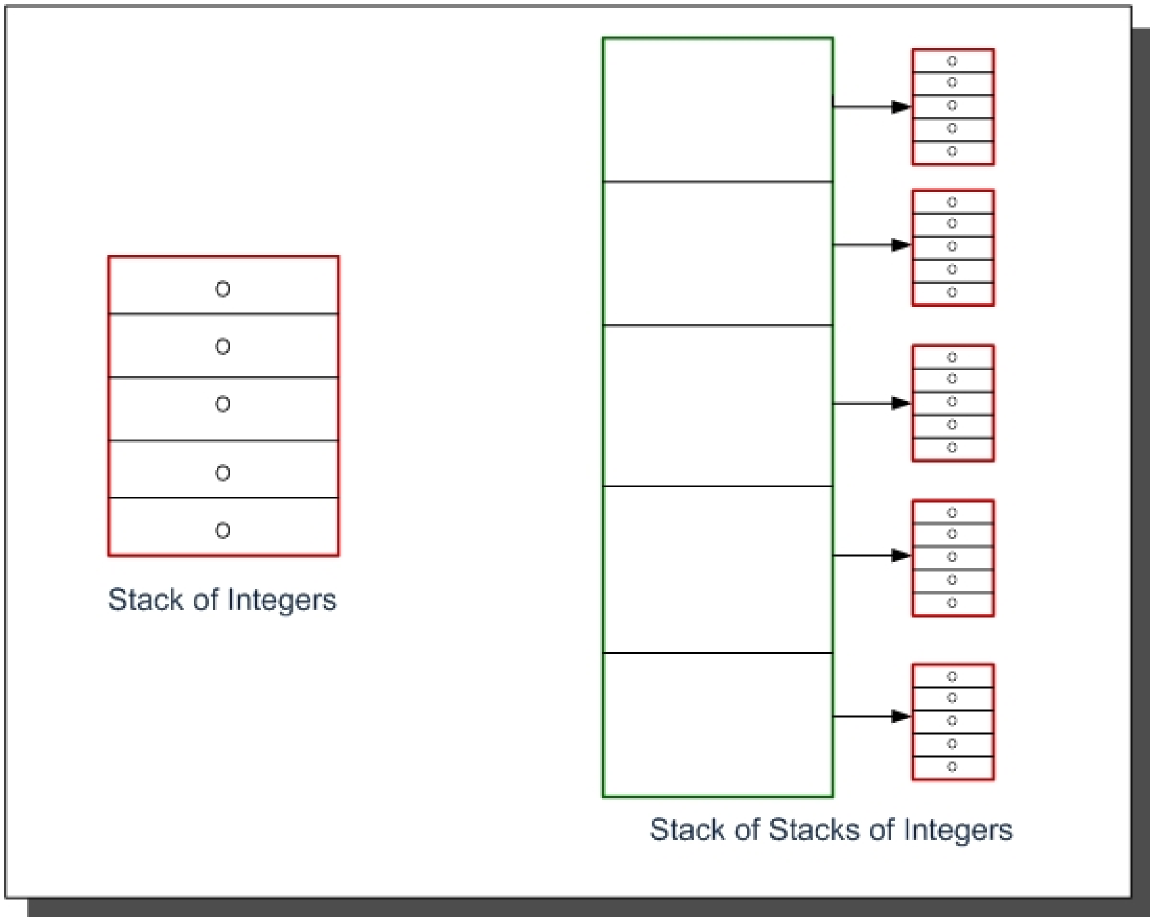


Figure 4.6: Representation of Nested Components - Stack of Stacks of Integers

```c
typedef struct ss_fac_rep{/* From SS_Fac.h file */
    r_type contents[5-1+1];
    stack contents_store[5-1+1];
    r_type top;
    long int int_top;
}ss_fac_rep;
```

Listing 4.18: Representation Structure for Nested Stack Components in C

The generated initialization is similarly nested. Listing 4.19 shows the initialization functions generated during Step 5 of concept instantiation. The function `ss_fac_array_realiz_init` initializes the array of `is_fac_rep` variables declared in global space by calling the `is_fac_array_realiz_init` (from Listing 4.15) as shown on line 4. The contents of the input `ss_fac_rep` variable are initialized with initialized `is_fac_rep` variables as shown on line 6 and 10.

```c
void ss_fac_array_realiz_init(ss_fac_rep* s){ /* From SS_Fac.c file */
    long int i_r,al_1; long int r =0;
    for(i_r = 0; i_r < 5;i_r++)
        is_fac_array_realiz_init(&rep[i_r]);
    for(al_1 = 0; al_1 < (5 - 1 + 1); al_1++){
        s->contents_store[al_1]= &rep[r];
        r++;
    }
    for(al_1 = 0; al_1 < (5 - 1 + 1); al_1++)
        s->contents[al_1]= &s->contents_store[al_1];
    s->int_top = 0;
    s->top = &s->int_top;
 }
```

Listing 4.19: Generated Initialization Functions for Nested Stack Components in C

4.6 Operations as Parameters

The RESOLVE language supports passing operations as parameters to realizations. These operations must be provided at the time concept instantiation. Listing 4.20 shows an example realization for the copying enhancement for `Stack_Template` taken from the RESOLVE examples [48]. The realization defines a copy operation as a parameter, where Entry is the type of element defined as a parameter in `Stack_Template`.

```
1  Realization Obvious_CC_Realiz
2  (
3      operation Copy_Entry(replaces Copy: Entry; restores Orig: Entry);
4          ensures Copy = Orig;
5  )
6  for Copy_Capability of Stack_Template;
7
8  Procedure Copy_Stack(replaces S_Copy: Stack; restores S_Orig: Stack);
9      ...
10     Copy_Entry(Entry_Copy, Next_Entry);
11     ...
12 end Copy_Stack;
13 end Obvious_CC_Realiz;
```

Listing 4.20: Use of an Operation as a Parameter to a Realization in RESOLVE [48]

As explained in the previous section, when instantiating `Stack_Template` with an enhancement, the facility declaration will be as shown on lines 10-13 in Listing 4.21. Since the enhancement realization, `Obvious_CC_Realiz`, defines an operation parameter, a `Copy_Integer` procedure is created as per the specification and passed as argument to `Obvious_CC_Realiz` in the facility declaration.

```
1 Facility Copy_Stack_Fac;
2     uses Std_Boolean_Fac,Std_Integer_Fac;
3
4     Operation Copy_Integer(replaces Copy: Integer; restores Orig: Integer);
5         ensures Copy = Orig;
6     Procedure
7         Copy := Orig;
8     end;
9
10    Facility SF is Stack_Template(Integer, 5)
11        realized by Array_Realiz
12              enhanced by Copy_Capability
13                realized by Obvious_CC_Realiz (Copy_Integer);
14
15    Operation Main();
16    Procedure
17        Var S_Orig: SF.Stack;
18        Var S_Copy: SF.Stack;
19        ...
20        SF.Copy_Stack(S_Copy,S_Orig);
21        ...
22    end Main;
23 end Copy_Stack_Fac;
```

Listing 4.21: Facility Using Copy Enhancement of Stack in RESOLVE

The translated C code for `Obvious_CC_Realiz` includes an additional parameter of type function pointer, to accommodate implementations of the `Copy_Entry` procedure. The implementation expects a function pointer with the same signature as defined in the instantiated operation specification. Listing 4.22 shows the translated code.

```
1 ...
2 void copy_stack(stack* s_copy, stack* s_orig, void(*copy_entry)(long int*, long int*)){
3  ...
4  copy_entry(&entry_copy, &next_entry);
5  ...
6  }
```

Listing 4.22: Generated C Source Code for `Obvious_CC_Realiz`

In the translated version of Copy_Stack_Fac, an implementation of Copy_Entry is generated, and the function's address is passed as parameter to the Copy_Stack procedure, as shown on line 18 in Listing 4.23. As before, the stack variables S_Orig and S_Copy are declared globally. The code also includes the initialization logic for the stacks prior to their first point of access.

```
1 ...
2 void copy_integer(long int* copy, long int* orig);
3 stack s_orig;
4 sf_rep s_orig_rep;
5 stack s_copy;
6 sf_rep s_copy_rep;
7 ...
8
9 void copy_integer(long int* copy, long int* orig) {
10     *copy = *orig;
11 }
12 void facilty_main(){
13     sf_array_realiz_init(&s_orig_rep);
14     s_orig = &s_orig_rep;
15     sf_array_realiz_init(&s_copy_rep);
16     s_copy = &s_copy_rep;
17     ...
18     copy_stack(&s_copy, &s_orig, copy_integer);
19     ...
20 }
21 int main(){
22     facility_main();
23 }
```

Listing 4.23: Generated C Source Code for Copy_Stack_Fac

4.7 Translator Optimizations

We have incorporated two optimization strategies in the translator. The first avoids wrapping scalar variables and scalar constants as `r_type` objects. The second introduces lazy initialization of arrays. These optimizations are intended to provide better memory utilization and run time efficiency, respectively.

4.7.1 Scalar Variables and Scalar Constants

As described earlier, every RESOLVE variable is translated to an `r_type` to enable constant time value swapping. In the case of scalar variables and constants - `Integers`, `Booleans` or `Characters` - there is no need to use a wrapper because standard value assignment takes constant time. So the compiler is optimized to translate scalar variables and constants to their corresponding C data types. Consider the example discussed in Section 4.3, where the variable `Var_1` is declared as an `Integer`. With this optimization enabled, that declaration statement will be translated as shown in Listing 4.24.

```
1 long int var_1 = 0 ;
```

Listing 4.24: Translated C Code for `Integer` Variable Declaration in RESOLVE

If a swap operator is used with two scalar operands (both variable and constants), the code is implemented using assignment statements. Consider the same example provided in Listing 4.6. Since `a` and `i` are both of scalar type, the translated code will be as shown in Listing 4.25. Note that the associated array `arr_a` is not created because the array is of type `Integer`. Also note that the swap operator is translated to assignment operator on lines 5-6.

4.7.2 Lazy Array Initialization

The second optimization strategy is to delay the initialization of array entries until the point of first access. This is implemented using bit flags. When an array is translated from RESOLVE to C, the compiler creates an associated array of type `uint8_t`. The compiler uses one bit flag for each array location. Therefore, if the original array is of size `n`, then the length of the bit array will be equal to `(n/8)`, if `n` is divisible by 8, or `(n/8) + 1`, if `n` is not divisible by 8. When an array location `i` is accessed, the compiler inserts an initialization check for the $\mathtt{i}^{\mathtt{th}}$ array location

using the bit array. If the bit indicates that the entry is not initialized, the compiler inserts a call statement to the function `initialize_array_element`, to initialize `i`th location.

```
1 long int a[4-1+1] = { 0 };
2 long int i= 0; long int ci_0 = 0;
3 void facility_main(){
4     i = 6; ci_0 = a[2-1];
5     a[2-1] = i; i = ci_0;
6 }
7 int main(){
8     facility_main();
9 }
```

Listing 4.25: Optimized Translated C code for Swap Operation on `Integer` Array

Listing 4.26 shows the definition of the `initialize_array_element` function created in the common C file. Since the default value of an object depends on its type, the `flag` parameter determines the value to be set. For example, `0` is default value for an `Integers`, and `FALSE` for `Booleans`. The parameter `array_element` is the pointer to the original array location to be initialized, `array_init` is the pointer to the base address of the bit array, and `index` is the array location to be checked for initialization. The `inited_index` variable stores the byte index of the bit array to be modified. The expression `(1 << mod_result)` sets the `index` bit to 1. On line 7, the condition checks whether the corresponding bit for the input `index` is set; if not, it initializes the `array_element`, and on line 10, the corresponding bit is set to 1.

```
1 void initialize_array_element(uint8_t flag, void* array_element, uint8_t array_init[],
2                                 int index){
3         uint8_t mod_result;
4         uint8_t inited_index;
5         mod_result = index % 8;
6         inited_index = (mod_result == 0 ) ? index /8 : (index/8) +1;
7         if(!(array_init[inited_index] & (1 << mod_result))){
8         // array_element is initialized to its default value
9         ...
10        array_init[inited_index] |= (1 << mod_result);
11        }
12 }
```

Listing 4.26: Array Initialization

For example, consider the RESOLVE code in Listing 4.27. Here an array of 100 `Integers` is declared. Prior to the optimized translation, this declaration would translated to an array of type `long int`, and all 100 locations would be initialized to 0. The `Main` procedure accesses only one array location - index 20. In this case, initializing all array locations is unnecessary. It is only necessary to initialize array index 20.

```
1 Type Arr = Array 1..100 of Integer;
2 Operation Main();
3 Procedure
4    Var A : Arr;
5    Var I : Integer;
6    I := A(20);
7 end Main;
```

Listing 4.27: Array Declaration and Access in RESOLVE

Listing 4.28 shows the optimized translated code in C. On line 3, the bit array is declared, and line 7 checks for the initialization of array index 20.

```
 1 long int a[100-1+1];
 2 long int i = 0;
 3 uint8_t a_init[(100 \% 8 == 0)? 100/8 : (100/8) +1];
 4
 5 void facility_main() {
 6     //a[20-1] initialization check
 7     initialize_array_element(0, &a[20-1], a_init,(((20-1) * 1 )));
 8     i = a[20-1];
 9 }
10 int main(){
11     facility_main();
12 }
```

Listing 4.28: Optimized Translated Code for Array Declaration and Access in C

In Chapter 5, an evaluation of the above mentioned optimization strategies is conducted using the sample applications as test cases.

Chapter 5

Validation and Evaluation

Embedded networked systems consist of devices that provide information to decision support systems in various fields such as health care, aeronautics, environmental monitoring, and others. The constituent devices collect data (*sense*) and communicate (*broadcast* and *receive*) that data to peer devices. These are the basic operations for any embedded sensing device irrespective of the field, the network size, and hardware/software setup. For this reason, we chose to validate the compiler translation using these basic operations and developed embedded application programs targeting the MoteStack [21] platform in RESOLVE. In this chapter, we describe two basic applications. The first can sense and broadcast sensor data over an embedded network, and the second can receive sensor data over an embedded network. We also present evaluation results focused on the optimization techniques adopted in our translation strategies.

The *MoteStack* is a sensing platform developed by the Dependable Systems Research Group at Clemson University. One of the key features of the platform design is that it supports hardware customization through a stackable board interface. The specifications for the hardware driver components are created as concepts in RESOLVE; the realizations are implemented in C. The following components are used to support the broadcast and receive applications.

- **Light Emitting Diode (LED)** component, used as a visual indicator. There are five LEDs on a platform and `Leds_Template` is the RESOLVE concept that specifies the operations on those five LEDs.

- **Analog to Digital Converter (ADC)** component, converts analog readings from a sensor

to digital value that can be processed by a microcontroller. `ADC_Template` is the RESOLVE concept that specifies the operations for an ADC.

- **Universal Asynchronous Receiver and Transmitter (UART)** component, provides asynchronous serial communication between the microcontroller and XBEE radio.
- **XBEE** radio component, provides wireless communication services over the 2.4 GHz band.

All MoteStack applications use these basic components. They are therefore provided as standard facilities. The LEDs concept is shown in Listing 5.1. The concept is modeled as a cartesian product of five boolean variables corresponding to the five LEDs on a MoteStack device. Each LED has two states: `on` and `off`; the initialization operation of every LED ensures that the LED is `off` (as shown on line 7 for `L.L0`) and ready to use.

```
1  Concept Leds_Template;
2      ...
3      Var L:Cart_Prod
4          L0:B;L1:B;L2:B;L3:B;L4:B;
5      end;
6      ...
7      Operation LED0_Init();
8          ensures L.L0 = false;
9      Operation LED0_Set(evaluates b: Boolean);
10         ensures L.L0 = b and L.L1 = #L.L1 and L.L2 = #L.L2
11                 and L.L3 = #L.L3 and L.L4 = #L.L4;
12     Operation LED0_Toggle();
13         ensures L.L0 = not(L.L0) and L.L1 = #L.L1
14                 and L.L2 = #L.L2 and L.L3 = #L.L3
15                 and L.L4 = #L.L4;
16     Operation LED0_Status(): Boolean;
17         ensures LED0_Status = L.L0;
18     --similar operations created for LED1, LED2, LED3, and LED4
19 end Leds_Template;
```

Listing 5.1: `Leds_Template` Specification in RESOLVE

The `Set` operation sets the LED state with the value passed as argument. The `Toggle` operation changes the current state of the LED from `True` to `False` and vice-versa. The `Status` operation returns the current state of the LED (`on` or `off`). As discussed in Section 3.3, `#L.L1` on

line 10 denotes the pre-conditional value of variable L.L1. The ensures clause of Set guarantees that LEDs L.L1, L.L2, L.L3, and L.L4 are not changed when L.L0 is set.

The ADC concept is shown in Listing 5.2. The ADC is modeled as a cartesian product of two boolean variables representing the on/off state of the attached sensor and the initialization state of the component. The ADC must be initialized before it can be used. ADC_Init specifies the initialization operation for the ADC, which ensures that the sensor is off. To read the sensor value, the sensor must be turned on by calling Sensor_On. The operations Sensor_On and Sensor_Off are used to provide and remove the power from the sensor, respectively. The efficient use of these operations can save energy and increase the node's life.

```
1 Concept ADC_Template;
2     uses Std_Integer_Fac;
3     Var ADC:Cart_Prod
4         Sensor_On:B;
5         Init:B;
6     end;
7     Facility_Initialization ensures
8         ADC.Init = false and ADC.Sensor_On = false;
9     Operation Sensor_On();
10        ensures ADC.Sensor_On;
11    Operation Sensor_Off();
12        ensures ADC.Sensor_On = false;
13    Operation ADC_Init();
14        ensures ADC.Sensor_On = false and ADC.Init;
15    Operation Read_ADC(evaluates I: Integer): Integer;
16        requires  ADC.Init = true and ADC.Sensor_On and 0 <= I <= 7;
17        ensures Read_ADC > 0 and ADC.Sensor_On = false;
18 end ADC_Template;
```

Listing 5.2: ADC_Template Specification in RESOLVE

Read_ADC is used to query the sensor to retrieve its current value. This operation ensures that the value read is positive. Read_ADC takes one Integer parameter. The mode evaluates ensures that the parameter contains constant data value, since the value for I corresponds to a port number from 0 to 7 that is used to attach a sensor.

The UART concept is shown in Listing 5.3. It is also modeled as a cartesian product of a natural number for UART baud rate and a Boolean variable for the initialization state of the

component. `UART_Init` takes one `Integer` parameter, corresponding to the desired UART baud rate. `UART_Send_Bytes_Blocking` is used to send the data value passed as argument via the UART. Similarly, `UART_Receive_Bytes_Blocking` is used to receive a data value via the UART. Both operations require that UART to be initialized.

The XBEE radio component is similarly defined; the details are hence omitted. The radio component is modeled as a cartesian product of two boolean variables, corresponding to the initialization state and the error state of the component, respectively. It includes specifications similar to the UART component, but communicates data over a wireless network via an XBEE radio module [44].

5.1 Broadcast Data Application

The Motestack broadcast application provides basic sensing and broadcasting functions. The ADC component is used to query an attached sensor, and the XBEE radio component is used to broadcast the collected data over a wireless network. Once the basic functions are verified, more complex sensing applications can be developed by reusing these verified functions.

For the sake of presentation, the broadcast application code is partitioned into separate listings. Listing 5.4 shows the facility declaration of the broadcasting application, `Broadcast_Data`. Lines 2 and 3 list the standard facilities used by the application. On line 4-7, using a facility, an instance of `Queue` concept for natural numbers is instantiated with the circular array based-implementation and the averaging enhancement. This enhancement extends the behavior of `Queue` to calculate the average of all its entries. `Circular_Array_Realiz` defines the representation as an array with minimum index 0 and maximum index as the concept parameter.

When translated to C, Listing 5.4 will be translated to the header files includes shown in Listing 5.5. These header files are translated concepts for the corresponding standard facilities used, as shown on lines 2-3 in Listing 5.4. Note that the header files `IQF.h` and `IQF_Averaging_Capability.h` are generated in the process of instantiating `Queue` concept, as explained in Section 4.5.

```
1 Concept UART_Template;
2     uses Std_Boolean_Fac, Std_Integer_Fac;
3     Var UART:Cart_Prod
4         Baud_Rate:N;
5         Init:B;
6     end;
7     constraint  UART.Baud_Rate = 1200 or UART.Baud_Rate = 2400 or
8                 UART.Baud_Rate = 4800 or UART.Baud_Rate = 9600 or
9                 UART.Baud_Rate = 14400 or UART.Baud_Rate = 28800 or
10                UART.Baud_Rate = 38400 or UART.Baud_Rate = 57600 or
11                UART.Baud_Rate = 76800 or UART.Baud_Rate = 115200;
12    Facility_Initialization
13        ensures UART.Init = false;
14    Operation UART_Init(evaluates baud_rate:Integer);
15        requires (baud_rate = 1200 or baud_rate = 2400 or
16                  baud_rate = 4800 or baud_rate = 9600 or
17                  baud_rate = 14400 or baud_rate = 28800 or
18                  baud_rate = 38400 or baud_rate = 57600 or
19                  baud_rate = 76800 or baud_rate = 115200);
20        ensures UART.Set_Speed and UART.Init;
21    Operation UART_Send_Bytes_Blocking(restores data:Integer;);
22        requires UART.Init;
23    Operation UART_Receive_Bytes_Blocking(alters data:Integer;): Integer;
24        requires UART.Init;
25 end UART_Template;
```

Listing 5.3: UART_Template Specification in RESOLVE

```
1 Facility Broadcast_Data;
2   uses  Std_Integer_Fac, Std_Boolean_Fac, Std_Leds_Fac,
3         Std_ADC_Fac, Std_UART_Fac, Std_XBEE_Fac;
4 Facility IQF is Queue_Of_N_Template(10)
5         realized by Circular_Array_Realiz
6             enhanced by Averaging_Capability
7                 realized by No_Overflow_Realization;
```

Listing 5.4: Facility Declarations in the Broadcast_Data Application

In Listing 5.6, the `Main` procedure is defined on lines 1-10. The variables `Garbage`, `Sample`, and `Average` are declared as `Integers`. `Garbage` is used to store the dequeued entry (to be discussed) from queue; `Sample` is used to store a sensor data read from the ADC; and `Average` is used to store the average value calculated over the entries in the queue. The variable `On` is declared as `Booleans`, used to set the LED state. `Data_Samples` is declared as a variable of type `IQF.Queue`, where `IQF` is the qualifier of the `Queue` type (this is the facility name declared on line 4 of Listing 5.4). It is used to store the collected sensor data to be averaged. Lines 6-10 initialize LED components, the ADC component, the UART component, and the XBEE radio component.

```
1 #include "Common.h"
2 #include "Integer_Template.h"
3 #include "Boolean_Template.h"
4 #include "Leds_Template.h"
5 #include "ADC_Template.h"
6 #include "UART_Template.h"
7 #include "XBEE_Template.h"
8 #include "IQF.h"
9 #include "IQF_Averaging_Capability.h"
```

Listing 5.5: Translated Facility Declarations in `Broadcast_Data` Application in C

```
 1 Operation Main();
 2 Procedure
 3     Var Garbage, Sample, Average : Integer;
 4     Var On: Boolean;
 5     Var Data_Samples: IQF.Queue;
 6     LED0_Init();
 7     --Similarly call Init for LED1, LED2, LED3, and LED4
 8     ADC_Init();
 9     UART_Init(9600);
10     XBEE_Init();
```

Listing 5.6: Variable Declaration and Component Initialization in the `Broadcast_Data` Application

The translated C code is shown in Listing 5.7. As discussed in Section 4.3, all local variables declared within `Main` are translated to global variables. These variables are initialized to their default values inside the main function. Note that the queue variable is initialized by calling `iqf_queue_initialize`, generated at the time of concept instantiation.

Listing 5.8 shows the program logic that reads and broadcasts sensor data within a non-terminating `While` loop. The `changing` clause lists the variables that may change during an iteration of the `While` loop. For example, the variable `Sample` is assigned a sensor value from the ADC component within the loop. The `maintaining` clause specifies the loop invariant. In this case, the value is set to `True`. Lines 5-8, check the remaining capacity of the `Data_Samples` queue. If the queue is full, one sensor reading is dequeued and LED0 is turned off.

```
1 long int garbage;
2 long int sample;
3 long int average;
4 boolean on;
5 queue data_samples;
6 iqf_queue_rep data_samples_rep;
7
8 void broadcast_data_main(){
9     garbage = 0;
10    sample = 0;
11    average = 0;
12    on = FALSE;
13    iqf_queue_initialize(&data_samples_rep);
14    data_samples = &data_samples_rep;
15    leds_template_led0_init();
16    /* Similarly call Init for LED1, LED2, LED3, and LED4 */
17    adc_template_adc_init();
18    uart_template_uart_init(9600);
19    xbee_template_xbee_init();
```

Listing 5.7: Translated C code for Listing 5.6

On Lines 9-12, a new sensor value is read into `Sample` from ADC port 0. This value is enqueued into the queue, and an average of all the entries in `Data_Samples` is calculated and stored in `Average`. To make the data more "readable" using the 5 LEDs on the MoteStack, the `Average` value is divided with 256 and the remainder is set back in `Average` variable. Depending on the remainder value, corresponding LED is set to On and the `Average` value is broadcasted by calling `UART_Send_Bytes_Blocking`. Listing 5.9 shows the translated C code.

```
1  While(On)
2      changing Average,Sample,Garbage,Data_Samples;
3      maintaining True;
4  do
5      If (IQF.Rem_Capacity(Data_Samples) = 0) then
6          IQF.Dequeue(Garbage, Data_Samples);
7          LED0_Set(!On);
8      end;
9      Sample := Read_ADC(0);
10     IQF.Enqueue(Sample, Data_Samples);
11     Average := IQF.Average(Data_Samples);
12     Average := Average mod 256;
13     If (Average >= 0 ) then
14         LED1_Set(On);
15     else
16         LED1_Set(!On);
17     end;
18     --Similarly  >= 64 sets LED2 , >= 128 sets LED3, >= 192 sets LED4
19     UART_Send_Bytes_Blocking(Average);
20     LED0_Set(On);
21 end;
22 end Main;
23 end Broadcast_Data;
```

Listing 5.8: Sensing and Broadcasting Logic in the `Broadcast_Data` Application

```
1 while(on){
2     if(integer_template_are_equal(iqf_rem_capacity(&data_samples), 0)){
3         iqf_dequeue(&garbage, &data_samples);
4         leds_template_led0_set(boolean_template_not(on));
5     }
6     sample = adc_template_read_adc(0);
7     iqf_enqueue(&sample, &data_samples);
8     average = iqf_average(&data_samples);
9     average = integer_template_mod(average, 256);
10    if(integer_template_greater_or_equal(average, 0)){
11        leds_template_led1_set(on);
12    }
13    else{
14        leds_template_led1_set(boolean_template_not(on));
15    }
16    /* Similarly >= 64 sets LED2, >= 128 sets LED3, >= 192 sets LED4 */
17    uart_template_uart_send_bytes_blocking(&average);
18    leds_template_led0_set(on);
19    }
20 }
21 int main(){
22     broadcast_data_main();
23 }
```

Listing 5.9: Translated Sensing and Broadcasting Logic in the `Broadcast_Data` Application in C

5.2 Receive Data Application

The MoteStack receive application implements the functionality to receive the sensor value. The XBEE radio component is used to receive data over a wireless network. Listing 5.10 shows the `Receive_Data` application created as a facility, constituent variables declared inside the `Main` function, and initialization logic for components and variables.

```
1 Facility Receive_Data;
2    uses Std_Integer_Fac, Std_Boolean_Fac, Std_Leds_Fac,
3         Std_UART_Fac, Std_XBEE_Fac;
4
5 Operation Main();
6 Procedure
7    Var Data, Bytes: Integer;
8    Var On: Boolean;
9
10   LED0_Init();
11   --Similarly call Init for LED1, LED2, LED3, and LED4
12   UART_Init(9600);
13   XBEE_Init();
```

Listing 5.10: Facility and Variable Declarations in the `Receive_Data` Application

The variables `Data` is declared as an `Integer` to store received data, and `Bytes` is declared as `Integer` variable to store the total count of integers received. `On` is declared as a `Boolean` variables and are initialized with `True`. The UART component is initialized with 9600 as the baud rate similar as in the case of `Broadcast_Data` application. The XBEE and LED components are also initialized. The translated C code is shown in Listing 5.11.

Listing 5.12 shows the data receiving logic within a non-terminating `While` loop. The `changing` clause lists the variables (`Data` and `Bytes`) that may change during an iteration of the loop. The `maintaining` clause specifies the loop invariant. On line 5, received data is stored in `Data` by calling `UART_Receive_Bytes_Blocking`. The number of bytes received is returned to `Bytes` depending on which `LED0` is set. The value in `Data` is checked against the same data values sent by the `Broadcast_Data` application and corresponding LEDs are set. The translated C code is shown in Listing 5.13.

```
1  #include "Common.h"
2  #include "Integer_Template.h"
3  #include "Boolean_Template.h"
4  #include "Leds_Template.h"
5  #include "UART_Template.h"
6  #include "XBEE_Template.h"
7
8  long int data;
9  long int bytes;
10 boolean on;
11
12 void receiveapplication_main(){
13     data = 0;
14     bytes = 0;
15     on = FALSE;
16     leds_template_led0_init();
17     /* Similarly call Init for LED1, LED2, LED3, and LED4 */
18     uart_template_uart_init(9600);
19     xbee_template_xbee_init();
```

Listing 5.11: Translated Facility and Variable Declarations in the Receive_Data Application in C

```
1 While(On)
2     changing Data, Bytes;
3     maintaining True;
4 do
5     Bytes := UART_Receive_Bytes_Blocking(Data);
6     If (Bytes = 0) then
7         LED0_Set(!On);
8     else
9         LED0_Set(On);
10    end;
11    If (Data >= 0 ) then
12        LED1_Set(On);
13    else
14        LED1_Set(!On);
15    end;
16    -- Similarly  >= 64 sets LED2, >= 128 sets LED3, >= 192 sets LED4
17    LED0_Set(!On);
18 end;
19 end Main;
20 end Receive_Data;
```

Listing 5.12: Data Receiving Program Logic in the Receive_Data Application

```
1  while(on){
2      bytes = uart_template_uart_receive_bytes_blocking(&data);
3      if(integer_template_are_equal(bytes, 0)){
4          leds_template_led0_set(boolean_template_not(on));
5      }
6      else{
7          leds_template_led0_set(on);
8      }
9      if(integer_template_greater_or_equal(data, 0)){
10         leds_template_led1_set(on);
11     }
12     else{
13         leds_template_led1_set(boolean_template_not(on));
14     }
15     /* Similarly >= 64 sets LED2, >= 128 sets LED3, >= 192 sets LED4 */
16     leds_template_led0_set(boolean_template_not(on));
17     }
18 }
19 int main(){
20     receive_data_main();
21 }
```

Listing 5.13: Translated Data Receiving Program Logic in the Receive_Data Application in C

The Broadcast_Data and Receive_Data application programs serve as useful test cases to validate sense, broadcast, and receive operations. As a part of validation process, each transmitted sensor value and the received sensor value were compared; the results were as expected. Since these are the basic operations of any embedded networked application, testing the correctness of the translated C programs on the MoteStack device increases our confidence of being able to develop more complex programs.

5.3 Optimization Results

To evaluate the optimizations applied in the compiler translation strategies, three test case application programs were created using the RESOLVE language and translated to C. The translated code was then installed on MoteStack devices to conduct the experiments. In this section, we explain

the experimental goals, the experimental setup and evaluation results.

5.3.1 Experimental Goals

We pursue the following experimental goals.

- **GOAL I**: *To evaluate the efficacy of the optimization strategy implemented to handle scalar variables and constants.* This goal is important because the optimization strategy is intended to decrease the overall memory usage of the translated C programs - a precious resource on resource-constrained embedded devices.
- **GOAL II**: *To evaluate the efficacy the optimization strategy implemented for lazy array initialization.* As explained in Section 4.4, the basic array translation strategy generates an array of equal size to the declared array and initializes all array locations. The optimized strategy as explained in Section 4.7.2, creates a supplemental bit array of equal dimension to the original array and introduces an initialization check each time an array index is accessed. This goal is important because the optimization strategy is intended to improve the runtime efficiency of translated C programs and guarantee constant-time initialization.

5.3.2 Experimental Setup

The experimental setup consists of two MoteStack embedded devices configured with LEDs, an ATMEGA644 microprocessor [11], and an XBEE radio module [44]. The test case programs were created using the RESOLVE language. They are compiled, verified, and translated to C programs using two versions of the compiler: Version 1 implements optimized translation strategies and version 2 implements non-optimized translation strategies.

The test cases for Goal I include the translated C code for `Broadcast_Data` and `Receive_Data`, described in Sections 5.1 and 5.2, respectively. The test case for Goal II is the RESOLVE code shown in Listing 4.27, which declares an array `A` of 100 `Integers` and assigns value at index 20 to the variable `I`. These translated programs are installed on the MoteStack embedded device using AVR Studio [12] and an AVRISP mkII In-System Programmer [13].

5.3.3 Experimental Results

- **Experiment I:** In this experiment, Goal I is targeted. Both the optimized code and the non-optimized code are compiled. The memory usage in both cases is shown in Table 5.1. The results show that the optimized code uses significantly less memory when compared to the non-optimized code.

	No Optimization		*With Optimization*	
Application	**Program (ROM)**	**Data (RAM)**	**Program (ROM)**	**Data (RAM)**
Broadcast_Data	10860	686	6990 (∼64.37%)	282 (∼41.11%)
Receive_Data	9634	432	5804 (∼60.25%)	126 (∼29.17%)

Table 5.1: Memory Usage of Applications Compiled for the ATMEGA644 Processor

- **Experiment II:** In this experiment, Goal II is targeted. The execution times are calculated for the optimized and non-optimized C code generated from the RESOLVE code in Listing 4.27. The execution times are compared for different array sizes. The results are shown in Figure 5.1. The vertical axis measures the execution time in microseconds and the horizontal axis measures the size of the array declared within the test program. The results show that lazy initialization yields better execution times as the size of the array increases, as expected.

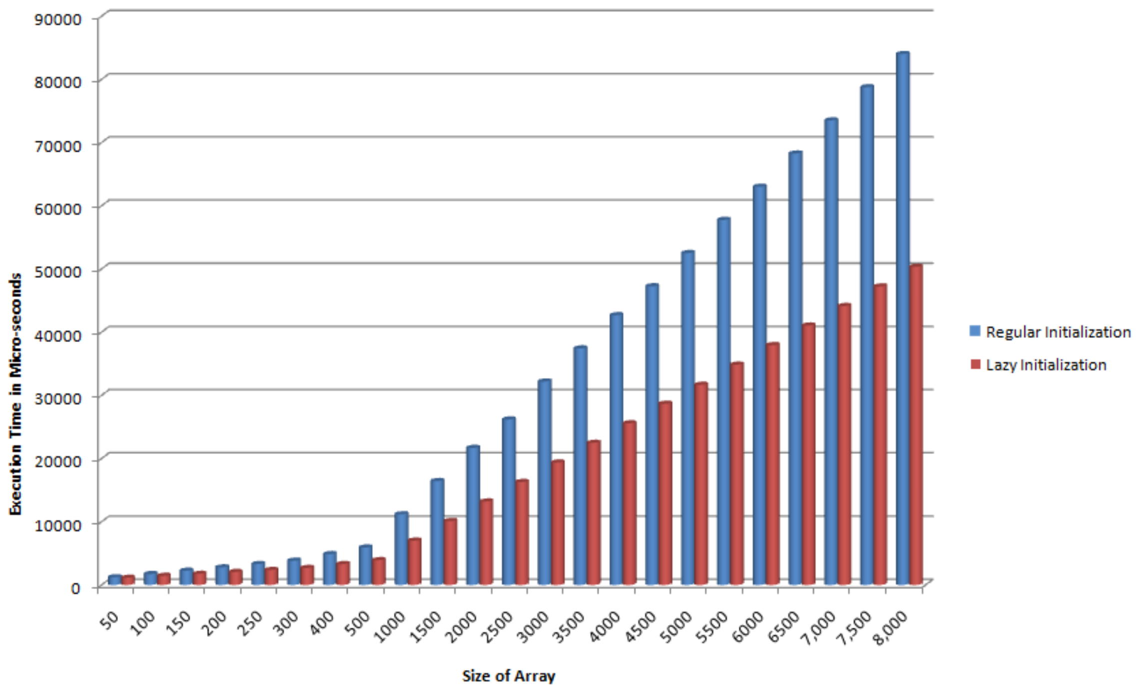


Figure 5.1: Execution Times for Basic and Lazy Array Initialization Strategies

Chapter 6

Conclusions

Embedded Networked Systems play a vital role in providing information to decision support systems in various fields such as healthcare, environmental monitoring, and others. These systems consist of networks of devices distributed in large numbers that can collect data and communicate with other devices within the network. Since embedded devices are resource-constrained, developers face challenges in designing systems with long life that can provide reliable data. When compared to other systems, software correctness is more critical for embedded networked systems as they involve significant maintenance costs, and errors can lead to loss of life. Our work focuses on realizing verified programs for embedded networks based on specifications and implementations written in the RESOLVE language. We extended the RESOLVE verifying compiler with a translator module used to generate verified C language programs that can be installed on an embedded device.

We provided an overview of the operators, datatypes, and modules of the RESOLVE language, and then described the strategies implemented by our translator module to generate equivalent C source code. The translator supports standard datatypes (`Integers`, `Booleans`, and `Characters`), arrays, and operators including `Swap`. We implemented a generic swap function using void pointers to support constant time data movement for all object types. One of the key features of the RESOLVE language is that it supports parameterized specifications (`Concepts`) and implementations (`Realizations`). Each specification is instantiated by passing its actual parameter values using RESOLVE `Facilities`. The translator module generates a unique object instance of a specification for each set of parameters. It also supports the extension modules (`Enhancements`) provided by RESOLVE. To enhance the efficiency of the translator, we imple-

mented two optimizations for translating variable declarations and array initializations, respectively.

Finally, we presented the results of experiments conducted to validate the translation strategies with test case programs implementing basic operations. We evaluated the variable declaration optimization using the same test case programs. We also evaluated the array initialization optimization with a test program using arrays. The results show that the optimizations offer decreased memory usage and better runtime efficiency, respectively. In future, we would like to provide initialization function for each type, provide better support for specifying drivers, and integrate the project with RESOLVE web tool to provide translated C code to all users.

Bibliography

[1] The perfect developer language reference manual. http://www.eschertech.com/papers/pdpaper .pdf.

[2] R. Acharya and K. Asha. Data integrity and intrusion detection in wireless sensor networks. In *Networks, 2008. ICON 2008. 16th IEEE International Conference on*, pages 1–5, Dec. 2008.

[3] A. Arora, E. Ertin, R. Ramnath, M. Nesterenko, and W. Leal. Kansei: A high-fidelity sensing testbed. *IEEE Internet Computing*, 10(2):35–47, 2006.

[4] The Motor Industry Softwre Reliability Association. Guidelines for the use of the c language in vehicle based software, 1998.

[5] R.D. Banker, S.M Datar, C.F Kemerer, and D. Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, 1993.

[6] J. Beutel, M. Dyer, R. Lim, C. Plessl, M. Wohrle, M. Yucel, and L. Thiele. Automated wireless sensor network testing. pages 303 –303, june 2007.

[7] S. Blazy. Experiments in validating formal semantics for c. In *C/C++ Verification Workshop*, pages 95–102, Oxford United Kingdom, 2007.

[8] P. Bucci, J.E. Hollingsworth, J. Krone, and B.W. Weide. Part iii: Implementing components in resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):40–51, 1994.

[9] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[10] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 142–151, New York, NY, USA, 2001. ACM.

[11] Atmel Corporation. Atmega644: 8-bit avr microcontroller with 64k bytes in-system programmable flash. http://www.atmel.com/dyn/resources/prod_documents/doc2593.pdf.

[12] Atmel Corporation. Avr studio 4 ide for 8-bit avr applications. http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725.

[13] Atmel Corporation. Avrisp mkii in-system programmer. http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3808.

[14] Moteiv Corporation. Tmote sky: Datasheet. http://sentilla.com/files/pdf/eol/tmote-sky-datasheet.pdf, 2006.

[15] P. Cousot and R. Cousot. Verification of embedded software: Problems and perspectives. 2211:97–113, 2001.

[16] D. Crocker and J. Carlton. A high productivity tool for formally verified software development. http://www.eschertech.com/product_documentation/LanguageReference/ LanguageReference-Manual.htm.

[17] D. Crocker and J. Carlton. Verification of c programs using automated reasoning. In *SEFM '07: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, pages 7–14, Washington, DC, USA, 2007. IEEE Computer Society.

[18] A.R. Dalton, S. Dandamudi, J.O. Hallstrom, and S.K. Wahba. A testbed for visualizing sensornet behavior. In *The Proceedings of The 17th International Conference on Computer Communications and Networks (IC3N'08)*, pages 1–7, Washington DC, USA, August 2008. IEEE Computer Society.

[19] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 15–28, New York, NY, USA, 2006. ACM.

[20] S.H Edwards, W.D Heym, T.J Long, M. Sitaraman, and B.W Weide. Part ii: Specifying components in resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):29–39, 1994.

[21] G.W. Eidson, S.T. Esswein, J.B. Gemmill, J.O. Hallstrom, T.R. Howard, C.J. Post, C.T. Sawyer, K.C. Wang, and D.L. White. The south carolina digital watershed: End-to- end support for realtime management of water resources. In *The 4th International Symposium on Innovations and Real-time Applications of Distributed Sensor Networks*, pages 9–16, Los Alamitos, CA USA, May 2009. IEEE.

[22] J. Filliâtre. The 'why' verification tool. http://why.lri.fr/index.en.html.

[23] J. Filliâtre and C. Marché. Multi-prover verification of c programs. *Formal Methods and Software Engineering*, pages 15–29, 2004.

[24] M. Fujita. Formal verification of c language based vlsi designs. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 93–100, 2004.

[25] M. Gallardo, P. Merino, and D. Sanán. Towards model checking c code with open/cæsar. In *Proceedings of the 4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2006*, pages 198–201, 2006.

[26] Y. Hanna, H. Rajan, and W. Zhang. Slede: A domain-specific verification framework for sensor network security protocol implementations. In *WiSec '08: Proceedings of the first ACM conference on Wireless network security*, pages 109–118, New York, NY, USA, March 31 – April 2 2008. ACM.

[27] D.E Harms and B.W Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Softw. Eng.*, 17(5):424–435, 1991.

[28] L. Hatton. Safer c: Developing software for high-integrity and safety-critical systems, 1995.

[29] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[30] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[31] G.J. Holzmann. Economics of software verification. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 80–89, New York, NY, USA, 2001. ACM.

[32] C. Huang, Y. Tseng, and H. Wu. Distributed protocols for ensuring both coverage and connectivity of a wireless sensor network. *ACM Trans. Sen. Netw.*, 3(1):5, 2007.

[33] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.

[34] A. Ireland. A practical perspective on the verifying compiler proposal. In *Proceedings of the Grand Challenges in Computing Research Conference, BCS, IEE and UKCRC, 2004*, 2004.

[35] F. Ivanicic, I. Shlyakhter, A. Gupta, M.K Ganai, P. Ashar, and Z. Yang. F-soft: Software verification platform. In *Computer-aided Verification, 2005.* Springer, 2005.

[36] F. Ivanicic, I. Shlyakhter, A. Gupta, M.K Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking c programs using f-soft. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 297–308, Washington, DC, USA, 2005. IEEE Computer Society.

[37] M.M.H. Khan, H.K Le, H. Ahmadi, T.F Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 99–112, New York, NY, USA, 2008. ACM.

[38] T. Kim, J. Kim, S. Lee, I. Ahn, M. Song, and K. Won. An automatic protocol verification framework for the development of wireless sensor networks. In *TridentCom '08: Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, pages 1–5, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[39] J. Kirschenbaum, H. K. Harton, and M. Sitaraman. A case study in automated verification. Technical Report RSRG-08-04, School of Computing, Clemson University, Clemson, SC, USA, June) 2008.

[40] G.W. Kulczycki. Direct reasoning, January 2004.

[41] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a c0 compiler: Code generation and implementation correctnes. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 2–12, Washington, DC, USA, 2005. IEEE Computer Society.

[42] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[43] Z. Manna, A. Anuchitanukul, N. Bjorner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H Devarajan, H Sipma, and T. Uribe. Step: The stanford temporal prover. Technical report, Stanford, CA, USA, 1994.

[44] MaxStream. Xbee^{TM}/xbee-pro^{TM} oem rf modules. http://ftp1.digi.com/support/documentation /manual_xb_oem-rf-modules_802.15.4_v1.xAx.pdf.

[45] W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying c compiler. In *C/C++ Verification Workshop*, July 2007.

[46] B. Sharma, S.D. Dhodapkar, and S. Ramesh. Assertion checking environment (ace) for formal verification of c programs. In *SAFECOMP '02: Proceedings of the 21st International Conference on Computer Safety, Reliability and Security*, pages 284–295, London, UK, 2002. Springer-Verlag.

[47] H. Smith, H. Harton, D. Frazier, R. Mohan, and M. Sitaraman. Generating verified java components through resolve. In *ICSR '09: Proceedings of the 11th International Conference on Software Reuse*, pages 11–20, Berlin, Heidelberg, September 2009. Springer-Verlag.

[48] Clemson Univeristy The RESOLVE Research Group. Resolve verication web tool. http://resolve.cs.clemson.edu/.

[49] B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 67, Piscataway, NJ, USA, 2005. IEEE Press.

[50] A. Tiwari, P. Ballal, and F.L Lewis. Energy-efficient wireless sensor network design and implementation for condition-based maintenance. *ACM Trans. Sen. Netw.*, 3(1):1, 2007.

[51] H. Tuch. Formal verification of c systems code. *J. Autom. Reason.*, 42(2-4):125–187, 2009.

[52] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: a wireless sensor network testbed. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 68, Piscataway, NJ, USA, 2005. IEEE Press.