Structuring Formal Control Systems Specifications for Reuse: Surviving Hardware Changes*

Jeffrey M. Thompson, Mats P.E. Heimdahl and Debra M. Erickson
Department of Computer Science and Engineering
University of Minnesota
4-192 EE/CS; 200 Union Street S.E.
Minneapolis, MN 55455 USA
+1 (612) 625-1381
{thompson,heimdahl,erickson}@cs.umn.edu

Abstract

Formal capture and analysis of the required behavior of control systems have many advantages. For instance, it encourages rigorous requirements analysis, the required behavior is unambiguously defined, and we can assure that various safety properties are satisfied. Formal modeling is, however, a costly and time consuming process and if one could reuse the formal models over a family of products, significant cost savings would be realized.

In an ongoing project we are investigating how to structure state-based models to achieve a high level of reusability within product families. In this paper we discuss a high-level structure of requirements models that achieves reusability of the desired control behavior across varying hardware platforms in a product family. The structuring approach is demonstrated through a case study in the mobile robotics domain where the desired robot behavior is reused on two diverse platforms—one commercial mobile platform and one build in-house. We use our language RSML^{-e} to capture the control behavior for reuse and our tool NIMBUS *to demonstrate how the formal specification can be validated and used as a prototype on the two platforms.*

Keywords: Requirements, Formal Models, Requirements Reuse, Control Systems, RSML^{-e}

*This work as been partially supported by NSF grants CCR-9624324 and CCR-9615088, and by NASA grant NAG-1-2242.

1 Introduction

Reuse of software engineering artifacts across projects has the potential to provide large cost savings. Traditionally, the research in the reuse community has focused on how to construct reusable software components, and how to classify and organize these components into libraries where they can be retrieved for use in a particular application. We know, however, that coding errors are not the main source of problems and delays in a software project; incomplete, inconsistent, incorrect, and poorly validated requirements are the primary culprit [4]. Thus, we hypothesize that reuse of requirements in conjunction with reuse of design and code will provide greater benefits in terms of both cost and quality. In this paper we present an approach to structuring formal requirements models for control systems that make the control requirements reusable across platforms where the hardware (sensors and actuators) may vary. We also illustrate the structuring approach with an example from the mobile robotics domain.

The beginnings of our approach is a high-level requirements structuring technique based on the relationship between *system requirements* and the *software specification*. We developed this structuring technique to enable a software development approach we call *specification-based prototyping* [23] where the formal requirements model is used as a prototype (possibly controlling the actual hardware—hardware-in-the-loop-simulation) during the early stages of a project. Here we present how this structuring approach also enables reuse of the high-level requirements across members of a product family with variabilities in the hardware components. The approach is

demonstrated via a case study in the mobile robotics domain where the desired robot behavior is reused on two diverse platforms—one commercial mobile robot and one built in-house. We use our language RSML^{-e} to capture the desired control behavior for reuse and our tool NIMBUS to demonstrate how the formal specification can be validated and used as a prototype on both platforms.

The rest of the paper is organized as follows. Section 2 describes related work on requirements reuse and product families. Then, Section 3 describes our approach to structuring the high-level system requirement and the software specification. Section 4 describes the mobile robotics platforms that we are using as the case study in the paper and presents a simple analysis of their commonalities and variabilities. The requirements of the mobile platforms in the family are presented in Section 5. The refinement of these *system requirements* to a *software specification* is presented in Section 6. In this section we also show how the system requirements are reused across the members of the product family. Finally, Section 7 presents a summary and conclusion.

2 Related Work

The foundations for reuse of can be traced back to the early work on program structure and modularity pioneered by David Parnas and others [3, 19, 20, 21]. This work establishes the basis for reuse: the concept of a self contained module with well-defined interfaces. Nevertheless, the guidelines for how to encapsulate and structure a model (in this case implementations) for reuse is not sufficiently addressed in this early work. Thus, subsequent research in the field of software reuse seeks to further define and provide additional tools and techniques for reuse.

In the area of requirements reuse, Lam *et al.* provides some guidance on specific techniques which can be used by organizations to introduce requirements reuse into their software process [14]. In addition, Lam addressed requirements reuse in the context of component-based software engineering [13]. Our area of interest is more in structuring of specifications to achieve reuse; nevertheless, this work presents some ideas about how to package and specify generic requirements and how to factor requirements into pluggable requirements parts [14]. Of particular interest is the relationship of their work to the product families work being done at Lucent Technologies [2, 24].

Product family engineering is related to the work presented in this paper; in particular, the FAST (Family Oriented Abstraction, Specification and Translation) approach is of interest. FAST provides a process for how to identify commonalities and variabilities across a product family. This commonality analysis can then be used to provide domain specific development tools that will greatly reduce the development costs for later generations of the product family. FAST does not explicitly address the structuring of product requirements. The FAST concepts of the domain analysis and the commonality analysis can, however, be directly applied to our work with formal specifications; FAST provided some of the inspiration for the work presented here.

Little work has been done on how to structure and develop a formal specification in a language such as RSML^{-e}. One notable exception is the CoRE methodology [22, 5, 6] developed by the Software Productivity Consortium. CoRE includes much useful information on how to perform requirements modeling in a semi-formal specification language (similar to the formal SCR defined at the Naval Research Laboratory [11]). Even so, the structuring mechanism proposed in the CoRE guidebook is based on the physical structure of the system as well as which pieces of the system that are likely to change together—these two (often conflicting) structuring mechanisms may or may not be beneficial to reuse. Furthermore, the way in which the structuring techniques achieve reuse is not specified in the guidebook—reuse is not specifically addressed. Our work is based on many ideas similar to those found in CoRE, but we have extended and refined these ideas to address structuring of state-based requirements models to achieve (1) conceptual clarity, (2) robustness in the face of the inevitable requirements changes to which every project is subjected, (3) robustness of the requirements as hardware evolves, and (4) reuse of models as well as V&V results.

3 Structuring

In our work we are primary interested in safety critical applications; that is, applications where malfunction of the software may lead to death, injury, or environmental damage. Most, if not all, such systems are some form of a process control system where the software is participating in the control of a physical system.

3.1 Control Systems

A general view of a software controlled system can be seen in the center of Figure 1. This model consists of a process, sensors, actuators, and a software controller. The process is the physical process we are

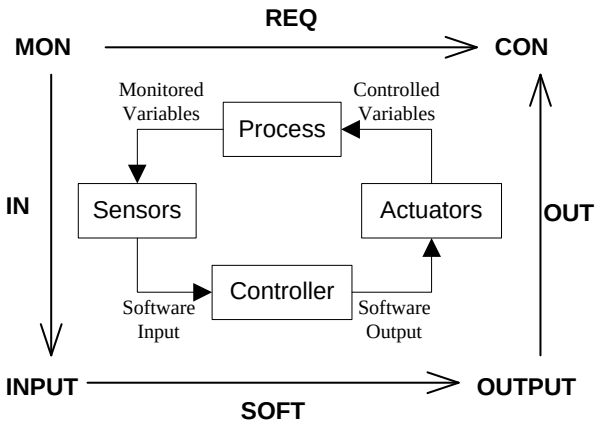

Figure 1. A traditional process control model (center) and how it is captured with the four variable model

attempting to control. The sensors measure physical quantities in the process. These measurements are provided as input to the software controller. The controller makes decisions on what actions are needed and commands the actuators to manipulate the process. The goal of the software control is to maintain some properties in the physical process. Thus, understanding how the sensors, actuators, and process behave is essential for the development and evaluation of correct software. The importance of this systems view has been repeatedly pointed out in the literature [18, 16, 11].

To reason about this type of software controlled systems, David Parnas and Jan Madey defined what they call the four-variable model (outside square of Figure 1) [18]. In this model, the monitored variables (MON) are physical quantities we measure in the system and controlled variables (CON) are physical quantities we will control. The requirements on the control system are expressed as a mapping (REQ) from monitored to controlled variables. For instance, a requirement may be that "*in case of a collision, the robot must back up and turn 90 degrees left.*" Naturally, to implement the control software we must have sensors providing the software with measured values of the monitored variables (INPUT), for example, an indication if the robot has collided with an obstacle. The sensors transform MON to INPUT through the IN relation; thus, the IN relation defines the sensor functions. To adjust the controlled variables, the software generates output that activates various actuators that can manipulate the physical process, for instance, a means to vary the speed of the robot. The actuator function OUT maps OUTPUT to CON. The behavior of the software controller is defined by the SOFT relation that maps INPUT to OUTPUT.

The requirements on the control system are expressed with the REQ relation; the system requirements shall always be expressed in terms of quantities in the physical world. To develop the control software, however, we are interested in the SOFT relation. Thus, we must somehow refine the *system requirements* (the REQ relation) into the *software specification* (the SOFT relation).

3.2 Structuring SOFT

The IN and OUT relations are determined by the sensors and actuators used in the system. For example, to determine if the robot has collided with an obstacle we may use a bumper with micro-switches connected to a digital input card. Similarly, to control the speed of a robot we may use a digital to analog converter and DC motors. Armed with the REQ relation, the IN relation, and the OUT relation we can derive the SOFT relation. The question is, how shall we do this and how shall we structure the description of the SOFT relation in a language such as RSML^{-e}?

As mentioned above, the system requirements should always be expressed in terms of the physical process. These requirements will most likely change over the lifetime of the controller (or family of similar controllers). The sensors and actuators are likely to change independently of the requirements as the controller is reused in different members of a family or new hardware becomes available; thus, all three relations, REQ, IN, and OUT, are likely to change over time. If either one of the REQ, IN, or OUT relations change, the SOFT relation must be modified. To provide a smooth transition from system requirements (REQ) to software specification (SOFT) and to isolate the impact of requirements, sensor, and actuator changes to a minimum, the structure of the software specification SOFT should be based heavily on the structure of the REQ relation [17, 23].

We achieve this by splitting the SOFT relation into three pieces, IN^{-1}, OUT^{-1}, and SOFT$_{REQ}$(Figure 2). IN^{-1} takes the measured input and reconstructs an estimate of the physical quantities in MON. The OUT^{-1} relation maps the internal representation of the controlled variables to the output needed for the actuators to manipulate the actual controlled variables. Given the IN^{-1} and OUT^{-1} relations, the SOFT$_{REQ}$ relation will now be essentially isomorphic to the system requirements (the REQ relation) and, thus, be robust if it is reused on a new platform (manifested as changes

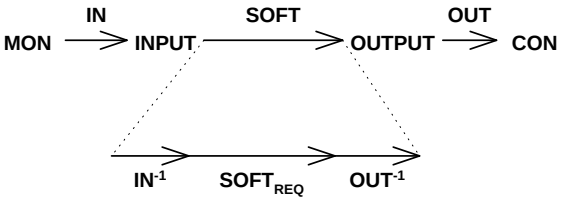

Figure 2. The SOFT relation can be split into three composed relations.

in the IN and OUT relations). Such changes would only effect the IN^{-1} and OUT^{-1} portions of the software specification. Thus, the structuring approach outlined in this section will makes the SOFT_{REQ} portion of the software specification reusable over members of a product family exhibiting the same high-level behavior.

4 Mobile Robotics Platforms

When evaluating our work, we wanted to find a domain were a variety of similar platforms could be constructed on a university budget in a timely and cost effective manner. Furthermore, we wanted this domain to be realistic—with the inclusion of noisy sensors and actuators and the possibility of complex sensor fusion and error detection. The mobile robotics domain seemed ideally suited for these needs.

The mobile robotics platforms that we are using in our research range in size from about the size of the Mars Pathfinder to a small lego-bot. The robots have a limited speed, and can operate either autonomously (via a radio modem or radio Ethernet) or via a tether cable going to a personal computer. The robotics platforms come from various vendors and have a wide variety of sensors and actuators available.

The platforms that are discussed in this paper are shown in Figure 3[1]. One platform, the Pioneer [1], is built and sold by ActivMedia, Inc. The Pioneer includes an array of sonar sensors in the front and sides that allow it to detect obstacles. To detect collisions, the Pioneer monitors its wheels and signals a collision when the wheels stall. The Pioneer includes an extensive control library called Saphira. The Pioneer is controlled by a radio modem that plugs in to the personal computer's serial port. Saphira manages the communication over the radio modem. Saphira is capable of implementing complex rule-based control functions; however, in our work we are using only the simplest

[1]Photograph by Timothy F. Yoon

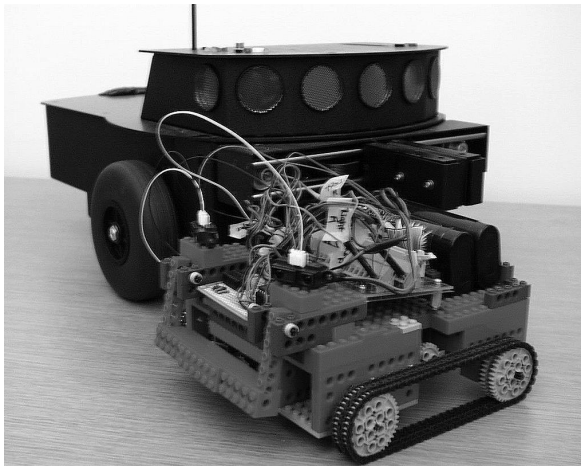

Figure 3. A picture of the robotic platforms used in this paper

of Saphira functions that allow us nearly direct access to the sensors and actuators. Nevertheless, the level of abstraction presented by the Saphira library is significantly higher than on the other platform in this case study: the lego-bot.

The lego-bot is a smaller platform built from Lego building blocks and small motors and sensors. The lego-bot uses a tank-like track locomotion system and has infrared sensors for range detection. The lego-bot is controlled via a tether to the robot from the personal computer. This tether is connected to a data-acquisition card and the software specification for the lego-bot behavior must directly manage the low-level voltages and signal necessary to control the robot; there is very little support for the actuators and sensors.

Despite the significant difference between the platforms, we wanted them to exhibit nearly identical visible behaviors; the only difference would be in the hardware determined speed of the robot's movements. Therefore, the visible behavior (the REQ relation) for each robot is the same. Note that we are not addressing non-behavioral requirements such as power consumption and wear and tear of hardware components in our discussions of reuse. We have focused solely on the *behavior* captured in the requirements.

5 The REQ relation

The first step in a requirements modeling project is to define the system boundaries and identify the monitored and controlled variables in the environment. In this paper we will not go into the details of how to

scope the system requirements and identify the monitored and controlled variables—guidelines to help identify monitored and controlled variables have been discussed in numerous other places [5, 12, 17]. Here it suffices to say that the monitored and controlled variables exist in the physical system and act as the interface between the proposed controller (software and hardware) and the system to be controlled.

For the mobile robots, the goal was to construct a simple reactive control behavior that would cause the robot to explore its environment. To accomplish this objective, the robot must be able to perform several tasks:

- If the robot detects an obstacle, it shall attempt to avoid it.
- If the robot collides with an obstacle, it shall attempt to recover from the collision and continue exploration.
- In the absence of a collision or obstacle, the robot shall proceed to move forward at full speed.

In this case study, we wanted all robots of the product family to exhibit the same exploratory behavior. To capture this behavior we must discover monitored and controlled variables in the environment that will allow us to build the formal model. In addition, while evaluating candidates for monitored and controlled variables we must keep in mind that the REQ model shall apply to all members of the product family.

We identified a robot's *speed* and *heading* as controlled variables. *Speed* ranges from 0 to 100 and can be mapped into a speed for each family member using the maximum speed of the particular robot. *Heading* ranges from -180 to 180 and indicates the number of degrees that the robot may have to turn to avoid an obstacle.

We identified *CollisionDetected*, *Range*, and *ObstacleOrientation* as monitored variables. The *CollisionDetected* variable is simply a Boolean value which is true when there is a collision and false otherwise. The *Range* variable is the distance from the robot to the nearest obstacle and the *ObstacleOrientation* denotes whether the obstacle is straight ahead, or on the right or left of the robot. These variables clearly reside in the system domain and are sufficient to model the desired behavior. If the monitored and controlled variables are chosen appropriately, the specification of the REQ relation will be focused on the issues which are *central* to the requirements on the system.

Since our work is based around a modeling language called RSML^{-e} (Requirements State Machine Language without events), a state-based language suitable for modeling of reactive control systems, we provide a short introduction to the notation before we continue with a discussion of the REQ relation for the mobile robots.

5.1 Introduction to RSML^{-e}

RSML^{-e} is based on the language Requirements State Machine Language (RSML) developed by the Irvine Safety Research group under the leadership of Nancy Leveson [16]. RSML^{-e} is a refinement of RSML and is based on hierarchical finite state machines and dataflow languages. Visually, it is somewhat similar to David Harel's Statecharts [9, 7, 8]. For example, RSML^{-e} supports parallelism, hierarchies, and guarded transitions. The main differences between RSML^{-e} and RSML are the addition in RSML^{-e} of rigorous specifications of the interfaces between the environment and the control software, and the removal of internal broadcast events. The removal of events was prompted by Nancy Leveson's experiences with RSML and a new language called SpecTRM-RL that she has evolved from RSML. These experiences have been chronicled in [15].

An RSML^{-e} specification consists of a collection of *state variables*, *I/O variables*, *interfaces*, *functions*, *macros*, and *constants*, which will be briefly discussed below.

In RSML^{-e}, the state of the model is the values of a set of *state variables*, similar to mode classes in SCR [11]. These state variables can be organized in parallel or hierarchically to describe the current state of the system. Parallel state variables are used to represent the inherently parallel or concurrent concepts in the system being modeled. Hierarchical relationships allow *child* state variables to present an elaboration of a particular *parent* state value. Hierarchical state variables allow a specification designer to work at multiple levels of abstraction, and make models simpler to understand.

For example, consider the behavioral requirements for our mobile robots outlined in the introduction to this section. The state variable hierarchy used to model the requirements on this system can be represented as in Figure 4. This representation includes both parallel and hierarchical relationships of state variables: *Failure* and *Normal* are two parallel state variables and *Robot_Recover_Action* is a child of *Normal*.

Next state functions in RSML^{-e} determine the value of state variables. These functions can be organized as *transitions* or *conditional assignments*. Conditional assignments describe under which conditions a state variable *assumes* each of its possible values. Transitions describe the condition under which a state

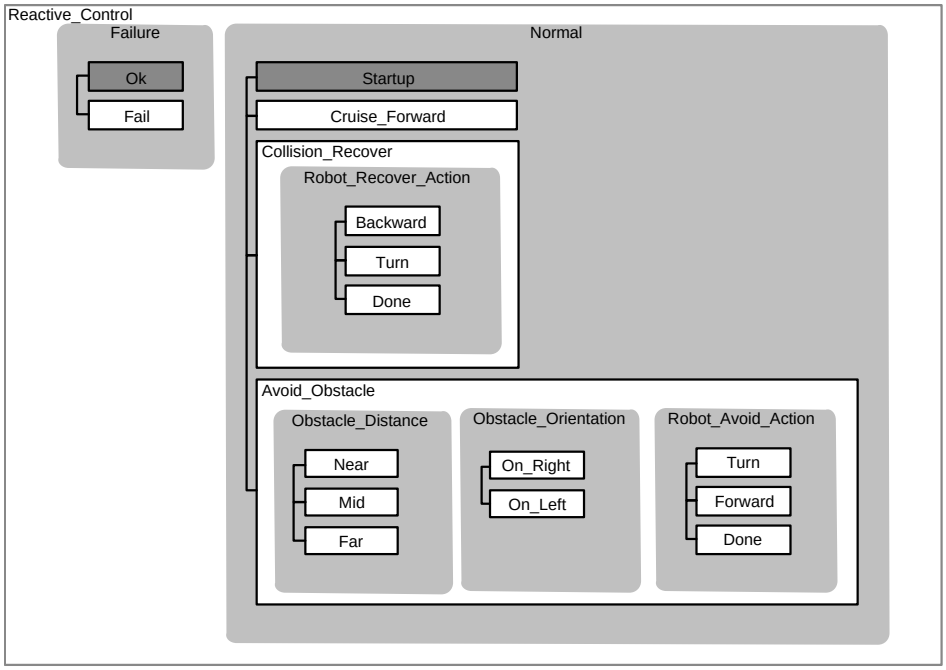

Figure 4. The REQ relation state hierarchy

variable is to *change* value. A transition consists of a source value, a destination value, and a guarding condition. The two state function types are logically equivalent; mechanized procedures exist to ensure that both types of functions are complete and consistent [10].

The next state functions are placed into a partial order based on data dependencies and the hierarchical structure of the state machine. State variables are data-dependent on any other state variables, macros, or I/O variables that are named in their transitions or condition tables. If a variable is a child variable of another state variable, then it is also dependent on its parent variable. The value of the state variable can be computed after the items on which it is data-dependent have been computed. For example, the value of the *Robot_Avoid_Action* state variable would be computed after the *Obstacle_Distance* state variable because the action to take is dependent upon the range of the obstacle.

Conditions are simply predicate logic statement over the various states and variables in the specification. The conditions are expressed in disjunctive normal form using a notation called AND/OR tables [16] The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated columns. An asterisk denotes "don't care." Examples of AND/OR tables can be found later in this section and in the next section.

I/O Variables in the specification allow the analyst to record the monitored variables (MON) or values reported by various external sensors (INPUT) (in the case of input variables) and provide a place to capture the controlled variables (CON) or the values of the outputs (OUTPUT) of the system prior to sending them out in a message (in the case of output variables).

To further increase the readability of the specification, RSML^{-e} contains many other syntactic conventions. For example, RSML^{-e} allows expressions used in the predicates to be defined as functions and familiar and frequently used conditions to be defined as macros. Finally, RSML^{-e} requires rigorous specification of *interfaces* between the environment and the model.

5.2 REQ Relation Overview

Due to space constraints, the entire model of the REQ relation cannot be given in this paper and we will focus on an illustrative subset. Figure 4 shows that the REQ relation definition at the top level is split between two state variables: *Failure* and *Normal*. The *Failure* state variable encapsulates the failure conditions of the REQ relation, whereas the *Normal* state variable de-

scribes the how the robot transitions between the various high-level behaviors discussed at the introduction to this section (obstacle avoidance, collision recovery, etc.). For the reminder of our discussion of REQ, we will focus on the *Normal* state variable where this aspect of the requirements is captured (Figure 5).

The *Normal* variable defaults to the *startup* value. This allows the specification to perform various initialization tasks and checks before the main behavior takes over. The first transition in Figure 5 states that after two seconds, the specification will enter the *Cruise_Forward* state.

The next two transitions govern the way that the *Normal* state variable can change from the *Cruise_Forward* value. If a collision is detected, then the state variable changes to the *Collision_Recover* state. If an obstacle is detected, then the specification will enter the *Avoid_Obstacle* state. Otherwise, the value of the *Normal* state variable will remain unchanged.

If a collision or obstacle is detected, the machine needs to begin the *Cruise_Forward* behavior when the avoidance/recovery action has been completed. We accomplished this in the mobile robotics specification by providing a "done" state in each of the sub-behaviors. This is illustrated by the fifth and sixth transitions in Figure 5.

Finally, it is also possible to transition from *Avoid_Obstacle* directly to *Collision_Recover* if, for example, the robot hits an undetected obstacle; this case is covered by the final transition in Figure 5.

Given this definition of the REQ relation high-level behaviors, the definitions of the sub-behaviors can be constructed in a similar and straightforward manner. For example, if the robot hits an obstacle, it will attempt to back up, turn, and then proceed forward again. This behavior is specified in the *Robot_Recover_Action* state variable by having the variable cycle though the values *Backward*, *Turn*, and finally *Done*.

6 The SOFT relation

When refining the specification from REQ to SOFT, we select the sensors and actuators that will supply the software with information about the environment, that is, we must select the hardware and define the IN and OUT relations for each platform. Consequently, we will also need to define the IN^{-1} and OUT^{-1} for each platform. We do not have the space to discuss the IN, OUT, IN^{-1}, and OUT^{-1} for every monitored and controlled variable. Instead, we will focus our discussion on two areas where the Pioneer

State Variable

Normal

Location: Reactive_Control

Transition: Startup→Cruise_Forward

Condition:

TIME > 2 s	T
..Failure IN_STATE Ok	T

Transition: Cruise_Forward→Collision_Recover

Condition:

CollisionDetectedMacro() = TRUE	T
..Failure IN_STATE Ok	T

Transition: Cruise_Forward →Avoid_Obstacle

Condition:

ObstacleDetectedMacro() = TRUE	T
CollisionDetectedMacro() = FALSE	T
..Failure IN_STATE Ok	T

Transition: Collision_Recover →Cruise_Forward

Condition:

Prev_Step(..Robot_Recover_Action IN_STATE Done)	T
..Failure IN_STATE Ok	T

Transition: Avoid_Obstacle →Cruise_Forward

Condition:

Prev_Step(..Robot_Avoid_Action IN_STATE Done)	T
..Failure IN_STATE Ok	T

Transition: Avoid_Obstacle →Collision_Recover

Condition:

CollisionDetectedMacro() = TRUE	T
..Failure IN_STATE Ok	T

Figure 5. The definition of the *Normal* state variable

and the lego-bot presented illustrative and challenging differences.

6.1 Obstacle Detection— Sonar versus Infrared

As members of the mobile robot product family that we specified in Section 5, both the Pioneer and the lego-bot have the ability to sense the distance to objects in their surroundings. Distance sensors typically function by emitting some sort of signal (for example, a sound in the case of sonar) and then measuring the amount of time between the emission of the signal and its being received back at the sensor. Given how fast the signal can travel, the distance to the closest object can be determined. Although the distance sensors may be somewhat similar in their operation, different sensors provide very different accuracies and ranges. For example, a laser range finder is far more accurate and has much less noise than the sonar sensors.

The Pioneer uses sonar sensors and the Saphira software package to accomplish obstacle detection whereas the lego-bot uses a set of simple infrared range finders. This significant difference in the type of sensors as well as differences in the number and placement of the sensors leads to two quite different IN relations. The differences of the IN relations necessitate different IN^{-1} in the computation of the estimated value of the *Range* monitored quantity.

Function

PTransformRange

Type: INTEGER

Parameters:

- iInRange IS INTEGER

:= 0 IF

iInRange <= 0	F	T
iInRange > 700	T	F

:= iInRange/7 **IF**

iInRange > 0	T
iInRange <= 700	T

Figure 7. IN^{-1} Range for the Pioneer

The difference between the SOFT relations for the two platforms (with respect to the range to obstacles) can be encapsulated in a function which transforms the input variables from the range sensors to estimates of the monitored quantity *Range*. The computation of

Function

LTransformRange

Type: INTEGER

Parameters:

- iInRange IS INTEGER

:= 0 IF

iInRange <= 200	F	T
iInRange > 900	T	F

:= (900 – iInRange)/8 **IF**

iInRange > 200	T
iInRange <= 900	T

Figure 8. IN^{-1} Range for the lego-bot

IN^{-1} for the Pioneer is pictured in Figure 7 and for the lego-bot is in Figure 8. For the Pioneer, the sonar inputs range from 0 to 700 and must be scaled appropriately to a number between zero and 100.

For the lego-bot, the transformation is more complex. Both the sonar and the infrared distance sensors have a certain range close to the sensor where the signals cannot be used for range detection (in the case of the sonars, the signals that are emitted bounce back to the sensor too fast for the sensor to detect). Thus, the sensor will report that no obstacle is present when, in fact, an obstacle is very close. In the case of the Pioneer, this problem is handled by the Saphira library. For the lego-bot, however, the RSML^{-e} specification must include a minimum threshold as well as a scaling factor for the maximum values. In our case, readings below 200 from the infrared sensor cannot be trusted and we simply treat any reading below 200 as if the distance is 0, indicating that no obstacle has been (or can be) detected (Figure 8).

Thus, we have shown that even though the sensors and the way in which we have access to the sensors differs widely between the Pioneer and the lego-bot, we can still use the same SOFT_{REQ} model for both robot platforms. In this way, we make the high-level behavior robust and reusable in the face of changes in the range finder.

6.2 Speed— Saphira versus Pulse Modulation

The previous section focused on platform dependent variabilities in the IN and IN^{-1} relations. The Pioneer and the lego-bot have more significant differ-

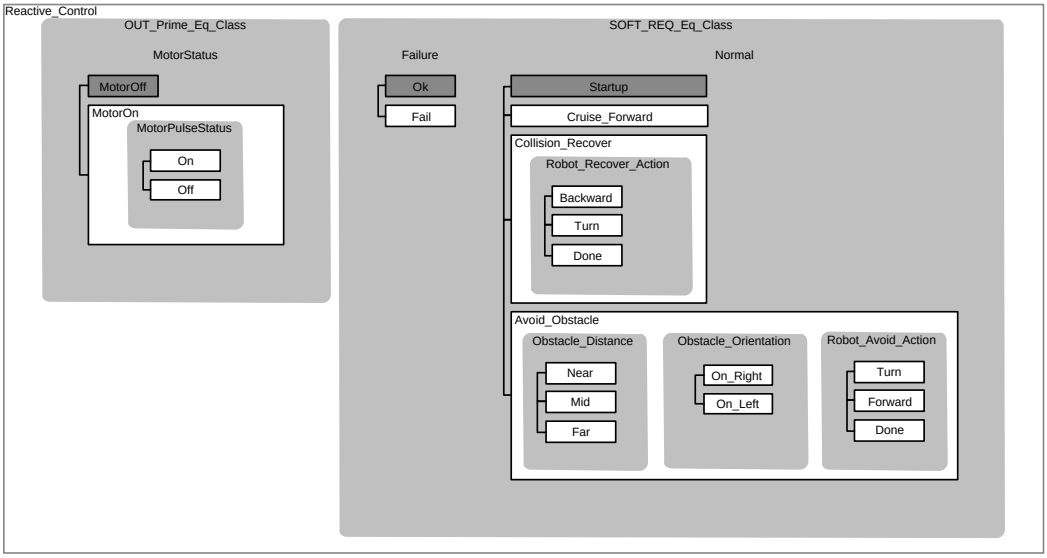

Figure 6. The state machine for the lego-bot

ences in the way that they control their propulsion and in their steering systems (the OUT and OUT^{-1} relations).

The Pioneer's Saphira library provides a high-level control of the Pioneer's motors so that the specification for SOFT on the Pioneer platform is very similar to REQ. The transformation of the desired speed performed in OUT^{-1} for the Pioneer (Figure 9) only requires some minor scaling with respect to the Pioneer's maximum speed. The result of this transformation can then be directly sent to the Pioneer platform and Saphira will control the hardware to achieve the desired speed.

Function

PTransformConSpeed

Type: INTEGER

Parameters:

- iConSpeed IS INTEGER

:= 0 IF

iConSpeed = 0	T

:= (PMaxSpeed * iConSpeed)/100 IF

iConSpeed = 0	F

Figure 9. OUT^{-1} Speed for the Pioneer

On the other hand, the OUT^{-1} specification for the speed of the lego-bot is significantly more complex.

State Variable

MotorPulseStatus

Location: ..MotorOn

Transition: Off ® On

Condition:

TIME >= PREV_STEP (..MotorPulseStatus TIME_ENTERED Off) + LMotorPWMTimeOut(Heading, ConSpeed)	T
PREV_STEP(..MotorPulseStatus IN_STATE Off)	T

Transition: On ® Off

Condition:

TIME >= PREV_STEP(..MotorPulseStatus TIME_ENTERED On) + LPWMOnTimeOut	T
PREV_STEP(..MotorPulseStatus IN_STATE On)	T
ConSpeed = 100	F

:= On

Condition:

ConSpeed = 100	T

Figure 10. The part of OUT^{-1} for the Lego-bot that performs the pulsing on the motors

This is because the SOFT relation for the lego-bot must control the motors directly with low-level hardware signals. The speed of the lego-bot is controlled by a technique called *pulse-width modulation* of the DC motors: the speed of the motors is determined by the length of time which passes between pulses of current applied to the motor. Therefore, the SOFT specification cannot simply output the speed value with some transformation applied; instead, we must use the computed value for the controlled variable *Speed* to determine the pulse width for the motors and then output the pulses accordingly; the motors will then provide enough propulsion to move the lego-bot at the desired speed.

This control strategy necessitates a more complex OUT^{-1} relation for the desired speed; the OUT^{-1} relation can no longer be a simple function—in this case we need to add an additional state machine. To model the pulse modulation we add a state variables to the SOFT specification so that the machine can output the required pulses. These additions are shown in Figure 6. The *MotorPulseStatus* state variable is the part of the OUT^{-1} specification that determines the pulse width. Figure 10 shows the definition of this state variable.

A key component of the pulse-width modulation is the LMotorPWMTimeOut function which determines the length of time to pulse the motors (Figure 11). Notice that because of the lego-bot's tank-track propulsion system, the motors must be pulsed both in the case of a turn and in the case that the robot is moving forward. Thus, the LMotorPWMTimeOut function takes as parameters the controlled variables for speed and heading and produces the correct timeout values.

The values for the pulse intervals were were chosen by running experiments to determine which pulse interval would achieve which speed. We have, therefore, encapsulated these constants so that if we were to change motors on the lego-bot in the future we could simply change the constants rather than having to revisit the pulse-width modulation process.

Thus, despite the fact that the Pioneer and the lego-bot differ significantly in the way that the motors are controlled, the SOFT_{REQ} relation can again be reused across the platforms. Furthermore, changes in the REQ relation (and analogous changes to SOFT_{REQ}) will be independent of changes in the OUT and OUT^{-1} relations.

Function

LMotorPWMTimeOut

Type: TIME

Parameters:

- iHeading IS INTEGER
- iConSpeed IS INTEGER

:= LSlowPWMOffTimeOut **IF**

iHeading = 90	T	F	F
iHeading = -90	F	T	F
iConSpeed = 25	F	F	T

:= LMidPWMOffTimeOut **IF**

iHeading = 45	T	F	F	F
iHeading = -45	F	T	F	F
iConSpeed = 50	F	F	T	F
iConSpeed = -50	F	F	F	T

:= LFastPWMOffTimeOut **IF**

iHeading = 20	T	F	F
iHeading = -20	F	T	F
iConSpeed = 75	F	F	T

:= 0 s **IF**

iConSpeed = 100	T

Figure 11. The timeout function for pulse-width modulation on the Lego-bot.

7 Conclusions

This paper describes how structuring the requirements based on the relationship between the system

requirements and the software specification can lead to benefits in terms of maintainability and reusability. Specifically, we describe a technique for structuring high-level requirements for reuse in the face of hardware changes.

From the four variable model for process control systems, we have described how the REQ relation can be refined to the SOFT relation while maintaining a separation between the part of SOFT which is related to REQ (SOFT_{REQ}) and the parts of SOFT which handle the particular sensors and actuators in the system design (IN^{-1} and OUT^{-1}). This allows us to separate changes in the requirements from sensor and actuator changes and achieve better maintainability and reusability.

This techniques was demonstrated on a case study in the mobile robotics domain using two quite different robots. One robot is commercially produced and is equipped with a rich control library that provides many complex control functions, for example, traveling at a requested speed. The other robot was build in-house from Lego building blocks and off-the-shelf motors and sensors. This robot is controlled completely by the software specification in RSML^{-e} through our NIMBUS toolset.

We demonstrated the usefulness of the structuring approach by reusing the high-level requirements (REQ) across a (currently quite small) family of mobile robots. Nevertheless, there are numerous issues left to address. In the future, we plan to define more complex control behaviors and investigate how individual behaviors (or operational modes) can be successfully reused.

References

[1] Activmedia robotics website. Makers of the Pioneer robot. http://www.activrobots.com/.

[2] Mark A. Ardis and David M. Weiss. Defining families: The commonality analysis. In *Nineteenth International Conference on Software Engineering (ICSE'97)*, pages 649–650, 1997.

[3] K.H. Britton, R.A. Parker, and D.L. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Fifth International Conference on Software Engineering*, 1981.

[4] F. P. Brooks. No silver bullet – essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[5] S. Faulk, J. Brackett, P. Ward, and J Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.

[6] Stuart Faulk, Lisa Finneran, James Jr. Kirby, Sudhir Shah, and James Sutton. Experience applying the CoRE method to the lockheed C-103J software requirements. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS)*, pages 3–8, 1994.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[8] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[9] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[10] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[11] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

[12] Michael Jackson. The world and the machine. In *Proceedings of the 1995 Internation Conference on Software Engineering*, pages 283–292, 1995.

[13] W. Lam. Developing component-based tools for requirements reuse: A process guide. In *Eighth International Workshop on Software Technology and Engineering Practice (STEP'97)*, pages 473–483, 1997.

[14] W. Lam, J.A. McDermid, and A.J. Vickers. Ten steps towards systematic requirements reuse. In *Third IEEE International Symposium on Requirements Engineering (RE'97)*, pages 6–15, 1997.

[15] Nancy G. Leveson, Mats P.E. Heimdahl, and Jon Damon Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.

[16] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[17] Steven P. Miller. Modeling software requirements for embedded systems. Technical report, Advanced Technology Center, Rockwell Collins, Inc., 1999. In Progress.

[18] David L. Parnas and Jan Madey. Functional documentation for computer systems engineering (volume 2). Technical Report CRL 237, McMaster University, Hamilton, Ontario, September 1991.

[19] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.

[20] D.L. Parnas. Designing software for ease of extension and contraction. In *Third International Conference on Software Engineering*, 1978.

[21] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3):256–266, 1985.

[22] Software Productivity Consortium. *Consortium Requirements Engineering Handbook*, 1993. SPC-92060-CMC.

[23] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

[24] David M. Weiss. Defining families: The commonality analysis. Technical report, Lucent Technologies Bell Laboratories, 1000 E. Warrenville Rd, Naperville, IL 60566, 1997.