Verified Generation of Glue Code for ROS-based Control Systems[*]

Wenrui Meng, Junkil Park, Oleg Sokolsky, Stephanie Weirich, and Insup Lee

University of Pennsylvania

Abstract. The paper considers the problem of automatic generation of platform-specific glue code for platform-independent controller code. We present a code generator, ROSGen that generates the glue code based on a declarative specification of platform interfaces. Our implementation targets the popular Robot Operating System (ROS) platform. We demonstrate the code generation process is amenable to formal verification. The code generator is implemented in Coq and relies on the infrastructure provided the CompCert and VST tool. We prove that the generated code always correctly connects the controller function to sensors and actuators in the robot. We use ROSGen to implement a cruise control system on the LandShark robot.

1 Introduction

Modern cyber-physical systems are typically constructed from separately developed components. A system architectural model of the system is used to describe the necessary components and relationships between them. Once individual components are developed, they are deployed on a middleware platform that can be configured to implement connections between the components according to the architecture.

We envision a model-based approach that targets both component development as well as deployment. Model-based development is a promising approach for developing safety critical systems, in particular autonomous robotic systems. It has the potential to detect and avoid design errors during the early phases of system development. In developing components, component behaviors are abstractly specified by data models, state charts, or diagrams. These diagrams can be expressed using design tools such as Simulink/Stateflow [1], UPPAAL [2], or SCADE/Lustre [3]. Code generation tools then convert these diagrams into code, typically platform-independent C source code. Such generative approach helps us to preserve properties verified at the modeling level, making sure that component implementation also satisfies these properties.

The second phase of the model-based development approach is to deploy the generated platform-independent code of each component on the chosen platform.

[*]This research is supported in part by DARPA HACMS program under agreement FA8750-12-2-0247. The views expressed are those of the authors and may not reflect the official policy or position of the Department of Defense or the U.S. Government.

An architectural model of the system specifies the desired platform configuration. The architectural model specifies (1) how each component should be executed (for example, periodic execution with a given period may be specified), (2) how system inputs, such as sensor streams, should be routed to inputs of components processing the streams, and (3) how outputs of each component should be routed to inputs of other components or to system outputs (such as actuators and operator displays). A faulty deployment undermines the benefits of provably correct implementation of individual components. Platform configurations, therefore, should be automatically generated from the architectural model to ensure correct integration of individual components.

A significant part of platform configuration is providing a platform-specific wrapper for the platform-independent component implementation. The wrapper (also known as glue code) uses platform APIs to schedule component execution and to obtain inputs for the component and forward its outputs.

In this paper, we address the problem of automatically generating provably correct glue code for a particular deployment platform from a given architectural model. We use the Robot Operating System (ROS) [1] as our target platform, a "thin, message-based, peer-to-peer" [4] robotics middleware designed for mobile manipulators. The ROS platform has recently gained popularity in the robotics community, because it raises the level of abstraction in embedded control system development. ROS-based applications are assembled from multiple ROS nodes that run concurrently. ROS supports communication between these nodes using a publish/subscribe-based message system.

To that end, we develop ROS glue code Generator (ROSGen), a tool that automatically generates such glue code from system architecture specifications. The input language for our code generator is a domain-specific language, called a `ROS node model`, that specifies the ROS nodes that comprise the system and ROS topics that the nodes subscribe to and publish on.

Of course, by generating code we eliminate some sources of programmer error in system development. However, for safety critical systems, we want the highest level of assurance. We would like to prove that output of our code generator satisfies strong correctness and safety requirements. One can take two approaches for the verification of generated code; first, one may verify every output individually. Alternatively, which is generally much harder, one may verify the code generator itself.

Our code generator is designed to support formal verification. ROSGen is implemented using the Coq proof assistant [5], making the full higher-order logic of Coq available for reasoning about both the output of the generator (represented as a Coq data structure) and the code generator itself (represented as a Coq function). In this context, we have used both approaches to verification.

We have applied ROSGen as part of a case study of glue code generation for the Black-i Robotics LandShark platform. The LandShark is an unmanned ground vehicle typically used to extend human capabilities, often in dangerous environments such as at a chemical spill or for sentry duty. ROSGen can generate

[1]www.ros.org

glue code for this platform, and we have proven that the generated code satisfies a crucial `Data Delivery Correctness (DDC)` property: that the arriving sensor message will be correctly delivered to the control function and that the output of the control function will be correctly delivered to the actuators. We express and prove this property using the Verified Software Toolchain (VST) tool [6], which provides a higher-order separation logic for reasoning about memory usage in C programs. Our proof has been mechanically checked by Coq.

Moreover, we were able to prove the correctness of the code generator itself. That is, we can show that `every` output of ROSGen satisfies the same `DDC` property that we have shown for the LandShark instance. In general, this is a very hard problem. However, in our case, because of the relatively simple code structure and the property of interest that is concerned with data transfer, we were able to generalize the proof of instances of the generated code to the proof of the generator itself.

In summary, this paper makes the following contributions:

– We introduce a domain specific language for describing the ROS nodes. We develop a code generator ROSGen to generate the robotics glue code according to a given ROS node model (Section 4).
– We demonstrate an application of ROSGen to a case study of a robotic control system and prove, using a suite of Coq-based tools, that the glue code correctly delivers data according to the ROS node model of the controller (Section 5).
– Finally, we verify that, given a well-formed ROS node model, ROSGen always generates code that satisfies the data delivery correctness property (Section 6).

The rest of the paper is organized as follows. We introduce the relevant work that our code generation is dependent on in Section 2. Section 3 explains the architecture of the ROS based control system and introduces the LandShark case study. In Sections 4, 5 and 6, we explain our code generation approach for ROS based control system and the verification for the generated code and code generator itself. We discuss related work in Section 7 and conclude in Section 8.

The Coq implementation of the code generator can be downloaded from `http://rtg.cis.upenn.edu/HACMS/codegen.html`. The relevant parts of the case study, including the ROS node model and complete generated code, can also be found on the same web page.

2 Proof Environment

Figure 1 shows the tools underlying ROSGen, which are briefly described below.

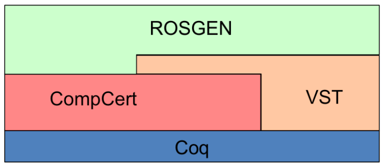

Fig. 1. ROSGen dependency structure

2.1 Coq

Coq[2] is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs.

2.2 CompCert

CompCert [7] is a formally verified optimizing compiler for the C programming language that currently targets PowerPC, ARM and 32-bit x86 architectures. The compiler is specified, implemented and proved correct using the Coq proof assistant. It targets embedded systems programming, with stringent reliability requirements. CompCert's source language, a large subset of C called Clight, is the target language of our code generator; our generator produces abstract syntax values for Clight.

The formal semantics of Clight is mechanized using Coq. It supports types including integral types (integers and floats in various sizes and signedness), array types, pointer types (including pointers to functions), function types, as well as struct and union types. A Clight program is composed of a list of declarations for global variables (name and type), a list of functions and an identifier naming the entry point of the program (the main function in C).

2.3 Verified software toolchain

The goal of the Verified Software Toolchain (VST[3]) project is to verify that the assertions claimed at the top of a software toolchain really hold in the machine language program, running in the operating system context, on a weakly-consistent-shared-memory machine. It defines `Verifiable C`, a higher-order concurrent separation logic for Clight. `Verifiable C` has been proven sound with respect to the operational semantics of CompCert C [6].

The `Verifiable C` program logic extends Hoare logic by including separation logic constructs to support reasoning about mutable data structures such as arrays and pointers. In separation logic, an assertion holds on a particular subheap and assertions on different subheap are independent. As a result logical

[2]http://coq.inria.fr/
[3]http://vst.cs.princeton.edu/

reasoning is modular. VST provides a tactic system for proving correctness properties, specified by the VST assertions, of C light programs. The most significant of these are the `forward` tactic, which symbolically executes the code, and the `entailer` tactic, which simplifies and often solves VST assertions [8].

3 ROS-based control system

3.1 Robot operating system

ROS is a widely used middleware for robotic system applications. ROS is component-based. A software component in ROS is called a ROS node. A ROS application usually consists of multiple ROS nodes running concurrently. The ROS nodes asynchronously communicate with each other. Communication in ROS is based on the Publish/Subscribe paradigm and uses structured message types. ROS Services are the mechanism to implement remote procedure calls in ROS, which are synchronous and blocking. ROS also provides services for synchronous, blocking communication, which are not considered in this paper.

```
void callback(MessageType msg) { ... };
main(){
  Subscribe(..., callback);
  Advertise(...);
  while( ros_ok() ){
    SpinOnce();
      /* Process the input to the controller */
    Controller_step();
      /* Process the output of the controller */
    Publish(...); }}
```

Fig. 2. ROS-based controller system skeleton

Figure 2 shows the skeleton of a ROS-based control system. In order to subscribe to a topic in ROS, users need to define a callback function. A callback function for a topic is a message handler that is invoked to process the new messages when they arrive. `Subscribe` is a function from the ROS API that registers subscription information: a topic name, the message type, the internal buffer size and the callback function for those messages. If a new message is received, it is stored in an internal buffer. It replaces the oldest message in the buffer if the buffer is already full. When the ROS API function `SpinOnce` is invoked, all registered callback functions are invoked for every message in the internal buffers. In order to publish a topic in ROS, users should use ROS API `Advertise` first to create a publisher with topic name, message type, and internal buffer size. ROS API `Publish` is used to publish a message.

ROS C Wrapper. ROS APIs described above are available as a C++ library. However, our proof environment currently supports only C language. We built a minimal wrapper that allow us to access APIs as C functions, and also allows access to ROS message types as C data structures.

3.2 Case study of LandShark control system

In this section we illustrate a typical ROS-based control system using the LandShark robot. The LandShark is an electric unmanned ground vehicle, shown in Figure 3, manufactured by Black-I Robotics.[4] Our case study develops a constant-speed cruise control algorithm that is resilient to attacks on vehicle sensors. The LandShark uses three sensors, GPS, a left wheel encoder and a right wheel encoder, to estimate its current velocity. These sensors can be compromised by attacks, such as GPS spoofing, that cause confusion in estimating the current velocity of the vehicle. The attack-resilient cruise controller of LandShark uses multiple independent sensors and the knowledge of the system model in order to correctly estimate the current velocity of the vehicle and drive the vehicle with a given constant velocity [9].

Fig. 3. LandShark robot

Figure 4 shows the architecture of the LandShark control system, which consists of sensor/actuation/controller nodes and connection between them through topic-based pub/sub communication. The ROS nodes landshark_gps and landshark_base are associated with sensors that read GPS and wheel encoder values respectively and publish them. The ROS node landshark_wheel_velocity subscribes the series of wheel encoder values and publishes the velocity of vehicle calculated from them. The ROS node landshark_base also plays a role as an actuation node in that it subscribes the actuation commands and actuates the vehicle according to them. The ROS node landshark_controller is the controller

[4]http://www.blackirobotics.com/

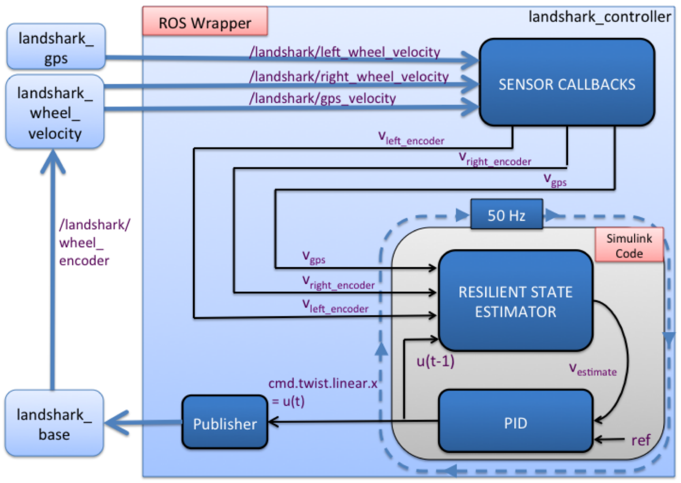

Fig. 4. LandShark control system architecture

node for landshark that subscribes sensor value messages and publishes actuation commands. The landshark_controller node is periodically invoked at the rate of 50 Hz to execute the Simulink-generated step function. In each invocation, the callback functions are invoked by `SpinOnce` to process the messages received. The callback functions store the sensor messages in global variables. The sensor values in the global variables are transferred to the input data structure of the control algorithm function that is generated by Simulink. The step function is executed to calculate the actuation command, which is encapsulated in a ROS message variable and published by the publisher.

4 Code Generation

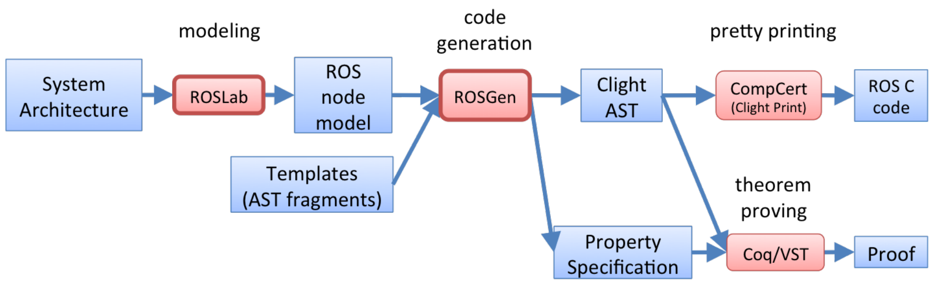

Fig. 5. Verified code generation toolchain

Our toolchain for verified code generation appears in Figure 5. The ROSLab tool supports the design of system architectures. The diagram block in ROSLab can be exported as a ROS node model. With the ROS node model, ROSGen produces an abstract syntax tree for a subset of C called Clight, by instantiating a Clight AST template. In addition, ROSGen also generates a VST specification for each function, describing its `DDC` properties. We can prove that the generated code satisfies these specifications, as we demonstrate in Section 6. The final C code, which is run on the LandShark, is produced by the CompCert compiler using its pretty printer.

4.1 ROSLab tool

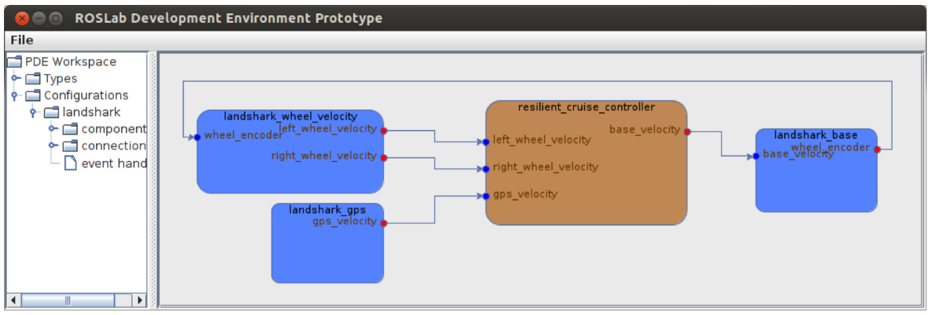

Fig. 6. LandShark system diagram in ROSLab

ROSLab is a modular programming environment for robotic applications based on ROS. Figure 6 shows a screenshot of ROSLab. ROSLab enables users to model an architecture of an ROS application that consists of a set of ROS nodes and the connection based on pub/sub between them. The interfaces of some commonly used ROS nodes such as sensor and actuator nodes are pre-defined in ROSLab. Users can define a new ROS node and its interface by selecting the pub/sub channels to add to the interface of the node. ROSLab automatically generates the glue C code of a ROS node and the glue code to interface with a Simulink generated code.

4.2 ROS node model

A diagram block in ROSLab can be exported as ROS node model. A ROS node model includes the period at which the node is to be invoked; the list of topics that the node publishes or subscribe to; the name and the I/O interface of the controller function that the node will run; finally, a mapping from subscribed and published topics to inputs and outputs of the controller function.

The ROS node model for the `landshark_controller` ROS node in Figure 4 is shown in Table 1. The name of the node, the period of the controller, and the

Node Information				
period	node name		controller name	
20	landshark_controller		Controller	
ROS Topics				
type	topic name	message package	message type	buffer size
S	/landshark/left_wheel_velocity	geometry_msgs	TwistStamped	1
S	/landshark/right_wheel_velocity	geometry_msgs	TwistStamped	1
S	/landshark/gps_velocity	geometry_msgs	TwistStamped	1
P	/landshark_control/base_velocity	geometry_msgs	TwistStamped	1
Controller Interface				
I/O	name	record type		
I	Controller_U	(In1, double), (In2, double), (In3, double)		
O	Controller_Y	(Out1, double)		
Interface Relation				
type	topic		controller	
SI	/landshark/left_wheel_velocity, twist, linear, x		Controller_U, In1	
SI	/landshark/right_wheel_velocity, twist, linear, x		Controller_U, In2	
SI	/landshark/gps_velocity, twist, linear, x		Controller_U, In3	
PO	/landshark_control/base_velocity, twist, linear, x		Controller_Y, Out1	

Table 1. ROS node model for LandShark

name of the controller function that the node will execute are shown at the top of the table. Published topics are indicated by the letter P and subscribed topics are indicated by the letter S. For each topic, the unique topic name and the type of messages are given. Next, the ROS node model specifies the controller function interface. The function, in our case study, is generated from the Simulink model of the controller, and names and types of input and output variables are following the Simulink code generator conventions. Finally, the interface relation represents the mapping from relevant fields of subscribed sensor messages to the fields in the input data structures of the controller function, and similarly for outputs of the controller function to published actuator messages.

4.3 ROSGen

4.3.1 Symbol table As the first step in code generation, ROSGen constructs a Coq data structure representing symbols to be used in the generated code. The names are obtained by parsing the ROS node model. Types for the controller function interface are given in the node model. Types for ROS messages reference in the node model are obtained by parsing the corresponding C header files.

4.3.2 Code templates Code generation proceeds by instantiating templates that are Clight AST fragments. We introduced a top-level template, representing the whole program, and a set of local templates. The top-level template is shown in Figure 7. The program contains a list of global definitions and the name for main function. A global definition can be either a variable definition or a function definition. One of the global definitions is the definition of the main function, which is partially constructed in the top-level template. Light-colored triangles in the top-level template represent holes that are filled with instantiations of local templates. Local templates are used to capture global definitions, such as callback function definitions, global variables used to transfer data from callback functions to the main function, and also glue code functions explained in more detail below. Holes in local templates can represent statements, as well as variable ids and types that are filled with references to the symbol table. Once all the templates are instantiated, the final C code is produced by CompCert pretty printing.

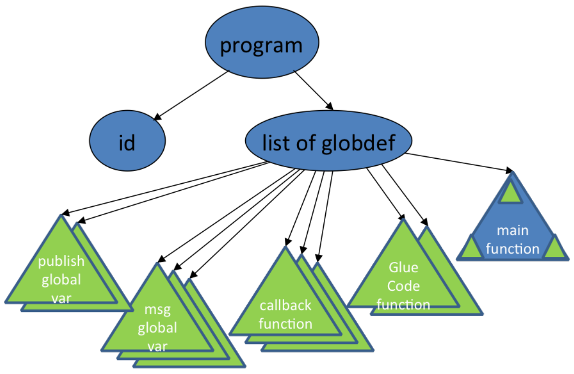

Fig. 7. Top-level Template

To make proofs more efficient, we modularize the body of the main function from Figure 2 into several functions. The while loop is encapsulated as loop function. Within the loop function, we wrap the code for transferring data from global variable to controller input and controller output to publish input as input_glue and output_glue function, respectively. Figure 8 shows the generated code for the glue functions.

5 Code Proof

We use the VST to make proof for `DDC` property of the generated Clight AST. Since VST is based on the programming axiom semantic, we specify the `DDC` property by storing the original value and checking the relation of the destination value and original value.

5.1 Data delivery correctness property of glue code

The main purpose of the ROS glue code is linking the sensor input, controller function and actuator, so the critical property of glue code should capture the

```
void input_glue(){
  double temp;
  temp = landshark_left_encoder_velocity_msg.twist.linear.x;
  Controller_U.In1 = temp;
  temp = landshark_right_encoder_velocity_msg.twist.linear.x;
  Controller_U.In2 = temp;
  temp = landshark_gps_velocity_msg.twist.linear.x;
  Controller_U.In3 = temp;
  return;
}
void output_glue(TwistStamped *command){
  double temp;
  temp = Controller_Y.Out1;
  command->twist.linear.x = temp;
  return;
}
```

Fig. 8. Input and output glue functions

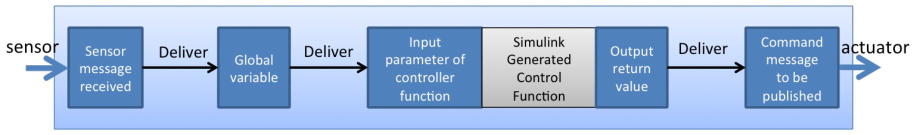
sensor
Sensor message received
Deliver
Global variable
Deliver
Input parameter of controller function
Simulink Generated Control Function
Output return value
Deliver
Command message to be published
actuator

Fig. 9. Data delivery correctness property for ROS-based control system

correctness of the linking. In ROS glue code, the linking correctness means that the sensor message is delivered into controller function input correctly. In addition, the output of the controller function correctly is delivered into the actuator input. We specify the linking correctness property of the ROS glue code as a `DDC Property`. This property indicates that the information from the origin should be consistent with the system specification when it arrives at the destination. For example, we design the system in the way that the sensor message is directly stored into global variables. So the `DDC` property of this operation is that the original value of the sensor message is equal to the value of the updated global variable. We can also do some arithmetical transformation on the original value, then the `DDC` property should specify the relation of the original value and destination value according to the arithmetical operation.

5.2 Generating function specifications

ROSGen automatically generates VST function specifications according to ROS node model for both generated functions and ROS API functions. In VST, users specify properties through function specifications, so we wrap our glue code as functions. This glue code includes callback functions, as well as input and output glue functions for the controller step. In addition, modularizing the code as separate functions is also helpful for VST proof efficiency, since it reduces the number of memory predicates involved in the proof. So we also wrap the while loop statements in the main function as a separate loop function.

As shown in Figure 9, the specifications of the callback functions, input and output glue functions capture the DDC property of the generated AST instance. The callback functions are responsible for transferring sensor messages to global message variables; the input glue function is responsible for transferring global message variable to the input parameter of controller function; and the output glue function is responsible for transferring output of controller function to the parameters of publish function. For each part, the DDC property specification defines the precondition that the original value is stored in the memory and the post condition that the destination contains the desired value according to the original value. For example, Figure 10 defines the specification of the callback

```
Definition landshark_left_encoder_velocity_callback_spec :=
 DECLARE _landshark_left_encoder_velocity_callback
  WITH sh : share, data : reptype' t_struct_TwistStamped, msg_val : val
  PRE [_msg OF (tptr t_struct_TwistStamped)]
     PROP(writable_share sh)
     LOCAL (`(eq msg_val) (eval_id _msg))
     SEP (`( data_at sh t_struct_TwistStamped (repinj _ data)) `(msg_val);
              `( data_at_ sh t_struct_TwistStamped)
               (eval_var _landshark_left_encoder_velocity_msg
                    t_struct_TwistStamped))
  POST [ tvoid ]  PROP() LOCAL()
      SEP (`( data_at sh t_struct_TwistStamped (repinj _ data)) (`msg_val );
                `( data_at sh t_struct_TwistStamped (repinj _ data))
                 (eval_var _landshark_left_encoder_velocity_msg
                       t_struct_TwistStamped)).
```

Fig. 10. Callback function specification

function shown in Figure 11, which copies the value from the sensor message to the global variable. In this definition, the keyword DECLARE relates the specification to the function id which is _landshark_left_encoder_velocity_callback in this example. WITH notation provides names of Coq variables that can be mentioned in both the precondition and the postcondition. It uses data_at to indicate the memory of variable contains some value. This data_at predicate requires the memory access permission which is described as share. It can be a read permission share, write permission share or other permissions. VST uses eval_id to get the memory address of the pointer pointing to and eval_var to get the memory address of a variable. reptype is a function to transfer the Clight data structure type to Coq type, so we can provide Coq data value as witness when calling this function. `repinj` indicates that the value is a initialized value. `PRE` notation gives the precondition of the function, which is parameterized with variable ids and their types. In this example, the precondition indicates that the function holds the writing permission on the memory; the value of the pointer _msg is equal to msg_val; value data is stored in the memory address msg_val; and the global variable _landshark_left_encoder_velocity_msg is stored in memory of address with unknown value. The postcondition, provided by `POST`, indicates that the value stored in the _msg and _landshark_left_encoder_velocity_msg are both data. There are more details about the function specification in the book [8].

```
void landshark_left_encoder_velocity_callback(TwistStamped* msg)
{
  double temp;
  temp = (*msg).twist.linear.x;
  landshark_left_encoder_velocity_msg.twist.linear.x = temp;
  temp = (*msg).twist.linear.y;
  landshark_left_encoder_velocity_msg.twist.linear.y = temp;
  temp = (*msg).twist.linear.z;
  landshark_left_encoder_velocity_msg.twist.linear.z = temp;
  temp = (*msg).twist.angular.x;
  landshark_left_encoder_velocity_msg.twist.angular.x = temp;
  temp = (*msg).twist.angular.y;
  landshark_left_encoder_velocity_msg.twist.angular.y = temp;
  temp = (*msg).twist.angular.z;
  landshark_left_encoder_velocity_msg.twist.angular.z = temp;
  return;
}
```

Fig. 11. Generated callback function C code

The function specifications of input glue and output glue are similar to the callback function. The difference is that the postcondition specifies value of the `Controller_U` of input_glue function and command value of output_glue function according to the interface relation. As shown in Figure 12, we specify for the input_glue function that the value of `Controller_U` is from fields of those three global message variables.

5.3 Specification of ROS API functions.

For the code proof, we have to supply specifications of ROS API functions called by the code. These specifications are treated as assumptions in the proof. Here, specification of SpinOnce presents a challenge. The function implicitly invokes the registered callback functions to update global variables with new sensor values. The straightforward way to specify SpinOnce is to refer to the specifications

```
Definition input_glue_spec :=
  DECLARE _input_glue
  WITH sh : share, data1 : reptype t_struct_TwistStamped,
                   data2 : reptype t_struct_TwistStamped,
                   data3 : reptype t_struct_TwistStamped
  PRE []  PROP() LOCAL()
  SEP(
    `( data_at sh t_struct_TwistStamped data1)
     (eval_var _landshark_left_encoder_velocity_msg t_struct_TwistStamped);
    `( data_at sh t_struct_TwistStamped data2)
     (eval_var _landshark_right_encoder_velocity_msg t_struct_TwistStamped);
    `( data_at sh t_struct_TwistStamped data3)
     (eval_var _landshark_gps_velocity_msg t_struct_TwistStamped);
    `( data_at_ sh t_struct_ExternalInputs_Controller)
     (eval_var _Controller_U t_struct_ExternalInputs_Controller))
  POST [tvoid]
  PROP() LOCAL()
  SEP(
    `( data_at sh t_struct_TwistStamped data1)
     (eval_var _landshark_left_encoder_velocity_msg t_struct_TwistStamped);
    `( data_at sh t_struct_TwistStamped data2)
     (eval_var _landshark_right_encoder_velocity_msg t_struct_TwistStamped);
    `( data_at sh t_struct_TwistStamped data3)
     (eval_var _landshark_gps_velocity_msg t_struct_TwistStamped);
    `( data_at sh t_struct_ExternalInputs_Controller
     ((fst(fst data1), (fst(fst data2), fst(fst data3)))))
     (eval_var _Controller_U t_struct_ExternalInputs_Controller)
).
```

Fig. 12. Input Glue Functions Specification

of the callbacks. However, currently, VST does not support using other function specifications to construct a specification. Therefore, we specify the SpinOnce function using the global variables update directly, as shown in Figure 13, essentially incorporating callback specifications directly into the SpinOnce specification. This specification has the precondition that the global variables are stored somewhere of memory and postcondition that the global variables are updated to the provided data.

5.4 Code proof strategy

We use the tactics from VST proof automation to prove the property specified by the function specification. For each function, the proof starts with the function precondition as proof context. We then apply the VST tactics for the current statement of function body. Each tactic execution updates the proof context by calculating the postcondition of the statement and advances to the next

```
Definition spinOnce_spec :=
 DECLARE _ros_spinOnce
  WITH sh : share, data1 : reptype t_struct_TwistStamped,
                   data2 : reptype t_struct_TwistStamped,
                   data3 : reptype t_struct_TwistStamped
  PRE []  PROP() LOCAL() SEP(
    `( data_at_ sh t_struct_TwistStamped)
     (eval_var _landshark_left_encoder_velocity_msg t_struct_TwistStamped);
    `( data_at_ sh t_struct_TwistStamped)
     (eval_var _landshark_right_encoder_velocity_msg
t_struct_TwistStamped);
    `( data_at_ sh t_struct_TwistStamped)
     (eval_var _landshark_gps_velocity_msg t_struct_TwistStamped))
  POST [tvoid]
  PROP() LOCAL()
  SEP(
    `( data_at sh t_struct_TwistStamped data1)
     (eval_var _landshark_left_encoder_velocity_msg t_struct_TwistStamped);
    `( data_at sh t_struct_TwistStamped data2)
     (eval_var _landshark_right_encoder_velocity_msg t_struct_TwistStamped);
    `( data_at sh t_struct_TwistStamped data3)
     (eval_var _landshark_gps_velocity_msg t_struct_TwistStamped)).
```

Fig. 13. SpinOnce Function Specification

statement, until the end of the function body is reached. At that point, the context should imply the function postcondition.

6 Code Generator Proof

6.1 Property of the code generator

We develop the code generator in Coq, which makes it possible to verify properties of the code generator itself. One of the interesting properties is the generalized `DDC` property that every generated ROS glue code from a valid ROS node model will satisfy the `DDC` property defined in the Section 5. Intuitively, we should prove that for any input ROS node model, our function template instance satisfies our function specification instance. However, VST tactics can only reason about closed code, it cannot specify properties of our AST templates. Therefore, we cannot directly verify these templates. Instead, we analyze what properties are required of code generation in order to guarantee the `DDC` property of the generated code.

The `DDC` property of generated code states that the destination variable holds the desired value according to the ROS node model before it is used. This `DDC` property can be implied by three code generation properties. We would like to use the input_glue function to analyze how the following three code generation properties imply the DDC property of generated code.

```
Definition input_glue_body_statement
   global_expr control_expr: statement :=
  (Ssequence
    (Sset temp_id (global_expr)
    (Sassign (control_expr)
    (Etempvar temp_id temp_type))
  ).
```

Fig. 14. Input Glue Function Body Template

Let us first look at one piece of statement template of the input glue function body as shown in Figure 14. In this template, it has two parameters global_expr field expression of global message variable and control_expr field expression of Controller_U. It generates two statements: one is copying the message filed value (global_expr) to temporary variable (temp_id); the other is setting one field (control_expr) of controller variable with temporary variable. We want the DDC property of input glue_function described in Section 5.2 to be satisfied by generated instance of this template.

The DDC property of input glue_function is specified in Figure 12. The first code generation property is that the origin (global_expr) and the destination (control_expr) should keep the corresponding relation according to the ROS node model. It ensures that the data is delivered from the right origin to the right destination according to the ROS node model. In this case, global_expr and control_expr in the input_glue function should be consistent with the interface relation. This property guarantees that the Controller_U fields will be assigned by the values from corresponding fields shown in Table 1.

The second property is the valid assignment property, which requires only that the left and right sides of an assignment have the same type. This property implies that the destination variables do get the assigned value after this assignment according to the axiom semantics of VST. In this case, Controller_U will hold the value from field x of those three global message variables in Table 1. So with the first and second code generation property, the input_glue function specification shown in Figure 12 is guaranteed. The last code generation property is that the destination variable is not re-assigned by other values before it fulfills its duty. The third property guarantees that the value of Controller_U is preserved until the Controller_step function is invoked.

6.2 Proof of the generalized DDC property

In this section, we discuss the proof of the three code generator properties presented above. The first property is that we instantiate the input_glue function assignment template correctly according to the input ROS node model interface relation. We maintain a list of expressions for each side in the resulting assignments. For the input glue function body, there are lists for global_expr and control_expr. The first property can be proven by showing that the lists of expressions are consistent with ROS node model interface relation, as stated by the lemma in Figure 15. In this lemma, lg_expr is the list expression for global_expr, while lc_expr is the list of expressions for control_expr. lir is the list of interface relation from Table 1. To prove the consistency, we verify that the fields of those expression list is identical of the fields in the interface relation.

```
Lemma relation_consistency_checking :
  forall (lir : list irelation) (lg_expr lc_expr : list expr),
    lg_expr = gen_list_global_variable_expr_input_glue lir →
    lc_expr = gen_list_controller_expr_input_glue lir →
    relation_consistency_checking lir lg_expr lc_expr.
```

Fig. 15. Relation consistency of the input glue function

For the valid assignment property, we only need to check that the lists of types for the left and right sides of assignment are consistent. The type checking function for the input_glue function is shown in Figure 16. Since user may specify an inconsistent ROS node model, mapping a ROS message field with one type to controller input with a different type, the generated assignment can be invalid. The type checking function is applied before generating the input_glue function. If type checking returns `FALSE`, ROSGen set the `error` flag to true and stops generating code. In this way, we guarantee that the generated code always satisfies the valid assignment property.

```
Fixpoint type_checking_input_glue (ltype_global_fields
   ltype_controller_fields : list type) : bool :=
  match ltype_global_fields, ltype_controller_fields with
    | [],  []  ⇒ true
    |tg:: ltypeg, tc:: ltypec ⇒ andb (type_equal tg tc)
      (type_checking_input_glue ltypeg ltypec)
    |_,  _ ⇒ false
  end.
```

Fig. 16. Type checking for input glue function

For the third property, we verify the preservation property by checking that there is no new assignment for the destination variable between input_glue function and Controller_step function. This is quite strait forward for our cases, because there is no other statements between input_glue function and Controller_step function in our loop function template. Furthermore, if we have to add additional statements between input_glue and Controller_U calling statements, it is reasonable to add constraint that they don't involve manipulation on the Controller_U heap, because Controller_U is destined as input of Controller_step by copying value of global message. According to the separation logic of VST, the value of Controller_U is still preserved if those statements manipulate variables in different heap.

7 Related Work

There has been much work on automatic generation of platform-specific glue code based on the architectural model of the system and the underlying platform specification. In [10,11], code generation for a variety of platforms is performed using AADL models to represent hardware and software architectures and their properties relevant for code generation. None of these papers targeted the ROS platform. More importantly, they do not consider verification of the generated code nor the code generator itself.

There is also a similarity between the intent of our approach and verification of model transformations in domain-specific languages. Most of that work, however, is done in the context of behavioral models, with the goal of ensuring that syntactic constraints are preserved by the transformation [12,13,14]. By contrast, we start with an architectural model, where behavior is implicit, and generate executable code.

8 Conclusions

We propose a verified framework ROSGen for generating glue code for ROS-based control systems. We start with a model of a ROS node capturing external connections of the node and parameters needed to execute the node. The code generator, implemented in Coq, uses this model to instantiate Clight templates and use the VST toolset to reason about the code. We then use CompCert utilities to generate C source code from Clight AST. We discuss how to generalize the proof of data delivery correctness for the generated code to a proof of data delivery correctness for the code generator itself. We apply the approach to the cruise control system for the LandShark robotic vehicle.

Our plans for future work include extending the proof approach to directly reason over quantified Clight templates, allowing for a more natural proof of the code generator correctness. Furthermore, we plan to extend the framework to cover the step function, to be able to reason about control-related properties of the code, in addition to the data delivery properties.

Acknowledgment

Thanks to Andrew W. Appel and Joey Dodds for help on applying VST and separation logic.

References

1. James B Dabney and Thomas L Harman. *Mastering Simulink 4*. Prentice Hall PTR, 2001.
2. Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.
3. Nicolas Halbwachs. A synchronous language at work: the story of lustre. *Formal Methods for Industrial Critical Systems: A Survey of Applications*, pages 15–31, 2005.
4. Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.
5. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
6. Andrew W Appel. Verified software toolchain. In *Programming Languages and Systems*, pages 1–17. Springer, 2011.
7. Xavier Leroy. The compcert c verified compiler, 2012.
8. Andrew W Appel, Dockins Robert, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
9. Miroslav Pajic, Nicola Bezzo, James Weimer, Oleg Sokolsky, Nathan Michael, George J Pappas, Paulo Tabuada, and Insup Lee. Demo abstract: Synthesis of platform-aware attack-resilient vehicular systems. In *Cyber-Physical Systems (ICCPS), 2013 ACM/IEEE International Conference on*, pages 251–251. IEEE, 2013.
10. Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In *Reliable Software Technologies: Ada-Europe 2009*, volume 5570 of *Lecture Notes in Computer Science*, pages 237–250. Springer Berlin / Heidelberg, 2009.
11. BaekGyu Kim, Linh TX Phan, Oleg Sokolsky, and Insup Lee. Platform-dependent code generation for embedded real-time software. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pages 1–10. IEEE, 2013.
12. Anantha Narayanan and Gabor Karsai. Towards verifying model transformations. In *Proceedings of the 5^{th} International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*, pages 191–200, 2008.
13. Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
14. Levi Lucio and Hans Vangheluwe. Model transformations to verify model transformations. In *Proceedings of the Workshop on Verification of Model Transformations*, June 2013.