Modeling a Real Time Operating System Using SpecC

A thesis

submitted to Auckland University of Technology
in partial fulfillment of the requirements of the degree of

Master of Engineering

School of Engineering

Auckland University of Technology

by

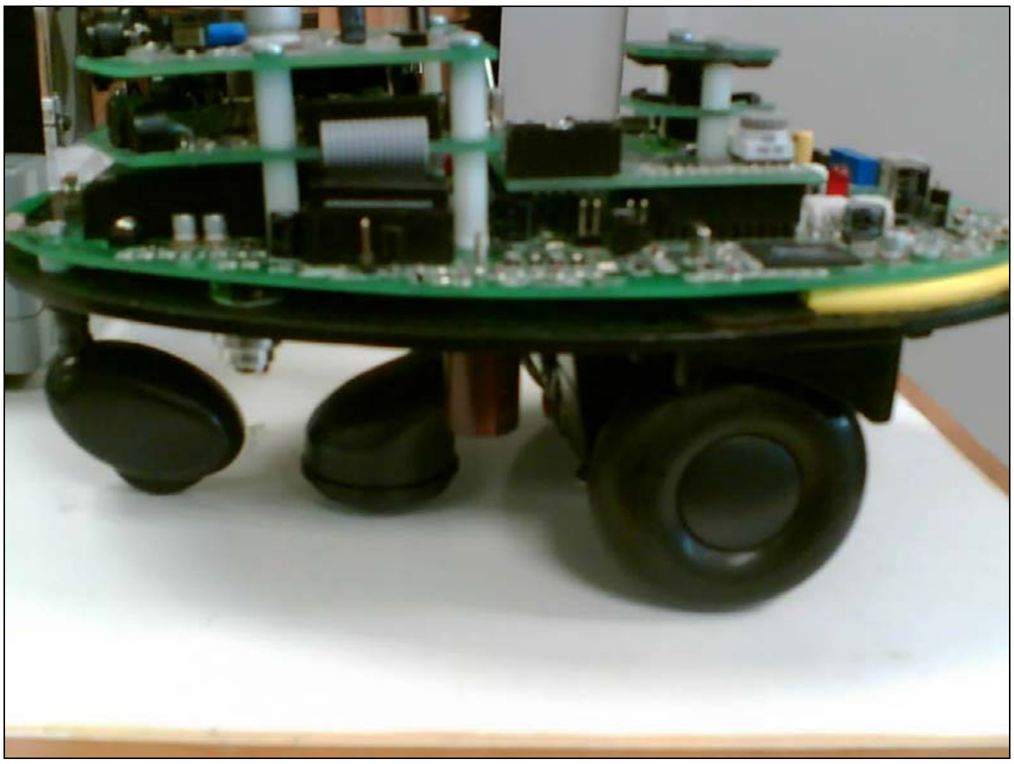
Akilesh Nukala

under the supervision of

Dr. John Collins

AUT UNIVERSITY
TE WĀNANGA ARONUI O TAMAKI MAKAU RAU

2007



The Hedgehog Robot

Acknowledgements

This thesis is by far the most significant scientific accomplishment in my life and it would have been impossible without the people who supported me and believed in me.

First, I thank my supervisor Dr. John Collins, for his continuous support in the masters program. John was always there to listen and to give advice. His enthusiasm, broad view and in-depth knowledge in research and his mission for providing 'only high-quality work and not less', has made a deep impression on me. I am grateful to him for having shown me this way of research. He could not realize how much I have learned from him.

I would like to thank Brett Holden, senior technician of AUT for fixing errors quickly and giving me the updated manual. I would also like to thank all the technicians of the Auckland University of Technology Electrical and Electronic Engineering Department for giving me the opportunity to work with the “Hedgehog” robot, which allowed me to develop a real world real-time application.

I would like to thank the librarians for helping me to source the books and references that I needed to write my thesis.

I would like to thank Dr. Robyn Ramage, for teaching me the essential tool “End Note” which made the job of referencing articles much easier.

I would like to thank my mom, dad and brother Aditya, as without their blessings and wishes, I couldn’t achieve what I have achieved today.

Lastly, how can I forget to thank mates and colleagues, as without their help and friendly attitude, it was highly impossible to complete my thesis.

Statement of Originality

‘I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for qualification of any other degree or diploma of a university or other institution of higher learning, except where due acknowledgement is made in the acknowledgements.’

Akilesh Nukala

Abstract

In today's digital (electronics) world, people's desire for electronic goods that ease their life at work, and leisure is increasing the complexity of the products of the embedded systems industry. For example, MP3 players for listening to music and cell phones for communicating with people.

The gap between the hardware and software parts of embedded systems is being reduced by the use of System Level Design Languages (SLDL) that can model both hardware and software simultaneously. One such SLDL is SpecC.

In this thesis, a SpecC model of a Real Time Operating System (RTOS) is constructed. It is shown how RTOS features can be incorporated into a SpecC model. The model is used to develop an application involving a robot avoiding obstacles to reach its destination. The RTOS model operates similar to the actual RTOS in the robot.

The application includes a testbench model for the robot, including features such as interrupts, sonar sensors and wheel pulses, so that its operation closely resembles the actual robot. The sensor model is programmed to generate the values from the four sensor receivers, similar to the behaviour of the sensors on the actual robot. Also the pulses from the wheels and associated interrupts are programmed in the model so that it resembles the interrupts and wheel pulses present on actual robot.

Keywords: SLDL, SpecC, RTOS, robot, obstacle avoidance

Contents

List of Figures

List of Tables

List of Abbreviations

ASM	**A**bstracted **S**tate **M**achine
FSM	**F**inite **S**tate **M**achine
HDL	**H**ardware **D**escription **L**anguage
ISR	**I**nterrupt **S**ervice **R**outine
PE	**P**rocessing **E**lement
PWM	**P**ulse **W**idth **M**odulation
RTL	**R**egister **T**ransfer **L**evel
RTOS	**R**eal **T**ime **O**perating **S**ystem
SLDL	**S**ystem **L**evel **D**esign **L**anguage
SOC	**S**ystem **O**n **C**hip
UML	**U**nified **M**odeling **L**anguage

1 Introduction

In the embedded systems industry, design complexity continues to increase with man's desire for more and more sophisticated electronic devices. According to "Moore's Law", the number of transistors on a silicon chip doubles every two years. As a result, electronic technology has grown to the extent of having millions of transistors on a chip.

This has led designers to build complex systems. In designing such complex systems, achieving correct functionality is becoming more important and difficult than minimizing silicon area or program memory size. The performance of systems depends on the correct partitioning between hardware and software to obtain the best performance with low implementation cost.

Because of this increasing complexity, there is a growing pressure for software and hardware engineers to work at higher levels of abstraction, using new methodologies, tools and languages which not only simplify their work but also help in the fast and accurate development of products. However, raising the level of abstraction means raising the level of abstraction in specification level, architecture level, communication level, components, tools, methods and methodologies. In other words, the higher abstraction level has to be achieved throughout the design process.

Before the existence of System Level Design Languages (SLDL), when designing a system, software and hardware engineers used different languages like C, VHDL and Verilog, to develop the software and hardware parts of the system. Problems often arise during the system integration phase when hardware and software are combined. SLDLs allow both hardware and software to be modeled in the same language, allowing earlier detection of incompatibilities between hardware and software.

There are several SLDLs available in the market with SpecC [1] and SystemC [2] being among the most popular. These SLDLs are both based on the C language so they are immediately usable by most embedded systems engineers. SpecC has the additional advantage of having a well-defined methodology and refinement rules. For this reason, SpecC was chosen for this application.

The SpecC language [3, 4] and the supporting SpecC methodology [5] do not distinguish between hardware and software, allowing both to be modeled with the same language in a single model program. As a result, the SpecC methodology automatically allows for software/hardware co-design.

Without the SpecC methodology, the designers were facing difficulty in building systems remaining the prisoners of the past, while adaptation of SpecC methodology helps designers to work at higher abstraction levels creating new exciting systems and products eventually freeing designers out of the difficulty they faced.

In this thesis, a SpecC model of a Real Time Operating Systems (RTOS) is constructed. Our RTOS model includes the semaphore, message box, queue and event flags. In the RTOS model, the memory manager feature is utilized to optimise memory usage for our application. The model [6] is used to develop a small application involving a robot avoiding obstacles to reach its destination. The application includes a testbench model for the robot, including features such as interrupts, sonar sensors and wheel pulses, so that its operation closely resembles the actual robot. The sensor model is programmed to generate the values from the four sensor receivers, similar to the behaviour of the sensors on the actual robot. Also the pulses from the wheels and associated interrupts are programmed in the model so that it resembles the interrupts and wheel pulses present on actual robot. All the features incorporated in this model make the model behave in a very similar way to the robot in the real world.

Chapter 2 describes related work on this subject. Chapter 3 describes the SpecC methodology and language. Chapter 4 describes the RTOS, particularly intertask communications. Chapter 5 describes the SpecC model of the RTOS. Chapter 6 describes the robot application involving obstacle avoidance. Chapter 7 describes the SpecC model of the robot application. Chapters 8 and 9 describe the results and conclusions and, most importantly, what can be done in the near future.

2 Related Work

A hierarchical approach with high levels of abstraction is used to deal with complex systems to reduce the number of components being managed. A hardware system, at transistor level composed of tens of millions of transistors, is reduced to thousands of components at register transfer level, and is further reduced to a few components including processing elements, memories and buses at system level.

In a top down design methodology, the system level is said to be at the highest or top level of abstraction. The model gets refined as it moves down to lower levels. On the other hand, a bottom-up methodology starts using components at the lowest level. These components can be used to build more complex components at higher levels [7]. This is illustrated in Figure 1.

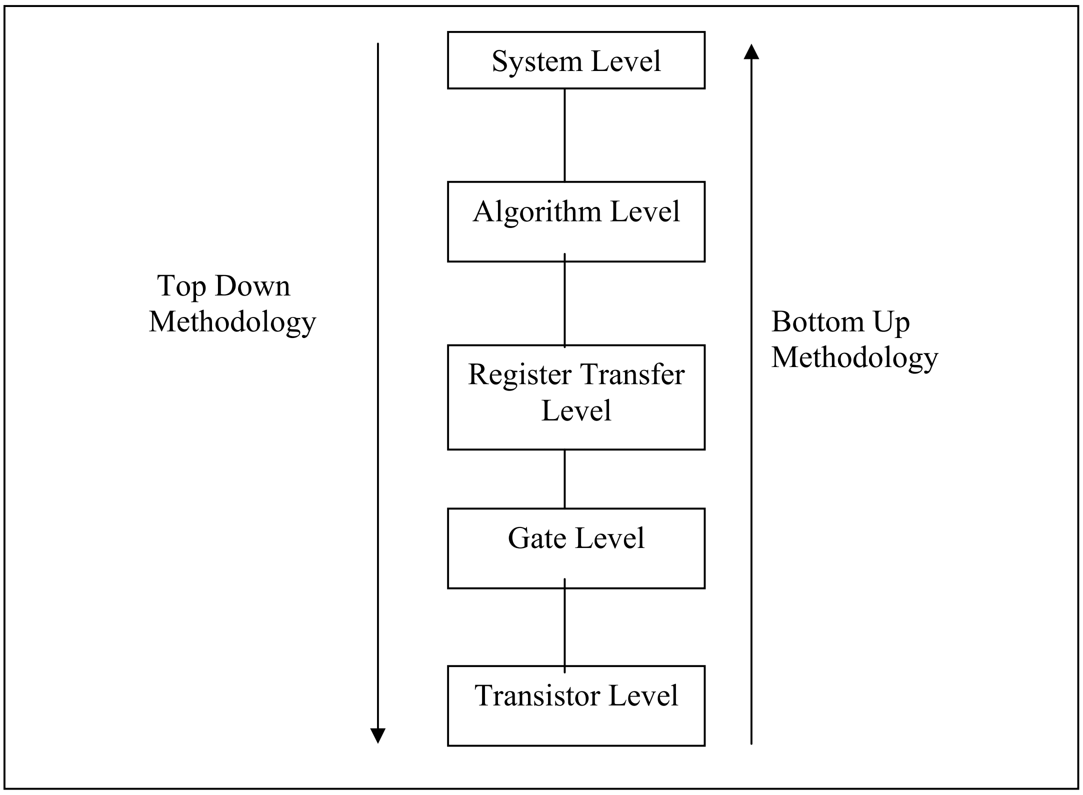


Figure 1 System Methodologies

Using the top-down methodology, the specification of a system is a set of models and a set of transformations between the models that refines the design to the lowest level of abstraction.

From Y-Chart [8], four levels of abstraction are

a) System level

b) Register-transfer level (RTL)

c) Gate level

d) Transistor level

At each level, the designer works with a specific set of objects. The design process has to focus on more details of the system as it reaches lower levels of abstraction.

At each level [8], the design object at that level can be described from three different views:

a) A behavioral view describes the functionality of the design in terms of concepts independent of implementation details.

b) A structural view describes the design as a netlist of lower level components and their connectivity.

c) A physical view describes the spatial layout of the lower level components on the chip, i.e. a floor plan of how the components and their interconnections are placed on the chip.

C language is commonly used for embedded software applications. C based SOC (System On Chip) design approaches cover both hardware and software aspects using System Design Languages (SDL). There are various features of the C language, which are not appropriate for hardware, such as pointers and recursive calls. To provide explicit concurrency and other embedded system features, C based system level design languages such as SpecC and SystemC[2] are used. In this thesis, SpecC [5] is used in a design methodology at system level to model a real time operating system where a memory manager, event flags and various channels comprising of semaphores, message boxes and message queues are used for the communication.

[9] describes the extensions to the C language that are defined in SpecC and compares this approach with the library based approach of SystemC.

[10] uses SystemC for hardware and C with RTOS for software to model an application involving synchronization of software execution with hardware clock events, and communication between the software model and the hardware model. In this thesis, the RTOS is modeled in SpecC, with synchronization and communication between tasks to make the application behave as required.

[11] describes how to apply formal verification techniques to SpecC system descriptions. Formal verification is often used in the hardware design industry but is much more difficult to apply to high-level tools such as SpecC. [15] gives a description of system level design methodologies for SOC design and formal verification technologies for system level specification, using SpecC and its associated design methodology. [24] describes techniques for the verification of synchronization properties of SLDL designs. In this thesis, formal verification techniques have not been used.

Various system level design languages and methodologies have been proposed in the past to address the issues involved in the system level design. These system level design languages deal with synthesizing the hardware part of the system.

The Unified Modeling Language (UML)[12] is often used for software system design but this does not integrate easily with the hardware design. HardwareC[13] is one of the earliest C like Hardware Description Languages(HDL), having arbitrary length bit vector data types and an extended set of bit vector operators aimed at a rather low hardware level featuring inter-process communication by means of channels. SpecC[1] has features including constructs for state machines, arbitrary length bit vectors and channels, used for synchronization and communication between tasks. HandleC[14] is very similar to SpecC including the syntax for extensions but is not as popular as SpecC when it comes to language usage in the market.

[16] uses SpecC to model a serial input/output device driver for the real time kernel μITRON4.0.This real time kernel is described in [17]. In this thesis we apply SpecC to model the MicroC/OS II RTOS [54].

[18] provides a detailed definition of the semantics of the SpecC language including the wait, waitfor, par and try statements from the SpecC Language Reference Manual [3].

[19] describes the design of a SpecC model of an autonomous real time emulator for an electric drive system. Using the SpecC methodology, a specification model of the emulator is transformed to a communication model and implemented making the product ready for manufacturing. The model is verified by comparing it with the manufactured product. The DSP 56600 processor [20] was used in the application.

In [21], the SpecC methodology is used to design control systems of power electronics and electric drives using DC motor drive with a control system based on DSP for motion control, ASIC for current control and three additional hardware components for I/0 processes. In this thesis, for the robot application, SpecC is used to model MicroC/OSII RTOS features running on an Atmel Atmega 128 [22] microcontroller.

[4] discusses the SpecC language and its methodology. This paper describes the semantics of the SpecC language [23] for hardware designers and software designers at system level design. Most of these SpecC language features are used in the thesis.

In[25], a SpecC specification is used to synthesize a gate-level circuit using state-based logic synthesis.

In[26], a SpecC model is used to evaluate scheduling algorithms to avoid the need to tune code later in the development process.

Several abstract modeling techniques have been proposed including graphical finite state machines(Statemate)[27], DSP graphical programming[28] and synchronous programming languages such as Esterel [29]. In [30], a method for automatic software generation of system level design is given. In [31], a method for combining static state scheduling and dynamic scheduling in software synthesis is proposed. In [32], a technique for modeling fixed priority pre-emptive multi tasking systems based on concurrency and exception handling mechanisms provided by SpecC is shown. However, the model has limited support for inter task communication. In [33], a high level model called SoCOS is introduced as a high level RTOS model that supports software generation as well as hardware modeling.

In [34], a timed operating system simulation model was proposed to enable fast and accurate evaluation of software and hardware implementation of on-chip communication. They calculated execution delay values using a delay function and showed how the system model communicates when transforming from macro architecture level to micro architecture level.

In [35], an RTOS model was proposed and used for a mobile application.

[36] experiences challenges faced by C-like languages used for hardware synthesis. The major issues are identified as modeling concurrency and timing.

There are several SLDLs for hardware modeling. Cones[37] is an automated synthesis system that connects C code into digital logic. HardwareC[5] supports hardware structure and structural hierarchy. TransmogrifierC[38] supports loops, conditional and integer arithmetic operators. SystemC[39] supports hardware and system modeling, handling, both combinational and sequential processes. C2Verliog[40] supports pointers, recursion, dynamic memory allocation and broadly supports ANSI C. Cyber[41] accepts a C variant behavior description language that contains hardware extensions but prohibits recursive functions and pointers. HandelC[42] supports constructs for parallel statements. BachC[43] supports explicit concurrency and rendezvous communication and supports arrays and not pointers. HardwareC, SystemC use process level constructs. HandleC and SpecC can also group concurrent statements. SystemC parallelism resembles Verilog and VHDL. Most of the languages cannot be easily extended to model software.

[44] proposes a technique to check the functional equivalence of models before and after scheduling behaviors in the architecture level.

[45] identifies the major design tasks generated at each level from the specification model to the implementation model. These tasks form the basis of the SpecC methodology used in this thesis.

In [46], a new kernel is proposed that handles hardware and software modeling, using multiple heterogeneous models of computation.

[47] describes a C-based methodology for hardware design and verification that uses C to HDL translation and then RTL-C to RTL-Verilog flow.

[48] examines the properties of different abstraction levels and models for system design. A JPEG encoder is designed to demonstrate the application of these techniques. In this thesis, we use these levels of abstraction to develop a model of an obstacle-avoiding robot.

In [49], a methodology is proposed to perform early design stage validation of hardware/ software systems using a HW/SW interface simulation model.

In [50], a methodology is proposed that focuses on methods to make the design flow smooth, efficient and easy by making use of three languages: SpecC at specification level, VCC at architecture level and SystemC at communication level. In this thesis, SpecC is used for all the three levels.

[51] describes how C++ can be used for system modeling. The ideas in this paper have been developed into the SystemC SLDL.

By taking advantage of the SLDL's existing modeling capabilities, our RTOS model is simple to implement yet powerful and flexible, and it can be directly integrated into any system model. In [35], an RTOS was applied to the design of a voice codec for a mobile physical device. Their model did not include features such as time delays, latency and dead time. In the model in this thesis, the RTOS was applied to the design of software for a robot to navigate to reach its destination avoiding obstacles. It is shown that the RTOS model behaves in a very similar way to an actual robot. The modeling concepts can be applied to any other SLDL (such as SystemC) that includes support for event handling and time.

3 SpecC

3.1 SpecC Methodology

The SpecC methodology is a design methodology to implement an embedded system design from specification to implementation involving four levels of modeling. They are the specification model, architectural model, communication model and implementation model.

The SpecC design flow shown in Figure 2[5] starts with the specification model of the desired system behavior. It is written by the user to specify the desired system functionality. The specification model is a purely functional model, free of any implementation details. In general, the specification is hierarchically composed of behaviors. The ordering of events in the system is based on causal relationships only and there is no notion of time. The Specification model describes how the system is going to respond to different inputs since we know its behavior. The behavior is purely functional without any timing or other information.

The architecture model is an intermediate model created after architecture exploration. Architecture exploration selects a set of processing elements (PEs) and maps the computation behavior of the specification onto the PEs. The Architecture model represents this mapping, thus exposing the communication between the components to be implemented by the following communication synthesis task. In the architecture model, the system is described with a set of interconnected components, so we model how to assemble the system from its parts. Each system has its own response time.

The communication model is the final output of the system level design process after architecture exploration implements computation on PEs and communication synthesis implements communication over the buses of the system architecture. The communication model represents the mapping of computation and communication onto PEs and buses respectively.

The communication model describes the system in terms of connections and communication protocols. Protocols are described in terms of wires changing values in real time.

The implementation model is the result of scheduling the functionality mapped onto the PEs (both computation and communication functionality) into register transfers per clock cycle. Therefore, the implementation model is clock accurate at the register transfer level. The implementation model describes hardware in terms of register transfers executed in each clock cycle for custom hardware or in terms of the instruction sequence for software

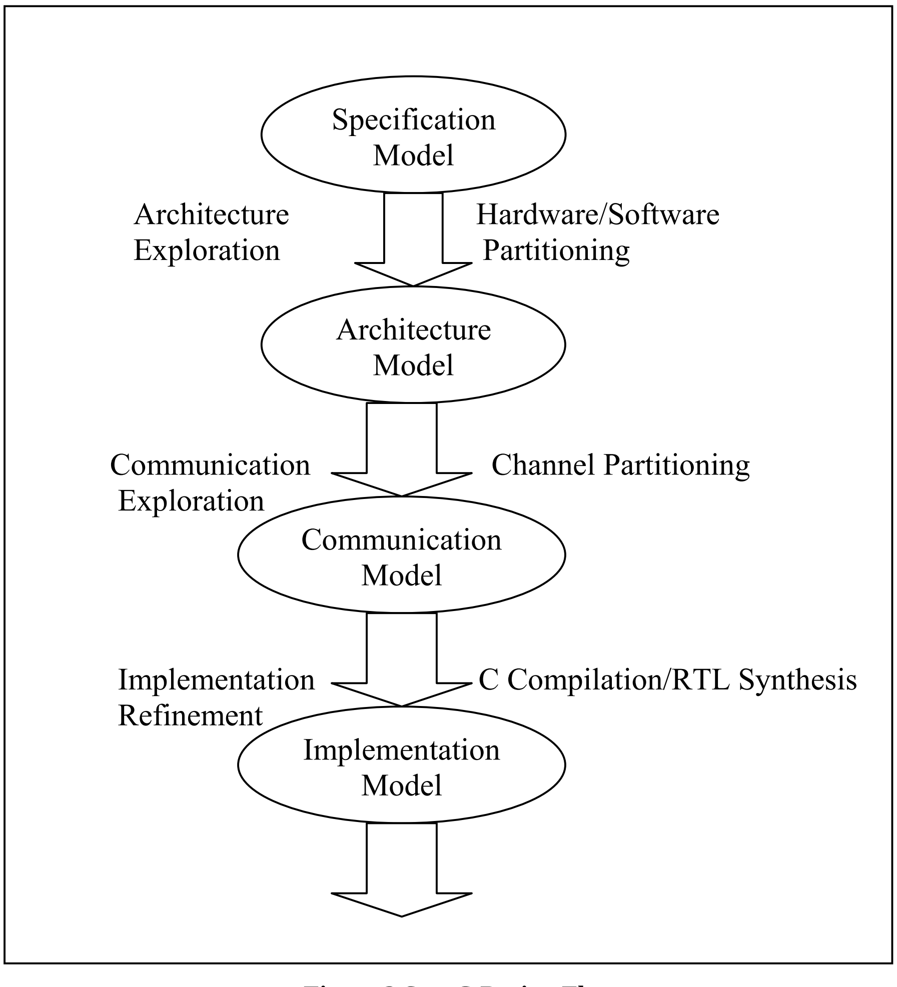


Figure 2 SpecC Design Flow

3.2 SpecC Language

SpecC is a system level design language. SpecC is a true super set of ANSI-C, so every C program is also a SpecC program. The SpecC language includes extensions for hardware design, which are added as a minimal, orthogonal set of concepts. It is a real language with its own keywords and grammar. A SpecC program consists of a set of behaviors, channels and interface declarations.

A b*ehavior* is a class consisting of a set of ports, a set of component instantiations, a set of private variables and functions and a public main function. Through its ports, a behavior can be connected to other behaviors or channels in order to communicate.

A behavior is called a *composite behavior* if it contains instantiations of child behaviors, otherwise it is called a *leaf behavior*. The functionality of a behavior is specified by its functions, starting with the main function.

A *channel* is a class that encapsulates communication. It consists of a set of variables and functions, called methods, which define a communication protocol. A channel can be hierarchical i.e. it can have subchannels which perform lower level communication. A channel can model a semaphore, message or queue in software or physical connections in hardware.

An *interface* represents the link between a behavior and a channel. An interface specifies the public methods that are defined in the channel. In order to define a channel, its interfaces must be defined first.

In SpecC, "*wait/notify*" statements are used for synchronization. The semantics is that a "*wait*" statement suspends the current thread from execution until one of the specified events is "*notified*" by another thread.

One key point in SpecC is the clear separation between communication and computation in system level descriptions used both for software and hardware development.

The communication between processes is done through channels and control mechanisms for communication are contained in the description of the channels.

From SpecC source code, the SpecC compiler generates C++ code, which will then be compiled by a standard C++ compiler in order to produce an executable file for simulation.

3.3 SpecC Scope

SpecC was developed to represent four levels of abstraction i.e. specification, architecture, communication and implementation. There are other languages that can be used for some of these abstraction levels but SpecC can produce efficient, simple, and synthesizable output for all these levels, allowing a seamless top down design.

SpecC supports agile System On Chip (SOC) design and smooth integration, for example for product on demand (POD) technology. SpecC is a starting point for the paradigm shift to Intellectual Property (IP) centric design by providing standardized encapsulation and interfacing for IPs, as well as attributes and models for IP databases.

3.4 Computational Models

A computational model is a formal model of the intended system. Computational models are commonly used in SOC design. They differ significantly in expressive power, features and complexity to simplify the problem and to give the required output.

The following are the computational models commonly used in system level design-

- Finite State Machine(FSM)

  A FSM can be implemented easily in hardware as a controller consisting of a state register and a block of combinational logic.

- Data Flow Graph(DFG)

This is the basic computational model where nodes of the graphs represent operations and arcs in the graph represent dependencies among those operations.

- Finite State Machine with Datapath(FSMD)

This model combines the features of the FSM and DFG representing control and computation used in behavioral synthesis.

- Super-State Finite State Machine with Datapath(SFSMD)

SFSMD is a FSMD with complex, multi-cycle states called super states. Each super state can be changed into several standard states where each state takes only one clock cycle during target implementation. Each super state is described in a standard programming language.

- Hierarchical Concurrent Finite State Machine(HCFSM)

Hierarchy and concurrency are very important concepts for embedded system design. Hierarchy eliminates the problem of state explosion in FSMs and concurrency describes multiple FSMs running in parallel in the same system. A popular example of the HCFSM model is Statecharts.

- Program State Machine(PSM)

PSM combines the features of HCFSM and SFSMD. The PSM model used for the SpecC language is easy to understand and sufficiently powerful for large complex designs as shown in Figure 3.

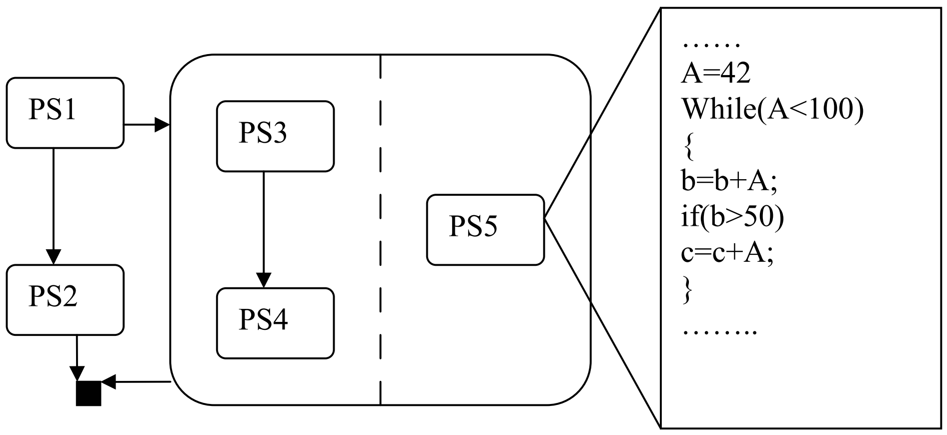


Figure 3 Program State Machine

3.5 Comparison of SpecC with Other Languages

Figure 4 [52] compares SpecC with other languages taking into account a set of system level language requirements. This is based on Table 1 that shows which languages fully support, partially support or do not support the language requirements. All these languages apart from SpecC have deficiencies for modeling embedded systems.

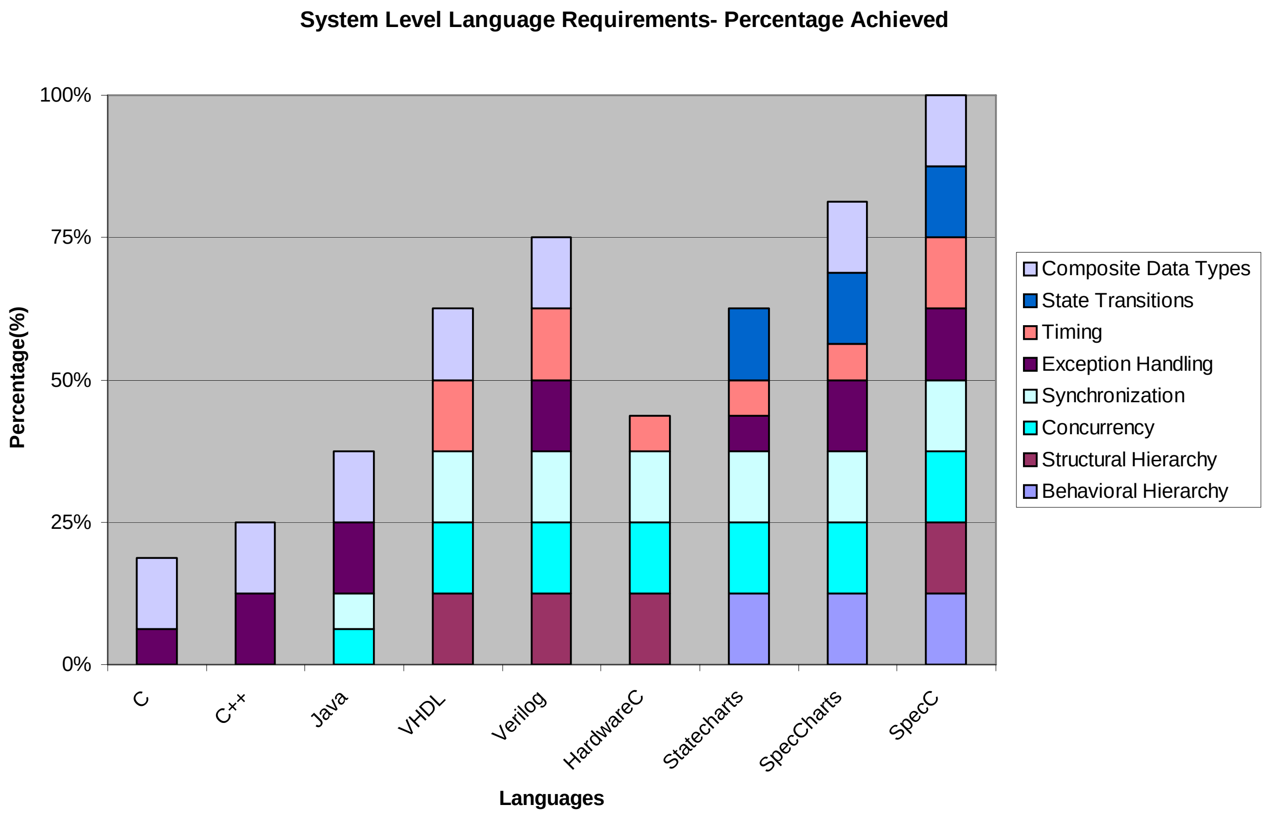


Figure 4 Comparison of SpecC with Other Languages

System level language Requirements								
Languages	*Behavioral Hierarchy*	*Structural Hierarchy*	*Concurrency*	*Synchronization*	*Exception Handling*	*Timing*	*State Transitions*	*Composite Data Types*
C	0%	0%	0%	0%	50%	0%	0%	100%
C++	0%	0%	0%	0%	100%	0%	0%	100%
Java	0%	0%	50%	50%	100%	0%	0%	100%
VHDL	0%	100%	100%	100%	0%	100%	0%	100%
Verilog	0%	100%	100%	100%	100%	100%	0%	100%
HardwareC	0%	100%	100%	100%	0%	50%	0%	0%
Statecharts	100%	0%	100%	100%	50%	50%	100%	0%
SpecCharts	100%	0%	100%	100%	100%	50%	100%	100%
SpecC	100%	100%	100%	100%	100%	100%	100%	100%

Table 1 Comparison of SpecC with Other Languages

From the chart, it can be seen that the C, C++ and Java languages fully support composite data types. C++ and Java fully support exception handling and C partially supports exception handling. VHDL and Verilog fully support structural hierarchy, concurrency, synchronization and timing, Verilog fully supports exceptional handling but VHDL does not support exceptional handling.

HardwareC is a hardware description language, which doesn't support behavioral hierarchy, exception handling and partially supports timing which makes it less powerful than SpecC.

Statecharts fully support behavioral hierarchy, concurrency, synchronization and state transitions, partially support exception handling and timing and does not support structural hierarchy whereas SpecC incorporates system language requirements not supported by Statecharts.

SystemC is C++ based language, which is tedious to profile because of C++ library burden. There are no such limitations with SpecC. SystemC do not fully support behavioral hierarchy whereas SpecC fully supports it.

In SystemC, variable and event can only be used either inside the modules or globally. On the other hand, SpecC supports scheduling using events and data transfer using variables without any constraint [53].

3.6 SpecC Vs VHDL

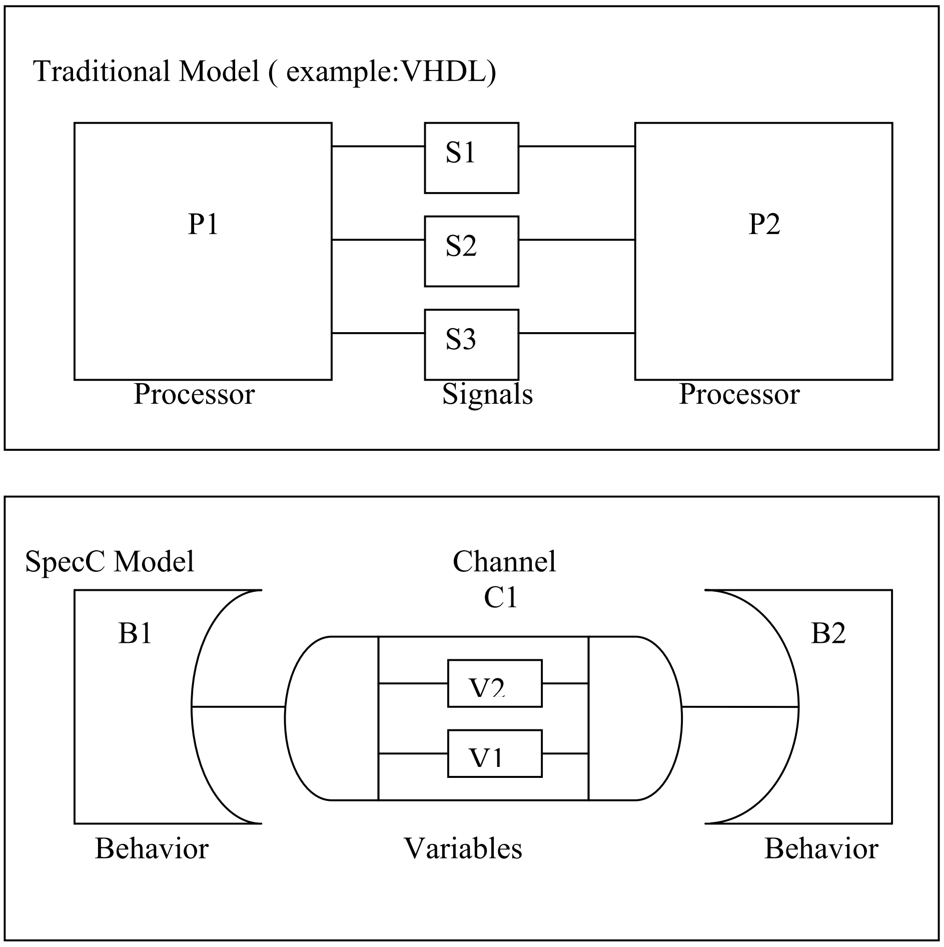


Figure 5 Block Diagram of a Traditional Model and a SpecC Model

In the traditional model such as VHDL, two processors P1 and P2 are communicating via signals S1, S2 and S3. The processors P1 and P2 contain code for both communication and computation. Communication and computation are typically so mixed in the code that it is difficult for the programmer to separate them and work with them individually.

In the SpecC model, the two behaviors B1 and B2 are communicating via channel C1. The behaviors B1 and B2 perform computation and channel C1 performs communication. In order to communicate, behaviors call the functions (methods) provided by the connecting channel. This model separates computation and communication.

3.7 Structural Hierarchy

SpecC supports structural hierarchies of behaviors in which each behavior can itself contain a hierarchical network of behaviors and channels.

The basic structure consists of:-

- Top Behavior
- Child Behaviors
- Channels
- Interfaces
- Variables (Wires)
- Ports

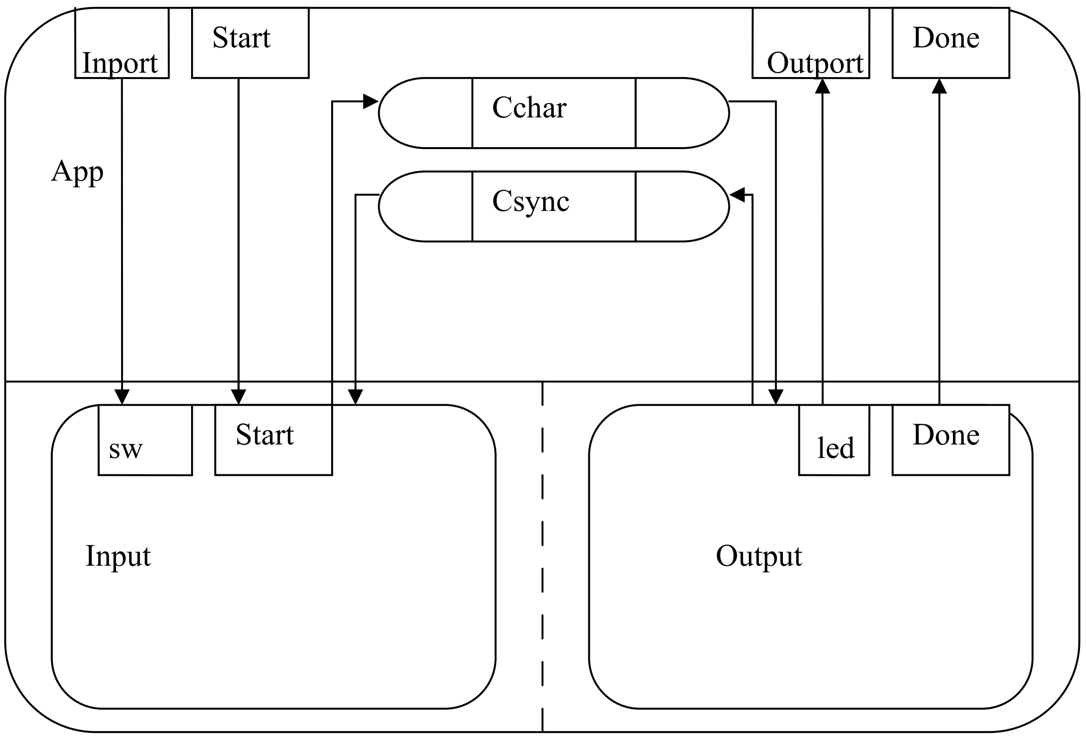


Figure 6 SpecC Model Showing Communication between two Behaviors Input and Output

The example in Figure 6 shows the behavior App with two ports Inport and Outport, through which it can communicate with its environment. These ports are connected internally to child behaviors Input and Output. These child behaviors can communicate using the channels Cchar and Csync. Here, the port sw (switch) is an input of the Input behavior and the port led is an output of the Output behavior. The child behaviors Input and Output may also contain networks of behaviors and channels. The dashed line shows that the Input and Output behaviors run concurrently.

3.8 Behavioral Hierarchy

Behavioral hierarchy is where a parent behavior contains a number of child behaviors. Child behaviors in SpecC can either be executed sequentially or concurrently. Standard sequential statements can specify sequential execution by calling main methods of the instantiated behaviors in the desired order or as a finite state machine (FSM) with explicit state transitions. Both Moore and Mealy type FSMs can be modeled with the SpecC fsm construct. Concurrent execution is either parallel (using the par construct) or pipelined (using the pipe construct).

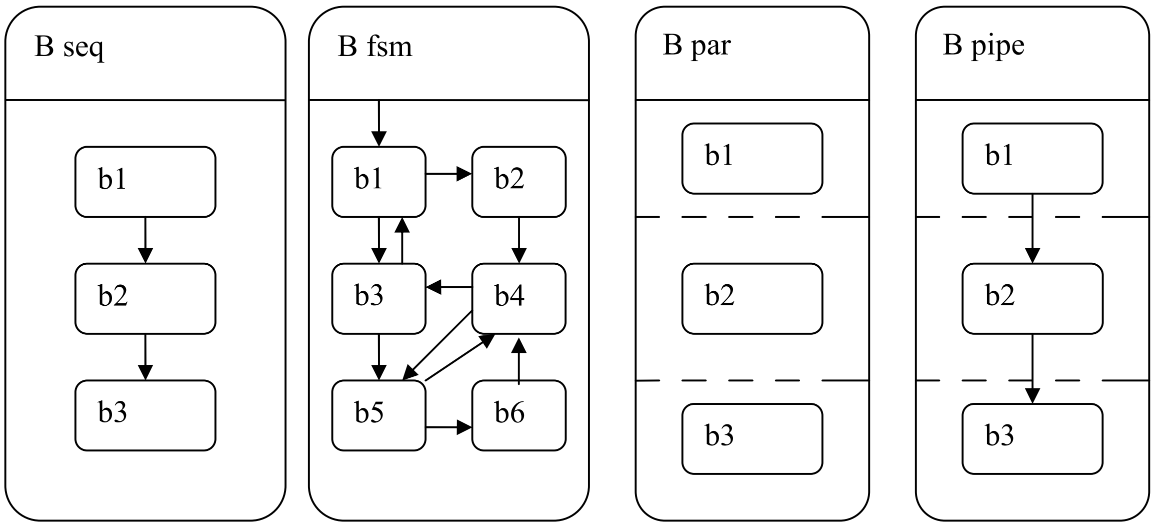


Figure 7 Diagram Representation of the Behavioral Hierarchy in SpecC

In the behavior B seq, the child behaviors b1, b2, b3 run sequentially one at a time.

In the behavior B fsm, the child behaviors b1, b2, b3, b4, b5, b6 represent states of a finite state machine.

In the behavior B par, the child behaviors b1, b2, b3 run in parallel.

In the behavior B pipe, the child behaviors b1, b2, b3 form a three-stage pipeline of behaviors. When the pipeline is started, only b1 is executed. When b1 completes, b2 starts and the second iteration of b1 also starts, so b1 and b2 are executed in parallel. Finally, in the third and every other following iteration, all three-child behaviors are executed in parallel. This implements a pipeline execution.

3.9 Communication

Communication is performed using variables or channels or hierarchical channels between behaviors.

Variables represent shared memory or wires for communication. They can be accessed through the ports that are mapped onto them.

A message can be passed between two behaviors using a channel. The communication functions of the channel are made available to behaviors through the interfaces of the channel.

A channel within a channel gives rise to a hierarchical channel.

3.10 Synchronization

Synchronization is required to allow cooperation among concurrently executing behaviors. In SpecC, the built in type event serves as the basic data type for synchronization. An event is used with *wait*, *notify* and *notifyone* statements which all take lists of events as arguments.

A *wait* statement suspends execution of the current behavior until another behavior notifies any event in the wait statement. Then execution of the waiting behavior resumes.

The *notify* statement activates one or more events so that all the behaviors waiting on one of these events can resume their execution.

The *notifyone* statement allows only one of the waiting behaviors to resume its execution.

3.11 SpecC Example

In SpecC, Figure 6 can be written as follows:

```
//interface contains function definitions helps in communication
//of behaviors
   interface ISendChar
   {
       void send(char v);
   };
   interface IRecvChar
   {
       void recv(char *v);
   };
   channel CChar()//Channel helps in transferring values among
   //behaviors
   implements ISendChar ,IRecvChar
   {
       char buf;
       event e;
       void send(char v)
       {
           buf=v;
           notify(e);
       }
       void recv(char *v)
       {
           wait(e);
           *v=buf;
       }
   };
   interface ISend
   {
       void send(void);
   };
   interface IRecv
   {
       void recv(void);
   };
   channel CSync()// Channel looks after synchronization among
   //behaviors
   implements ISend ,IRecv
   {
       event e;
       void send(void)
       {
           notify(e);
       }
   void recv(void)
       {
           wait(e);
       }
   };
   behavior Input(in unsigned char sw, in event Start, ISendChar
   cchar, IRecv csync)
   {
      void main(void)
      {
           while(1)
           {
                wait(Start);
                 cchar.send(sw);//sw  value  is  sent  to  Output
                               //behavior
                 csync.recv();
           }
      }
   };
   behavior  Output(out  unsigned  char  led,  out  event  Done,
   IRecvChar cchar, ISend csync)
   {
    void main(void)
      {
       char v1;
           while(1)
           {

                 cchar.recv(&v1);// value is received from Input
                 //behavior
                 led = v1;
                 csync.send();
                 notify(Done);
           }
      }
   };
   behavior  App(in  unsigned  char  Inport,  out  unsigned  char
   Outport, in event Start, out event Done)
   {  CChar cchar;
      CSync csync;
      Input In(Inport, Start, cchar, csync);
      Output Out(Outport, Done, cchar, csync);
      void main(void)
       {
           par
            {
                 In.main();
                 Out.main();
            }
       }
   };
```

```
/**************TestBench*********************/
behavior IO(out unsigned char inport, in unsigned char outport,
out event start, in event done)
{
     void main(void)
     {
          while (1)
           {
               printf("Input for switch: ");
               scanf("%c%*c",&inport);// values from the
               //keyboard
               fflush(stdin);//keeps memory free
               notify(start);
               wait(done);
               printf("Output from switch = %c\n", outport);
          }
     }
};
behavior Main
{
     unsigned char inport, outport;
     event start, done;
     IO io(inport, outport, start, done);
     App app(inport, outport, start, done);
     int main (void)
     {
        par
          {
               io.main();//simulates hardware input and output
               app.main();//models application
          }
           return 0;
     }
};
/***************************************************/
```

In this example, the objective is for the value of the Inport input port to be passed through to the Outport output. In the behavior App, behavior Input and behavior Output run in parallel. The value of Inport is passed in from the testbench to the sw (switch) input of behavior Input. The testbench simulates the environment of the application. In the behavior Input, when event Start occurs then the value of sw is passed to the behavior Output using the channels CChar and CSync.

The channel CChar passes the sw value from behavior Input to behavior Output whereas the other channel CSync indicates the sw value has been successfully received by behavior Output. The Output behavior transfers the led value to the App behavior output port Outport.

3.12 Timing

Time is an important requirement for system level design languages. SpecC supports two types of timing specification. They are

- Exact timing

This is specified by use of the *waitfor* statement. The time delay is given in the form of an argument and must be of an integral constant type that is evaluated at compile time.

- Timing constraints

This is specified by use of the *do-timing* construct. The do statement specifies a set of labeled action statements and the timing block contains the actual timing constraints.

The unit of time can be chosen arbitrarily, depending on the time scale of the application being modeled.

3.13 Exception Handling

SpecC supports two types of exception handling. They are

- Abort or Trap

This is implemented by use of the *trap* keyword.

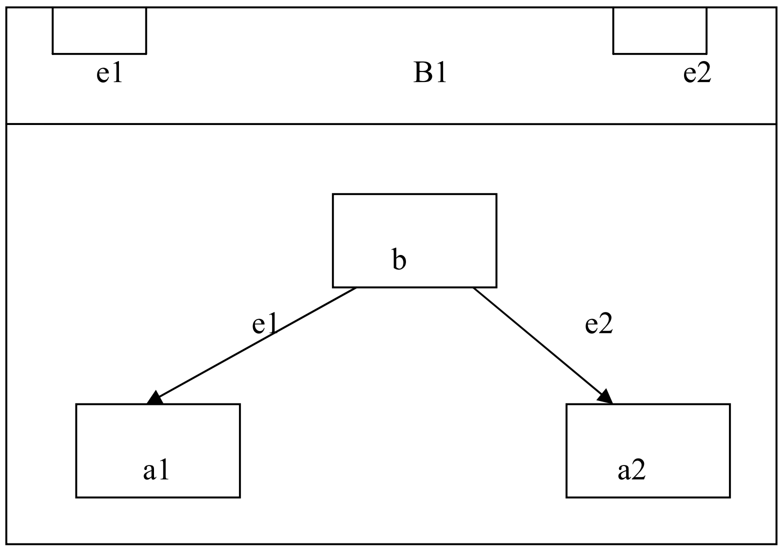


Figure 8 SpecC Abort

In SpecC

```
    behavior B1(in event e1,in event e2)
    {
             B b,a1,a2;
             void main(void)
             {
                  try
                  {
                       b.main();
                  }
                  trap(e1)
                  {
                       a1.main();
                  }
                  trap(e2)
                  {
                       a2.main();
                  }
             }
     };
```

The try-trap construct, shown in Figure 8, aborts behavior b immediately when one of the events e1 or e2 occurs i.e. the execution of behavior b is terminated without completing its computation, and control is transferred to behavior a1 in case of event e1, or to behavior a2 in case of event e2. This type of exception is usually used to model the reset of a system.

- Interrupt

This is implemented by use of the *interrupt* keyword.

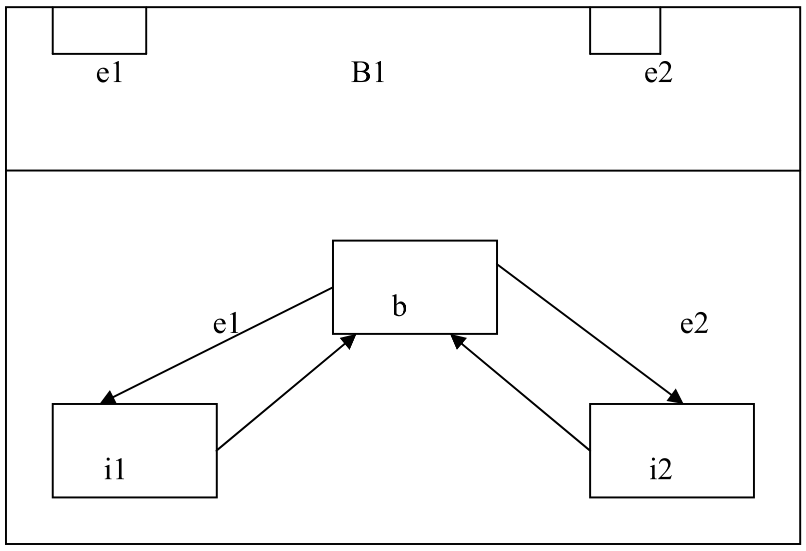


Figure 9 Interrupt In SpecC

In SpecC

```
     behavior B1(in event e1,in event e2)
     {
          B b,i1,i2;
          void main(void)
          {
               try
               {
                    b.main();
               }
               interrupt(e1)
               {
                    i1.main();
               }
               interrupt(e2)
               {
                    i2.main();
               }
          }
     };
```

The try interrupt construct shown in Figure 9, can be used to model interrupts. Here again, execution of behavior b is stopped immediately for event e1 or e2, and behavior i1 or i2 respectively, is started to service the interrupt. After completion of interrupt handlers i1 or i2, control is transferred back to behavior b, and execution is resumed exactly at the point at which it was interrupted.

4 Real Time Operating System (RTOS)

4.1 Definition

“A *RTOS* is a program that schedules execution in a timely manner, manages system resources and provides a consistent foundation for developing application code” [54].

The application code using an RTOS could be for a small application such as a digital watch or for a large and complex application such as for navigation or an IPOD.

In some applications, the RTOS can comprise of only a kernel, which provides scheduling, resource and management algorithms. In other applications, the RTOS can be a combination of various modules including a kernel, system files, I/O devices, device drivers, networking protocols and support libraries. In this thesis, we use the MicroC/OSII RTOS[54].

4.1.1 Kernel

The *Kernel* is the core piece of the operating system. It is a piece of software responsible for the communication between hardware and software components. It is basically a housekeeping program that runs at the highest level, manages the computer’s resources and allows other programs to run. This involves tasks such as:

- Memory Management
- Process Management
- Communication

4.2 Task

A *task* is a small independent program that performs a specific activity. Each task is assigned a priority and its own stack area.

Each task can be in any of the following states:

- Dormant state

  The *dormant* state corresponds to a task that resides in memory but has not been made available to the kernel.

- Ready state

  A task is said to be *ready* when it can execute but its priority is less than the running task.

- Running state

  A task is said to be in *running* state when it has control of the CPU.

- Waiting state

  A task is *waiting* when it requires the occurrence of an event to become ready.

- Interrupted state

  A task is said to be in the *interrupted* state or *ISR* state when an interrupt has occurred and the CPU is in the process of servicing the interrupt.

4.3 Intertask Communication

It is necessary for tasks and Interrupt Service Routines (ISR) to communicate information to other tasks. This can be done in several ways:

- Global variable
- Semaphore
- Message Mailbox

- Message Queue
- Event Flags

4.3.1 Global Variable

When using global variables, each task or ISR must ensure that it has exclusive access to the variables. The only way to ensure exclusive access to common variables is to disable interrupts.

In the MicroC/OSII RTOS, one is able to disable and enable interrupts by calling the macros OS_ENTER_CRITICAL ( ) and OS_EXIT_CRITICAL ( ) respectively.

For example:

```
OS_ENTER_CRITICAL();
countright = 0;
countleft = 0;
OS_EXIT_CRITICAL();
```

4.3.2 Semaphore

This is a key acquired by the code in order to continue with its execution. Semaphores are used to :

- Control access to a shared resource
- Signal the occurrence of an event
- Allow two tasks to synchronize their activities

The three operations performed on a semaphore are:

- INITIALIZE (CREATE)
- WAIT (PEND)
- SIGNAL (POST)

The initial value of the semaphore must be provided when it is first initialized. A semaphore is not available when its value is zero, and it is available when its value its positive. Its value is never negative.

A task desiring the semaphore performs a WAIT operation. If the semaphore is available, the value of semaphore is decremented and the task continues execution. If the semaphore is not available (value is 0) then the task desiring the semaphore is placed in the wait list for the semaphore.

A task releases a semaphore by performing a SIGNAL operation. If no task is waiting for the semaphore then the value of semaphore is incremented. If any task is waiting for the semaphore then one of the waiting tasks is unblocked and made ready to run, and the semaphore value is not incremented. The task that receives the semaphore is the highest priority task waiting for the semaphore.

The following diagram shows communication using a semaphore.

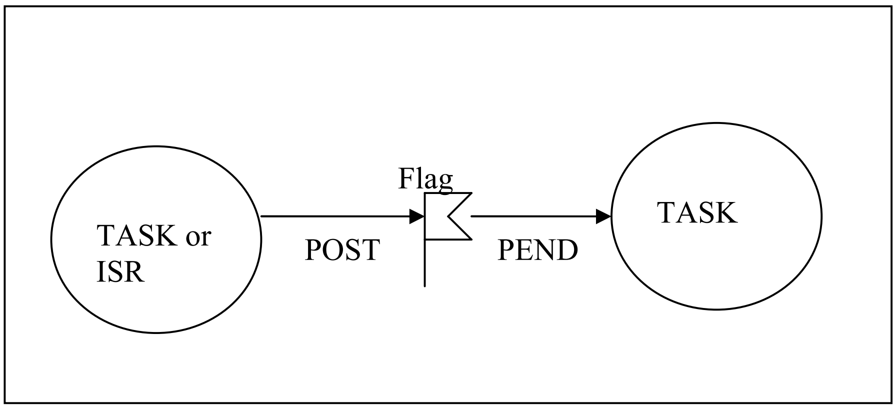


Figure 10 Communication Using Semaphore

4.3.3 Message Mailbox

A *message mailbox* is a way of sending data from a task or ISR to another task. A task or an ISR can deposit a message into the mailbox through a service function provided by the kernel. One or more tasks can receive messages through the kernel.

Each mailbox has a wait list of tasks waiting to receive messages through the mailbox. A task desiring a message from an empty mailbox is blocked and placed on the wait list until a message is placed in the message box by another task.

In general, the kernel allows the task waiting for a message to specify a timeout. If a message is not received before the time out expires, the waiting task is made ready to run and an error code is returned to it. When a message is placed in the mailbox, the waiting task with the highest priority is given the message.

The Kernel provides mailbox services to

- Create the mailbox
- Allow tasks to deposit a message into the mailbox (POST)
- Waits for message to be deposited into the mailbox (PEND). If the mailbox contains a message, the message can be extracted from the mailbox by the waiting task.

The Message mailbox is shown in Figure 11.

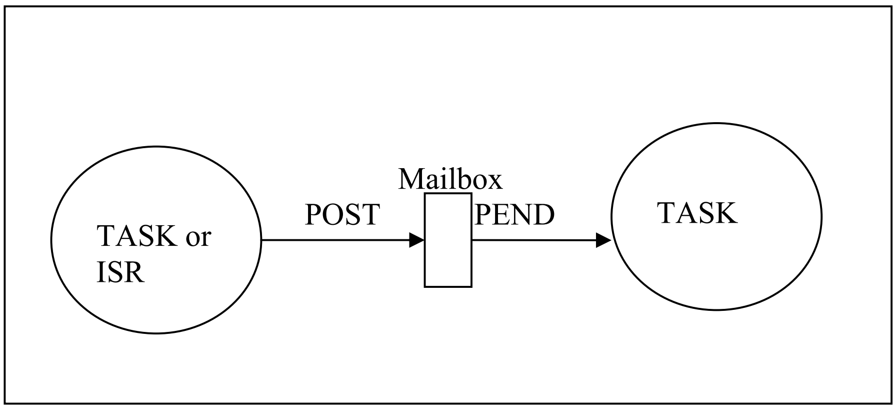


Figure 11 Communication Using a Message Mailbox

4.3.4 Message Queues

A *message queue* is a mailbox that can store more than one message. A task or an ISR can place a message into the message queue through a post service provided by the kernel. In the same way, one or more tasks can receive messages from the queue by calling the pend service provided by the kernel.

As with the mailbox, each message queue is associated with a waiting list, in case more than one task wants to receive messages through the queue. A task desiring a message from an empty queue is blocked and placed on the waiting list until a message is placed in the queue by another task. In general, the kernel allows the waiting task to specify a timeout.

If a message is not received before the timeout expires, the requesting task is made ready to run and an error code is returned to it. When a message is placed in the queue, the task with the highest priority is given the message. The queue delivers messages on a First In First Out (FIFO) basis.

The message queue is depicted in Figure 12.

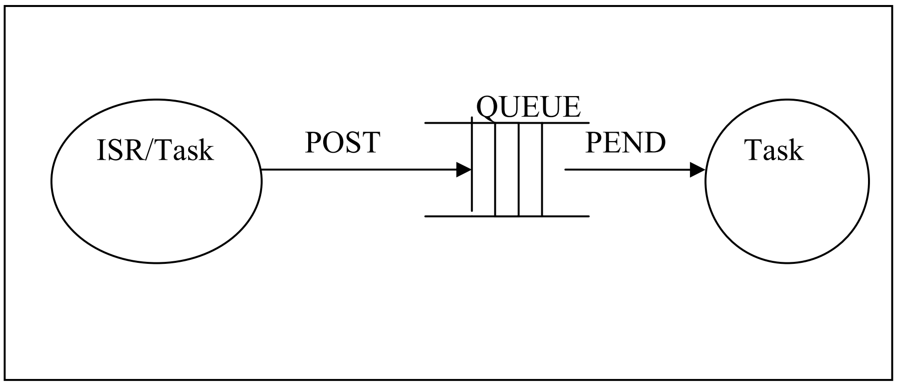


Figure 12 Communication Using a Message Queue

4.3.5 Event Flags

An *event flag* is process synchronization primitive in the operating system. An event flag is actually a group of flag bits, each of which is set on the occurrence of a particular event. An event flag has two possible states, set or cleared. The basic operations are:

- Set event flag
- Clear event flag
- Wait for event flag

When a task waits for an event flag, the task is blocked while the event flag is clear. Event flags allow a task to be blocked waiting for a combination of events.

Additional synchronization operations are:

- Disjunctive Synchronization (logical OR)

  The task waits for any of the specified event flags to be set.

- Conjunctive Synchronization (logical AND)

  The task waits for all specified event flags to be set.

4.3.6 Memory Partitions

The RTOS memory manager provides a simplified memory management system avoiding the use of the C malloc and free functions because they have unpredictable timing requirements. The memory manager maintains a pool of fixed size memory blocks. Tasks call a memory get function to obtain a memory block and a memory put function to release the memory block.

4.4 Interrupts

An *interrupt* is a hardware mechanism to inform the CPU that an asynchronous event has occurred. When an interrupt is recognized, the CPU saves the program context and jumps to a special subroutine known an interrupt service routine (ISR).

The ISR processes the event and after completion of the ISR, the context is restored, program returns to the interrupted task, resuming from where it was interrupted. If the interrupt unblocks a higher priority task then the ISR returns to this task instead.

4.5 Clock Ticks

The RTOS has a timer interrupt which occurs periodically called the *clock tick*. The time between interrupts is usually between 10 and 200ms. The clock tick interrupt allows a kernel to delay tasks for an integral number of clock ticks and to provide timeouts when tasks are waiting for events to occur. The faster the tick rate, the higher the overhead. The clock tick interrupt manages time delays and timeouts.

In the MicroC/OSII RTOS, the function OSTimeDly( ) allows the calling task to delay itself for a number of clock ticks. The calling task is blocked and this forces the RTOS to execute the next highest priority task that is ready to run. The task that called OSTimeDly( ) is made ready to run when the time specified expires or if another task cancels the delay by calling OSTimeDlyResume( ).

5 SpecC Model of RTOS

5.1 Semaphore In SpecC

A task releases a semaphore by calling the semaphore post function. If a higher priority task is waiting for the semaphore then the waiting task is unblocked and the RTOS will resume execution of that task instead of returning to the calling task. If no task is waiting for the semaphore then the value of the semaphore is incremented and the calling task continues running.

When used as a flag, the semaphore is initialised to zero. A task calls the post function to signal the flag and a task calls the pend function to wait for the flag.

The RTOS also has an accept function that allows a task to acquire the semaphore, without being blocked if the semaphore is not available.

The semaphore in SpecC is implemented as a channel as follows:

```
interface ISendSem // send interface
{
    void post(void);
};
interface IRecvSem // receive interface
{
    bool accept(void);
    void pend(void);
};
channel cSem(void) // channel definition
implements ISendSem, IRecvSem
{
    event e;
    unsigned int n = 0;

    void pend(void) // pend
    {
      if (n = = 0)// wait if semaphore is not available
        {
          wait e;
        }
          n--;// return when semaphore is available
    }
```

```
     void post(void)
     {
        n++;//increment semaphore
       notify e;//unblock waiting tasks
     }
     bool accept(void)
     {
       if (n > 0)// if semaphore is available
       {
           n--;
           return(true);
       }
       else // semaphore is not available
       {
           return(false);
       }
     }
};
```

The interface contains declarations of the channel functions (post, pend and accept) and the channel definition contains the definitions of the functions.

In the channel, the variable n is the value of the semaphore. The functions pend, post and accept model the functions in the RTOS.

5.2 Memory Manager In SpecC

In SpecC, the memory manager channel is a counter whose value is the number of available memory blocks. The code is similar to a semaphore controlling access to a shared resource. The get function is similar to the semaphore pend function.A task calls the get function to obtain a memory block. The put function is similar to the semaphore post function.A task calls the put function to return a memory block to the memory manager. The get function returns the address of the memory block. It returns NULL if there is no memory block available.

The memory manager in SpecC is implemented as a channel as follows :

```
#define NUM_MEM_BLOCKS 10
#define MEM_BLOCK_SIZE 100//memory size
interface MemMgmt
{
   void* get(void);
   void put(void *mem);
};
```

```
channel cMem(void) implements MemMgmt
{ unsigned int n = NUM_MEM_BLOCKS;// number of available channels
   void* get(void)
   {
      void *mem;
      if (n = = 0)// return NULL if no memory available
      {
      mem = NULL;
      while (1);
      }
      else // get memory and return its address
      {
      mem = malloc(MEM_BLOCK_SIZE);
      n--;// decrement memory block counter
      }
      return mem;
   }
   void put(void *mem)
   {
      if (mem != NULL)// check for valid memory address
      {
         free(mem);
         n++;//increment memory block counter
      }
   }
};
```

5.3 Message Box In SpecC

A message box is used to pass a message to another task. This is different from a semaphore because data is transferred from one task to another. Usually the sending task places the message in memory obtained from the memory manager. The format of the message can be any data type. The address of the message is posted to the message box. The task that receives the message extracts the contents of the message, and then frees the memory by calling the memory manager’s put function. Note that the message box can only contain one message. The accept function allows a task to check the message box without being blocked when the message box is empty.

The channel message box in SpecC is implemented as follows:

```
interface ISendMbox //message box for sending messages
{
   void send(void *message);
};
```

```
interface IrecvMbox//message box for receiving messages
{
    void recv(void **message);
    void accept(void **message);
};
channel  cMbox()//communication  of  sending  and  receiving  of
//messages
implements ISendMbox ,IRecvMbox
{
    unsigned long m,b;
    void *buf = 0;// memory space
    event e;
    void send(void *message)
    {
     b = (unsigned long)buf;
     m = (unsigned long)message;
     if (buf == 0)//storing a message into memory
     {
          buf=message;
          notify(e);
     }
    }
    void recv(void **message)
    {
     b = (unsigned long)buf;
     if (buf == 0)
          wait(e);
     b = (unsigned long)buf;//
        *message=buf;
     buf = 0;
    }
    void accept(void **message)
    {
     b = (unsigned long)buf;
        *message=buf;
     buf = 0;
    }
};
```

5.4 Queue In SpecC

A queue is also used to pass messages to another task. The difference between the message box and queue is that queue can store more than one message pointer at a time. The queue stores these messages in an array of pointers that can be read by the receiving task on a first in-first out basis.

In SpecC, a queue is implemented as a channel as follows:

```
interface i_receiver
{
    void receive(void **d);
};
interface i_sender
{
    void send(void *d);
};
#define QMAX 100 // maximum number of queues
channel c_mem implements i_sender, i_receiver
{
     void *queue[QMAX];
     int  head=0,  tail=0,  count=0;//first  element,  last  element
//and count of no. of queues respectively
     event         req,
                  ack;
     bool          v = false,
                   w = false;
     void receive(void **d)
     {
      if (!count)
      {
         w = true;
         wait req;
         w = false;
      }
      *d = queue[head];//value coming out of the queue
      count--;
      head++;
      if  (head  >=  QMAX)//if  the  no.  of  first  element  greater
//than max. no. of queues
           head = 0;
      if (v)
       {
           notify ack;
       }
     }
    void send(void *d)
    {
     if (count >= QMAX)//if no. of available queues is greater
//than max. no. of queues
     {
           v = true;
           wait ack;
           v = false;
     }
     if (count < QMAX)//if no. of available queues is less
//than max. no. of queues
     {
           queue[tail] = d;//queue receiving the value
           count++;
           tail++;
           if (tail >= QMAX)// if the no. of last element greater
//than max. no. of queues
           {
            tail = 0;
           }
           if (w)
           {
           notify req;
           }
     }
    }
};
```

5.5 Event Flag In SpecC

An event flag gives us the information of the flag bits giving an indication what values are coming out of the message box to the behavior that helps us in calculations related to the robot's movement. An event flag is implemented as a SpecC channel as follows:

```
interface IEventFlag
{
    unsigned int pend(unsigned int flags, bool consume);
    void post(unsigned int flag);
    unsigned int accept(unsigned int flags, bool consume);
};
channel cEventFlag() implements IEventFlag
{
    event e;
    unsigned int n = 0;
    unsigned int pend(unsigned int flags, bool consume)
    {
      unsigned int flag_rdy;
      if (n = = 0)
      {
            wait e;
      }
      flag_rdy = n;
      n = 0;
      return flag_rdy;
    }
    void post(unsigned int flag)
    {
      n = n | flag;
      notify e;
    }
    unsigned int accept(unsigned int flags, bool consume)
    {
      unsigned int flag_rdy;
      flag_rdy=n&flags;
      if(flag_rdy!=0)
      {
           if (consume= =true)
           {
                n&=~flag_rdy;
           }
      }
      return(flag_rdy);
    }
};
```

The model features described above, such as the semaphore, message box, queue, memory manager and event flag all function the same as in the MicroC/OSII RTOS. The operation of these features is generally compatible with the operation of any other RTOS and is therefore is of general use.

6 Real Time Application – Robot with Obstacle Avoidance

6.1 Hardware

The technicians of Auckland University of Technology built the hedgehog robot. It has a detachable CPU board that can be replaced with any appropriate purpose built board using a microcontroller or FPGA. All programming interfaces go via this board. For this application, the CPU board uses an Atmel ATMega128L microcontroller.

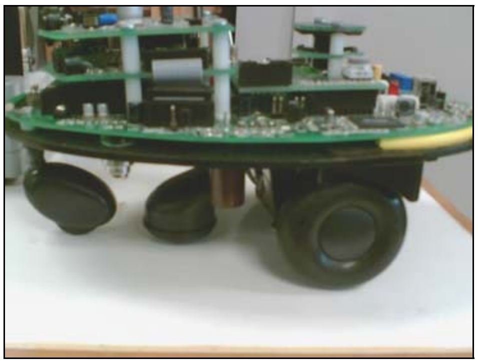


Figure 13 Side View of Robot

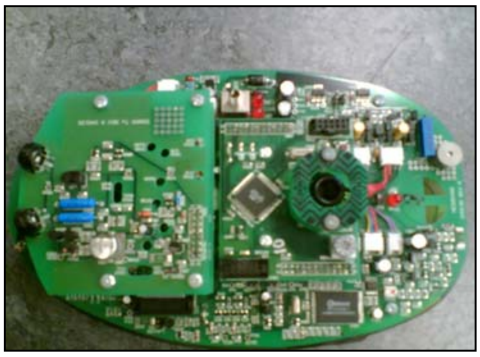


Figure 14 Top View of Robot

Figure 15 Front View of Robot

Figure 16 Back View of Robot

6.1.1 Motor Driver

The robot has two driven wheels, each with its own motor. There are two inputs for each motor connected to two outputs of the micro. In total, there are four motor connections:

Left Motor:	Drive	PB5 (OC1A)
	Direction	PA6
Right Motor:	Drive	PB6 (OC1B)
	Direction	PA7

The motors can be driven by pulse width modulation signals providing fine control of the motor speeds.

6.1.2 Wheel Pulse Generators

There are two independent optical wheel pulse generators, one on the left and one on the right wheel. Each pulse generator produces 24 pulses per rotation of a wheel.

The outputs of the wheel pulse generators are connected to:

Left Wheel	PD0 (INT0) and PD4 (IC1)
Right Wheel	PD1 (INT1) and PE7 (IC3)

The wheel pulse generators are connected to external interrupts (INT0, INT1) allowing the pulses to be counted in an interrupt service routine. The wheel pulse generators are also connected to input capture inputs (IC1, IC3) allowing the time between pulses to be measured.

6.1.3 Sonar System

The Hedgehog has two boards mounted at the front, on top of the battery box. The upper one is the sonar transmitter, which has two ultrasonic speakers (left and right) operating at a frequency of 40 Khz.

The ultrasonic pulses sent from these are reflected back (by objects in front of the Hedgehog) and then received by four ultrasonic microphones (2 left, 2 right) on the sonar receivers (1 left, 1 right), which are mounted on the board underneath.

The field of vision from left to right sensor is 110° and is symmetrical. The left and right receivers are 66mm apart and turned outwards 22.5°. Their angle of view is about 75°.

The outputs from the Sonar Receivers are connected to:

Left Close:	PA0
Left Far:	PA1
Right Close:	PA2
Right Far:	PA3

The possible states of the sonar receivers are shown below:

Possible valid states					
Left Receiver		Right Receiver		Condition	Possible states
Close	Far	Close	Far		Yes/No
Low	Low	Low	Low	No object detected	Yes
Low	High	Low	Low	Far object detected left	No
Low	Low	Low	High	Far object detected right	No
Low	Low	High	High	Close object on right	Yes
High	High	Low	Low	Close object on left	Yes
High	High	High	High	Close object on left & right	Yes
Low	High	Low	High	Far object on left & right	Yes
High	High	Low	High	Close object on left Far object detected right	No
Low	High	High	High	Close object on right Far object detected left	No

Table 2 Possible Sonar Receiver States

6.1.4 Serial Port

The name "serial" comes from the fact that a serial port "serializes" data. That is, it takes a byte of data and transmits the 8 bits in the byte one bit at a time. The advantage is that a serial port needs only one wire to transmit the 8 bits (while a parallel port needs 8). The disadvantage is that it takes 8 times longer to transmit the data than it would if there were 8 wires. Serial ports lower cable costs and make cables smaller.

Before each byte of data, a serial port sends a start bit, which is a single bit with a value of 0. After each byte of data, it sends a stop bit to signal that the byte is complete. It may also send a parity bit.

Serial ports are bi-directional. Bi-directional communication allows each device to receive data as well as transmit it at the same time. Serial devices use different pins to receive and transmit data.

Serial ports rely on a special controller, the Universal Synchronous and Asynchronous Receiver/Transmitter (USART), to function properly. The USART takes a byte and transforms it into serial form for transmission through the serial port.

The serial port pins on the Hedgehog robot are:

Serial Out:	TxD0
Serial In:	RxD0

6.2 Software

6.2.1 PWM for Motor

Pulse width modulation (PWM) is a powerful technique for controlling analog circuits with a processor's digital outputs. PWM is employed in a wide variety of applications, ranging from measurement and communications to power control and conversion.

By controlling analog circuits digitally, system costs and power consumption can be drastically reduced. Many microcontrollers and DSP’s already include on-chip PWM controllers, making implementation easy.

PWM is a way of digitally encoding analog signal levels. Through the use of high-resolution counters, the duty cycle of a square wave is modulated to encode a specific analog signal level. The PWM signal is still digital because, at any given instant of time, the signal is either fully on or fully off. The voltage or current source is supplied to the analog load by means of a repeating series of on and off pulses. The on-time is the time during which the DC supply is applied to the load and the off-time is the period during which, that supply is switched off.

Within the resolution due to the number of bits controlling the PWM signal, any analog value between fully on and fully off can be encoded with PWM.

In the ATMega128 microcontroller, each timer/counter has its own Timer/Counter Register (TCNT) and Output Compare Register (OCR). The Output Compare Register is compared with the Timer/Counter value at all times. The result of the compare used by the waveform generator produces a PWM signal.

In the robot, a 10-bit PWM mode was used.

6.2.2 Input capture and Overflow Interrupts

The Timer/Counters also have an input capture unit that can capture external events indicating time of occurrence. When a specified edge occurs on the Input Capture Pin (IC), a capture will be triggered. When a capture is triggered, the value of the counter (TCNT) is written to the Input Capture Register (ICR). The Input Capture Flag (ICF) is set at the same time as the TCNT value is copied into the ICR Register. If enabled, the input capture flag generates an input capture interrupt.

When using the input capture interrupt, the ICR Register should be read as early in the ISR (interrupt service routine) as possible, because if the processor has not read the captured value in ICR before the next event occurs, ICR will be overwritten with a new value giving an incorrect capture result.

In the program, interrupt [TIM1_CAPT] void LeftInputCapture(void) used on left wheel and interrupt [TIM3_CAPT] void RightInputCapture(void) used on right wheel helped in getting the count of pulses from the two wheels. Sometimes due to the time lag between the pulses of the two wheels, it leads to problem of overflow of pulse values from one of the two wheels that pose a problem in calculating the position of the robot. To avoid the situation, interrupt [TIM1_OVF] void LeftOverflow(void) used for left wheel and interrupt [TIM3_OVF] void RightOverflow(void) used for right wheel helped in retaining the overflow of values out of the two wheels.

6.2.2.1 Glitches in Exceptional Handling of Interrupts

While working on SpecC, I found out there is something wrong with the exceptional handling of interrupts as shown per the book i.e. SpecC compiler didn't allow nested interrupts in a main function especially for my application.

For example

```
int main (void)
     {
          setup.main();
          printf("Operation: starting\n");
          try
          {
               operate.main();
          }
          interrupt (eLeft)
          {
               intLeft.main();
          }
          interrupt (eRight)
          {
               intRight.main();
          }
          return 0;
     }
};
```

In this case, when an event eLeft or eRight is notified, an interrupt occurs and behavior operate is stopped immediately in its execution. The appropriate interrupt behavior such as intright and intleft, is then executed. Once the interrupt behavior finishes, the main behavior operate can resume its execution right from the point where it was stopped.

But SpecC compiler didn't allow the above code in my case as I was using interrupts as counters to the wheels of a robot and threw errors on the screen so I had to make some changes which is shown below in the code provided:

```
 int main (void)
      {
            setup.main();
            printf("Operation: starting\n");
            try
            {
                  operate.main();
            }
            interrupt (eInt)
            {
                  isr.main();
            }
            return 0;
      }
};
```

In this case, when an event eInt is notified, an interrupt occurs and behavior operate is stopped immediately in its execution. The appropriate interrupt behavior such as isr, is then executed. Once the interrupt behavior finishes, the main behavior operate can resume its execution right from the point where it was stopped.

In the function isr, I used a variable intFlag, which takes care off the counters of the wheels of the robot thereby taking care of the problem and it worked well with the compiler. The source code can be seen in the appendix.

6.2.3 Sensor Polling

Polling is to check the status of an input or memory location to see if a particular external event has occurred. Here, polling is done on the sonar sensors to get values on the sensor receivers of the robot. To avoid obstalces, the robot movement goes into different states depending on the values of the sonar recievers.

6.2.4 Serial Data Transmission

In the robot application, the serial port is used to transmit messages containing the position of the robot, sonar values and motor parameters to a computer screen, to help us monitor the robot operation. From the serial output, we are able to tell whether the robot is functioning as required.

6.3 Position Calculation

The position of the robot is calculated from the pulses generated by the right and left wheels. The robot position is calculated as its x and y co-ordinates and the heading (direction) in which the robot is pointing.

The position is recalculated whenever the wheels generate pulses. From the dimensions of the robot, it is calculated that a wheel moves d=7.11mm for each wheel pulse. Due to time delays before performing this calculation, several wheel pulses may occur for each calculation.

The position calculation is one of two cases:

Case I: When the number of right wheel pulses (n) is equal to the number of left wheel pulses (m), the robot moves in a straight line.

Suppose O is the origin and robot is at point A with co-ordinates (x1,y1) and heading θ.

The new position B(x2,y2) is given by :

$x2 = x1+n*d*\cos\theta$

$y2=y1+m*d*\sin\theta$

The heading θ does not change.

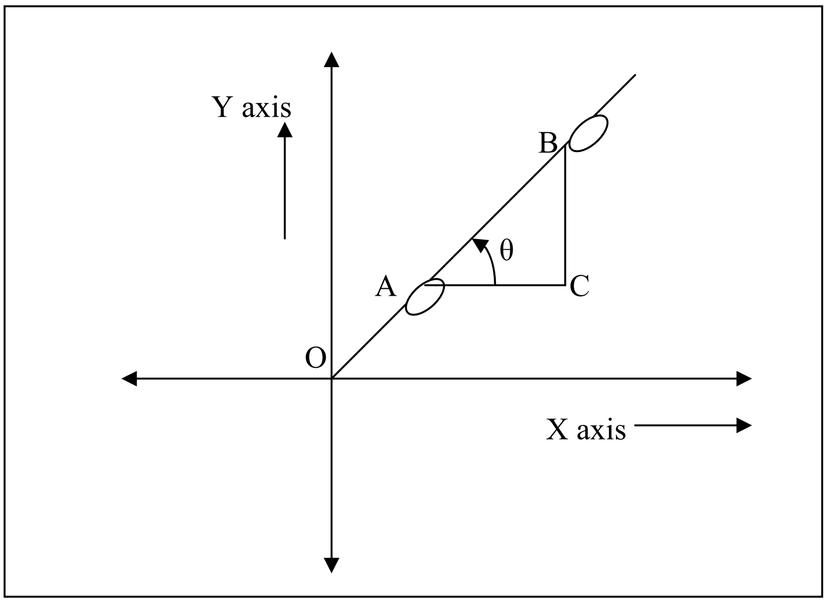


Figure 17 Position of robot when pulses from wheels are equal

Case II: When the number of right wheel pulses is not equal to the number of left wheel pulses

Suppose the left wheel moves from A to B, the right wheel moves from E to F and the midpoint between the wheels moves from C to D. The initial robot position is C (x1, y1) and heading θ.

Let

l= number of pulses from left wheel

r= number of pulses from the right wheel

d= distance moved by wheel for each pulse

sep= seperation of wheels

We assume the robot moves in an arc whose centre is at P as shown in Figure 18. The radius PC of this circle is R and the angle subtended at the centre of the circle is φ. Then:

$\Delta\theta=\varphi$

$R=((l+r)/(l-r))*(sep/2)$ (1)

$(r*d)-(l*d)=sep*\varphi$ (2)

By arranging the terms in (2), we get

$\varphi=(d/sep)*(r-l)$ (3)

Using the values of R and φ from equations (1) and (3) we get the values of new position of the robot as :

$\Delta x = x1-(R*\sin\theta)+ (R*\sin(\theta+ \varphi))$
$\Delta y = y1-(R*\cos\theta)- (R*\cos(\theta+ \varphi))$

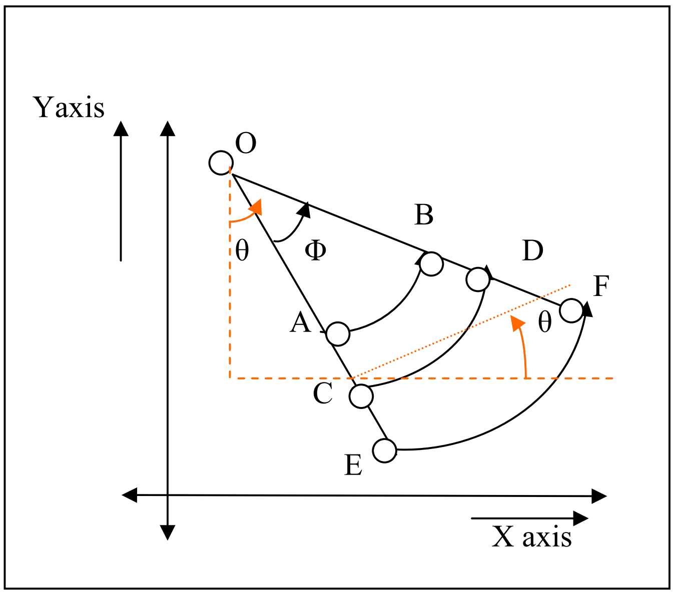


Figure 18 Position of robot when pulses from wheels are not equal

6.4 Obstacle Avoidance

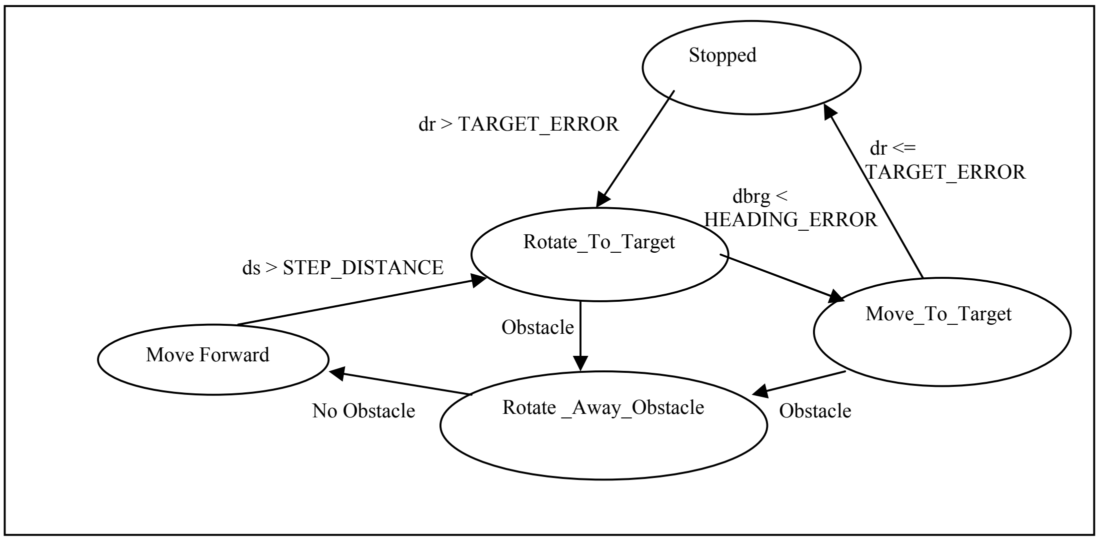


Figure 19 State Machine for Obstacle Avoidance

The robot uses the state machine shown in Figure 19 to move towards the target while avoiding obstacles. The robot enters the state machine in the Stopped state. If the robot is not at the target, then it rotates to point towards the required target position and moves towards the target. If an obstacle is encountered, the robot rotates to point parallel to the obstacle and moves forward a fixed distance before rotating towards the target position again. This procedure is necessary because the robot sonar sensors can only detect obstacles in front of the robot.

The states operate as follows:

- **Stopped**

In this state, the robot remains stopped if it within 10cm (TARGET_ERROR) of the target position. Otherwise the robot goes to the Rotate_To_Target state.

- **Rotate_To_Target**

The robot rotates until it is pointing to within 10 degrees (HEADING_ERROR) of the target. If there is an obstacle then the robot goes to the Rotate_Away_Obstacle state, otherwise it goes to the Move_To_Target state.

- **Move_To_Target**

The robot moves forward towards the target. The TARGET_ERROR has been set to 10 cm so the robot goes to the Stopped state when it is within 10cm of the target position. If the robot detects an obstacle then it goes to the Rotate_Away_Obstacle state.

- **Rotate_Away_Obstacle**

The robot rotates until the sonar sensors do not detect an obstacle. Then the robot goes into the Move Forward state.

When the left sonar receiver detects an obstacle, the right motor is stopped so the robot rotates right. When the obstacle disappears, the robot goes to the MoveForward state. Similarly, when the right sonar receiver detects an obstacle, the left motor is stopped until the obstacle disappears and the robot goes to the MoveForward state.

If both receivers detect an obstacle, the robot starts rotating towards the target destination and continues to rotate in this direction until the obstacle disappears. Then the robot goes into the MoveForward state.

- **MoveForward**

In this state the robot is attempting to get round an obstacle. The robot moves forward 50 cm (STEP_DISTANCE) then goes to the Rotate_To_Target state. This procedure is necessary because the robot cannot detect obstacles to the side.

7 SpecC Model of the Robot Application

7.1 Specification Model

In the specification model, there is one behavior that represents the entire robot control system. The inputs and outputs of this behavior are the system inputs and outputs. For the hedgehog robot the inputs are the sensor values and the counter pulses from the left and right wheels, and the outputs are the motor speed and direction values and the serial output. The model also includes the testbench behaviors. A diagram of the robot controller specification model (without the testbench) is shown in Figure 20.

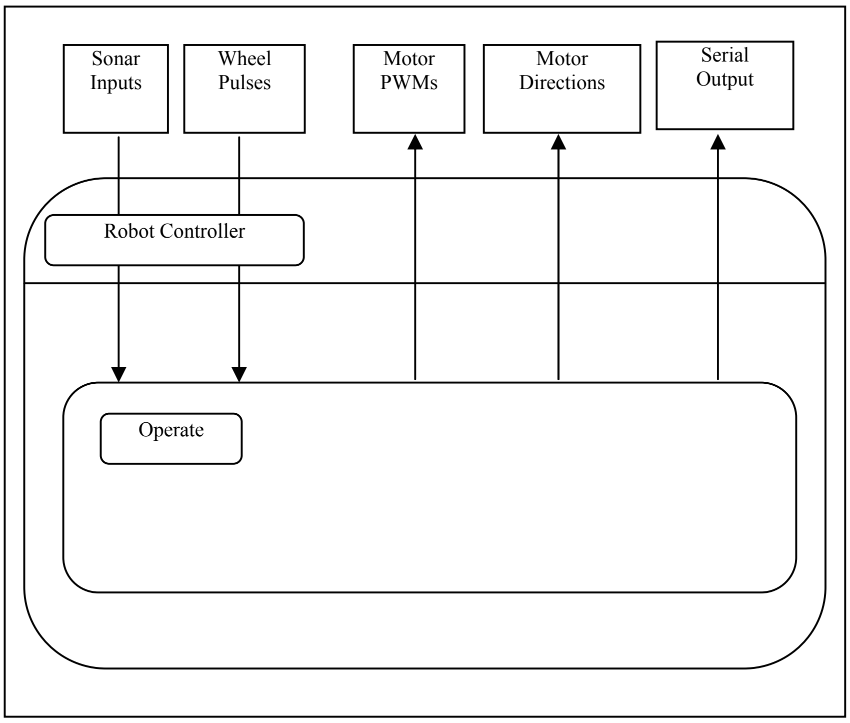


Figure 20 SpecC Specification Model

7.2 Architecture Model

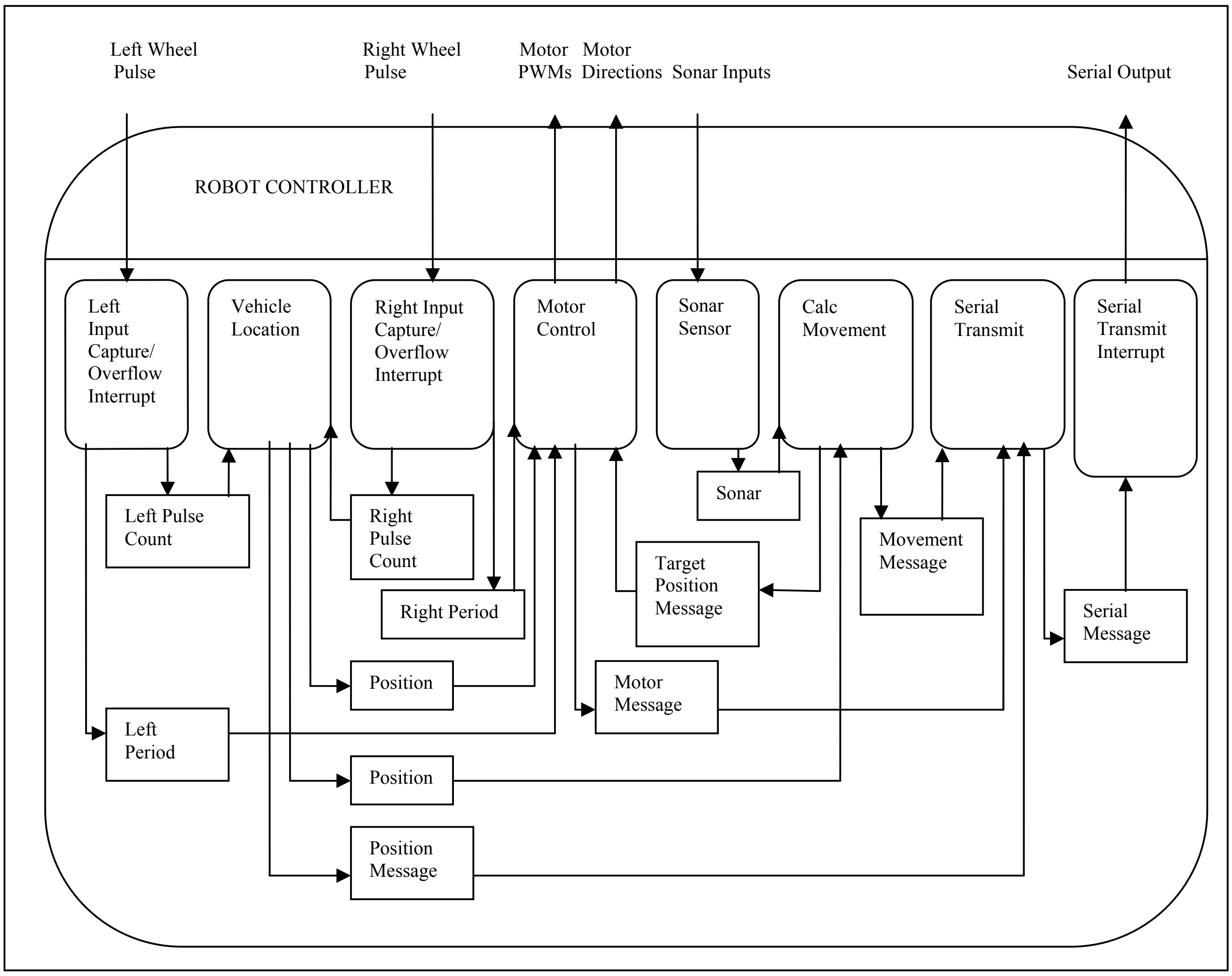
In the Architecture Model, the system is divided into tasks and the system behavior now contains these task behaviors as sub-behaviors. Each task is represented by a different individual behavior. Values are passed between these sub-behaviors as global variables in the main system behavior.

The Hedgehog robot application uses several structures to do this:

- struct Counters contains values of the number of pulses for the left and right wheels,
- struct Position contains values of the X and Y co-ordinates of the robot and the heading (orientation) and
- variable Sonar contains values received from the sonar system.

These global variables allow communication between the task sub-behaviors.

Figure 21 Architecture Model



7.2.1 Software Tasks

In order to make robot move to reach its destination avoiding obstacles coming into its way, the application has six main tasks. They are

- Input Capture and Overflow Interrupts
- Sonar Sensor
- Vehicle Location
- Calculate Movement
- Motor Control
- Serial Transmit

Input Capture and Overflow Interrupts

These interrupts count the wheel pulses and measure the time between pulses from the left and right wheels.

Sonar Sensor

This task periodically reads the values of the sonar receivers.

Vehicle Location

This task calculates the position of the robot from the pulses generated by the two wheels.

Calc Movement

This task implements the state machine for obstacle avoidance using the sonar values and position of the robot.

Motor Control

This task controls the PWM outputs to the wheel motors. When the robot is moving in a straight line, it uses the time between pulses to adjust the PWM values to equalize the right and left wheel speeds.

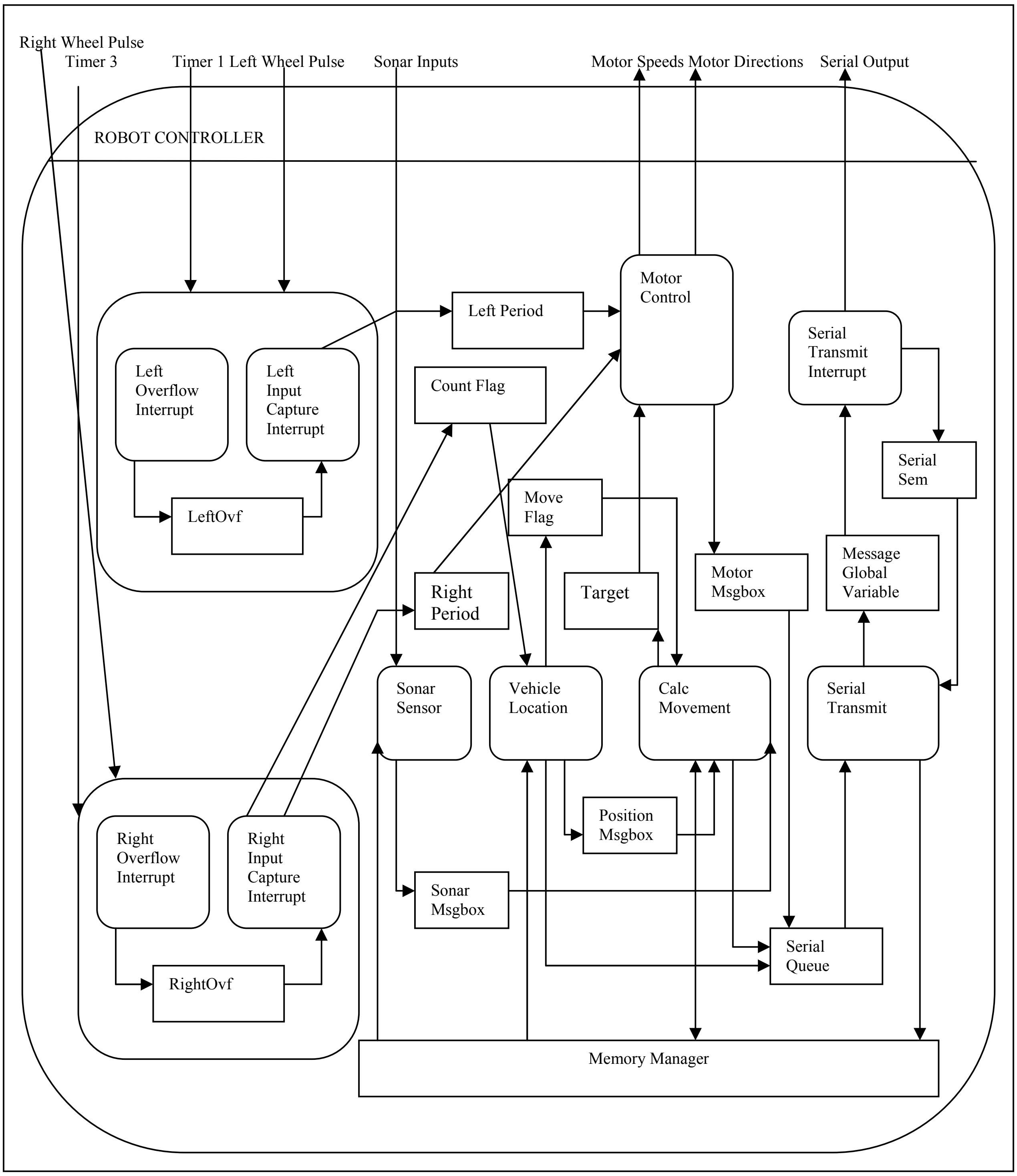
Serial Transmit

The values of the motor speeds and the position values of the robot are sent to the Serial Transmit task and the values are transmitted to a computer screen for diagnostic purposes.

7.3 Communication Model

In the Communication Model, communication channels replace the global variables. Communication between task behaviors is performed through channels representing the semaphores, message boxes, queues and event flags of the RTOS. To optimize memory utilization the communication model also uses the RTOS memory partition feature. This is modeled as a channel from which task behaviors can get memory for messages and to which memory can be returned when the message has been received. Eventflags are used to allow the Vehiclelocation, CalcMovement and Motor Control tasks to block for combination of events.

Figure 22 Communication Model



7.3.1 Task Interaction

Pulse counts generated by interrupts from the right and left wheels are sent to the Vehicle Location task to calculate position of the robot. The sonar receiver values are periodically obtained by the Sonar Sensor task and position values from the Vehicle Location task are passed to the CalcMovement task for the state machine. The CalcMovement task sends a short-term target position and the other motor control values to the Motor Control task. The Motor Control task also receives the measured time between pulses from the Input Capture interrupts. All the tasks can send messages to the Serial Transmit task.

These interactions are shown in Figure 23, along with the RTOS features (semaphores, message boxes, queues and event flags) used to implement them. The event flags allow the receiving task to wait for combinations of events.

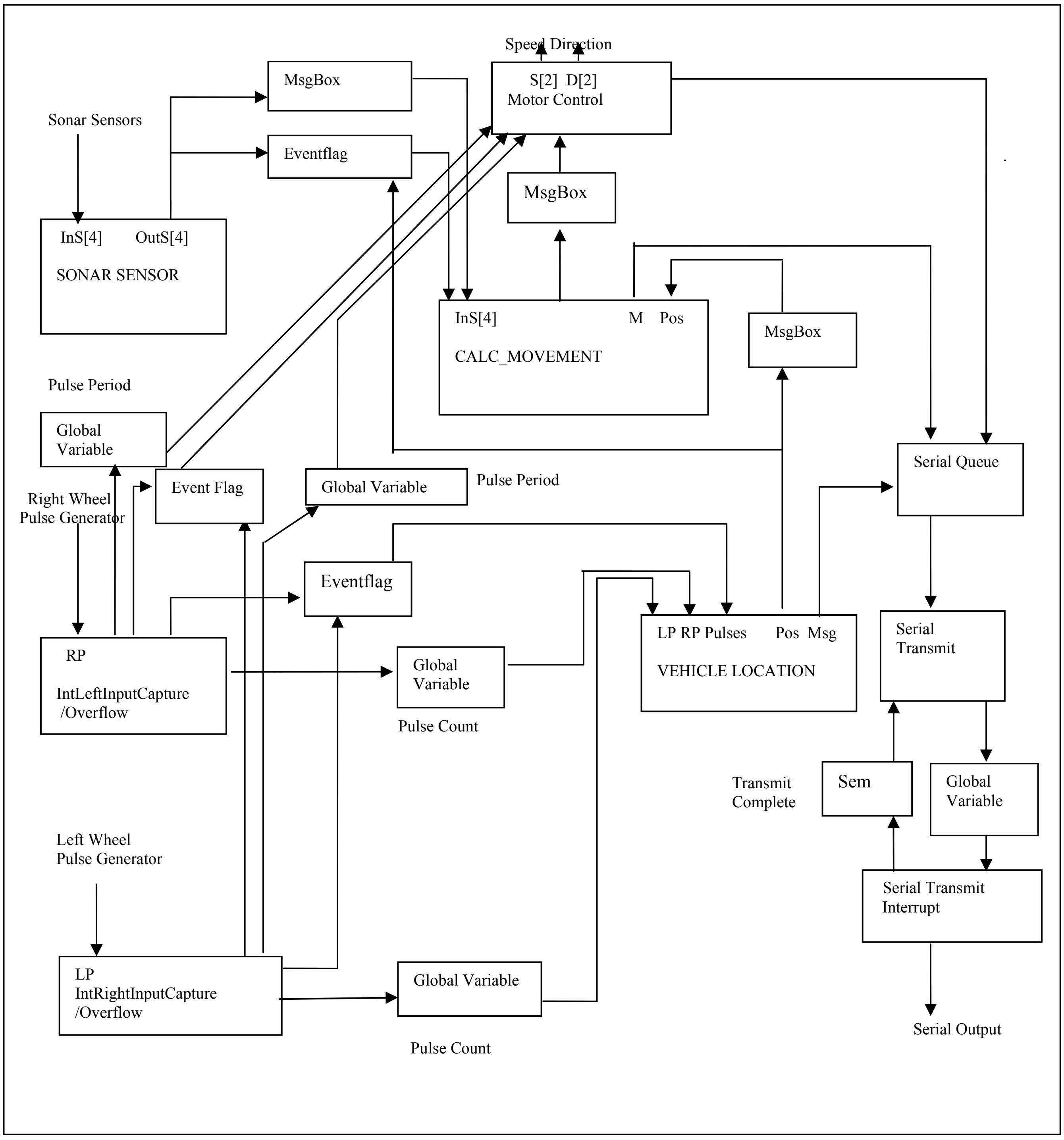


Figure 23 Task Interaction

7.4 Implementation Model

In the implementation model, the software and hardware parts of SpecC that are separated out in the architecture model are brought together and implemented on target hardware. In this thesis, only the software side has been developed so the implementation model consists of the software implementation of the SpecC model.

Here all the behaviors in the SpecC communication model are transformed to tasks in the RTOS. The obstacles and inputs defined in the SpecC testbench are removed as the real world environment of the robot replaces them. The communication channels using semaphores, messageboxes, queues and eventflags in SpecC are replaced by the corresponding RTOS features and function calls. The interrupts modeled in SpecC are replaced by interrupt service routines with C code implementation.

7.5 SpecC Testbench

The SpecC model must contain a *testbench* to simulate the external inputs of the system being modeled. The testbench consists of behaviors that perform this simulation. These behaviors use the system outputs to calculate values for the system inputs.

The sonar input behavior contains the main portion of the testbench is also the main portion of the whole simulation having information about the location of obstacles and the characteristics of the sonar sensor system and uses the calculated location of the robot to periodically generate values for the sonar inputs. Similarly, the wheel pulse input behavior uses the wheel speed to calculate the time between wheel pulses, and generates simulated interrupts at the appropriate times.

In the Specification Model, the testbench main functions

```
init.main();     // testbench initilization
setup.main(); //  application initilization
while (1)
{

 input.main();  // calculates testbench inputs
 operate.main();// application
 output.main(); // handles testbench controls
}
```

Parallel execution of behaviors in system specification is achieved by par statement.

In Architectural Model, the task Operate used in the testbench would make other tasks sonarinput, intleftinputcapture, intrightinputcapture, sonarsensor, calc_movement, vehiclelocation, motorcontrol,serialtransmit and intserialtransmit run in parallel which in SpecC is changed to :

```
par
{
     sonarinput.main();//testbench sonar receiver values
     intleftinputcapture.main();//testbench left wheel pulse
     intrightinputcapture.main();//testbench right wheel pulse
     sonarsensor.main();//sonar sensor values from 4 receivers
     calc_movement.main();//maneuvering of the robot
     vehiclelocation.main();//distance calculations
     motorcontrol.main();//looking out robot speed and direction
     serialtransmit.main();//transmitting messages on the CPU
     intserialtransmit.main();
}
```

In Communication Model, the task Operate used in the testbench would make other tasks sonarinput, intleftinputcapture, intrightinputcapture, sonarsensor, calc_movement, vehiclelocation, motorcontrol, serialtransmit, intserialtransmit, incNow run in parallel which in SpecC is changed to:

```
par
{
     sonarinput.main();
     intleftinputcapture.main();
     intrightinputcapture.main();
     sonarsensor.main();
     calc_movement.main();
     vehiclelocation.main();
     motorcontrol.main();
     serialtransmit.main();
     intserialtransmit.main();
     incNow.main();//providing behavioral time information
}
```

The task SonarInput used in SpecC provides details about the obstacle, how the receiver of the sensor would get values to be used in movement of robot.

7.5.1 Testbench Sonar Input model

In SpecC, I used structures to construct the model of the obstacle as

```
#define NUM_OBSTACLES 2
struct Point
{
      float x;
     float y;
};
struct Quad
{
      struct Point p[5];
};
struct Quad obs[NUM_OBSTACLES] =
{{{{-0.3, 0.5}, {0.3, 0.5}, {0.3, 0.7}, {-0.3, 0.7}, {-0.3,
0.5}}},{{{0.3, 1.0}, {1.0, 1.0}, {1.0, 1.3}, {0.3, 1.3}, {0.3, 1.0}}}};
```

This in real time is taken care off by the sensors.

The obstacles are defined in the program code as polygons. The co-ordinates of the corners of the obstacle polygons are defined in the input module.

The Sonar sensors are modeled as shown in Figure 22. The dimensions used in this model have been obtained experimentally. The sonar system detects an obstacle when any boundary line of the sensor pattern intersects any edge of an obstacle. The obstacles are all large enough that this guarantees detection.

While working with robot in the real environment, Sonar sensors present on the robot take care of the values received by the sensor and robot avoids the obstacle as per the user but in SpecC, user himself has to sort out a way to achieve change the values of the sensor as well.

Sensors are represented by a shape as F'B'G'A' and FBGA in SpecC shown in Figure 24 nearly similar to the specifications of the sensors in the real time environment. Assuming the receivers at the far end of robot would turn on when the robot reaches 15cm away from the obstacle and the receivers at the near end of robot would turn on when the robot reaches 5cm away from the obstacle and using the formula for finding intersection of two straight lines that is of obstacle and sensor and the position of the robot, sonar receiver values were generated.

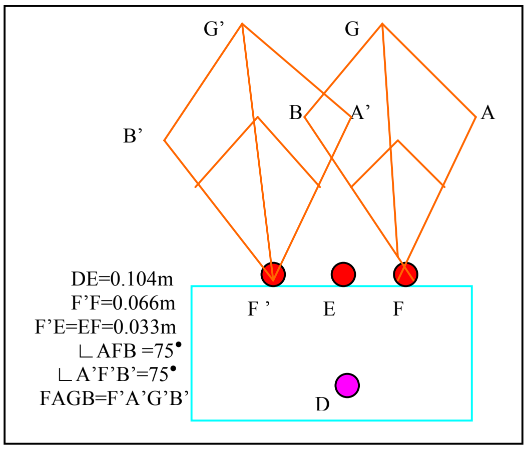


Figure 24 Robot Model In SpecC for Sonar Sensor Calculations

7.5.2 Testbench Wheel Pulse Input Model

This part of the testbench model uses the motor speeds and dimensions of the wheels to calculate the time between pulses from wheel and generates pulse inputs at the appropriate time.

7.5.3 Testbench Output Model

For this application the output function displays serial port message on the computer screen.

7.6 Software Execution

The SpecC language is easy to learn for anyone familiar with the C programming language. SpecC has additional features for modeling embedded systems. In addition the user must write testbench code that drives the SpecC model to run an application in the desired manner.

To execute SpecC using Windows, it is necessary to use Cygwin, a Linux like environment for Windows to run the SpecC compiler.

8 Results

This research describes the design of a SpecC model of a robot avoiding obstacles. Using the SpecC methodology, a specification model of the robot is transformed to an architecture model and then to a communication model. The model is verified by comparing it with the actual robot performance.

In the communication model, most of the essential features of the RTOS have been incorporated. The research shows the interaction of the major design tasks (behaviors) generated at each level from the specification model to the communication model, and the refinement of communication elements (global variables and channels) as the model was developed. In this research, a state machine was used to make the robot avoid obstacles, taking into account various elements such as the sonar inputs, the root location and the desired target position.

8.1 Robot Manoeuvring Pictorial Representation

The Figure 25 below shows the path followed by the robot avoiding the obstacles to reach its target destination. The two blocks (shown in red) are the obstacles and the line (shown in blue) is the path of the robot traveled avoiding the obstacles.

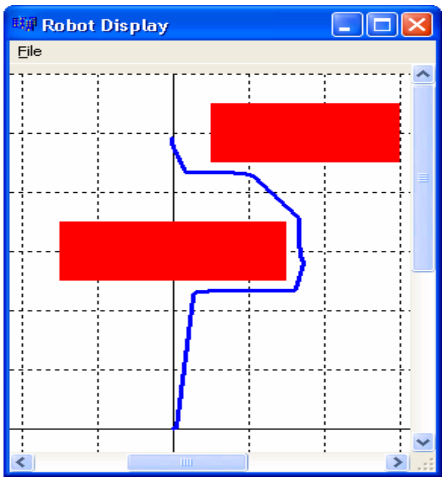


Figure 25 Example of Obstacle Avoidance

8.2 Testing the Model

The SpecC model has been tested by implementing the model in C code on the Hedgehog robot. The performance of the Hedgehog robot was then compared with the SpecC model performance.

Typical results are shown in Figures 26 and 27. The robot was required to travel forward 3 metres, avoiding two obstacles as shown in the figures. The robot paths are shown in each case.

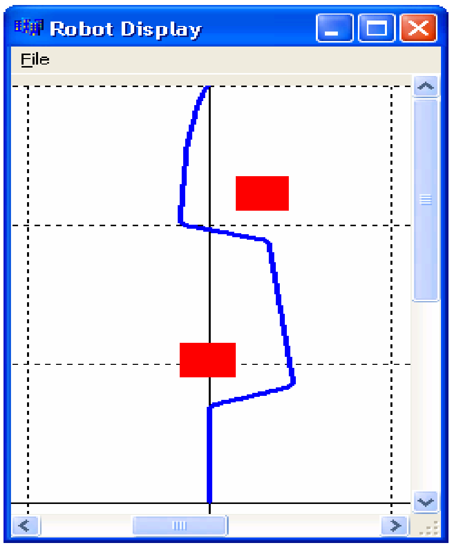


Figure 26 SpecC Model Performance

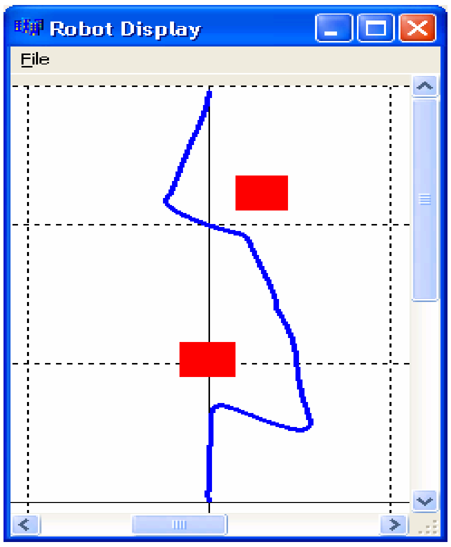


Figure 27 Actual Robot Performance

9 Conclusions and Future Work

In summary, this research has developed a model of a RTOS with the SpecC SLDL and a real-time application in which a robot avoids obstacles to reach its destination. The model gives similar results to the actual robot. This research has produced satisfactory results and hence the methodology adopted is potentially promising for future work.

This research shows how the SpecC SLDL can be used to model a RTOS incorporating essential features and model a small real time application for robot obstacle avoidance. The SpecC model depicts the real world application and allows the exploration of different architecture and communication methods. It is very difficult for a programmer to model an application in SpecC to make it function as it would in the real time environment. It was challenging to control every aspect that was involved, and this research has made every effort to minimize any biases or flaws, which could greatly affect the results. The model functions similar to the application in the real world but not exactly the same.

Further work is required to complete the model of the MicroC/OS-II RTOS using SpecC. We have not modeled all the features of the RTOS, although most could be modeled in the same style as we have presented here.
We have also not considered the important feature of timing, which would allow exploration of the time taken to run tasks and RTOS operations. An accurate timing model would allow investigation of the responsiveness of systems using the RTOS. It is a vital feature for the engineer as it reveals where most of the time is being consumed in the behaviors. This provides information that will identify which behaviors need to be improved for the application to run as required. This may necessitate some behaviors being shifted from software to hardware.

Differences between the SpecC model performance and the actual robot performance are caused by the dynamics of the robot that have not been included in the SpecC model. For example, at present the model assumes the motor speed changes as soon as the PWM output to the motor is changed. The model should also include some random variation in the timing of the wheel pulses, as this is a significant issue in controlling the robot.

In future, an FPGA could be used as the controller. The SpecC model could be converted to a hardware description language such as VHDL or Verilog to build an application and test it on FPGA.

The FPGA could include a processor element running the software part of the implementation with the hardware part being in the same FPGA.

References

[1] SpecC, SpecC Open Technology Consortium, “HomePage,” http://www.specc.gr.jp/eng/index.html, visited on 08/12/2006.

[2] SystemC, SystemC Welcome, “Home Page,” http://www.systemc.org, visited on 08/12/2006.

[3] R. Doemer, A. Gerstlauer and D. Gajski. *SpecC Language Reference Manual, Version 1.0.*SpecC Technology Open Consortium, March (2001)

[4] M. Fujita and H. Nakamura, “the standard SpecC language”, *Proceedings of the 14 th International Symposium on System Synthesis*, Montreal, Canada, pp 81-86 (2001).

[5] D. Gajski, J.Zhu et al. *SpecC: Specification Language and Design Methodology*, Kluwer Academic Publishers, Norwell, Massachusetts (2000).

[6] J. Collins, A. Nukala, “SpecC RTOS Model for Robot Obstacle Avoidance”, *International Conference for Autonomous Robots and Agents (ICARA 2006)*, Massey University, Palmerston North, New Zealand, 11-14 December (2006).

[7] R. Domer and D. Gajski, “reuse and protection of intellectual property in the SpecC system”, *Design Automation Conference Asia and South Pacific 2000*, Yokohama, Japan, pp 49-54 (2000).

[8] D. Gajski and R.H. Kuhn, “guest editor’s introduction: new VLSI tools”, *IEEE Computer* 16, pp 11-14 (1983).

[9] J. Zhu and D. Gajski, “compiling SpecC for simulation”, *Design Automation Conference Asia and South Pacific 2001*, Yokohama, Japan, pp 57-62 (2001).

[10] M.K. Chung, S. Yang, S.H. Lee and C.M. Kyung, “system level HW/SW co-simulation framework for multiprocessor and multithread SoC”, *International Symposium on VLSI Design, Automation Test, 2005,* Taiwan, pp 177-180 (2005).

[11] H. Jain, D. Kroening and E. Clarke, “verification of SpecC using predicate abstraction”, *Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design 2004, MEMOCODE ’04*, San Diego, California, pp 7-16 (2004).

[13] D. Ksu and G.De Micheli, “HardwareC -a language for hardware design(version 2.0).Technical Report CSL-TR-90419”,*Stanford University* (1990).

[14] I. Page, “constructing hardware-software systems from a single description”, *Journal of VLSI Signal Processing*, pp 87-107 (1996).

[12] UML, IBM Rational Software-Unified Modeling Language, “Home Page.” http://www.rational.com/uml/index.html, visited on 08/12/2006.

[15] M. Fujita, "system level methodologies from the viewpoint of formal verification", *Proceedings of the 5th International Conference ASIC*, Shanghai, China, pp 6-10 (2003)

[16] S. Honda and H. Takada, "evaluation of applying SpecC to the integrated design method of device driver and device", *Design, Automation and Test in Europe Conference and Exhibition,* Munich, Germany, pp 138-143 (2003).

[17] TRON Association, µITRON 4.0,TRON Documents, "Title Page", http://www.assoc.tron.org/eng/document.html , µITRON 4.0 Specification Ver.4.00.00 http://www.assoc.tron.org/spec/itron/mitron-400e.pdf ,visited on 08/12/2006

[18] W. Mueller, R. Domer and A. Gerstlauer, "the formal execution semantics of SpecC", *Proceedings of the 15 th International Symposium on System Synthesis*, Kyoto, Japan, pp 150-155 (2002)

[19] S.B. Saoud, D. Gajski and A. Gerstlauer. "co-design of emulators for power electric processes using SpecC methodology", *IECON 02, 28 th Annual Conference of the Industrial Society*, Spain, volume 3,pp 2143-2148 (2002).

[20] Motorola, Inc., Semiconductor Products Sector, DSP Division, DSP 56600 16-bit Digital Signal Processor Family Manual, http://www.freescale.com/files/dsp/doc/user_guide/DSP56600FM.pdf, DSP56600FM/AD visited on 08/12/2006.

[21] S.B. Saoud, D. Gajski and A. Gerstlauer, "seamless approach for the design of control systems for power electronics and electric drives", *IEEE International Conference on System, Man and Cybernetics*, Tunisia, pp 6 (2002).

[22] Atmel Corporation,Atmel ATMega128L microcontroller Manual, http://www.atmelchips.com/dyn/resources/prod_documents/doc2467.pdf , visited on 0812/2006.

[23] SpecC, SpecC Reference Compiler, "Title Page," http://www.ics.uci.edu/~specc/reference/ , visited on 08/12/2006.

[24] T. Sakunkonchak, S. Komatsu and M. Fujita, "synchronization verification in system level design with ILP solvers", *Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design 2005,*Italy, pp 121-130 (2005).

[25] T. Yoneda, A. Matsumoto, M. Kato and C. Myers, "high level synthesis of timed asynchronous circuits", *11th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2005*, New York, pp 178-189 (2005).

[26] H.Yu, A.Gerstlauer and D. Gajski, "RTOS scheduling in transactional level models", *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Co-design and System Synthesis*, CA,USA, pp 31-36 (2003).

[27] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull - Trauring, M. Trakhtenbrot, “Statemate: a working environment for the development of complex reactive systems”, *IEEE Trans.on Software Engineering*, pp 403-414 (1990).

[28] J.L. Pino, S. Ha, E.A. Lee and J.T. Buck, “software synthesis for DSP using ptolemy”, *Journal of VLSI Signal Processing*, (1995).

[29] F. Boussinot and R.de Simone, “the ESTEREL language”, *Proceedings of IEEE*, pp 1293-1304 (1991).

[30] L. Gauthier, Y. Sungjoo, A.A. Jerraya, “automatic generation and targeting of application-specific operating systems and embedded systems software”, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pp 1293-1301 (2001).

[31] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe and A.L. Sangiovanni-Vincentelli, “task generation and compile time scheduling for mixed data-control embedded software”, 37th *Design Automation Conference*, Los Angeles, California, USA, pp 489-494 (2000).

[32] H. Tomiyama, Y. Cao and K. Murakami, “modeling fixed priority preemptive multi task systems in SpecC”, *Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI)*, Nara, Japan, pp 93-100 (2001).

[33] D. Desmet, D. Verkest and H.D. Man, “operating system based software generation for system on chip”, *37th Design Automation Conference*, Los Angeles, California, USA, pp 396-401 (2000).

[34] S. Yoo, G. Nicolescu, L. Gauthier and A.A. Jerraya, “automatic generation of fast timed simulation models for operating systems in SoC design”, *Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, pp 620-627 (2002).

[35] A.Gerstlauer, H. Yu and D.D.Gajski, “RTOS modeling for system level design”, *Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany, pp 130-135 (2003).

[36] S.A.Edwards, “the challenges of hardware synthesis from C-like languages”, *Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany volume1, pp 66-67 (2005).

[37] C.E. Stroud, R.R. Munoz and D.A. Pierce, “Behavioral model synthesis with cones”, *IEEE Design & Test of Computers*, pp 22-30(1988).

[38] D. Galloway, “the TransmogrifierC hardware description language and compiler for FPGAs”, *In Proc, FCCM*, Napa Valley, CA, pp 136-144 (1995).

[39] T. Grötker, S. Liao, G. Martin and S. Swan. *System Design with SystemC* , Kluwer Publishers, Norwell, Massachusetts (2002).

[40] D. Soderman and Y. Panchul, “implementing C algorithms in reconfigurable hardware using C2Verilog”, *In Proceedings to the conference on FPGAs for Custom Computing Machines*, Napa Valley, CA, USA, pp 339-342 (1998).

[41] K. Wakabayashi, “C-based synthesis experiences with a behavior synthesizer – Cyber”, *In Proceedings to the conference on Design, Automation and Test In Europe DATE*, Paris, France, pp 390-393 (1990).

[42] Handel C, Technology Library Celoxica, “Title Page”, http://www.celoxica.com/techlib/, *HandelC Language Reference Manual,* visited on 08/12/2006.

[43] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, T. Nomura , “A C-based synthesis system, Bach, and its application”, *In Proc. ASP-DAC*, Yokohama, Japan, pp 151-155 (2001).

[44] S. Abdi and D. Gajski, “Functional Validation of System Level Static Scheduling”, *Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany, volume 1, pp 542-547 (2005).

[45] L.Cai and D. Gajski, “transaction level modeling: an overview”, *Proceedings of the 1st IEEE/ACM/IFIPinternational conference on Hardware/software codesign and system synthesis*, Newport Beach, CA, USA, pp 19-24 (2003).

[46] D. Björklund and J. Lilius, “a language for multiple models of computation”, *Proceedings of the 10 th International Symposium on Hardware/Software Co-design*, Estes Park, Colorado, USA, pp 25-30 (2002).

[47] L. Séméria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake and B. Pangrle, “RTL C-based methodology for designing and verifying a multi-threaded processor”, *Proceedings of the 39th conference on Design automation*, New Orleans, Louisiana, USA, pp 123-128 (2002).

[48] A. Gerstlauer and D.D. Gajski, “system level abstraction semantics”, *Proceedings of the 15 th International Symposium on System Synthesis*, Kyoto, Japan, pp 231-236 (2002).

[49] A. Bouchhima, S. Yoo and A. Jerraya, “fast and accurate timed execution of high level embedded software using hw/sw interface simulation model”, *Proceedings of ASP-DAC 2004, Design Automation Conference, Asia and South Pacific*, Yokohoma, Japan, pp 469-474 (2004).

[50] L. Cai, P. Kritzinger, M. Olivares and D. Gajski, “top down system level design methodology using SpecC, VCC, SystemC”, *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, pp 1137 (2002).

[51] D. Verkest, J. Kunkel and F. Schirrmeister, “system level design using C++”, *Proceedings of the Conference on Design, Automation and Test in Europe,* Paris, France, pp 74-83 (2000).

[52] A. Gerstlauer, R. Domer, J. Peng and D. Gajski. *A Practical Guide with SpecC*. USA: Kluwer Academic Publishers, Norwell, Massachusetts (2001).

[53] L. Cai, S. Verma and D.D. Gajski, “comparison of SpecC and SystemC languages for system design Technical Report CECS-03-11”, *University of California* (2003).

[54] J.J.Labrosse. *MicroC/OS-II The Real Time Kernel*, Second Edition. USA: CMP Books, Lawrence, Kansas (2002).

Appendix

Companion CD

This thesis includes a CD that contains all the source code for the SpecC models. It is assumed that you have a Microsoft Windows 95, 98, NT, 2000, or XP computer system running on an 80x86, and Pentium-class, or AMD, processor.

To run the executable files, you need to have the Cygwin environment that can be freely downloaded from the website http://www.cygwin.com/

Insert the CD into your CD-ROM drive, and execute the file tb.exe in Cygwin. These are executable files for the specification model, architectural model and communication model in appropriate folders. The files with an extension of .sc are the SpecC files that can be viewed using a text editor.

When the executable is run, you will see a continuous flow of messages that describes how the robot is moving to avoid the obstacles to reach its destination.