# CONTRACT BASED DEVELOPMENT AND REFINEMENT IN SIMULINK®

**Sujit Kumar Muduli**

**Master of Technology Thesis**

June 2017

International Institute of Information Technology, Bangalore

# CONTRACT BASED DEVELOPMENT AND REFINEMENT IN SIMULINK®

Submitted to International Institute of Information Technology, Bangalore
in Partial Fulfillment of
the Requirements for the Award of
Master of Technology

by

**Sujit Kumar Muduli**
**MT2015113**

International Institute of Information Technology, Bangalore
June 2017

*Dedicated to*

*My Parents*

## Thesis Certificate

This is to certify that the thesis titled **Contract Based Development and Refinement in Simulink®** submitted to the International Institute of Information Technology, Bangalore, for the award of the degree of **Master of Technology** is a bona fide record of the research work done by **Sujit Kumar Muduli**, **MT2015113**, under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Meenakshi D'Souza

Bengaluru,

The 10th of June, 2017.

# CONTRACT BASED DEVELOPMENT AND REFINEMENT IN SIMULINK®

## Abstract

Software systems are becoming large and complex day by day while the reliability and correctness is still an issue. Softwares are also dominantly used in controls system for controlling hardwares in many different domains like avionics, heath case, defense and many safety critical systems. So any error in the production code can cause disastrous effects on life and money. Many of these industries use Model Based Design techniques like Simulink® to model their control system and auto generate code from model which is then embedded in their system and along with tools like SLDV™, Simulink allows us to perform verification task on the model. Verification is performed at the stage where the complete model is ready, so if any errors has been found it still takes a lot of time to fix it in a complex model. But there exists methodology such as stepwise refinement technique, where starting from an abstract specification of system we gradually move towards a concrete system model which is fully implemented and by verification in each refinement step we ensure the modeling decision is correct and will not lead to any bad behavior. Checking for correctness in intermediate stages in the development process helps detect errors made by developers at a very early stages. Now in this work we've tried to use this style of development in a Simulink environment. In order to achieve this we introduced the concept of contract based development in this modeling environment and also a generic workflow has been proposed to perform refinement using contract based design concept.

## Acknowledgements

I would like to express my gratitude to all those people who helped me through this journey and pushing me hard to achieve this goal.

First and foremost I would like to thank my thesis advisor Prof. Meenakshi D'Souza, for her guidance. She is extremely good in her field of work and her teaching helped me a lot in building good understanding formal methods. I am very grateful to her for allowing me to do thesis under her supervision and thank her for supporting me in writing this thesis.

I would also like to thank Dr. Prahlad Sampath who has contributed a lot of his time in guiding me through. I feel very lucky to get a chance to work with him. I deeply thank him for his motivation and advice without which it may not have been possible.

I am also extending my thanks to MathWorks and RAEngg (Royal Academy of Engineering) for sponsoring my thesis work.

Next, I would like to thank Dr. Manoj Dixit and the entire SLDV team at Mathworks for helping me in gaining good understanding in Simulink and MATLAB. Also like to thank Prof. Deepak D'Souza, Prof. Sujit Kumar Chakrabarti, Prof. Sumesh Divakaran and Dr. Jeremy L Jacob for their amazing feedback on my research work.

I also thank my friends Subhas, Shishir, Sayyam, Supromit, Soumyadip, Prachi Prakash, Piyush at IIIT Bangalore for their encouragement.

Last but not the least I thank my parents for their blessings. Special thanks to my two brothers for standing by me all the time.

May God bless you all.

# Contents

# List of Figures

# List of Abbreviations

| | |
|---|---|
| **CBD** | Contract Based Development |
| **EML** | Embedded MATLAB |
| **MBD** | Model Based Design |
| **SLDV** | Simulink® Design Verifier™ |

# CHAPTER 1

# INTRODUCTION

The ultimate goal of a software development process is to design and build a complete error free system which must be satisfying all its requirements. This thesis study is about formalism of refinement workflow in building control systems in Simulink® Modeling environment. Control systems are used in various safety critical systems and generate control systems to operate any physical/mechanical entity. Our study involves the best use of the theoretical concept concept of *Refinement* in practice and its scope in practical implementation in model based design environment like Simulink.

The issues those have been investigated by this thesis work,

- Defining semantic and a generic workflow of refinement in Simulink Model Based Design environment.
- Bringing the concept of contract based development into Simulink design environment and investigated what advantages it can give over the traditional way of model based design in Simulink.
- Identifying the areas where tool support can be provided for automation.
- Reasoning about the correctness of the methodology followed in refinement.

The structure of this thesis is as follows, in next section 1.1 we'll have an overview about proving correctness of programs and how refinement process can help in constructing a correct program much faster than the usual method of system development. In chapter 2 we'll discuss elaborately about model based design and Simulink editor, chapter 4 gives an overview of using contracts in system design and then in chapter 3 the idea of using contracts is taken up to implementation level in designing control system in Simulink.

In this chapter an overview about program correctness followed by a short description about refinement process has given, this aims to give a basic idea about the background of this study.

## 1.1 Correctness of a Program

A program is said to be correct if we can show that it satisfies its specification. Specification acts like a contract between client and supplier, which describes what a program should do or how it should behave. A program specification consists of precondition and postcondition , i.e. a client calling the program with necessary parameters must satisfy a given condition called *precondition*, and if the program is executed on satisfying corresponding precondition upon termination output must satisfy the *postcondition*. If program doesn't satisfy its specification might have one of the following issues,

- A precondition violation indicates bug in the client (caller), as the caller doesn't observe the condition imposed on the correct call.
- A post-condition violation is a bug in the supplied program. Programmer failed to deliver on its promise.

If one can show for all valid inputs i.e. all input values for which per-condition is satisfied post-condition must be satisfied by outputs if program gets executed, then that would be enough to argue about the correctness of the program. This method is well known as *Software Testing*, which is one of the most popular methods for finding errors in a Program. Tester produces a test suite which contains a set of test inputs using which a program is going to be tested. For each of valid test inputs of a test suite, output produced by the program is observed and if the output is not as expected then we say there is a bug in the program and that has to be isolated and fixed. But it is very hard in testing method to generate test cases to hit all the execution path and find all errors in a program.

*“ Program testing can be used to show the presence of bugs, but never to show their absence! ”*

*– Edsger W. Dijkstra*

We have another method known as *Model Checking* which is a being used in verification of programs for safety critical systems, to verify whether or not a program satisfies its specification. Model checking consists of mathematical techniques to find errors in a program without even running it. Model checking algorithm is invoked on the transition system model of the program. From the transition system of the program, model checker will try to search if there exists an execution path such that a program might enter into an undesired state i.e. any state where a property about the system will fail. If there exists such execution path it is returned as a counter example by model checker.

A usual practice of using model checking to catch errors after a complete model of a system is formed and testing can be used once programming task is done. At this point finding and fixing an error will consume a lot of time and money which is undesirable.

However combining design phase with verification phase, at the end of this process designer will have a system design that is correct by its construction. This is achievable using the concept of *Refinement* along with property proving using model checkers and since at each step of refinement model checking is performed it restrains designer from introducing design errors in the intermediate steps. Refinement is a very simple yet powerful idea which provides a guided and systematic way of software development. The concept of refinement is briefly described in section-1.2.

## 1.2 Refinement in Software Engineering

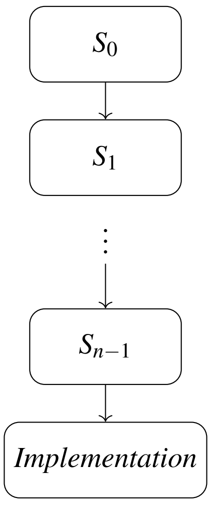


Figure FC1.1: Refinement Steps

Software development begins with gathering all requirements which talk about what needs to be achieved. In refinement based workflow we start with a very abstract or high-level specification say $s_0$ for a system which talks about system behavior or what the system is required to do. In each step of refinement, we add some details to the abstract specification and create a more refined specification for system. Every step in refinement contains some design decision. After a few step of refinement we'll have a full specification of the system and this low level specification can be then implemented by any programming language. In this process of refinement we also need to show that when a specification is being updated the correctness described by the previous version of specification must be preserved by this updated specification and this can be checked by property proving techniques. At initial stage when system specification was abstract it can lead up to different final implementation of the system. But in each step of refinement the design decisions performed reduces choices and process ends up in one of the possible implementation among many options [1].

Refinement process offers various benefits to software developers,

- To verify different implementations for same part of program, which helps in deciding a suitable alternative based on time or space complexity.
- Stepwise refinement process reduces the probability of introducing errors into program, since in each step of refinement every decision is verified against system properties so if any mistakes has been made that could be caught immediately. Time and cost overrun in whole development phase can be suppressed to a large extent.
- Provides scope for easy maintenance and scalability of programs. Each step we perform simple and manageable refinement which can be logged down and can be referred by programmers later.

## 1.3 Summary

In this chapter we had a brief idea about how to determine the correctness of a software system and also looked at refinement concept and how it can give a much guided way of development starting from very abstract system specification towards a full implementation of system along with proving system properties at each step which prevents developer from introducing errors in system model. In chapter 4 refinement concept is described w.r.t contracts and it’s support for model based design environment like Simulink.

# CHAPTER 2

# MODEL BASED DESIGN

The traditional way of software development process involves many steps such as requirements gathering and analysis, followed by design, implementation, verification and maintenance. The design phase where system architect writes specification for system and produces a specification document for programmers. In the implementation phase developers write programs according to software specification. So, if an error is introduced in the design phase it will also be introduced in to the corresponding codes as well. Another problem at this phase is that sometimes these specification documents have ambiguities which are misinterpreted by developers and software is not implemented correctly.Besides this reducing development time and increasing software reliability is also one of many challenges in software development.

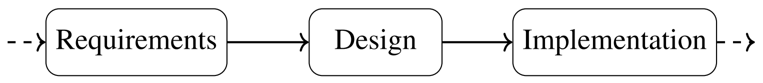


Figure FC2.1: Stages of Software Development

Model Based Design concept was introduced to tackle these challenges. In Model based design methodology design and implementation phase has combined into a single process. It uses a high level modeling language for system specification, from which target code is generated automatically. This allows an engineer to focus more upon design instead of implementation in programming languages. The model is an executable specification of system which can be simulated to check if system is modeled correctly.

Model based design has been a very popular choice in developing software for embedded systems, because modeling is platform independent and it allows us to cope up with the rapid changing hardware in the industry. Secondly, it provides easy integration of new functionality into old models which helps in handling complex systems effectively. It reduces overall development time and helps in cutting down cost overrun, by detecting errors at a very early stage.

## 2.1 Simulink®

Simulink® is a product of MathWorks which provides a graphical programming interface for model based design. It is used in modeling physical systems and simulate them in computer. A rich set of domain specific block libraries are provided for modeling purpose and allows users to create their own custom blocks also. Simulink has built-in solvers for simulation of both continuous and discrete systems, which applies numerical methods to solve a set of equations that represent a block and determines the state of blocks in next time step. Computing state of a system at each time step, from simulation start time to end time, these series of time steps will describe the characteristics of the system. By simulating a system helps us in deciding whether or not a modeled system following expected behavior or not. Simulink predominantly used for modeling control systems in automotive industries, using auto-code generation feature it generates code for embedded controllers which can be written on chip.

### 2.1.1 Simulink Design Verifier

Simulink Design Verifier™ (SLDV) tool in Simulink is used for different purposes like property proving, test case generation and finding design errors in model [2]. Design errors involves finding integer overflow,dead logic, array access violations, division by zero. It uses model checking technique in proving some property about a model and if property doesn't hold by the model it generates a counter example. For this purpose SLDV uses Prover plug-in developed by Prover Technology. SLDV is used for generating test cases and computes model coverage for condition, decision, modified condition/decision and custom coverage objectives. However not all Simulink models are compatible with SLDV. It can only supports models not having continuous library blocks or any algebraic loops in it.

Basic workflow for model analysis using SLDV is given below,

1. Checking Model compatibility for verification. `sldvcompat` command is used to check if a model is compatible with SLDV or not. If a model is found to be not compatible because of some incompatible blocks like continuous blocks or algebraic loops following measures can be taken by the user.
    (a) If model contains continuous block replace it with corresponding discrete blocks or with an equivalent model.
    (b) For each algebraic loop introduce one memory or delay block.
2. When we have a model compatible with sldv then we can perform following analysis on model
    (a) Design error detection
    (b) Property proving
    (c) Test case generation
3. Generate analysis report or create a harness model to simulate model under test cases generated from SLDV.

SLDV also has the notion of partial compatibility. SLDV can perform automatic stubbing to allow users to perform analysis on system. And also to generate test cases for model user can replace the block in model that is not recognised by SLDV with something else that is function equivalent such as a look up table.

## 2.2 Design Error Detection

This step is done before a model is simulated to ensure it is free from design errors such as integer overflow,dead logic, array access violations, division by zero as mentioned earlier. SLDV performs static analysis performed on models to track these errors and generates warnings for designers. All models needed to be checked for design errors in order to avoid such errors to be inherited to code generation stage.

## 2.3 Property Proving using SLDV

SLDV allows user to verify some properties about a model, properties can be modeled either using Simulink blocks or by using EML Function blocks. A property is defined as an a logical expression over signal values in model. It can not only be used for verifying static requirements but can also be used in verifying temporal properties as well. Simulink Library has few blocks to model temporal requirements and for verification purpose it has a set of blocks like Assumption block, Proof Objective block, Assertion block and Verification Subsystem to be used.

(a) Assumption Block (b) Proof Objective Block

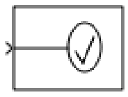
Figure FC2.2: Simulink Design Verification Blocks

(a) *Assumption Block* is used for making an assumption regarding a logical statement in Simulink environment, i.e a logical expression over signal value is modeled which output is then fed into this block. This ensures that logical expression modeled will hold always during simulation time or during verification. It can also be used to fix a signal value to a particular range such that it always remains within the given bound and based on this, user can perform any desired analysis of the model.

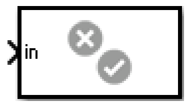
(b) *Proof Objective Block* is used for denoting a proof objective. First a property that is needed to be verified is modeled and then output of this property model is given to Proof Objective block. Then SLDV tries to proof whether this modeled property will hold always during simulation, if any simulation exists that can violate this property it generates a counter example which a test case. Similar to assumption block one also can define a range of values for proof objective.

(c) *Assertion Block* is similar to an assertion statement in programming language. If input to assertion block is ture or non-zero it does nothing, but if input becomes false or zero it stops the simulation. It is added to model as a self verification and if present in the model it acts as a proof objective. However range of input value it can handle is boolean where in proof objective block one can define a custom range of values for a proof objective to hold.

(d) *Verification Subsystem* is like a block within which one can specify proof or test objective to be used in verification task. To specify a proof objective over a set of signals, set those signals as inputs to verification subsystem. Then proof objective block is used to declare this specification as a proof objective. The advantages of Verification Subsystem is that it is ignored by Simulink auto code generator and it doesn't show up in programs generated.



(a) Assertion Block



(b) Verification Subsystem Block

Figure FC2.3: Additional Support for Verfication in Simulink

### 2.3.1 Modeling Requirements in Simulink

We just discussed properties that are needed to be proved has to be modeled first and then verification task can be applied. In this section we'll see through examples, how this actually being done in Simulink environment and end of this section we'll have a basic idea about requirement modeling. Before that let's spend some time in discussing types of requirements i.e *static* and *temporal* requirements that can be modeled using Simulink.

#### 2.3.1.1 Static Requirement

Static requirements are those requirements which should be satisfied by a system all time from beginning till end of simulation. This does not have any essence of time, it is just like an assertion statement for entire simulation time. Following are few examples of this type of requirements.

(i) **Comparision with a Constant**

Suppose one wants to verify that a particular signal value in relation with a constant value. For example, Value of `out1` never goes above 1.

(ii) $(cond) \Rightarrow (result)$, is a generic static requirement type for Simulink Models. Suppose a requirement is described as "If $(x > 10)$, then $y$ must be `true`", then it can be modeled as below.

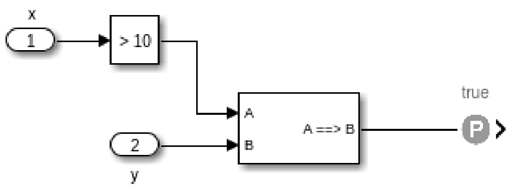


Figure FC2.4: Static Requirement Modeling Example

#### 2.3.1.2 Temporal Requirement

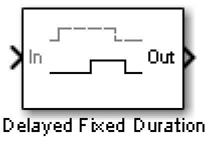
Temporal requirements has the essence of time within them. In case of Simulink time is measured in unit of simulation time step which can be set according to user for simulation. A usual temporal requirement is written something like this,

*If* `condition` *is satisfied for* <u>*at least*</u> $t_1$ *time steps, then* `signal` *value should become high for* <u>*at most*</u> $t_2$ *time steps.*

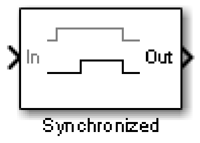
Well this is one example of temporal requirement, all we try to see if a signal value attains certain level for at least, at most, exactly for a range of consecutive time steps. Similarly for detection of signal properties with non-consecutive time steps like, *signal value should satisfy certain property for less than x times, in a time scale of 't' steps.* To model these types of requirements blocks like *Extender* and *Detector* provided in Simulink library.

1. **Detector Block**

   This block detects true duration on input and construct output true duration. It has two modes of operation (FC2.5),



(a)



(b)

Figure FC2.5: Simulink Detector Block Functionality

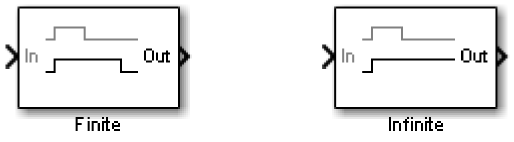
(i) *Delayed Fixed Duration* mode helps user in achieving two things firstly to detect input is high for a fixed consecutive number of time steps, then user can set the width or duration for which output signal will remain true also till how much duration output needs to be delayed by this block.

(ii) In *Synchronized* mode is only for input detection. Suppose one wants to check if input signal remains high for $T$ time steps. If it is, then at $T_{th}$ step output of detector block in synchronized mode will become true and will remain true until input remains true.

Detector block also has a *reset* parameter using which block can be reset feeding any external signal into it. This will make output to be false for that moment and will start detection process once again from beginning.

2. **Extender Block**

Extender block is used for extending the true duration of a signal value. This block has two modes of operation *finite* and *infinite* mode shown in FC2.6. In *finite* mode user gives a finite amount for which it wants to extend input true duration on its output and when set to *infinite* once true occurs on input, output of this block will always remain true from that point till end of simulation time. Extender block also has a similar reset parameter as detection block to reset corresponding block using external signal values.



(a) (b)

Figure FC2.6: Simulink Extender Block Functionality

#### 2.3.1.3 Examples of Modeling Temporal Requirements

(a) *Whenever signal x equals 1 for more than 6 steps, the signal y shall be equal to 2.*

Modeling above requirement task can be divided into two segments first we need to perform input detection. To detect *x* equals 1 for more than 6 steps consecutively i.e at least for 7 steps and secondly, the time step when detection condition is satisfied value *y* must be equal to 2. FC2.7 models this requirement, it uses a detector block in synchronized mode to check if *x* remains 1 for at least 7 steps. Then an implication block to check when condition on signal *x* is true whether or not condition on signal *y* holds, so an implication block is used for this purpose.

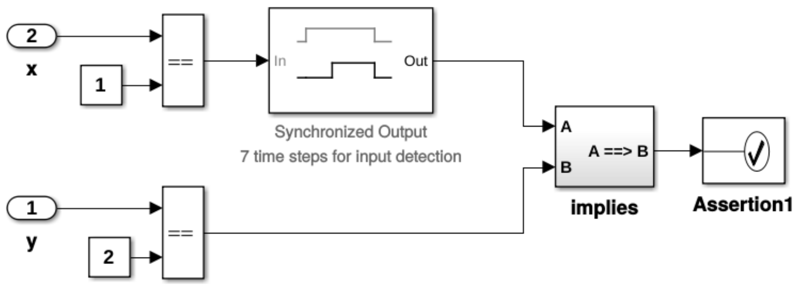


Figure FC2.7: Example Modeling Temporal Requirements

### 2.3.2 Modeling Requirements using EML Function

Simulink also provides facility to include any EML function as a block into a model. EML stands of Embedded Matlab, using EML Function block user can write m-code which will define the functionality of this block and it can be used in any Simulink model. It has MATLAB Function block in library where one can define a function

```
function [out1, out2] = fname(in1, in2)
out1 = in1 - in2;
out2 = in1 + in2;

obj = (out1 > 0);
sldv.prove(obj);
end
```

Figure FC2.8: Modeling Requirements using EML

in eml. The block will behave as per the function definition. Input parameters are the signals passed to in-ports of MATLAB Function block. A typical function definition in EML is given below. SLDV has provided built-in function calls (APIs) like `sldv.prove, sldv.assume` to be used for performing verification task inside a function definition. One can form a logical expression using eml and pass that expression to either `sldv.prove` or `sldv.assume`. These will serve similar purpose as assumption and proof objective block in Simulink model.

## 2.4 Test Case Generation

Simulink Design Verifier can also be used for generating tests which replicates design errors, achieve test objectives or exercising model to meet coverage criteria. It allows user to exercise it's model over and over with already existing test input files or by generating new set of test inputs for model.

### 2.4.1 Model Based Testing

Through Model Based Testing one can perform verification and simulation on reference models, subsystem in a model. It also allows to test functionalities of subsystems in isolation by extracting it into a harness model and simulate it with different test inputs.

# CHAPTER 3

# CONTRACTS IN SYSTEM DESIGN

This chapter is dedicated only to study of contracts and their applications in system design. Section 3.1 gives a brief info about the work of B. Meyer on contracts. Section 3.2 summarizes the work done in the research report by Albert Benveniste et al. [3] which talks about different challenges in System Design and how to tackle those issues. Here, the use of contracts in system design and an elaborative study on contract has been formalized. After this chapter one will have basic understanding about what do we mean by contract and how it can be used in system design process. This chapter gives us theoretical overview of contracts and various operation among contracts are also being explained.

## 3.1 Design By Contract

The use of contracts in programming and software design can be traced back to the paper named Applying “Design by Contract” [4] published in year 1992. This paper talks about a methodology or guideline to be followed in order to increase reliability in Object oriented programming. It talks about how it is achieved by producing less number of bugs while building software components using contracts that has been carefully designed. It says a programming task can include different subtasks for example a main method can include different function calls to perform different subtasks. But the reliability depends upon two things, on the function call made second the function definition. The function call should provide a valid input parameter and the upon executing the function it should return correct result. So, a client or caller of a function can't blame the supplier or the function designer if client has not followed the appropriate guidelines or conditions while calling the function. In this case guideline or condition is known as precondition. Upon satisfying the precondition the supplier should guarantee a correct result and this result must satisfy certain conditions called post-condition in order to verify function execution happened correctly. So, there must be a understanding between the client and supplier and this is being captured by a *Contract* which is a pair (Precondition, Postcondition). The client and supplier are different programming units in this case. An appropriate example would be following sub-routine which computes greatest integer value of square root of an number.

```
int floor_sq_root(int num)
{
        /*
         *      value of num should be greater that or equal to 1.
         */
        for(int i=1; pow(i,2) <= num; ++i)
        {
                if(pow(i,2) <= num && pow(i+1, 2) > num)
                        return i;
        }
        return -1;
}
```

Above function will work properly as long as value of parameter `num` is greater than or equal to 1. But if the value of `num` is set to 0 the expected output of any function computing greatest integer function of square root would return 0 as result. But this function is designed only for numbers greater than or equal to 1. If invoked for `num=0` it will give -1 as result, which is not correct. The client who called the function can't complain about the result being wrong as the client has not met the necessary precondition of the function call.

## 3.2 Contracts for Systems

Now the concept of contracts discussed in previous Section 3.1 is being extended to be used in system design as in developing control and safety critical systems. An component has some input ports and output ports and is interfaced with its environment i.e with other components it interacts. As seen in the previous example calling a subroutine is meaningful only when caller obeys the guidelines then only correct result is expected, similarly in this case also a component is expected to produce correct result if it operates in right environment and upon placed in right environment component should guarantee correct behavior. In order to make a reliable component its designer should also bound by a contract and in this case a contract should talk about its indented behavior of its implementation by suppressing too much details i.e. it is an abstraction of system's actual expected behavior. So, a contract in system design can be viewed as a pair $(E,M)$, where $E$ is set of environments where contract can operate and $M$ is a set of possible implementations.

Any contract must have following two properties,

a) **Consistency**

A contract is called consistent if it is implementable or there should exist at least one possible implementation in corresponding programming environment or modeling environment i.e $M \neq \phi$.

b) **Compatibility**

A contract is called compatibility if there should be at least one environment where it can operate i.e $E \neq \phi$.

| $C_1$ | | |
|---|---|---|
| variables | : | inputs $:x,y$ |
| | | outputs $:z$ |
| types | : | $x,y,z \in \mathbb{R}$ |
| assumptions | : | $y \neq 0$ |
| guarantees | : | $z = x/y$ |

Figure FC3.1: Specification of contract $C_1$

It is very obvious and makes no sense of making any contract which is neither operable nor implementable. Figure FC3.1 shows a specification of contract $C_1$ and its corresponding implementation $M_1$ is shown in Figure FC3.2. Contract $C_1$ gives a short description about the system interface like what are the input and ouput variables and type of each variables along with their data types. Besides this it has two other sections *assumptions* and *guarantees*, which gives designer or developer a hint about the type of environment it can be used and on correct invocation of its implementation what the implementation should guarantee. If we interpret contract $C_1$ it says that the implementation upon receiving real values on inputs $x,y$ it will generate output $z$ which is equal to $x/y$ assuming that environment is not generating $y = 0$. It doesn't talk about the behavior when $y = 0$ is encountered and leaves it to the designer how they handle this exception. Depending on how this exception is being handled there could be many implementations of $C_1$. $M_1$ is one such implementation where in *behavior* section we have mentioned when $y = 0$ is encountered it should produce 0 as result. Similarly there could be infinitely many environments which might not be generating $y = 0$ where it could be used. So, we can call this contract said to be both consistent and compatible. A variation of contract $C_1$ given in Figure FC3.3 will be inconsistent where the assumptions is set to **TRUE**. Which says that the implementation must also compute $x/y$ even if $y = 0$ which is not possible. Contracts don't have power to restrict any environment it can only talk about restrictions and constraints on implementations.

The following characteristics of contracts what makes it more interesting to be used in system design,

i. Contracts are made abstract such that it is sufficient enough to talk about the necessary behavior of its implementation. It can be either very close to implementation or only one property is being specified by it. It allows users to replace complex implementation with a simple contract while doing some analysis on the whole system or at a component level.

ii. It distinguished the role between environment and component, so this characteristics will help us in using contracts for both as a check for implementation and as a abstract implementation.

How these two characteristics of contracts is being used in Simulink Model Based Design environment is discussed thoroughly in Chapter 4. In next section 3.3 different operations that can be performed on contracts have been discussed.

## 3.3 Contract Operators

### 3.3.1 Composition

A parent component can be composed of multiple sub-components and these sub-components should be designed in such a way that it can interact with each other [3]. So we define composability as follows,

i. Common variable types or the interfaced variable types must match.

| $M_1$ | | |
|---|---|---|
| variables | : | inputs $: x, y$<br>outputs $: z$ |
| types | : | $x, y, z \in \mathbb{R}$ |
| behaviors | : | $y \neq 0 \rightarrow z = x/y$<br>$\wedge$<br>$(y = 0 \rightarrow z = 0)$ |

Figure FC3.2: $M_1$ impliments contract $C_1$

| $C'$ | | |
|---|---|---|
| variables | : | inputs $: x, y$ <br> outputs $: z$ |
| types | : | $x, y, z \in \mathbb{R}$ |
| assumptions | : | **TRUE** |
| guarantees | : | $z = x/y$ |

Figure FC3.3: Specification of contract $C'$

ii. Under composition these contracts and implementations must be able to interact with each other, such an environment should exist.

| $C_1 \otimes C_2$ | | |
|---|---|---|
| variables | : | inputs $: I$ <br> outputs $: O$ |
| types | : | $x, y, z \in \mathbb{T}$ |
| assumptions | : | $A_{C_1 \otimes C_2}$ |
| guarantees | : | $G_{C_1 \otimes C_2}$ |

Figure FC3.4: Specification of contract $C_1 \otimes C_2$

The operator for composition of implementations is denoted by symbol $\times$ and $\otimes$ denotes composition operation of contracts. We're interested only in learning how contracts can be composed and from this composition of implementations will follow through as we know implementations respects corresponding contract. Let $C_1$ and $C_2$ be two contracts then their composition denoted as $C_1 \otimes C_2$ is specified in Figure FC3.4. $I$ and $O$ are set of inputs and outputs and $\mathbb{T}$ is the data type. $A_{C_1 \otimes C_2}$ and $G_{C_1 \otimes C_2}$ are assumptions and guarantees are defined as follows,

$$G_{C_1 \otimes C_2} = G_{C_1} \wedge G_{C_2} \qquad \text{(Eqn 3.1)}$$

$$A_{C_1 \otimes C_2} = max \left\{ A \;\middle|\; \begin{array}{c} A \wedge G_{C_1} \Rightarrow A_{C_2} \\ \text{and} \\ A \wedge G_{C_2} \Rightarrow A_{C_1} \end{array} \right\} \qquad \text{(Eqn 3.2)}$$

Equation Eqn 3.1 says that the composed contract should now guarantee both the guarantees of $G_{C_1}$ and $G_{C_2}$. Where the assumption is computed as per equation Eqn 3.2, which says the weakest assumption possible for which the two implications $(A \wedge G_{C_1}) \Rightarrow A_{C_2}$ and $(A \wedge G_{C_2}) \Rightarrow A_{C_1}$ will hold. *max* operator here is interpreted as a extraction of the weakest predicate. This ensures whenever first contract is put in the context of an implementation of $C_2$ the assumptions of $C_1$ should met and vice versa. If the assumptions $A_{C_1 \otimes C_2} \neq$ **FALSE** then two contracts $C_1$ and $C_2$ are said to be compatible. An example of such contracts $C_1$ and $C_2$ are given in Figure FC3.1 and FC3.5. It can be observed that the assumptions for $C_1 \otimes C_2$ is not equals to **FALSE**, so $E_{C_1 \otimes C_2}$ will be a non-empty set and hence this contract is compatible.

| $C_2$ | | | |
|---|---|---|---|
| variables | : | inputs | $: u$ |
| | | outputs | $: x$ |
| types | : | $x, u \in \mathbb{R}$ | |
| assumptions | : | $y \neq 0$ | |
| guarantees | : | $z = x > u$ | |

Figure FC3.5: Specification of contract $C_2$

### 3.3.2 Refinement

We've had already seen the basic definition of refinement in 1.2 now we extended this concept to contracts. In order to say a contract $C'$ is a refinement of contract $C$, the following properties must satisfy

i. An implementation of $C'$ should also satisfy the behaviors defined in $C$.

$$M \models^m C' \Rightarrow M \models^m C$$

ii. $C'$ should also be operable in the environment defined by $C$ i.e any environment of $C$ is also an environment of of $C'$.

$$E \models^e C \Rightarrow E \models^e C'$$

We denote refinement using symbol $\preceq$. We say $C' \preceq C$ if and only if $M_{C'} \subseteq M_C$ and $E_{C'} \supseteq E_C$, which again can be brought down at a level where we can say to call $C'$ is a refinement of $C$ it is sufficient to show that the followings are true.

(a) $A_C \Rightarrow A_{C'}$

(b) Whenever $(A_{C'} \Rightarrow G_{C'})$ holds, $(A_C \Rightarrow G_C)$ should also hold.

#### 3.3.2.1 Independent Component Development

Let $C_1, C_2$ be two contracts and it is known $C_1' \preceq C_1$ and $C_2' \preceq C_2$. Then we can show that $C_1' \otimes C_2' \preceq C_1 \otimes C_2$ given that $C_1$ and $C_2$ are composable. This allows us to perform independent development of components. Consider a contract $C_0$ which represents system wide requirements, this can be further decomposed into multiple subsystems say $C_{01}, C_{02}$ and $C_{03}$ with the property that $(C_{01} \otimes C_{02} \otimes C_{03}) \preceq C_0$. Each of these newly created components can also be refined and developed independently by different development teams.

### 3.3.3 Conjunction

A full component specification of component, doesn't contain function specifications only rather it is a *conjunction* of multiple viewpoints such as functional, timing, safety etc. These are different aspects of design and one can keep separate contracts for each of these aspects.

## 3.4 Design-by-contract by Boström et al.

In this section we've discussed the work done by Boström et al. [5]. In this paper they have given a basic definition of contract which is nothing but a pair $(A : assumption, G : guarantee)$ similar to a precondition and postcondition of the block. In terms of Simulink they've viewed it as an non-deterministic abstract specification which can't be simulated yet can be used in verification process. The methodology they've followed for checking whether or not an component satisfies corresponding contract by interpreting the component i.e a Simulink model, using action system formalism. Later uses theorem provers to analyse contracts with respect to corresponding action system. Also, this paper describes how contracts along with refinement calculus allows to perform compositional reasoning on models. The similar objective is also achieved in this thesis work with a little different approach by not translating models into any other semantics rather using semantics of Simulink and with help of SLDV this goal is achieved.

# CHAPTER 4

# CONTRACT BASED DEVELOPMENT

We've seen the theoretical background of contracts in chapter-3 and in this chapter, we've used the idea of contract based development in developing control systems in Simulink. The idea behind contract based development in Simulink is to have both contract and implementation under same design entity called system component. This component is used as a unit of design in modeling control systems and components are connected together to form a system [6]. In this chapter, we present a methodology for *contract based design* in Simulink and compare it with the traditional way of system design in Simulink modeling environment and various benefits have been discussed. The following points also have been addressed in this chapter,

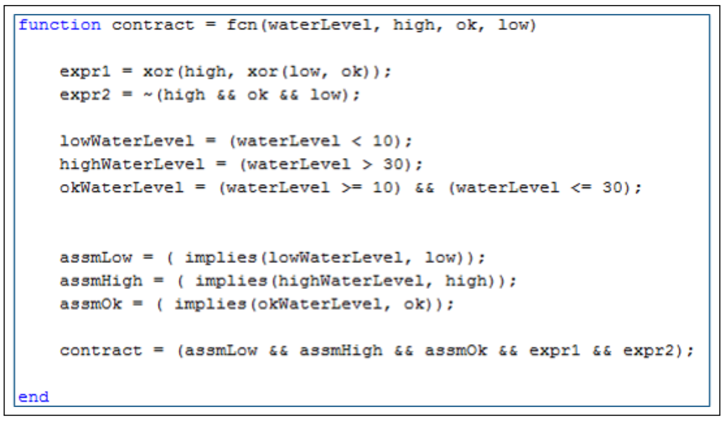
- Support to incorporate contract based development concept into Simulink.
- Different use cases where this concept can find applications.

Section 4.1 of this chapter contains a detailed description about contract and implementation and relationship between them and section 4.2 discusses how to support this idea in Simulink.

## 4.1 Contract and Implementation

**Definition 4.1** *A **contract** in Simulink is defined as an abstract specification for a block or system component. Every block or component behavior must satisfy the contract definition or in other words, a component should implement the corresponding contract.*

In Simulink, a contract can be defined by simply specifying component behavior either by an EML block or by modeling it using Simulink library blocks. An example of defining contract using EML function block is shown in Figure FC4.1. This block talks about the functionality required for the component i.e. what should corresponding implementation guarantee. First two variables `expr1` and `expr2` combined says that at most one of the outports will be high at any given point in time. This is kind of a safety requirement for the component. `assmLow` says that when water level will be low, then make the outport `low` True. Similarly, `assmHigh, assmOk` talks about when the high and ok outports should be activated or made high.



```
function contract = fcn(waterLevel, high, ok, low)

    expr1 = xor(high, xor(low, ok));
    expr2 = ~(high && ok && low);

    lowWaterLevel = (waterLevel < 10);
    highWaterLevel = (waterLevel > 30);
    okWaterLevel = (waterLevel >= 10) && (waterLevel <= 30);


    assmLow = ( implies(lowWaterLevel, low));
    assmHigh = ( implies(highWaterLevel, high));
    assmOk = ( implies(okWaterLevel, ok));

    contract = (assmLow && assmHigh && assmOk && expr1 && expr2);

end
```

Figure FC4.1: Contract Definition using EML

**Definition 4.2** *An **Implementation** is a complete Simulink model which satisfies the contract definition of corresponding component.*

A Simulink implementation for the contract in Figure FC4.1 is given in Figure FC4.2. In contract block tells how it should behave based on water level value that is now converted into a Simulink model using basic switch blocks in Simulink Library.

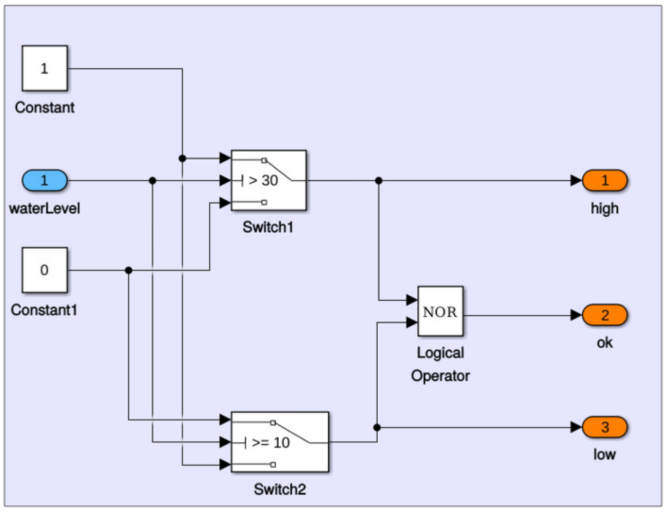


Figure FC4.2: Implementation

Since we've defined what are contract, implementation and relationship between them we now ask the question how one would be able to verify whether a component implements a contract ? The answer is simple, we can use model checking for this purpose and in case of Simulink models SLDV does this work for us. To perform this checking we use this contract as a proof objective for the model. Since contract is an abstraction of corresponding Simulink model and behavior defined in contract must be captured in corresponding Simulink model, if there is violation then it would be reported by SLDV in form of a counter example.

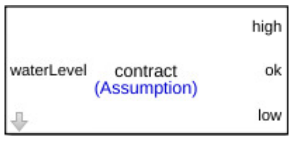
Now let's proceed and understand what is the motive behind this and what is that we're achieving here ?

- Consider a case where in a system model which is a composition of different components, and we don't have a complete implementation for that block or component and yet we want to reason about the correctness of the whole system or to test whole or some part system. This can be easily done in contract based development process by imposing contract on them and making them to behave as per contract definition. This will help user to set right context of other blocks to operate safely.

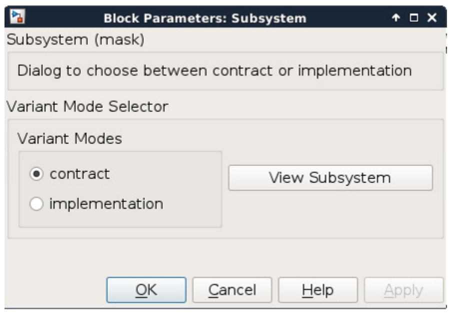
- Some parts of a huge model can have a very complex implementation and while performing analysis of system it can consume a lot of time by any tool. This complex implementation part of model can be replaced with a simple contract which will give an abstract view for that block, this will reduce the number of proof objective and helps in faster verification.

## 4.2 Simulink Support

Bringing this style to Simulink environment was easily achievable using different blocks provided in Simulink Library. A custom block has been designed using Simulink library blocks to realise the idea of having one block which can act in two different modes i.e. contract mode and implementation mode. When block or component is used under contract mode we impose the contract on the component and when implementation mode selected actual Simulink model overrides the block behavior and contract is used as a proof objective for the implementation. This, in turn, helps us to check whether underlying Simulink model indeed implements corresponding contract. A contract puts some amount of functional restriction over component to be built yet it provides freedom to developer to design system as it likes. This whole mechanism is performed under a masked subsystem block named as *Contract-Implementation Block* that has been design for this purpose only. For the convenience of user a control dialog has been provided which allows to set the mode of operation and using `View Subsystem` button user can manipulate the underlying contract of implementation.



(a) Masked Subsystem Block



Subsystem (mask)

Dialog to choose between contract or implementation

Variant Mode Selector

Variant Modes

[✓ contract]
implementation

(b) Control Dialog

Figure FC4.3: Contract-Implementation Block

## 4.3 Applications in System Design

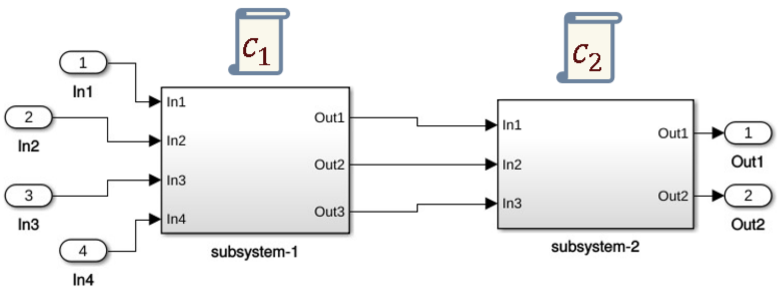
In this section few scenarios where contract based development style can be very beneficial for developers have been discussed.

### 4.3.1 Independent Subsystem Development

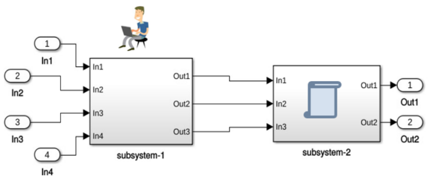
Contract Based Development could be very useful while developing systems independently. Consider a scenario where a control system that has to be designed is composed of two components as shown in FC4.4a. For each of these component we've high level description and we use that as a contract for each of these blocks. Then whole development task can be now divided between two developers where each of them set another component in contract mode and start developing corresponding component and verify it against the contract.

### 4.3.2 Adding New Features to Systems

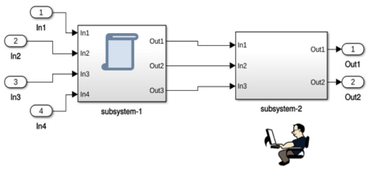
Most of control systems that has been designed and implemented in industries are incremental or evolutionary. In each iteration or release multiple new features are also being included in control systems. Previously existing models are already being verified



(a) System Structure and High-Level Description



(b) Developing Component-1 Independently



(c) Developing Component-2 Independently

Figure FC4.4: Different Use Cases of Refinement

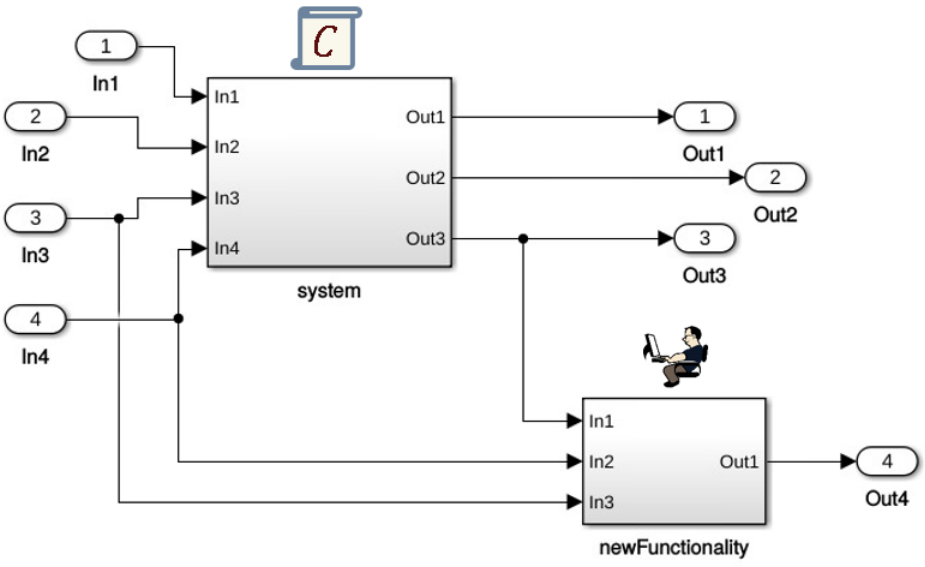


Figure FC4.5: Verification of new features in controller and known to satisfy all safety conditions and while implementing a new feature on top of these models we need not perform verification task all together on the whole system. All we want to do is to test our new feature going to be introduced is working perfectly. So we put the control system into contract mode which will help in converting all proof objectives into assumptions. Now, control system on contract mode will give an abstract view for original controller and yet allows us to verify our new feature by reducing complexity up to great extent.

### 4.3.3 Application in Simulink Model Refinement

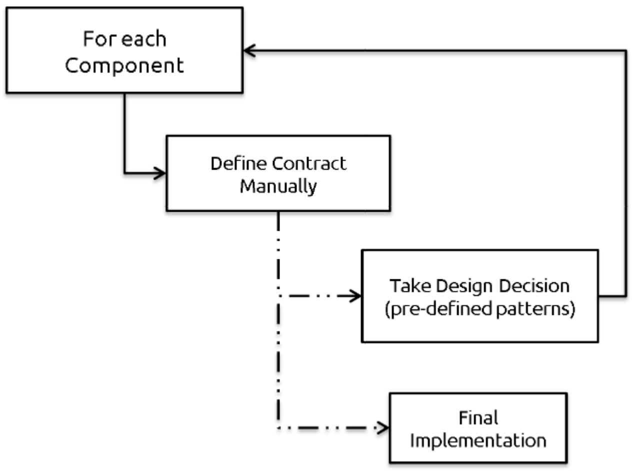


Figure FC4.6: Refinement Workflow for Model Based Design in Simulink

We can use the idea of refinement on contracts to perform refinement in Simulink Model design for developing control systems. Since we've designed a subsystem block which can operate in contract mode and at the top levels of abstraction where we don't have concrete implementation of any component and we always use these blocks in contract mode. These blocks can be treated as contracts as in Chapter-3 and further decomposition into multiple blocks and each of these new blocks can be refined independently after computing contracts for them. The figure FC4.6 shows a proposed workflow for refinement in Simulink. First of all we need to have a contract for the component which we'll be refining as of now defining contract is done manually. In each step of refinement the system component is broken into smaller sub-components along with new contracts for each of these blocks have been created. Decomposition of blocks is being done using some per-defined patterns that we created in to be used for this specific purpose. SLDV can be used here to prove that the refinement that has been performed is correct with respect to the composition of newly created contracts. This is described in 4.3.3.1 by using theory of contracts that was discussed in chapter-3,

#### 4.3.3.1 Proving Correctness of Refinement using SLDV

Consider a abstract system component $B$ implemented using *contract-implementation* block is defined by a contract $C$. When refinement is performed block is broken into three component $B_1, B_2$ and $B_3$ each of these is now defined by contracts $C_1, C_2$ and $C_3$ respectively. Once this refinement is performed in upper level we have contract $C$ as a proof objective and in lower level we have three new blocks whose behavior is described by three contracts $C_1, C_2$ and $C_3$ respectively which gives an abstraction of actual implementation. Each contract defines assumptions and guarantees it provides, in order to prove the correctness of the refinement we need to show two things,

1. $A_{C_1 \otimes C_2 \otimes C_3} \Rightarrow A_C$
2. If $A_{C_1 \otimes C_2 \otimes C_3} \Rightarrow (G_1 \wedge G_2 \wedge G_3)$, then $A_C \Rightarrow G_C$

Alternatively we also can prove the correctness by treating blocks in contract mode as an abstract implementation. We can say $B_1, B_2$ and $B_3$ has implementations $M_1^\alpha, M_2^\alpha$ and $M_3^\alpha$ for contracts $C_1, C_2$ and $C_3$ respectively. And we just need to show that composition of these implementations $M_1^\alpha \times M_2^\alpha \times M_3^\alpha$ implements contract $C$. This can be checked by showing that all behaviors of this composition is also a behavior in $C$ this can be done using SLDV.

#### 4.3.3.2 Decomposition Using Predefined Patterns

Using refinement in designing control system is based on decomposition and independent system development. How this decomposition is going to happen is based on the choice of the developer and varies from one developer to another. Performing these decomposition is also called as design decisions in system development which is made automated through a set of scripts implementing predefined decomposition patterns. A library of few scripts have been provided to user to choose the decomposition pattern and apply it. All the design decision is performed inside the contract-implementation block which makes system design hierarchal and staged across different levels of abstraction. Following are few examples of decomposition patterns that has been observed to be followed in Simulink based system design. These scripts are nothing but MATLAB scripts which with the help of programmatic modeling concepts creates new models based on selected pattern.

1. **Selective Output Decomposition**

   This decomposition is performed by grouping a sub-set of outports which can be controlled or processed independent of others. In Figure FC4.7a output decomposition script is run after selecting a sub-set of outports and then the resulting model is shown in Figure FC4.7b.

2. **Selective Input Decomposition**

   Selective Input Decomposition FC4.8 is similar to the previous one where instead of performing decomposition based on outports it is performed by selecting subset of inputs. The motivation behind is allow user to perform some preprocessing it's like creating a subtask and the its output is fed into the main logic block which then generates control signals. After choosing set of inports and executing script, an input dialog is being generated for user to insert names for signals to be generated by preprocessing block. The execution action is shown in Figure FC4.8.

3. **Serial Composition**

   Figure FC4.9 shows how this decomposition is being performed. It is a compact version of selective input decomposition where previously we're supply all input signals to the main logic block along with pre-processing block's output. Here the second block only relies upon signals generated from firt block only.

4. **Modes of Operation**

   In this pattern we have multiple pre-computation function implemented in different subsystem and based on some selection condition one of the subsystem's output is selected which is then handled by the main logic block as shown in Figure FC4.10.

Another script that has been added to the library which is not a decomposition pattern but rather can be used in system design for checking invariants. As shown in Figure FC4.11 an user can select the signal lines and upon executing this script it automatically adds up a Simulink block where invariant can be added as a proof objective.

### 4.3.4 Defining a Plant Model

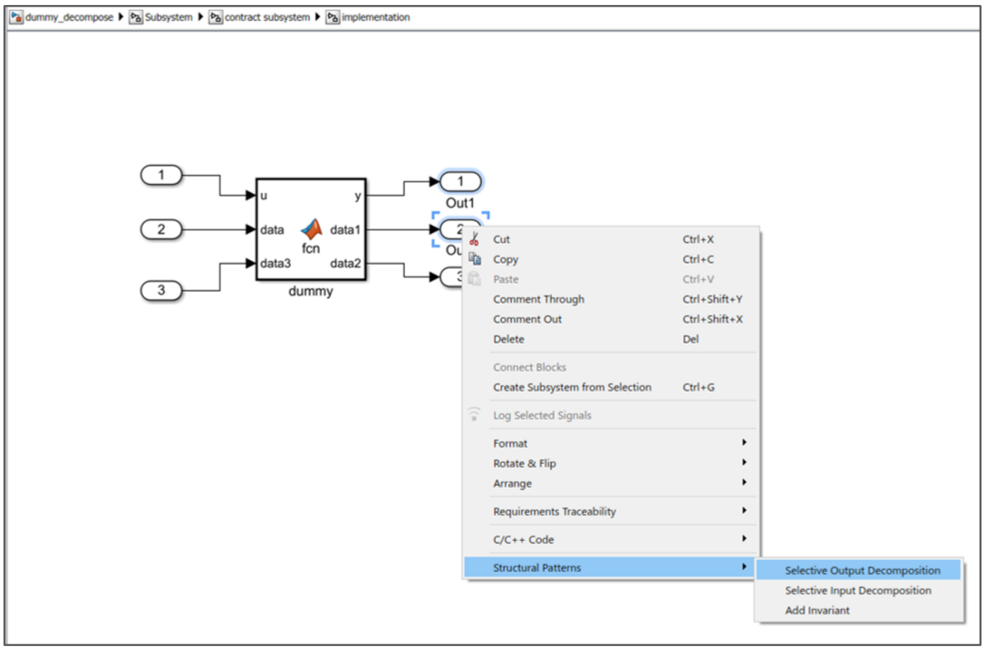
Sometimes while designing a control system we need a plant in place in order to mimic the real environment and give feedback to controller based on which control signals have been generated. In this case one has to develop both plant and controller's Simulink model. In Contract Based Development we can use a plant in contract mode by creating a contract which is nothing but all the assumptions regarding the plant's behavior according to control signals. Along with plant development user can also perform refinement of the plant in order to introduce more behavior into plant. This helps in developers to focus mostly on designing controller without explicitly designing a plant which cuts down development time and makes system verification much fast and efficient.

**Algorithm 4.1:** REFINESYSTEM

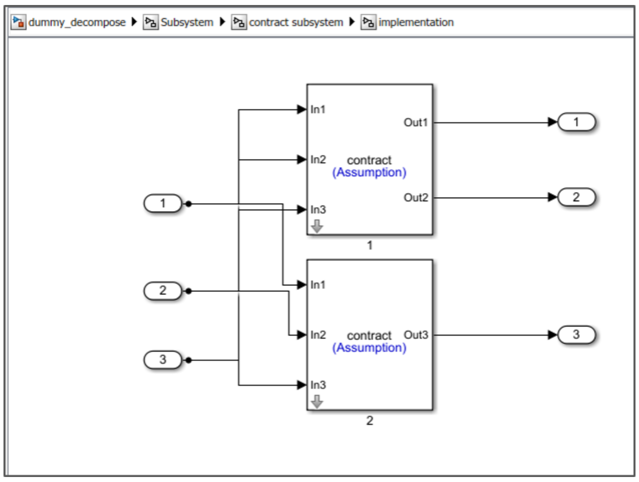
```
Input:
   1. $S$, Simulink subsystem block with abstract specification.
   2. List of all system requirements and invariants.
   3. Assumptions about environment.
Init: blockList = [ ]
Output: $S'$, a full Simulink Implementation.
begin
    if $S$ is a full specification then
        ▷ no scope for refinement
        IMPLEMENT($S$);
        blockList ← blockList \ currBlock;
        return;
    end
    switch userInput do
        case New Requirement
            $S$.contract ← UPDATECONTRACT($S$.contract, NewRequirement);
        end
        case Decomposition
            newBlockList ← DECOMPOSESYSTEM($S$);
            for $u \in$ newBlockList do
                $u$.contract ← CREATECONTRACT($S$.contract)
            end
            PROVEPROPERTY(root);
            blockList ← blockList ∪ newBlockList;
        end
        for $u \in$ blockList do
            REFINESYSTEM(u);
        end
    endsw
end
```

## 4.4 Simulink System Design using Refinement

The general idea behind using refinement is to get a system which is correct by construction at the end of the process. Till now we've only talked about refinement in software engineering and refinement in terms of contracts. Using the concept of contract based development we try to perform refinement in designing Simulink model. As seen in Figure FC4.6 for each block we define contract manually after performing decomposition. Another case is where refinement is performed by introducing new requirement at each of subsystem level and updating the contract for the corresponding component. Algorithm 4.1 gives us a refinement workflow that can be followed while performing System Design in Simulink. In this algorithm it can be seen at line 7 the decision of performing decomposition or updating the contract is solely depends on the designer. The patterns library along with SLDV provides support the user for testing the correctness of its decisions. Following this process designer ends up with a correct system model.

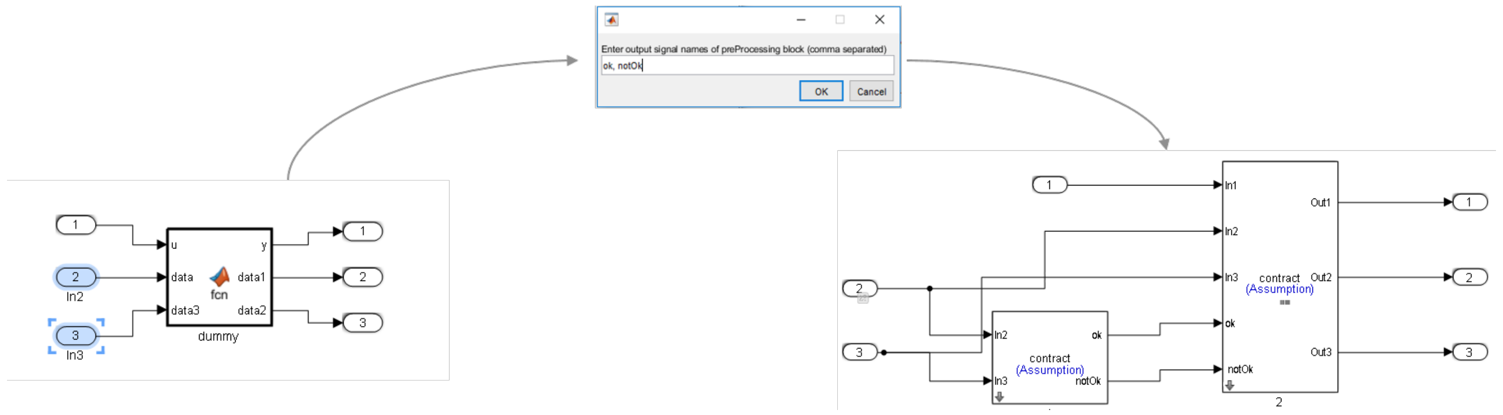

dummy_decompose ▸ Subsystem ▸ contract subsystem ▸ implementation
1
2
3
u
data
data3
y
data1
data2
fcn
dummy
Out1
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Comment Through Ctrl+Shift+Y
Comment Out Ctrl+Shift+X
Delete Del
Connect Blocks
Create Subsystem from Selection Ctrl+G
Log Selected Signals
Format
Rotate & Flip
Arrange
Requirements Traceability
C/C++ Code
Structural Patterns
Selective Output Decomposition
Selective Input Decomposition
Add Invariant

(a) Executing selective output decomposition script



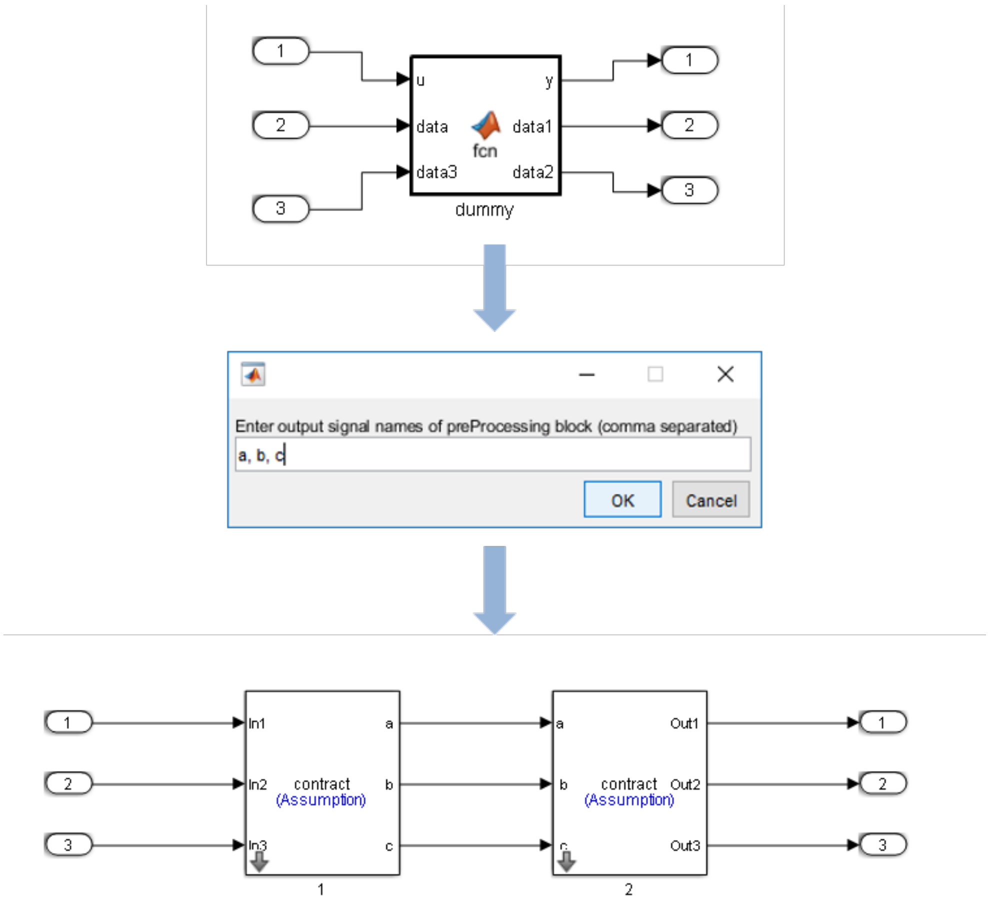
(b) Result after execution of selective output decomposition

Figure FC4.7: Selective Output Decomposition Pattern


Enter output signal names of preProcessing block (comma separated)

ok, notOk

Figure FC4.8: Selective Input Decomposition



Enter output signal names of preProcessing block (comma separated)

a, b, c

Figure FC4.9: Serial Decomposition Pattern

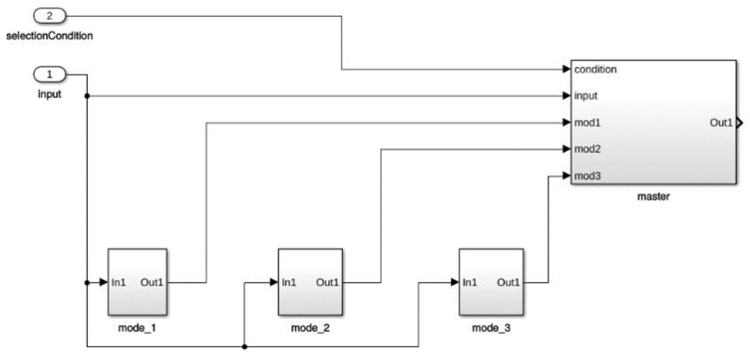


Figure FC4.10: Modes of Operation Pattern

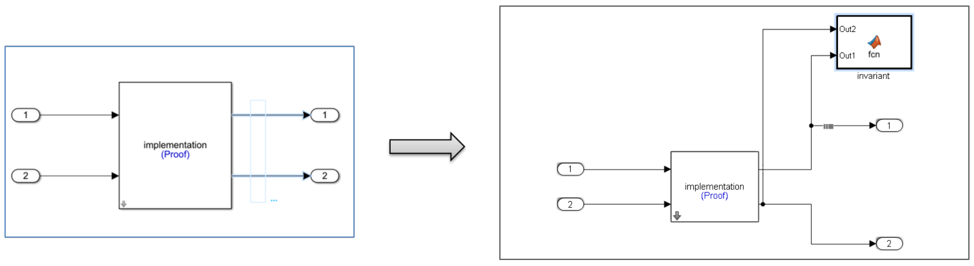


Figure FC4.11: Checking Invariant

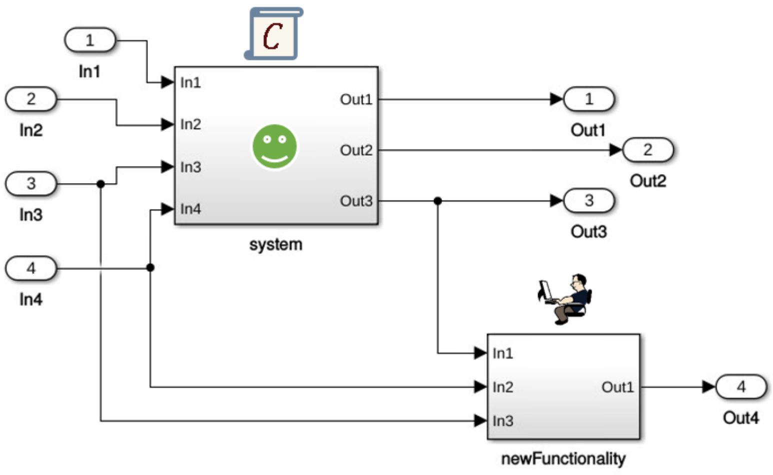


Figure FC4.12: Plant in Contract Mode in Closed Loop System

# CHAPTER 5

# CASE STUDY

The concepts that we formulated in previous chapters is being implemented and tested through this case study. In this case study a simple controller for a steam boiler plant is being designed using contract based development methodology for Simulink. The plant description, controller interface and controller requirements is stated in next section.

## 5.1 System Description

**Plant Components**

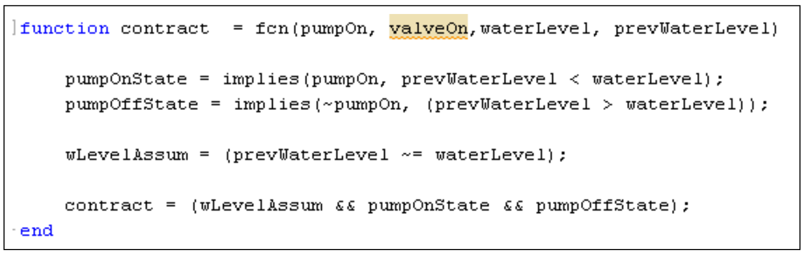
1. Water container
2. Water inlet
3. Steam outlet (valve)

**Sensors**

1. Water Level Sensor

**System Requirements**

1. When water level is high, pump should be **off** and valve(steam outlet) should be **on**.



```
function contract  = fcn(pumpOn, valveOn,waterLevel, prevWaterLevel)

    pumpOnState = implies(pumpOn, prevWaterLevel < waterLevel);
    pumpOffState = implies(~pumpOn, (prevWaterLevel > waterLevel));

    wLevelAssum = (prevWaterLevel ~= waterLevel);

    contract = (wLevelAssum && pumpOnState && pumpOffState);
end
```

Figure FC5.1: Assumptions Regarding Plant Behavior

2. When water level is low, pump should be **on** and valve should be **off**.
3. Pump and valve are not open simultaneously.
4. If water level is in between low and high, pump state doesn't change.

## 5.2 Assumptions Regarding Plant

Since it is a close loop system the controller has to work along with the steam boiler plant, we've made certain assumption regarding the plant. The assumption is written in EML and treated as a contract for the plant as shown in Figure FC5.1. It says that what should be the expected behavior of the plant if pump is **on** or **off** by setting a relationship between `prevWaterLevel` and `waterLevel`(current water level).

1. If Pump is on then water level should increase which is captured in the statement `implies(pumpOn,prevWaterLevel < waterLevel)`.
2. If Pump is off then water will evaporate which will result in water level drop which is captured in `implies(~pumpOn,prevWaterLevel > waterLevel)`.
3. We've also made an assumption that the water level is either strictly increasing or strictly decreasing never remains constant at any point of time i.e. captured in `prevWaterLevel ~= waterLevel` in the contract.

The `contract` is imposed on the plant model and helps in defining plant functionality. The `prevWaterLevel` value is got by using a delay block on `waterLevel` signal. After setting up the plant we move on to design the controller part.

## 5.3 Refinement Steps

### Model-0

Now we start modeling the controller. At the first step of refinement we don't perform any implementation. Everything at this stage is only specified. Here we specify a very abstract behavior of controller which addresses two of the requirements (1) and (2) which is reflected in status variables `expr1` and `expr2` as shown in Figure FC5.3. And we define the contract for this component as `contract = expr1 && expr2`. Now this contract describes an abstract behavior of this component.

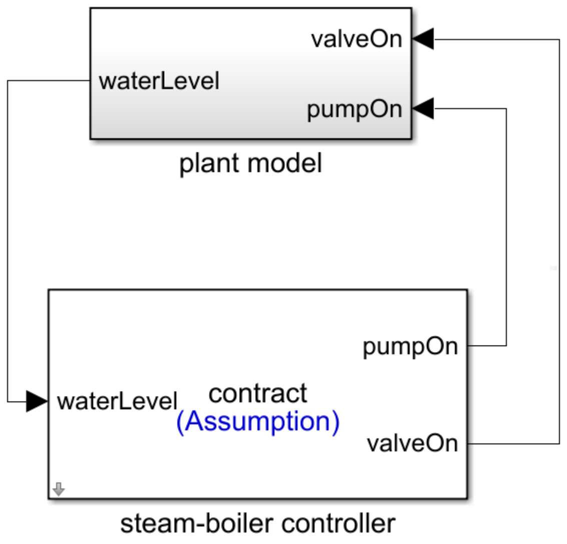
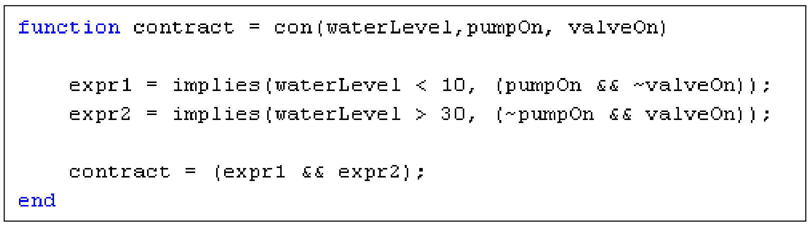


Figure FC5.2: Plant and Controller in Loop

### Model-1

Now in next step we observed that `pumpOn` and `valveOn` signals can be handled independently as there exits no dependency between them. We took the design



```
function contract = con(waterLevel,pumpOn, valveOn)

    expr1 = implies(waterLevel < 10, (pumpOn && ~valveOn));
    expr2 = implies(waterLevel > 30, (~pumpOn && valveOn));

    contract = (expr1 && expr2);
end
```

Figure FC5.3: Plant and Controller in Loop

decision to break the component *steam-boiler-controller* in to two new components one to handle `pumpOn` and second to handle `pumpOff` signal as show in Figure FC5.4. This decomposition is performed inside this contract-implementation block as it is hierarchal in nature using selective output decomposition pattern. Then for each of these newly created components we write the contracts. For pumpController component in Figure FC5.4 the contract is given in Figure FC5.5 which talks only about the behavior of `pumpOn` signal based on water level value. For valveController component contract is given Figure FC5.6 which describes behavior of only `valveOn` signal based on water level. Now, since we've performed one step of refinement where we broke the contract of parent component into two new contracts, we also need to verify that this decision has been taken is correct and as discussed in previously we'll use Simulink Design Verifier to discharge this proof. In this case the contract of the parent block i.e steam-boiler controller block in Figure FC5.2 will be used as a proof objective. This proof objective is verified by using Simulink Design Verifier and proof has been discharged.

**Model-2**

Next we choose the block pumpController in Figure FC5.4 and refine that model. We perform a selective input decomposition on this particular block as shown in Figure FC5.7. The contract for each of these new blocks decisionBlock and mainLogic is shown in Figure FC5.8 and FC5.9 respectively. The contract for

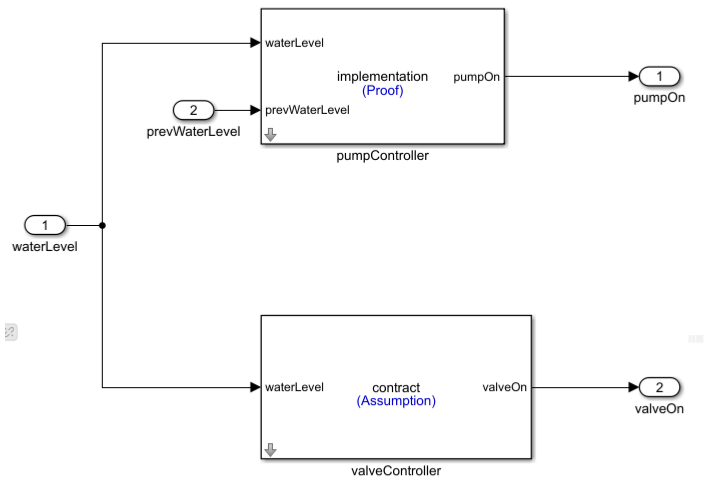


Figure FC5.4: First Refinement on Controller

```
function contract = con(waterLevel, pumpOn)

    pumpOnCondition = implies(waterLevel < 10, pumpOn);
    pumpOffCondition = implies(waterLevel > 30, ~pumpOn);

    contract = (pumpOnCondition && pumpOffCondition);

end
```

Figure FC5.5: Contract for pumpController Component of Figure FC5.4

```
function contract = con(waterLevel, valveOn)

valveOnCondition = implies(waterLevel > 30, valveOn);
valveOffCondition = implies(waterLevel <= 30, ~valveOn);

contract = (valveOnCondition && valveOffCondition);

end
```

Figure FC5.6: Contract for valveController Component of Figure FC5.4

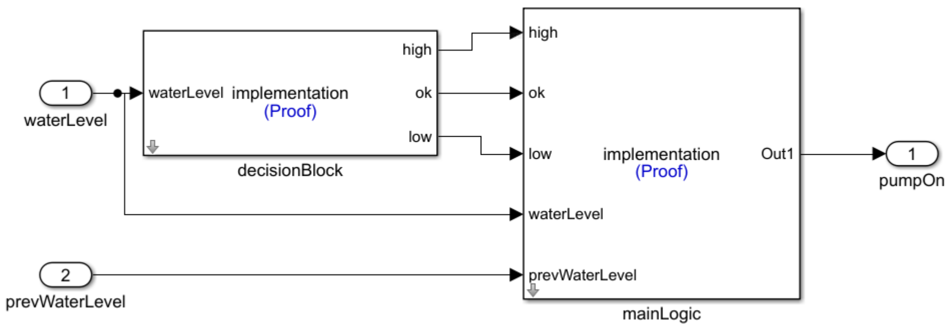


Figure FC5.7: Decomposition of pumpController in Figure FC5.4

decisionBlock makes one of the output true among the three based on water level value either `high,` `ok` or `low` and as a safety condition it says that at most one of the outport of decisionBlock can be active. The contract of mainLogic block has been updated with an additional requirement i.e requirement (4) from the list which talks about the behavior of controller when water level is in between high and low. It says if water level is rising then pump should be on until max limit is reached and if water level is decreasing then pump should remain off until water level reaches the lowest level. Now at this level all the top level contracts are being used as a proof objective in conjunction. The SLDV discharges all the proof objectives.

### Model-3

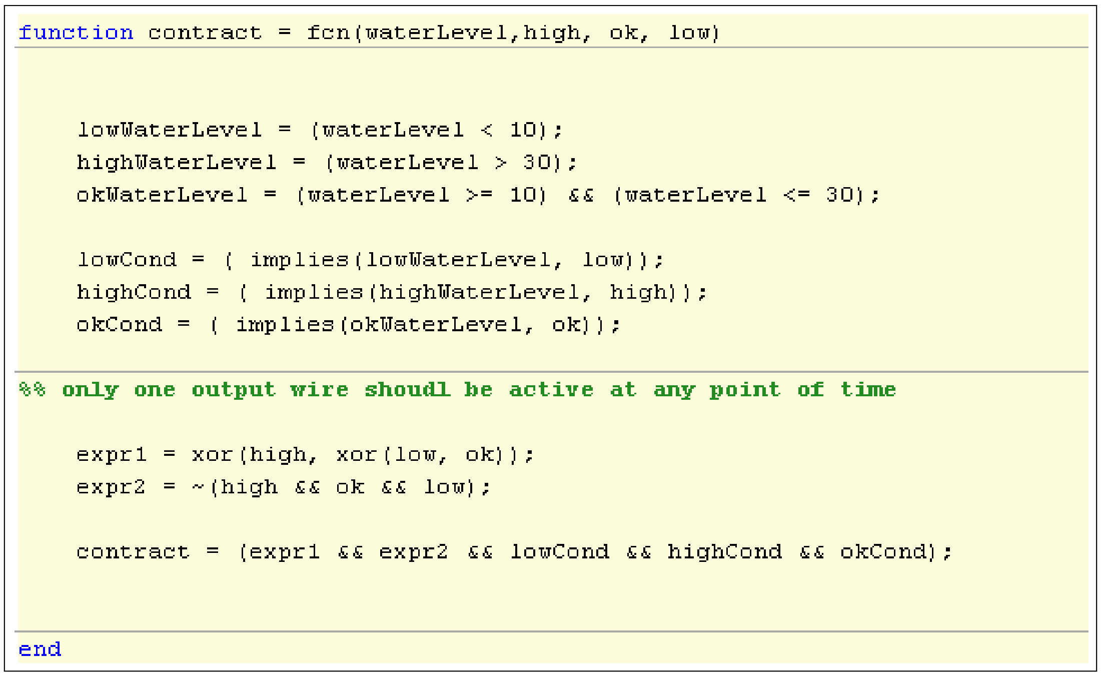
Since for decisionBlock in Figure FC5.7 has a fully specified contract it could now be translated into a corresponding Simulink model easily. Figure FC5.10 shows the final implementation for decisionBlock which is again verified against its proof objective i.e contract in Figure FC5.8 using SLDV and found to be correct.

### Model-4

Similar to decisionBlock, the appropriate implementation Figure FC5.11 for main-Logic block is also made which satisfies the contract in Figure FC5.9 which again tested using SLDV.

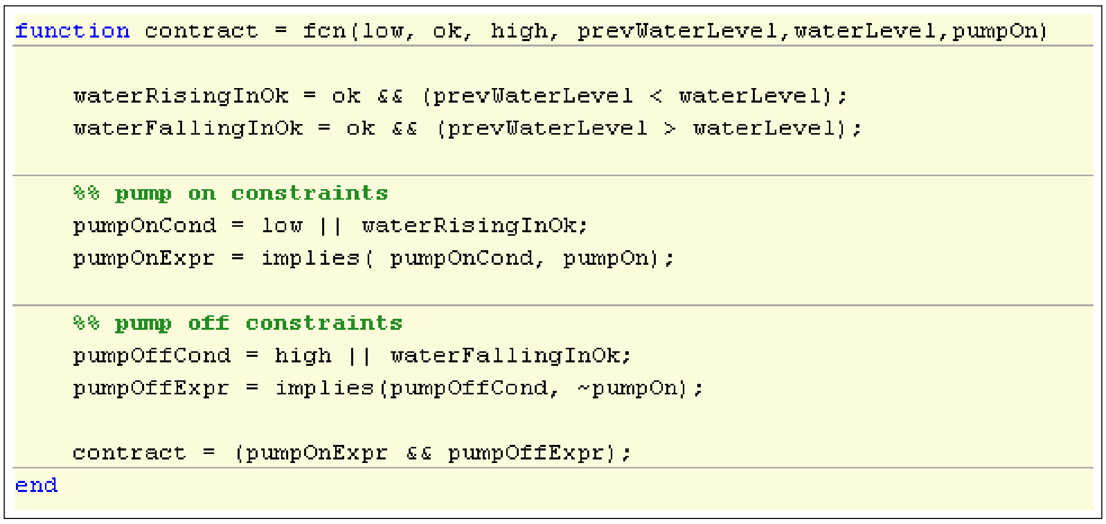
### Model-5

The final refinement is performed by implementing the valve controller contract given in Figure FC5.6. Its implementation is shown in Figure FC5.12 which is a simple switch block. The process of refinement and verification is performed in each step in order to ensure correctness of model is preserved at each step. After this a final run of sldv has been made, at this point all the contracts have been fully implemented and contacts are working as proof objectives at each level of refinement. The verification is performed on whole model against the contracts using Simulink Design Verifier . Now the final model is verified to be correct by construction as all properties are satisfied this model.



```
function contract = fcn(waterLevel,high, ok, low)


    lowWaterLevel = (waterLevel < 10);
    highWaterLevel = (waterLevel > 30);
    okWaterLevel = (waterLevel >= 10) && (waterLevel <= 30);

    lowCond = ( implies(lowWaterLevel, low));
    highCond = ( implies(highWaterLevel, high));
    okCond = ( implies(okWaterLevel, ok));

%% only one output wire shoudl be active at any point of time

    expr1 = xor(high, xor(low, ok));
    expr2 = ~(high && ok && low);

    contract = (expr1 && expr2 && lowCond && highCond && okCond);


end
```

Figure FC5.8: Contract for decisionBlock in FC5.7



```
function contract = fcn(low, ok, high, prevWaterLevel,waterLevel,pumpOn)

    waterRisingInOk = ok && (prevWaterLevel < waterLevel);
    waterFallingInOk = ok && (prevWaterLevel > waterLevel);

    %% pump on constraints
    pumpOnCond = low || waterRisingInOk;
    pumpOnExpr = implies( pumpOnCond, pumpOn);

    %% pump off constraints
    pumpOffCond = high || waterFallingInOk;
    pumpOffExpr = implies(pumpOffCond, ~pumpOn);

    contract = (pumpOnExpr && pumpOffExpr);
end
```

Figure FC5.9: Contract for mainLogic in Figure FC5.7

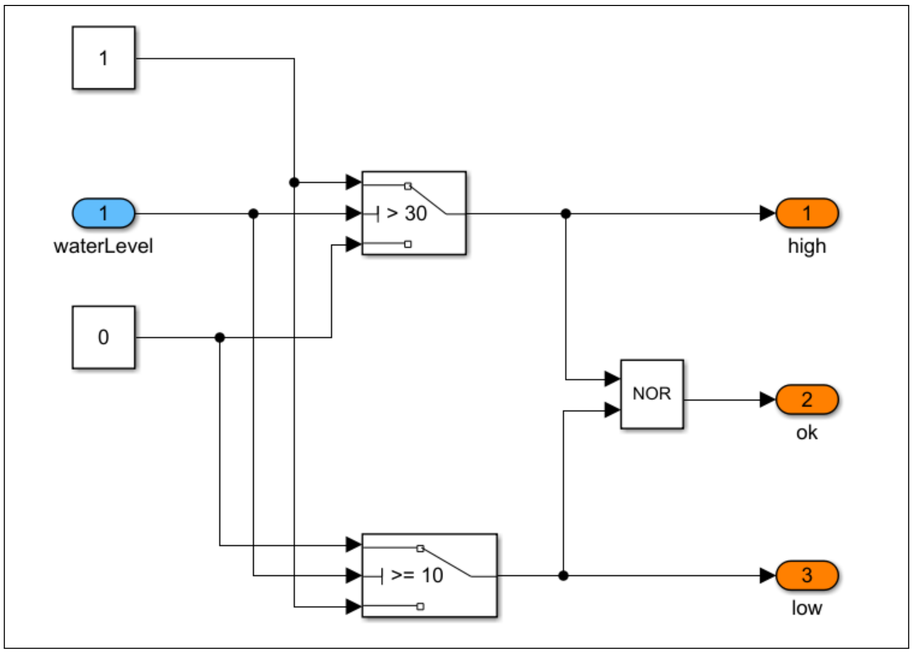


Figure FC5.10: Simulink implementation for contract in Figure FC5.8

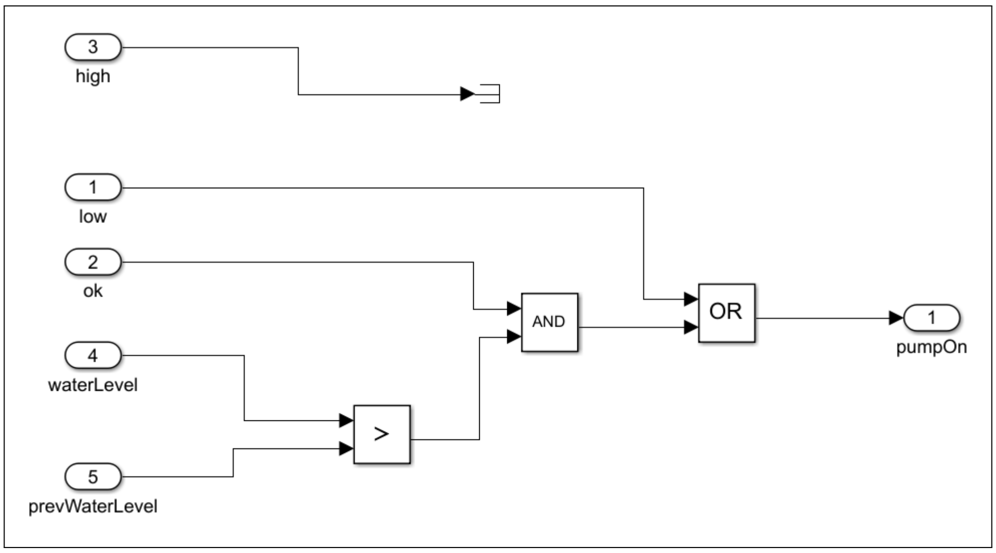


Figure FC5.11: Simulink implementation for contract in Figure FC5.9

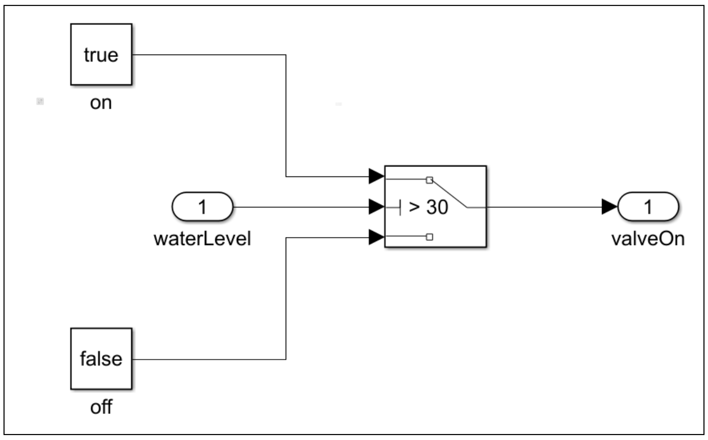


Figure FC5.12: Simulink implementation for valveController contract in Figure FC5.6

# CHAPTER 6

# CONCLUSION

This thesis work is mostly focused towards designing workflow and tool support that will enable developers to use the concept of contract based development and refinement conveniently. The same has been tested through case study, where we perform a top-down approach of developing control system with the help of refinement and different operations on contract. At the end of it, we had a controller which is correct by its construction. The idea of using a block in Simulink as both implementation and specification and choosing mode whenever required gives a lot of flexibility to system testing and verification. In each step we used SLDV to discharge the proofs which increases the reliability of the final product by not allowing intermediate design errors which ensures that no design errors has been carried forward to the programming stage. This saves a lot of time in debugging and fixing the error. A few use cases have also been discussed which shows the applicability of this thought in constructing control system. Some decomposition patterns are also been provided for the development process which makes this process more convenient to use.

## 6.1 Future Scope

For very large and complex models the level of abstractions to be performed will also be high. The *contract-implementation* subsystem which is used as unit of design in development process is hierarchal in nature. Each decomposition performed is housed inside this block. So it can be viewed as a node in a tree and the new sub-systems formed by decomposition will be children of this node in the tree. So, whole refinement process can be visualised through tree like structure shown in Figure FC6.1.

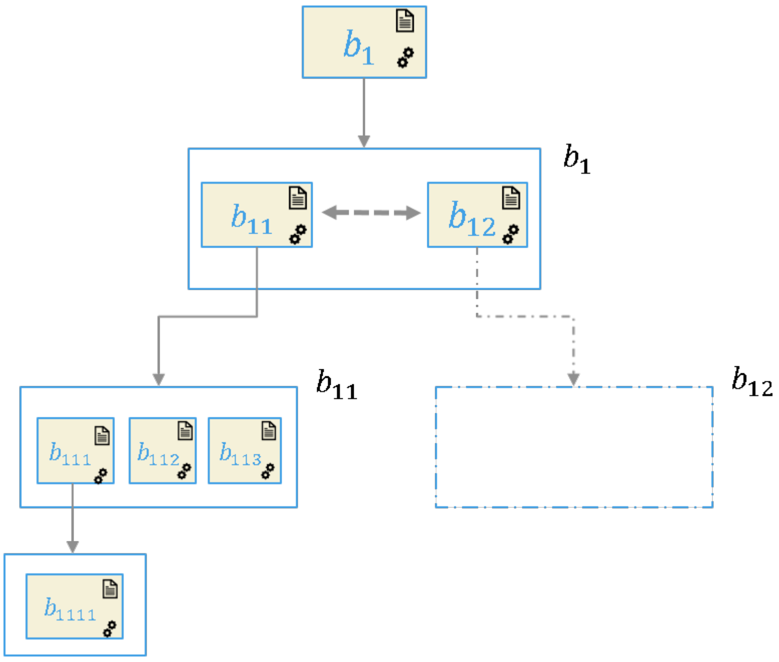


Figure FC6.1: Refinement Tree

Refinement tree captures hierarchal composition of the entire system developed using contract-implementation block. However this doesn't captures how the sibling components are interacting. This sort of gives an skeleton view of the whole system which can be used in different purposes in the development process.

i. It can be used an external controller for the whole models where each of the nodes has a reference to corresponding block such that it mode switching can performed externally. The motivation behind this thought is that the model will become much sophisticated in terms of the number of hierarchies which can be painful for the developers. This will allow user in localising the erroneous parts in the model which violates the contracts.

ii. Other application is in project management where after each refinement a new model is generated. If multiple developers are working on a designing controller, it shall capture the sequence of refinements been performed during modeling process by independent actors.

As mentioned earlier the contracts are defined manually by users, but we can provide some tool support for automating decomposition of contracts based on the pattern of decomposition performed. Also for each component there might be different implementations possible so we also need to provide support for including different implementation into model and to switch between one implementation to another and perform system analysis.

# Bibliography

[1] Anne E Haxthausen. An Introduction to Formal Methods for the Development of Safety-critical Applications. pages 1–32, 2010.

[2] MathWorks. Simulink design verifier documentation, r2017a, 2017.

[3] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim G Larsen. *Contracts for system design*. PhD thesis, Inria, 2012.

[4] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[5] Boström, Pontus, Lionel Morel and Marina Waldén. Stepwise development of Simulink models using the refinement calculus framework. *International Colloquium on Theoretical Aspects of Computing*, 2007.

[6] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *International Symposium on Formal Methods for Components and Objects*, pages 200–225. Springer, 2007.