FORMAL SPECIFICATION OF A WALL-CLIMBING ROBOT USING Z – A CASE STUDY OF SMALL-SCALE EMBEDDED HARD REAL-TIME SYSTEM

RADZIAH MOHAMAD¹, DYG. NORHAYATI ABG. JAWAWI², SAFAAI DERIS³ & ROSBI MAMAT⁴

Abstract. The task of checking whether a real-time system satisfies its timing and concurrency specifications is extremely important. One major area of research addressing software reliability aspect is called formal method, which attempts to prove the correctness of programs with respect to system specifications. Since, timing and concurrency properties can be very important in the operation of real-time systems, there is a need for applying formal methods to verify timing properties. This paper investigates the process of building a formal specification of a small-scale embedded hard real-time systems using Z. It is expected that the formal specification presented in this paper can provide assistance in analysing design trade-offs early in the development process. It is also expected that this paper can act as the foundation for any upcoming formal methods related project especially for small-scale real-time systems project.

Key words: Software reliability, formal specification, Z, hard real-time, small-scale systems

1.0 INTRODUCTION

The development of a wall climbing robot is currently requested strongly, on behalf of human operator, to perform dangerous operations on the surface of a wall. The
potential applications for the robot are for operations such as fire fighting in high-rise
buildings, wall surface cleaning or decorating and transporting along wall surfaces.
A wall-climbing robot (WCR) is currently under development at Universiti Teknologi
Malaysia. The WCR can be categorized as a small-scale embedded hard real-time
system.

Real-time systems must produce their results within specified time intervals. The
correctness of the system is therefore not only dependent on the logical results but
also the time at which results are produced. According to the timing constraints,
real-time systems can be classified as either hard real-time systems or soft real-time
systems. In hard real-time systems, timing is critical where the lateness of the results
is not permitted under any circumstances since late response are either useless or
even dangerous. In soft real-time systems, timing correctness is important but not
critical. Solving timing constraints and co-ordinating the communication between
co-operating processes are therefore important in ensuring the software reliability.

Small-scale embedded hard real-time systems are becoming more sophisticated
and usually offer many functions in one product. As a result, software development
for small-scale embedded hard real-time systems are growing in scale and becoming
very complex over the years. Furthermore, a real-time system is inherently concur-
tent and multitasking since it has to react to and process numerous events simulta-
neously. Thus, developing software for even small-scale embedded real-time sys-
tems can be very difficult.

Due to the complexity and the nature of the control software for the WCR sys-
tem, proving the correctness of the software requirements of the robot earlier is
important so as to reduce the costs of requirement errors occur in later phases of
software development life cycle. The purpose of this paper is to present the process
of building a formal specification of a small-scale embedded hard real-time systems,
in particular, the specification of control firmware for a four-legged WCR.

The organization of this paper is as follows: Section 2 will discuss the informal
specification of the WCR. This will be followed by the formal specification of the
WCR in the Section 3. Specifications for each component of WCR requirement
were derived using Z. Section 4 will then conclude the paper.

2.0 INFORMAL SPECIFICATION OF THE WALL-CLIMBING
ROBOT FIRMWARE

The main function of the embedded digital controller is to move the four legs of the
robot with a predefined sequence during climbing operation. The block diagram of
the embedded controller for the wall-climbing robot is shown in Figure 1.

The main functional operation of the robot controller can be divided into three
major groups i.e.:
The embedded controller monitors its environment using some sensors *i.e.* collision sensors, proximity sensors and pressure sensors. Collision sensors send signal to controller when the sensor collides any obstacles. Obstacle sensors detect the presence of a distance obstacles. This environment must be monitored typically every 500 milliseconds to detect the presence of obstacles using the collision and the obstacle sensors during the forward and reverse movement of the robot. Three position sensors in the form of rotary potentiometers at each leg measure the joint angles of the leg during the leg movement. Position sensors read the current position of the legs joint angle. The highest priority task in the controller software is the motor control task with a cycle of 50 milliseconds. This is to ensure the correct movement of the legs.

### 3.0 FORMAL SPECIFICATION OF THE WALL-CLIMBING ROBOT Firmware

Formal methods have been studied for a long time for the development of software. Formal methods have been thought of as having great promise for the development
of real-time systems [1]. The term formal method refers to the use of the techniques from logic and discrete mathematics in the specification, design and construction of computer systems and software [2]. The essence of the formal methods is a formal specification. Since timing and concurrency properties are very important in the operation of real-time systems, there is a need for applying formal methods to specify timing properties. In order to increase the reliability of the software developed, developing specification that can be verified during an early stage of the development is hence needed.

The formal specification notation \( Z \) is one of the most popular and widely used notations for the formal specifications and development of software and hardware systems [3]. \( Z \) is a formal specification language based on a standard set theory and uses a mathematical notation. The specification is divided into a number of units or blocks [4]:

(a) Basic type definition defining the values which constants and variables may take.
(b) Global constant and variable definition within the system and any constraints on them.
(c) Schemas defining the state and operations of the system.

Traditionally, \( Z \) is widely perceived as being unsuitable to specify concurrent and temporal behaviour of the system [3]. Recently however, there has been considerable interest in applying \( Z \) to specify a concurrent and temporal behaviour of the system. Much of this work has concentrated on integrating \( Z \) with formalisms better suited to specifying concurrent and temporal behaviour, such as Petri Nets [5], Temporal Logic [3], CSP [6] and CCS [6]. However, a main disadvantage shared by all these approaches is the difficulty of reconciling the semantics of the separate notations, resulting in problems of compatibility with the standardized definition of \( Z \) and existing \( Z \) tools for proof and type checking. Moreover, techniques for reasoning with the resultant hybrid notations generally makes poor use of the excellent proof system offered by \( Z \). This paper shows a somewhat different approach to specifying concurrent and temporal behaviour of the system in \( Z \). Rather than integrating \( Z \) with yet another formalism, modifications are made to the \( Z \) specification based on the \( Z \)'s generic model proposed by [7]. Next subsection introduces the formal specification of the WCR using \( Z \).

### 3.1 The Embedded Robot’s Control Software

This section introduces the basic data items of the robot controller’s software requirement specification. The set of controller’s operational modes is defined by the free type \( Modes \). \( YesNo \) is the type consisting of the two constants \( Yes \) and \( No \). \( OnOff \) is the type consisting of the two constants \( On \) and \( Off \).
The following types are used to describe state information of sensors. 

- **Collision_Sensor** denotes the state of a collision sensor; **On** denotes the sensor collides any obstacles, while **Off** denotes the sensor does not collide any obstacles.
- **Obstacle_Sensor** denotes the state of an obstacle sensor; **On** denotes the sensor detects the presence of a distance obstacle, while **Off** denotes the normal state (does not detect the presence of obstacle).
- **Pressure_Pad** denotes the state of the suction pads of the robot leg; **Yes** denotes enough pressure is maintained in the suction pads, while **No** denotes the pressure maintained in the suction pads is not enough.
- **PressureActivate** is used to describe whether an indicator for specific button on a pressure pumps is activated (**active**) or deactivated (**passive**).
- **Pcommand** denotes the command from the PC to the controller; **Start_Move** denotes the command to the controller to trigger the robot to start moving, while **Stop** denotes the stop moving command. **RobotState** refers to the state of the robot at time t; **Moving** denotes the robot's leg or body is moving, while **Stop_Move** denotes the leg or body is stop from moving; **Idle** denotes the leg or body is not moving.

```
RobotState == {Moving, Stop_Move, Idle}
Collision_Sensor == {On, Off}
Obstacle_Sensor == {On, Off}
Pressure_Pad == {Yes, No}
PressureActivate := active | passive
Pcommand := Start_Move | Stop
```

The abstract system’s state of the robot controller is specified by the schema **RobotController**. There is one major system variable **mode** of type **Modes** representing the current operational mode. The other system variables can directly be derived from the system specification.

```
RobotController
mode : Modes
pc : Pcommand
leg : Modes
obstacle : Obstacle_Sensor
collar : Collision_Sensor
pad : Pressure_Pad
pos : N
```
During initialization, the controller is in the *Idle* mode and the Robot is in the *Idle* state.

\[
\text{RobotCtrlInit} \equiv [\text{RobotController} \mid \text{mode} = \text{Idle}]
\]
\[
\text{RobotInit} \equiv [\text{RobotController} \mid \text{mode} = \text{Idle}]
\]
\[
\text{Initialize} \equiv \text{RobotCtrlInit} \land \text{RobotInit}
\]

The following schemas define predicates on the controller’s state that are used to model the interdependencies between the dynamic behavior and the current controller’s state.

\[
\text{PressureIsEnough} \equiv [\text{RobotController} \mid \text{pad} = \text{Yes}]
\]
\[
\text{ObstaclePresent} \equiv [\text{RobotController} \mid \text{obstacle} = \text{On} \land \text{mode} = \text{Waiting_Pc}]
\]
\[
\text{CollisionActivated} \equiv [\text{RobotController} \mid \text{collide} = \text{On}]
\]
\[
\text{CollisionMode} \equiv [\text{RobotController} \mid \text{mode} = \text{Scanning_Collision}]
\]
\[
\text{ObstacleMode} \equiv [\text{RobotController} \mid \text{mode} = \text{Scanning_Obstacle}]
\]
\[
\text{PressureMode} \equiv [\text{RobotController} \mid \text{mode} = \text{Suctioning_Pad}]
\]
\[
\text{PositionMode} \equiv [\text{RobotController} \mid \text{mode} = \{\text{Scanning_Leg}, \text{Scanning_Body}\}]
\]

Informally, the specification above asserts that when the leg’s pressure is enough, the pad variable is set to *Yes*. In the presence of a distance obstacle, the obstacle sensor indicator will be triggered to *On* and the mode will be changed to *Waiting PC* command as to whether to keep on moving or stop from moving. Whenever the collision sensor collides with any obstacles, the collision sensor indicator will be triggered to *On*. The next four specifications assert the relevant mode of the controller as per operation.

### 3.2 Pressure Sensor Operation

Schema *PressureState* models the state of active, passive buttons defined for each state of the suction pads of the robot leg.

\[
\text{PressureState}
\]
\[
\text{display: Pressure_Pad} \rightarrow \text{PressureActivate}
\]
The following schema (PressureSensor) describes the operation of the pressure sensor. Here, the Δ convention is used to include the before and after components of the state schema PressureState, PressureLeg and PressureMode. If a pressure pump button indicator of suction pad, pad?, is passive, it will be activated only if the suction pad has no enough pressure (pad? = No). In the former case, this will be as a pre-selection, so that the pressure pump will automatically be activated as soon as the suction pad has no enough pressure. If the pressure pump is activated and the suction pad has enough pressure, pumping the pressure into the suction pad has no effect since when the suction pad has no enough pressure, the pressure pump will automatically be activated in any case.

Collision Sensor operation

Collision sensor sends signal to the controller when the sensor collides any obstacles. Schema CollideState models the state of Moving, Stop_Move of the robot’s leg or body defined for each state of the collision sensor.

The following schema (CollisionSensor) describes the operation of the collision sensor. If the state of the robot’s leg or body is Moving, the Stop_Move command will be sent to the robot by the controller if the collision sensor sensor? collides any obstacles (sensor?=On). In the former case, this will be as a pre-selection, so that the robot will automatically be stopped as soon as the sensor collides any obstacles.
3.4 Obstacle sensor operation

The obstacle sensor sends signal to a controller when it detects the presence of a distance obstacle. Schema \textit{ObstacleState} models the state of \textit{Moving}, \textit{Stop_Move} of the robot’s leg or body defined for each state of the obstacle sensor.

\begin{center}
\begin{tikzpicture}
    
ode at (0,0) {	extbf{ObstacleState}};
    
ode at (0,-1) {	ext{event 1: Obstacle Sensor $\rightarrow$ RobotState}};
\end{tikzpicture}
\end{center}

The following schema (\textit{ObstacleSensor}) describes the operation of the obstacle sensor. If the state of the robot’s leg or body is \textit{Moving}, the signal will be sent to the PC when the sensor \textit{sensor1} detects a distance obstacle (\textit{sensor1} = On) and the controller will then have to wait for a reply command from the PC as for the robot either to stop moving or keep on moving. While waiting for the PC’s command, the robot’s leg or body state will be maintained to a \textit{Moving} state. In the former case, this will be as a pre-selection, so that a signal will automatically be sent to the PC as soon as the sensor detects a distance obstacle.

\begin{center}
\begin{tikzpicture}
    
ode at (0,0) {	extbf{ObstacleSensor}};
    
ode at (0,-1) {	ext{event1: sensor1: Obstacle Sensor}};
\end{tikzpicture}
\end{center}
3.5 Position Sensor operation

Position sensor reads the current position of legs and body of the robot. The robot has four legs and a body. They are formally defined by the free type $\text{Leg}$ and $\text{Body}$.

$$\text{Body} ::= \text{body}$$

$$\text{Leg} ::= \text{leg1} \mid \text{leg2} \mid \text{leg3} \mid \text{leg4}$$

Set $\text{RobotPos}$ combines leg identifiers of the robot of type natural numbers. Set $\text{RobotPos1}$ combines body identifier of the robot of type natural numbers.

$$\text{RobotPos} == \{\text{leg1}, \text{leg2}, \text{leg3}, \text{leg4}\} \times \mathbb{N}$$

$$\text{RobotPos1} == \{\text{body}\} \times \mathbb{N}$$

Schema $\text{PosState}$ models the current position defined for each leg and body of the robot and the target position defined for each leg and body of the robot.

$$\text{Current} == \mathbb{N}$$

$$\text{Target} == \mathbb{N}$$

$$\text{PosState}$$

- $\text{CurrentPos}: \text{RobotPos} \rightarrow \text{Current}$
- $\text{TargetPos}: \text{RobotPos} \rightarrow \text{Target}$
- $\text{CurrentPos1}: \text{RobotPos1} \rightarrow \text{Current}$
- $\text{TargetPos1}: \text{RobotPos1} \rightarrow \text{Target}$

The following schema ($\text{PositionSensor}$) describes the operation of the position sensor. If the current position of the leg or body is different from its target position, a signal will be sent from the controller to the robot's leg or body to move to the target position (i.e. the current position + the error or the difference between the current position and the target position). In the former case, this will be as a pre-selection, so that the robot's leg or body will automatically be moved to the target position as soon as there is a difference between the target position and the current position.
3.6 Motor Control operation

The motor receives the computation of control signal (the error computation between the target position and the current position of each legs and body of the robot) from a controller to ensure the correct movement of each leg and body. Schema MotorControl describes the operation of the motor control with the output of a computation error, error! sent to the motor control.

3.7 Serial communication with external PC

PC commands focus on telling the robot to start moving or stop from moving. Schema External_pc describes the operation of the external PC.
3.8 Temporal specification of the WCR

The temporal specification of the WCR is based on the improved Z recipe by [7]. The next specifications define states which specify all the operations which interact. This process is called promotion. SendCollisionSignal for example, specifies all the individual operations which interact in sending the collision signal to the controller.

\[
\begin{align*}
\text{SendCollisionSignal} & \equiv \text{CollisionSensor} \land \text{CollideIsActivated} \land \text{CollisionMode} \\
\text{SendObstacleSignal} & \equiv \text{ObstacleSensor} \land \text{ObstaclePresent} \land \text{ObstacleMode} \\
\text{SendPressureStatus} & \equiv \text{PressureSensor} \land \text{PressureIsEnough} \land \text{PressureMode} \\
\text{SendPositionStatus} & \equiv \text{PositionSensor} \land \text{PositionMode} \\
\text{SendToMotorControl} & \equiv \text{MotorControl} \land \text{PositionMode}
\end{align*}
\]

The operations can now be used to add timing constraints to the operation of the controller.

\[
\text{TimedBehavior}
\]

\[
\begin{align*}
&: \text{compt}[\text{RobotController}] \\
& \text{t bounds } ((\text{SendCollisionSignal} \land \text{RobotController} \rightarrow \neg \text{RobotController}), 500..20000) \\
& \text{t bounds } ((\text{SendObstacleSignal} \land \text{RobotController} \rightarrow \neg \text{RobotController}), 2000..30000) \\
& \text{t bounds } ((\text{SendPressureStatus} \land \text{RobotController} \rightarrow \neg \text{RobotController}), 10000..30000) \\
& \text{t bounds } ((\text{SendPositionStatus} \land \text{RobotController} \rightarrow \neg \text{RobotController}), 5..20) \\
& \text{t bounds } ((\text{SendToMotorControl} \land \text{RobotController} \rightarrow \neg \text{RobotController}), 50..100)
\end{align*}
\]

Informally, this specification asserts that the controller must receive signals from a collision sensor at least 500 milliseconds but no more than 20 seconds, from an obstacle sensor at least 2 seconds but no more than 30 seconds, from a pressure sensor at least 10 seconds but no more than 30 seconds, from a position sensor 5
milliseconds but no more than 20 milliseconds, from a motor control at least 50 milliseconds but no more than 100 milliseconds.

### 3.9 Concurrency Specification

A dynamic specification describes the concurrent behavior in terms of the allowable sequences of behaviors that result from the execution of its operations. It is assumed that operations are atomic, i.e. they occur instantaneously. Concurrency is modeled by the non-deterministic interleaving of atomic operations. As with the temporal consideration, the approach used here is based on the improved recipe by [7].

The first step in modeling the concurrency behavior is to construct a next-state schema. This is the disjunction of the system’s operations. It represents the fact that an atomic step in the controller’s behavior may be caused by any one of its operations. The next-state schema for the controller is the disjunction of `SendCollisionSignal`, `SendObstacleSignal`, `SendPressureStatus`, `SendPositionStatus` and `SendToMotorControl`.

\[
\text{ControllerNS} \equiv \\
\quad \text{SendCollisionSignal} \lor \text{SendObstacleSignal} \lor \text{SendPressureStatus} \\
\lor \text{SendPositionStatus} \lor \text{SendToMotorControl}
\]

The controller’s initial state set and next-state relation must be obtained from `Initialize` and `ControlNS`. To do this, schema binding \( \emptyset \) is used to project out the relevant state components before direct substitution into `validcompt` [7]:

\[
\begin{align*}
\text{ControllerBehavior} \\
\quad t : \text{compt}[\text{RobotController}] \\
\quad \text{validcompt} \left( \{ \text{Initialize} \cdot \emptyset \text{RobotController} \}, \\
\quad \{ \text{ControllerNS} \cdot \emptyset \text{RobotController} \rightarrow \emptyset \text{RobotController}' \} \right)
\end{align*}
\]

Any behavior that satisfies \( t \) will be a valid behavior of the controller.

### 3.10 The Robot Behaviour

The robot is that component of the system that is needed to reason about properties of the complete system. It is assumed that the behavior of the robot is either moving
or stop from moving. So, as when the formal method is concerned, it is assumed that
the sequence consideration of the leg’s and body’s movement falls into another field
of study.

The robot has four independent legs and a body. A generic leg’s state and body’s
state are first specified:

```
LegState
state: Modes
```

```
BodyState
state: Modes
```

The leg or the body is either at the state of moving or stop from moving. Initially,
the leg and body is at the state of *Idle*.

```
InitLeg
LegState
state = Stop_Move
```

```
InitBody
BodyState
state = Stop_Move
```

The robot has four legs and a body called leg1, leg2, leg3, leg4 and body and
have ten discrete movements.

```
Robot_movement ::= move_leg1 | move_leg2 | move_leg3 | move_leg4 | move_body
| stop_leg1 | stop_leg2 | stop_leg3 | stop_leg4 | stop_body
```

Five schemas are now specified which will be used to promote the operations of
the generic leg and body to the robot's state:
Each of the generic leg and body operations are then promoted to operations of the robot legs and body:

\[
\text{WallClimbing} = \text{Robot schema} \land \text{RobotController schema}.
\]

The specification of the system needs an overall state so the operations which interact can be specified. In this project, the total state is called WallClimbing. Therefore, the WallClimbing is a conjunction of Robot schema and RobotController schema.

The operation Moving_Leg1 is a combination of WallClimbing, Leg1ToRobot, External_Pc and PosState with pos! of type Natural Numbers as an output and leg1? as an input. The predicate part denotes that if the PC sends the Start_Move command and the robot movement is move_leg1, then the state of the leg 1 is Moving. pos! denotes the output of the Moving state if the leg 1 which is the current position of the leg 1. The same explanation applies to Moving_Leg2, Moving_Leg3, Moving_Leg4 and Moving_Body operations.
FORMAL SPECIFICATION OF A WALL-CLIMBING ROBOT

\[ \text{FinalPosLeg1} \]
\[
\Delta \text{WallClimbing}
\]
\[
\text{External_Pc}
\]
\[
\Delta \text{Leg1ToRobot}
\]
\[
\text{pos1: N}
\]
\[
\text{leg1?: RobotPose}
\]
\[
\text{PosState}
\]
\[
\text{pc} = \text{Start_Move} \land \text{robot_move} = \text{move_leg1} \Rightarrow \text{leg1.state} = \text{Moving}
\]
\[
\text{pos1} = \text{CurrentPosLeg1}
\]

*FinalPosLeg1* denotes the final position of the leg 1 after correcting the error between the current position and the target position in which if there is an error, the leg or the body will be moved to the targeted position. The error computation will then be sent to the motor control. The same explanation applies to *FinalPosLeg2*, *FinalPosLeg3*, *FinalPosLeg4* and *FinalPosBody*.

\[
\text{FinalPosLeg1} \equiv \text{Moving_Leg1} \land \text{PositionSensor} \land \text{MotorControl}
\]
\[
\text{FinalPosLeg2} \equiv \text{Moving_Leg2} \land \text{PositionSensor} \land \text{MotorControl}
\]
\[
\text{FinalPosLeg3} \equiv \text{Moving_Leg3} \land \text{PositionSensor} \land \text{MotorControl}
\]
\[
\text{FinalPosLeg4} \equiv \text{Moving_Leg4} \land \text{PositionSensor} \land \text{MotorControl}
\]
\[
\text{FinalPosBody} \equiv \text{Moving_Body} \land \text{PositionSensor} \land \text{MotorControl}
\]

4.0 CONCLUSION

The main idea behind this paper is that in order to increase the readability and correctness of the computer software, in particular the hard real-time systems, it is desirable to separate the often contradictory aims of writing a clear and understandable program with one which is efficient. Formal specifications can greatly help in providing a rigorous and precise framework within which the specification can be written. This paper has shown that Z specification can be developed for the small-scale embedded hard real-time system. A case study, the embedded controller of the four-legged WCR was used to illustrate the approach.

The case study shows that the expressibility of Z specifications can be considerably improved upon, enabling important properties such as concurrent and temporal behavior [8]. The case study also shows that the introduction of timing and parallelism adds very little extra complexity compared with the traditional Z and is suitable to specify the static and dynamic behavior of the small-scale hard real-time systems.
ACKNOWLEDGEMENT

The authors would like to thank UTM Mobile Robot Research Group (MRRG) for their cooperation in defining the wall-climbing robot requirement specification.

REFERENCES


