Pós-Graduação em Ciência da Computação

"Behavioural Preservation in Fault Tolerant Patterns"

By

Diego Machado Dias

M.Sc. Dissertation

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, FEBRUARY/2012

Federal University of Pernambuco
Center for Informatics
Graduate in Computer Science

Diego Machado Dias

"Behavioural Preservation in Fault Tolerant Patterns"

A M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: *Juliano Manabu Iyoda*
Co-Advisor: *Adalberto Cajueiro de Farias*

RECIFE, FEBRUARY/2012

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Dias, Diego Machado
Behavioural preservation in fault tolerant patterns / Diego Machado Dias. - Recife: O Autor, 2012.
xi, 106 p. : il., fig., tab.

Orientador: Juliano Manabu Iyoda.
Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2012.
Inclui bibliografia.

1. Engenharia de software. 2. Métodos formais. 3. Computação tolerante a falhas. I. Iyoda, Juliano Manabu (orientador). II. Título.

005.1 CDD (23. ed.) MEI2012 – 060

Dissertação de Mestrado apresentada por **Diego Machado Dias** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Behavioural Preservation in Fault Tolerant Patterns**", orientada pelo **Prof. Juliano Manabu Iyoda** e aprovada pela Banca Examinadora formada pelos professores:

_____

Prof. Alexandre Cabral Mota
Centro de Informática / UFPE

_____

Prof. Rohit Gheyi
Departamento de Sistemas e Computação / UFCG

_____

Prof. Juliano Manabu Iyoda
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 2 de março de 2012.

_____

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Acknowledgements

The project described in this thesis would not have been possible without the support of several people that directly and indirectly influenced it.

Firstly, I would like to thank my advisor, Juliano Iyoda. I owe Juliano for his enthusiasm, availability, guidance and numerous discussions about this work that help me to understand the really meaning of doing research in computer science. I'm very thank to him not only for his supervision, but for the assistance dedicated to me along these two years of the master's course. I have no doubt that more than a advisor, he has become a truly friend.

I'm also grateful to Alexandre Mota, Augusto Sampaio, Giovanny Lucero and Adalberto Cajueiro for their suggestions to improve this work. The feedbacks received from them in the Formula's seminars were decisive for delimiting the scope of this work. I also devote special thanks to the anonymous reviewers of the paper [10] submitted to SBMF, their feedbacks give valuable contributions to this work.

I thank to all friends that made my stay in Recife a rich life experience, especially Adauto Trigueiro, Carlo Reillen, Orivaldo Vieira, Robson Silva, Weslley Torres, and colleagues of classes of dancing. Besides they, I thank my teachers at UFBA. They gave me evidences of how good is an academic career and motivated me to engaging a MSc course.

I could not finish the acknowledgements without thanking to Thays Amaral for her care, patient and attention during these two years; and to my parents for understanding the distance.

My MSc has been financially supported by the FACEPE (Fundação de Amparo à Ciência e Tecnologia do Estado de Pernambuco).

Resumo

No desenvolvimento de sistemas críticos é prática comum fazer uso de redundância para se alcançar níveis mais elevados de confiabilidade. Existem padrões de projeto bem estabelecidos que introduzem redundância e que são amplamente documentados na literatura e adotados na indústria. Entretanto, há relativamente poucos trabalhos que tentam verificar formalmente estes padrões com respeito à preservação de comportamento.

Este trabalho propõe uma abordagem para especificar de tais padrões, chamados aqui de *padrões de tolerância à falhas*, usando HOL. Utiliza-se o provador de teoremas HOL4 para provar a composicionalidade e corretude de padrões de tolerância à falhas. A abordagem proposta é ilustrada através da modelagem de três padrões de tolerância à falhas clássicos: redundâcia homogênea, redundância heterogênea e redundância modular tripla. O modelo formal desenvolvido neste trabalho considera que um sistema (sem redundância) computa uma determinada função, com determinado atraso e é susceptível a falhar aleatoriamente.

Para provar que um padrão de tolerância a falhas preserva o comportamento dos seus subsistemas, foram definidos novos conceitos de refinamento. Engenheiros de sistemas normalmente aceitam que padrões de tolerância a falhas não mudam a funcionalidade de um sistema. Entretanto, esta prática não é compatível com as noções clássicas de refinamento. Desta forma definiu-se noções de refinamento axiomaticamente para provar que os padrões de tolerância a falhas formalizados preservam comportamento.

Também provou-se que os padrões de tolerância a falhas analisados são composicionais, no sentido que eles podem ser aplicados consecutivamente um número arbitrário de vezes. O resultado desta aplicação é ainda um sistema, cujo o atraso, modelo de falhas e funcionalidade podem ser sistematicamente descobertos (através de prova) com quase nenhum esforço.

Para ilustrar a utilização dos padrões é aplicado o padrão de redundância modular tripla à um sistema simplificado de controle de elevação de um avião. Mostrou-se que tendo-se verificado previamente um padrão, a aplicação deste a um sistema específico e a prova de preservação de comportamento do sistema resultante torna-se trivial. Este trabalho foi desenvolvido em colaboração com a fabricante brasileira de aviões Embraer.

Palavras-chave: Padrões de tolerância à falhas, HOL4, preservação de comportamento, refinamento.

Abstract

In the development of critical systems it is common practise to make use of redundancy in order to achieve higher levels of reliability. There are well established design patterns that introduce redundancy and that are widely documented in the literature and adopted by the industry. However there have been few attempts to formally verify them with respect to behavioural preservation.

In this work, we purpose an approach to specify such design patterns, called here *fault tolerant patterns*, using HOL. We use the theorem prover HOL4 to prove the compositionality and correctness of the fault tolerant patterns. We illustrate our approach by modelling three classical fault tolerant patterns: homogeneous redundancy, heterogeneous redundancy and triple modular redundancy. Our model takes into account that the original system (without redundancy) computes a certain function with some delay and is amenable to random failures.

In order to prove that a fault tolerant pattern preserves the behaviour of its subsystems, we defined new notions of refinement. Systems engineers commonly accept the fact that fault tolerant patterns do not change the functionality of a system. However, this practise is not compatible with the classical refinement notions. Thus we defined axiomatic notions of refinement to prove that the formalised fault tolerant patterns preserve the behaviour of its subsystems.

We also proved that our fault tolerant patterns are compositional in the sense that we can apply fault tolerant patterns consecutively and for an arbitrary number of times. The result of that is still a system whose delay, failure model and functionality can be systematically discovered (by proof) with almost no effort.

In order to illustrate the usage of the patterns we applied the triple modular redundancy pattern to a simplified avionic Elevator Control System. We showed that once a fault tolerant pattern is verified, the application of it to a specific system and the proof of the behavioural preservation of the resulting system becomes trivial. This work has been done in collaboration with the Brazilian aircraft manufacturer Embraer.

Keywords: Fault tolerant patterns, HOL4, behavioural preservation, refinement.

Contents

List of Figures

List of Tables

1 Introduction

Software engineering has gained maturity over the years by incorporating techniques that automate the software development process and that assist the validation of an implementation with respect to its requirements. However the classical problems of software development are still present. Bugs are discovered in the last stages of the development, projects run over time and over budget. In the case of critical systems like flight control systems, nuclear power plant systems, radiotherapy machines, insulin pumps, this level of quality (specially regarding correctness) is not acceptable. The consequences of releasing a critical software or a critical hardware with bugs can put in risk our safety and can cause major financial loss. In these cases, it is important to guarantee the absence of bugs for all execution paths of a software or for all combinations of inputs of a hardware by using more rigorous techniques like Formal Methods.

Formal Methods comprise a set of techniques and tools based on mathematical modelling and formal logic that are used to specify and verify requirements and designs for computer systems [24]. Since the past decades, the adoption of Formal Methods among software developers has increased. The usage of Formal Methods in a project can vary from employing mathematical notations mixed with natural language to the sole use of formal languages. Revealing ambiguities, incompleteness and inconsistencies in a system are just some advantages that the use of Formal Methods brings. When used early in the system development process, they can reveal design flaws that otherwise might be discovered only during the testing and the debugging phases, when they are more costly. When used later, they can help to prove the correctness of a system implementation and the equivalence of different implementations [43]. In this work, we investigate how Formal Methods can be applied to the safety of critical systems. In particular, our aim is to propose a method that allows us to decide if the introduction and management of redundancy does not change the functionality of the original system.

Critical systems are developed in such a way that safety is addressed explicitly and under severe regulations. For instance, catastrophic failure of digital flight control systems must be extremely improbable to occur: the failure rate must be lower than 10^{-9} per hour [23]. In order to achieve such a restriction, critical systems are replicated in different ways in order to guarantee that a failure on one component is covered by another replica of it [11]. Therefore fault tolerance is mostly based on redundancy.

Redundancy is implemented in different ways. We can replicate a system with an identical copy of it (and add a monitor to check which one is working); or we can make copies of a system that have different design and implementation, but that computes the same function; or we can ask for a voter to output an average value of the output of the replicas; and so on. These design solutions, which we call here *fault tolerant patterns*, are widely used in industry.

Fault tolerant patterns are *de facto* standards in industry. Unfortunately, there has been few attempts to formally verify this catalogue of design patterns with respect to the behavioural preservation of the original system. If not handled correctly, the redundancy can itself become the primary source of system failure. Pioneering work on the verification of fault tolerant patterns was done in the nineties [6, 34, 38]. These works proved the correctness of fault tolerant patterns by hand [6], by using theorems provers [34], and with model checkers [38]. More recently, Dajani-Brown *et al.* [8] used SCADE's [1] model checker to verify the correctness of a triple redundant system. Jesus [9] modelled Simulink [26] block diagrams[1] in CSP [19] and found, using the FDR2 refinement checker [12], a condition that could cause a failure in an Elevator Control System of Embraer.

In this dissertation, we modelled in the HOL4 theorem prover [15] three fault tolerant patterns used in industry and proved that the application of these patterns preserves the behaviour of the original system. We regard that the hardware failures can be caused by deterioration, fatigue, wear, etc. Different from the previous works, our model regards functionality, delay and failures as separate entities. It allows us to analyse the behaviour of a system without making assumptions about its failure rate or delay. Moreover, during the proof of correctness, we discover that the patterns that output the average of the original system do not preserve the behaviour according to the traditional notions of refinement [3]. That led us to define new notions of refinement. We illustrate our approach with a case study inspired by Embraer's Elevator Control System.

[1] A block diagram is a diagram of a system composed of blocks (boxes) that represent functions, and lines that connect the blocks. This representation is commonly used in the engineering world in hardware design, electronic design, software design, and process flow diagrams.

This work is concerned with the behavioural preservation of the fault tolerant patterns. It is not concerned with the benefits of redundancy with respect to failure rate improvements. There are other studies [2, 25] that show the quantitative reliability benefits of using redundancy and discuss aspects as cost increase and modifiability analysis.

1.1 Overview of the Proposed Solution

The process to build a library of verified fault tolerant patterns is depicted in Figure 1.1. The strategy is divided in two phases. The first phase consists in developing a formal model of fault tolerant patterns in HOL4 (Chapter 3). A fault tolerant pattern is modelled as a predicate in the HOL4 logic that captures the pattern architecture. Usually, the fault tolerant patterns are modelled as block diagrams [22].

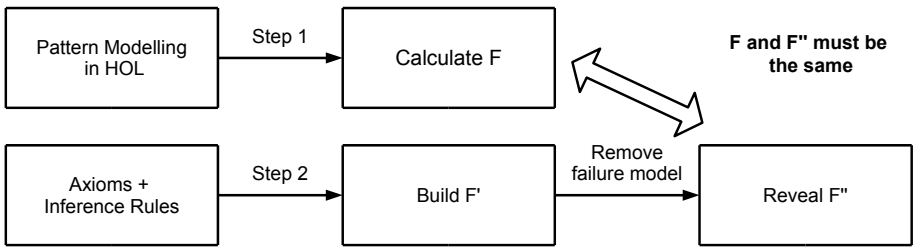


Figure 1.1 Overview of our proposed solution.

Given such a formal description of a fault tolerant pattern, it is possible to prove that this pattern is an implementation of a system that computes a certain function F (Step 1). We found out that the function F computes "almost the same" function of its replicated subsystems.

The second phase of the project formalised what "almost the same computation" means. We introduced new notions of refinement to capture the behavioural preservation of the fault tolerant patterns (Chapter 4). Step 2 of the strategy proves that the behaviour of each replicated subsystem of a fault tolerant pattern is refined by some function F'. Because of technical details of our refinement calculus, this function is generated with special annotations (the subsystem failure model) that, once ignored, results in a function F''. This function happens to be exactly F, thus proving (under our particular refinement notions) that the replicas are refined by the fault tolerant system.

Figure 1.1 shows the process of building formal fault tolerant patterns. This process is a once-and-forall activity to build a library of verified fault tolerant patterns. Whenever we have to *use* that library, no verification effort is needed.

1.2 Statement of the Contributions

As a result of the work presented in this dissertation, the following contributions can be highlighted:

- We propose a model in Higher Order Logic of fault tolerant patterns. Our model was applied to three classical fault tolerant patterns: Homogeneous Redundancy, Heterogeneous Redundancy and Triple Modular Redundancy. However, our model is not specific to these patterns. The basic components offered by our specification can be combined to model other patterns in future work.

- Different from previous works, our model specify systems that can suffer from random failure and takes into account the possibility of the signals to come with failure [10].

- We prove that the fault tolerant patterns behaviour is a refinement of its subsystem replicas. In order to prove this behavioural preservation we define new notions of refinement that capture the intuition and common practise of systems engineers.

- Our correctness theorems are compositional, allowing us to prove the correctness of a composition of fault tolerant patterns with minimum proof effort.

- We illustrate our approach with a case study in which we apply a Triple Modular Redundancy pattern to a simplified model of an avionics elevator control system. We show that it is easy to prove that the addition of the redundancy did not introduce more bugs to the original (non-replicated) system. This work has been done in collaboration with the Brazilian aircraft manufacturer Embraer.

1.3 Dissertation Structure

This dissertation is organised in seven chapters and two appendices.

- Chapter 2 presents a brief description of the HOL Logic and the HOL4 system — the logical system and its mechanisation — and describes how hardware is modelled and verified in HOL4. This chapter also presents an overview of the fault tolerant patterns modelled in this work.

- Chapter 3 introduces the specification of a generic system and the components that make up the implementation of the system. This chapter also develops the HOL4

specification of three fault tolerant patterns, namely, Homogeneous Redundancy, Heterogeneous Redundancy and Triple Modular Redundancy. Finally, we prove that all fault tolerant patterns implement a system that computes a certain function, has a delay and can suffer from random failures.

- Chapter 4 proposes a new notions of refinement to formalise behavioural preservation after introducing replicas. The axioms presented in this chapter can be applied to refine a system without fault tolerance by one that is fault tolerant.
- Chapter 5 illustrates our approach in a case study that introduces Triple Modular Redundancy to the control function of a simplified Elevator Control System (ECS) inspired by Embraer's ECS.
- Chapter 6 discusses related work on formalisation of fault tolerant systems and on formal specification of function block diagrams. We compare and discuss the main differences of these approaches from ours.
- Chapter 7 concludes and addresses future work.
- Appendix A contains the entire specification in HOL4, including the proof scripts and the case study.
- Appendix B contains the proofs of the Section 4.2 of Chapter 4 (New Notions of Refinement).

2

Background

This chapter introduces background material relevant for this dissertation. Section 2.1 gives an introduction to Higher Order Logic (HOL), Section 2.2 discusses how this formalism is used to model hardware, and Section 2.3 illustrates how prove the correctness of a hardware implementation. The same modelling style used to model hardware is later used to model fault tolerant patterns. Section 2.4 describes the fault tolerant patterns formalised in this work: homogeneous redundancy, heterogeneous redundancy and triple modular redundancy. Finally, Section 2.5 concludes.

2.1 Introduction to HOL Logic

HOL stands for Higher Order Logic and, in this work, it also refers to the particular formulation developed by Mike Gordon at the University of Cambridge in the 1980s [13]. HOL is a predicate calculus with terms from the typed lambda calculus [29]. The notation used to represent the logical operations in HOL is similar to conventional predicate calculus notation.

There are only four kinds of terms in HOL: constants, variables, applications and λ-abstractions. A lambda abstraction $\lambda x.P(x)$ denotes an anonymous function that maps its argument x to the value $P(x)$. For example, $(\lambda x.\ x{+}1)$ denotes the function that takes a number and returns its successor. Variables can range over functions (λ-abstractions) and predicates. Functions can take functions as arguments, return functions as results and can be partially applied to an argument. For example, when the function $(add\ n\ m = n{+}m)$ takes as argument $n = 3$, it returns a function that takes a number m as argument and adds 3 to it. These features characterise HOL as a 'higher order logic'. As in λ-calculus, application have the form $(M\ N)$ and denotes the result of applying the function M to the value N. For example $((\lambda x.\ x{+}1)\ 5)$ evaluates to 6. By convention, application is left-

associative: $f\ x_1\ x_2\ ...\ x_n = (((f\ x_1)\ x_2)...\ x_n)$. For convenience, there are many syntactic sugars in HOL. For instance, the HOL primitive definition for the logical constant *true* is $\mathsf{T} = ((\lambda x.x) = (\lambda x.x))$, while the "for all" is defined as $\forall = (\lambda P.P = \lambda x.T)$.

Table 2.1 summarises a subset of the (sugared) notation of HOL with a short description for each term. These terms represent the constants *true* and *false*, logical operations (negation, disjunction, conjunction and implication), equalities, variable quantification and a conditional operation. From now on we will use the syntax presented in Table 2.1 instead of HOLs primitive definitions.

Term	ASCII Representation	Informal semantics
T	`T`	*true*
F	`F`	*false*
$\neg t$	`~t`	*not t*
$t_1 \vee t_2$	`t1\/t2`	t_1 *or* t_2
$t_1 \wedge t_2$	`t1/\t2`	t_1 *and* t_2
$t_1 \Rightarrow t_2$	`t1 ==> t2`	t_1 *implies* t_2
$t_1 = t_2$	`t1 = t2`	t_1 *equals* t_2
$\forall x.\ t$	`!x. t`	*for all* $x : t$
$\exists x.\ t$	`?x. t`	*for some* $x : t$
if t *then* t_1 *else* t_2	`if t then t1 else t2`	*conditional*

Table 2.1 Terms of the HOL logic.

Every term is associated with a unique type, which denotes a set. There are four kinds of types in the HOL logic described by the following BNF grammar.

$$\mathsf{T} ::= \alpha \quad | \quad c \quad | \quad (\mathsf{T}_1, ..., \mathsf{T}_n)\ op \quad | \quad \mathsf{T}_1 \rightarrow \mathsf{T}_2$$

In the grammar before, α ranges over type variables (usually represented by Greek letters), c ranges over atomic types (e.g. *bool*, *num*, *integer*, *real*, etc.), *op* ranges over type operators (e.g. *prod*, the Cartesian product) and $\mathsf{T}_1 \rightarrow \mathsf{T}_2$ ranges over function types whose domain is of the type T_1 and range is of the type T_2 (e.g. $num \rightarrow bool$ represents the type of all functions from natural numbers to booleans). Some type operators also have the infix form. For example, the infix form of the Cartesian product $(\mathsf{T}_1, \mathsf{T}_2)\ prod$ is T_1 # T_2. By convention, infix operators as # and $\rightarrow$ are right-associative as illustrated by the equation next.

$$\mathsf{T}_1 \rightarrow \mathsf{T}_2 \rightarrow ... \rightarrow \mathsf{T}_{n-1} \rightarrow \mathsf{T}_n \ = \ \mathsf{T}_1 \rightarrow (\mathsf{T}_2 \rightarrow ... \rightarrow (\mathsf{T}_{n-1} \rightarrow \mathsf{T}_n))$$

As type variables are available in HOL, it is possible to take advantage of it to

define polymorphic operations. For example, the definition $ID = (\lambda x.\ x) : \alpha \rightarrow \alpha$ is a polymorphic definition built using type variables, where α denotes 'any type'. *ID* is the function that takes a value as input and returns it unchanged as output. Definitions can also be given without explicitly declaring the types. In this case, types are inferred from the context. An example of a definition where types are not defined explicitly is the function $EQUAL = (\lambda x\ y.\ x = y)$ that compares if its arguments are equals.

The deductive system of the HOL logic is specified by eight rules of inference and five axioms [28]. Every theorem in HOL can be obtained by repeatedly applying these primitive inference rules and the five axioms. The HOL4 system, one of the implementations of the HOL logic, has a large library of theories (e.g. natural numbers, lists, groups, etc.) built from the primitive definitions of the HOL logic. The HOL4 system is the implementation adopted in this work due to its widely available documentation for hardware verification [14, 18, 21]. Some of these works inspired us to formalise the concept of system, which is the unit of replication in this work. The notion of a system is discussed in Chapter 3.

The next section discusses how the HOL4 system can be used to model and prove properties about hardware.

2.2 Modelling Hardware in HOL

Hardware components are modelled in HOL by predicates that establish the relation between external signals (wires). A generic hardware component is regarded as a black-box that comprises *n* inputs and *m* outputs, as depicted in Figure 2.1.

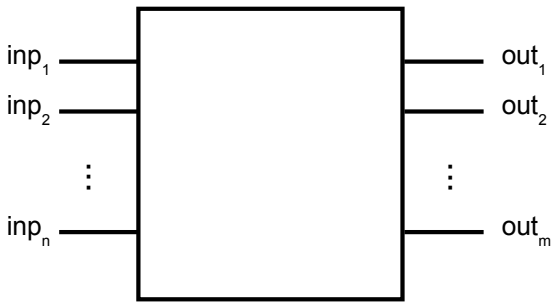
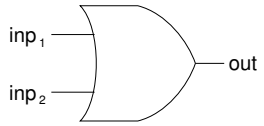


Figure 2.1 Generic hardware component

Signals are modelled as functions from time (natural numbers) to some type. For example, in HOL the natural numbers are represented by the type *num* and a signal that carries

booleans can be modelled as $sig : num \rightarrow bool$. The behaviour of a generic hardware component is described by a predicate with the form $P(inp_1, inp_2, ..., inp_n, out_1, out_2, ..., out_m)$. This predicate is true if and only if the relation specified between inputs and outputs is true. Usually, this relation is defined for each instant of time. For example, Figure 2.2 illustrates how a logical port *OR* is modelled in HOL.



$$OR\ (inp_1,\ inp_2,\ out)\ =\ \forall t.\ out\ t = (inp_1\ t)\ \vee\ (inp_2\ t)$$

Figure 2.2 OR logic gate

The signals inp_1, inp_2 and *out* are functions from time to boolean ($num \rightarrow bool$). The predicate $OR(inp_1, inp_2, out)$ specifies that the output signal is the pointwise application of the disjunction operation on the input signals inp_1 and inp_2. The first equality symbol establishes that $OR(inp_1, inp_2, out)$ is a shorthand for $\forall t.\ out\ t = (inp_1\ t)\ \vee\ (inp_2\ t)$. The second equality symbol defines the value of $out\ t$. Note that this component does not introduce delay to carry out the computation.

Another way of modelling this component is by encapsulating all input signals into a unique input signal of a compound type. The predicate OR' illustrates this approach. It receives a compound signal $inp : num \rightarrow (bool\ \#\ bool)$ and defines the output signal as the pointwise application of the disjunction operation on the components of the input signal.

$$OR'\ (inp : num \rightarrow (bool\ \#\ bool),\ out)\ =\ \forall t.\ out\ t = FST(inp\ t)\ \vee\ SND(inp\ t)$$

The operations *FST* and *SND* return the first and second element of an ordered pair, respectively. These operations are predefined in HOL4. The signal *inp* is a function from time (*num*) to a pair of booleans. This modelling captures the notion of a bus, which carries several bits in parallel. As the HOL4 system allows us to declare terms omitting their types, we can write the same predicate in a more readable form by removing the type declaration. In this dissertation, whenever possible we omit types to make the specification more readable. The OR' definition with implicit types is presented next.

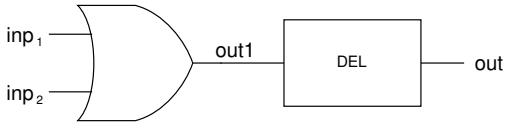
$$OR'\ (inp,\ out)\ =\ \forall t.\ out\ t = FST(inp\ t)\ \vee\ SND(inp\ t)$$

The next example is a delay component that introduces a unitary delay in a signal. Figure 2.3 specifies that the output at time $t+1$ is equal to the input at time t. Note that this is a partial specification: the output at time 0 is undefined.

$$DEL\ (inp, out)\ = \forall t.\ out\ (t+1) = (inp\ t)$$

Figure 2.3 Unitary delay component

To specify the connection among components in HOL, we just connect them using logical conjunction. The name of internal wires that connect components must match. The predicate constructed this way imposes that these wires must satisfy the specification of all connected components. To hide the internal wires from the specification, these wires are existentially quantified. For example, the connection between the components OR' and DEL is specified as depicted in Figure 2.4. Remember that the specification OR' models inp_1 and inp_2 using a unique input, which is $inp = (inp_1, inp_2)$. The wire $out1$ is a local wire hidden from the external environment.



$$OR'\_DEL(inp,\ out) = \exists out1.\ OR'(inp,\ out1)\ \wedge\ DEL(out1,\ out)$$

Figure 2.4 Connecting components OR' and DEL

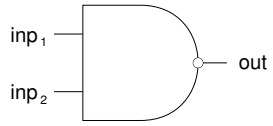
2.3 Hardware Verification in HOL

This section shows how to prove that a given circuit implements a specification. In order to illustrate the process of proof, we develop a simple example where a specification of an *AND* logic gate is implemented by a circuit that uses only *NAND* logic gates. We discuss the manual proof and how the proof is discharged in the HOL4 system.

First we define the formal specification of the system, i.e. a predicate that specifies the required behaviour of the circuit. In our example, the specification states that the output at time t is given by application of the conjunction on the input signals inp_1 and inp_2.

$$AND\ (inp_1,\ inp_2,\ out) = \forall t.\ out\ t = (inp_1\ t) \wedge (inp_2\ t)$$

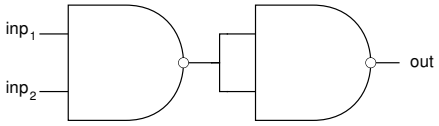
After specifying the desired behaviour of the system, an implementation must be formally defined. For simplicity, we provide an implementation of *AND* that uses only *NAND* logic gates. The *NAND* specification is described in Figure 2.5.



$$NAND\ (inp_1,\ inp_2,\ out) = \forall\ t.out\ t = \neg((inp_1\ t) \wedge (inp_2\ t))$$

Figure 2.5 NAND logic gate

The simplest implementation of an *AND* logic gate that uses only *NAND* logic ports is defined by the predicate *IMPL*. This implementation connects two *NAND* logic gates as depicted in Figure 2.6.



$$IMPL\ (inp_1,\ inp_2,\ out) = \exists out_1.NAND(inp_1,\ inp_2,\ out_1) \wedge NAND(out_1,\ out_1,\ out)$$

Figure 2.6 AND Implementation using NAND logic gates

Manual Proof

In order to prove that *IMPL* correctly implements the specification *AND*, we have to prove that if inp_1, inp_2 and *out* satisfy the constraints imposed by *IMPL*, then they must also satisfy the constraints imposed by *AND*. The notion of correctness is formalised by

a logical implication as illustrated by the following sentence.

$$\forall inp_1,\ inp_2,\ out.\ IMPL(inp_1,\ inp_2,\ out) \Rightarrow AND(inp_1,\ inp_2,\ out)$$

Proof: The proof starts by assuming that:

$$IMPL(inp_1,\ inp_2,\ out)$$

By expanding the definition of *IMPL* we have:

$$\exists out_1.\ NAND(inp_1,\ inp_2,\ out_1) \wedge NAND(out_1,\ out_1,\ out)$$

By expanding the definition of *NAND* we have:

$$\exists out_1.\ (\forall t.\ out_1\ t = \neg((inp_1\ t) \wedge\ (inp_2\ t))) \wedge (\forall t.\ out\ t = \neg((out_1\ t) \wedge (out_1\ t)))$$

Moving all the equations under the scope of a single $\forall$ quantifier[1] gives:

$$\exists out_1.\ \forall t.\ (out_1\ t = \neg((inp_1\ t) \wedge\ (inp_2\ t))) \wedge (out\ t = \neg((out_1\ t) \wedge (out_1\ t)))$$

Simplifying[2] the term $(out\ t = \neg((out_1\ t) \wedge (out_1\ t)))$ to $(out\ t = \neg(out_1\ t))$ gives:

$$\exists out_1.\ \forall t.\ (out_1\ t = \neg((inp_1\ t) \wedge\ (inp_2\ t))) \wedge (out\ t = \neg(out_1\ t))$$

Rewriting the term $(out\ t = \neg(out_1\ t))$ using the right-hand side of equation $(out_1\ t = ...)$ gives:

$$\exists out_1.\ \forall t.\ (out_1\ t = \neg((inp_1\ t) \wedge\ (inp_2\ t))) \wedge (out\ t = \neg(\neg((inp_1\ t) \wedge\ (inp_2\ t))))$$

Simplifying $(out\ t = ...)$ using double negation[3], the sentence becomes

$$\exists out_1.\ \forall t.\ (out_1\ t = \neg((inp_1\ t) \wedge\ (inp_2\ t))) \wedge (out\ t = (inp_1\ t) \wedge\ (inp_2\ t))$$

[1] Distributivity of $\forall$: $(\forall x.\ P \wedge Q) = (\forall x.\ P) \wedge (\forall x.\ Q)$
[2] Idempotence of conjunction: $P \wedge P = P$
[3] Double negation: $\neg\neg P = P$

Weakening the sentence using the commutativity of quantifiers[4] infers:

$$\forall t.\ \exists out_1.\ (out_1\ t = \neg((inp_1\ t) \wedge\ (inp_2\ t))) \wedge (out\ t = (inp_1\ t) \wedge\ (inp_2\ t))$$

Eliminating the existential quantifier by the one-point rule[5] gives:

$$\forall t.\ out\ t = (inp_1\ t) \wedge\ (inp_2\ t)$$

Using the definition of *AND*, the term becomes

$$AND(out_1,\ out_2,\ out)$$

Based on the assumption in the first step, we conclude that

$$IMPL(inp_1,\ inp_2,\ out) \Rightarrow AND(out_1,\ out_2,\ out)$$

Generalising[6] the free variables gives:

$$\forall inp_1, inp_2, out.\ IMPL(inp_1,\ inp_2,\ out) \Rightarrow AND(out_1,\ out_2,\ out)$$

Q.E.D.

Mechanised Proof

The manual proof described before could be mechanised in the HOL4 system using proof commands provided by the theorem prover. To start this proof in HOL4, we first need to define the *goal*, i.e. the predicate that we intend to prove. In this example, the goal is:

$$\forall inp_1, inp_2, out.\ IMPL(inp_1,\ inp_2,\ out) \Rightarrow AND(out_1,\ out_2,\ out).$$

The second thing to do is to use proof *tactics* in order to manipulate the goal. Tactics are functions that perform transformations in the goal by applying set of inference rules and logical identities. Some tactics are proof algorithms that allow us to reduce the effort involved in proving simple conjectures. Other tactics just make few changes in the goal. The tactics provided by HOL4 are described in its Reference Manual [30]. The first tactic we apply to the goal is STRIP_TAC: it removes the outermost quantifier. As there are

[4] Commute quantifiers: $(\exists x.\ (\forall y.\ P(x,y))) \Rightarrow (\forall y.\ (\exists x.\ P(x,y)))$

[5] One-point rule: $(\exists x.(x = t) \wedge P) = P[t/x]$, provided that x is not free in t

[6] Generalisation rule: $(\vdash P(x)) \Rightarrow (\vdash \forall x.P(x))$

three variables universally quantified (which is equivalent to three universal quantifiers nested), this tactic needs be applied thrice. After this application, the goal becomes:

$$IMPL(inp_1,\ inp_2,\ out) \Rightarrow AND(out_1,\ out_2,\ out).$$

By following the same reasoning of the proof done by hand, we could use a rewriting tactic to expand the definitions of *IMPL* and *NAND*. A rewriting tactic that we could use to do this is "PURE_REWRITE_TAC [...]". It receives a list of theorems and rewrites the goal by applying these theorems. In HOL4, the definitions constitute theorems and each theorem requires a name. Assuming that we have saved the definitions of *IMPL* and *NAND* as IMPL_def and NAND_def, respectively, then the tactic PURE_REWRITE_TAC[IMPL_def] followed by PURE_REWRITE_TAC[NAND_def] produces:

$$\begin{aligned}\exists out_1.\ (\forall t.\ out_1\ t = \neg((inp_1\ t) \wedge\ (inp_2\ t))) \wedge \\ (\forall t.\ out\ t = \neg((out_1\ t) \wedge (out_1\ t))) \Rightarrow \\ AND(out_1,\ out_2,\ out).\end{aligned}$$

The following step is to apply STRIP_TAC again. As the predicate already suggests a witness for the existential quantifier, this tactic removes the existential quantifier based on the one-point rule[7] and transforms the goal to

$$\begin{aligned}(\forall t.\ out_1\ t = \neg((inp_1\ t) \wedge\ (inp_2\ t))) \wedge \\ (\forall t.\ out\ t = \neg((out_1\ t) \wedge (out_1\ t))) \Rightarrow \\ AND(out_1,\ out_2,\ out).\end{aligned}$$

Next, we need to expand the definition of *AND* using PURE_REWRITE_TAC [AND_def] (assuming that AND_def is the name of the AND definition). Finally, we use the simplification tactic "ASM_SIMP_TAC bool_ss []". The parameter "bool_ss" is the name of a simplification set, i.e. a set of logical identities used by this tactic to perform transformations on the goal, and the second parameter (the empty list []) means that no additional theorems is provided for this simplification. At the end of proof, the HOL4 system presents the message "Initial goal proved".

For a complete and detailed description of HOL4 see Gordon and Melham [15] and the manuals of the HOL4 system [28, 29, 30, 31]. Along this dissertation we omit details of proofs. However, the complete specification with proof scripts is presented in Appendices A and B.

[7] One-point rule: $(\exists x.(x = t) \wedge P) = P[t/x]$, provided that x is not free in t

2.4 Fault Tolerant Patterns

Developing a fault tolerant system is an exercise in exploiting and managing *redundancy*: the property of having more of a resource than is minimally necessary to perform the job [25]. Fault tolerance aims to increase the reliability of a system (i.e. continuity of correct service), which is achieved by maintaining some form of correctness despite the presence of faults.

Fault tolerance addresses different classes of failures like systematic failures and random failures depending on the type of redundancy applied. A systematic failure is a flaw on the design of the system (i.e. a bug, in the software engineering terminology). A random failure is a failure caused by wear, deterioration, fatigue, etc. Mechanical and electronic hardware exhibit both kinds of faults, while software only exhibits systematic faults [11].

Redundancy comes in different flavours. We can duplicate a system, triplicate it, decide which replica is the “winner" (or is correct) by voting, install the replicas on top of a reliable architecture, duplicate a system by another that is slightly different from the original one (it may implement a different algorithm or use a different technology), and so on. There are plenty of such solutions that we refer to here as *fault tolerant patterns*. These patterns are abstract representations of how to manage redundancy in order to maintain the system operational when failures occur.

The original system which is the target of the replication is called *channel*. A channel is an end-to-end system that goes all the way from acquisition of relevant data (the input) to the generation of the output based on that data [11]. So, “channel” in this dissertation means any “system” and has no relation to the traditional usage of the word “channel” in the concurrent systems domain. Every subcomponent of a system can be regarded as a channel or the entire system can also be regarded as a channel. The choice of which components of a system should be replicated depends on the safety and the reliability analysis. A system is said safe if it is free of catastrophic consequences on the user(s) and the environment. The fault tolerant patterns discussed in this work act on channels, which are seen as black-boxes.

The following sections describe three classic fault tolerant patterns.

2.4.1 Homogeneous Redundancy

The homogeneous redundancy pattern (HR) is possibly the simplest existing fault tolerant pattern. The original channel is simply duplicated in order to improve reliability. The

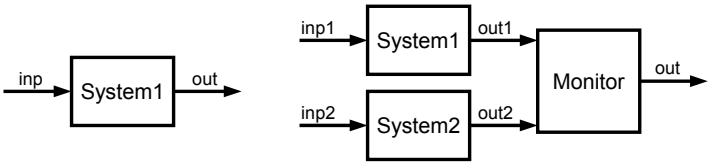


Figure 2.7 Homogeneous/Heterogeneous Redundancy

replicated channels operate in parallel, take input data from different sources, and produce outputs at the same time. One of the replicated channels is set as the *primary channel* (the main system) and the other, the *secondary channel* (the backup). The channels are managed using a monitor that implements an automatic *switch-to-backup* policy in case an error occurs in the primary channel. This policy specifies that, when the primary channel fails, the backup channel is used. The outputs of the channels are analysed by the monitor in order to detect failures in the channels.

The term *homogeneous* means that the replicas are exactly the same (same implementation, same technology, etc.). Note that as the replicas are exact copies of each other, this type of redundancy is still valid because it catches random failures. This pattern, however, does not address systematic failures, as the bugs are replicated. Figure 2.7 depicts a channel and the result of applying the HR to the system. In case of the HR, System2 is a copy of System1.

This pattern has several advantages when compared to other patterns. It is simple and easy to design and provide a good isolation from faults in contexts where random faults occur at a significant higher rate than systematic faults. The cost of applying this pattern comes from the additional secondary channel and the monitor.

2.4.2 Heterogeneous Redundancy

The heterogeneous redundancy pattern (HetR) improves reliability by offering channels with *dissimilar* design or implementation, i.e. different design or implementation for systems that do the same thing. Dissimilar channels are particularly useful to reduce the chances of replicating two systems with the same systematic failures (bugs). The HetR pattern operates like the homogeneous redundancy pattern: it uses a monitor that implements a switch-to-backup policy. The same Figure 2.7 used to depict HR can be used to depict the heterogeneous redundancy pattern as their architectures are the same. Nevertheless, differently from HR, here System2 is a dissimilar version of System1.

Actually, this pattern is a generalisation of HR as it does not impose the replicas be exactly the same. Besides addressing systematic faults, this pattern also addresses random

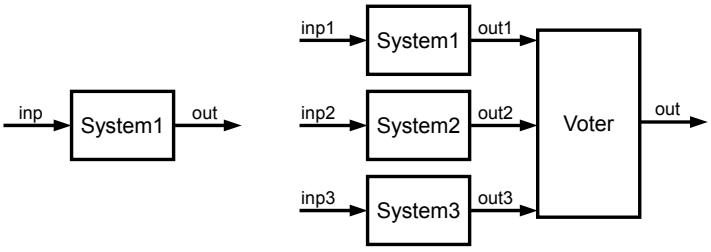


Figure 2.8 Triple Modular Redundancy

faults. This is a more expensive kind of redundancy because, not only has the replication cost increased, the development cost increases as well due to the effort required to develop dissimilar technologies [11].

2.4.3 Triple Modular Redundancy

The triple modular redundancy (TMR) is a variation of the homogeneous redundancy that consists of three identical channels that operate in parallel, and a voter that compares and averages the outputs of the channels. As in *HR*, only random faults can be addressed by this pattern, but it differs from *HR* by allowing the system to provide a valid output in the presence of up to two simultaneous random failures.

The voter (see Figure 2.8) plays a main role in this pattern by applying a voting policy that takes into account the majority of the valid outputs from the replicas [2]. Although the replicas are identical copies of each other, their outputs can diverge slightly due to divergences in the input signal of each replica, which comes from different sensors. The voter analyses the valid outputs and averages them to produce a more reliable result. The invalid outputs are not taken into account in the computation of the average.

2.5 Concluding Remarks

This chapter introduced the HOL logic and one of its mechanisation, the HOL4 system. We showed how to model circuits in HOL and how to prove the correctness of circuits. The same principles used to model hardware inspired us to develop a formal model for the fault tolerant patterns HR, HetR, and TMR.

We follow the convention of representing time as natural numbers (*num*). This formalisation is adopted in the various references to hardware specification in HOL [14, 18, 21]. The usage of natural numbers means that the hardware has discrete behaviour and that the same operation could be performed by software. However, we could have

also represented the time as real (*real*).

Although we have described just three fault tolerant patterns, there are many more replication patterns documented in the literature (for a catalogue, see Armoush's thesis [2]). Besides this, each pattern is usually customised before being applied. We do not intend to cover all patterns in this work but just a few that serve as a proof of concept of our model in HOL.

In the next chapter, we present a model of a generic system in HOL that computes a certain function, with a specific delay and that can present random failures. The model intends to represent any hardware that can be replicated and submitted to a replication pattern. We also present the HOL model for the HR, the HetR and the TMR patterns.

3

The HOL4 Model

This chapter presents the HOL4 specification of three fault tolerant patterns, namely Homogeneous Redundancy (HR), Heterogeneous Redundancy (HetR) and Triple Modular Redundancy (TMR). These patterns were chosen as a proof of concept of our approach, since they are classical patterns. This chapter is organised in seven sections. Section 3.1 describes how to model both the nominal and the faulty behaviour of a hardware. Section 3.2 presents the formal model of a system. Section 3.3 introduces components that make up the implementation of the patterns and presents an implementation of a system. Sections 3.4, 3.5 and 3.6 formalise the fault tolerant patterns HR, HetR and TMR respectively, and Section 3.7 briefly discusses the effort involved in the proof, and concludes.

3.1 Modelling Hardware Failures

Traditional hardware specifications commonly specify only its nominal behaviour, i.e. the behaviour in absence of failures (for instance, see the hardware specifications in Chapter 2). However, in this dissertation we are interested in modelling the possibility of failures in the hardware caused by deterioration, fatigue, wear, etc. These failures are not foreseeable. In fact, they are random failures. To model this feature, we added new concepts to the hardware model that are discussed next.

The first notion that needs to be well defined is the notion of failure. We define failures as random perturbations on a signal that destroy the information being carried. We assume that a failure at time t does not necessarily implies a failure at subsequent times. Failures in a hardware component can occur due to two causes: (i) input signals come with failures from the environment; or (ii) the hardware itself suffers a random failure (or both). We assume that a failure at time t cannot be fixed (or recovered) to

transform a bad signal into a good signal. However, the signal can "recover" at future times, but once a damage happens at time t, no information can be extracted from this particular data. This assumption is suitable for the patterns homogeneous redundancy, heterogeneous redundancy and triple modular redundancy, but excludes the possibility of modelling patterns that use recovery algorithms in a signal to rebuild the original signal without failure. The consequence of this assumption to our work is that our data and our failure information are modelled as a unique signal. When a failure occurs, the data itself is not present. It simplifies the representation of signals and still is generic enough to model several fault tolerant patterns. In order to deal with patterns that recovery the original signal, we need to drop this assumption and separate data and failure information.

The HOL4 system conveniently provides the *option* type operator that 'lifts' a type α to a new type α *option* containing all values of α wrapped by the constructor *SOME* plus a special value called *NONE*. For example, natural numbers with failure information can be modelled as *num option*. Thus values of the type *num option* are *NONE*, $SOME(0)$, $SOME(1)$, etc. To extract the value wrapped by the constructor *SOME*, the operator *THE* is used, e.g. $THE(SOME(13)) = 13$. HOL4 also provides the boolean functions $IS\_SOME()$ and $IS\_NONE()$ that test whether its argument is a proper value or not. For instance, $IS\_NONE(NONE) = \mathsf{T}$ and $IS\_SOME(SOME(3)) = \mathsf{T}$. Whenever $IS\_SOME()$ is true, $IS\_NONE()$ is false, and vice-versa.

We use the *option* type operator to model the data carried by signals. Values wrapped by the constructor *SOME* have no failure, and the value *NONE* represents a value with failure. For instance, two boolean signals could be modelled as a composite signal $inp : num \rightarrow (bool\ option \# bool\ option)\ option$. This model allows the first signal to break, the second signal to break (or both). In this case, the value *NONE* represents the simultaneous break of the component signals at a specific time, and the values $SOME(NONE, SOME(\mathsf{F}))$ and $SOME(SOME(\mathsf{T}), NONE)$ represent the break of the first and the second signals, respectively. Note that when the signal is *NONE*, the data itself is not present: the only information we know is that a failure has occurred.

Another important decision is how to model a random failure in a hardware component. We evaluated some approaches to model random failures. One of such approaches is to model hardware as devices that can fail at any moment in time. Assuming that the input signal has no failures, we could model an extended OR' component as described next.

$$\begin{aligned} UNSAFE\_OR'\ (inp,\ out) = \forall t.\ (out\ t = SOME((THE\ FST(THE\ (inp\ t))) \vee \\ (THE\ SND(THE\ (inp\ t)))) \vee \\ (out\ t = NONE) \end{aligned}$$

This specification states that the output at time t can be equal to *NONE* (the case a fault is injected) or equal to the pointwise application of the disjunction operator on the input signals (the case of the nominal behaviour). Note that, in order to operate on the components of the input signal we need to remove the constructor *SOME* twice: first the outermost *SOME* is taken, followed by the internal constructor which wraps the *bool* value. This specification style leaves the hardware behaviour too loose and allows any hardware specification to be implemented by a circuit that always produces corrupted values *NONE*.

Another approach to model random failures extends the hardware interface to accept an additional parameter: the failure model. The failure model is a boolean function that establishes, for every point in time, whether the hardware must present a random failure or not. Such a function deals with a random failure of the hardware itself, not the random failures of the input. For the input, the *option* type embeds this information on the input signal itself. As random failures are not foreseeable, we universally quantify the failure model in all theorems along this chapter. By using this approach and assuming that input signals come with no failure, the *UNSAFE_OR*′ could be specified as

$$\begin{aligned} &\mathit{UNSAFE\_OR}'\ (e : \mathit{num} \to \mathit{bool})\ (\mathit{inp},\ \mathit{out})\ = \\ &\qquad \forall t.\ \mathit{out}\ t = \mathit{if}\ e(t)\ \mathit{then}\ \mathit{NONE} \\ &\qquad\qquad\qquad \mathit{else}\ \mathit{SOME}((\mathit{THE}\ \mathit{FST}(\mathit{inp}\ t))\ \vee\ (\mathit{THE}\ \mathit{SND}(\mathit{inp}\ t))) \end{aligned}$$

This specification states that the output at time t is defined according to the failure model e. If the failure model decides that a failure must be introduced at time t, then the system outputs *NONE*. Otherwise the nominal behaviour is produced. In this dissertation we adopted this style of modelling as it allows us to predict the failure model of a composite system in terms of the failure model of its subsystems. It is important note all theorems presented later about the specification quantify universally the failure model. Then, no assumption about the failure model is made to prove the theorems.

In addition to failures introduced by the failure model, input signals can come with failure as well. When the input signal comes with a failure nothing can be done to recover the broken signal and the failure is carried over the system. In order to illustrate inputs with failure, we complement the *UNSAFE_OR*′ definition by removing the assumption

that inputs come with no failures.

$$
\begin{aligned}
&\mathit{UNSAFE\_OR''}\ e\ (\mathit{inp},\ \mathit{out})\ = \\
&\quad \forall t.\ \mathit{out}\ t = \mathit{if}\ (e(t) \vee \mathit{IS\_NONE}(\mathit{inp}\ t))\ \mathit{then}\ \mathit{NONE} \\
&\qquad\qquad \mathit{else\ if}\ \mathit{IS\_NONE}(\mathit{FST}(\mathit{THE}\ (\mathit{inp}\ t))) \vee \\
&\qquad\qquad\quad \mathit{IS\_NONE}(\mathit{SND}(\mathit{THE}\ (\mathit{inp}\ t)))\ \mathit{then}\ \mathit{NONE} \\
&\qquad\qquad \mathit{else}\ \mathit{SOME}((\mathit{THE}\ \mathit{FST}(\mathit{inp}\ t)) \vee (\mathit{THE}\ \mathit{SND}(\mathit{inp}\ t)))
\end{aligned}
$$

This specification states that the output at time t is *NONE* if any of these conditions occur: (i) the failure model define that hardware must break at time t; or (ii) the composite input signal is broken, i.e. $\mathit{IS\_NONE}(\mathit{inp}\ t)$ is true; or (ii) either the first or the second signals is broken: $\mathit{IS\_NONE}(\mathit{FST}(\mathit{THE}\ (\mathit{inp}\ t)))$ or $\mathit{IS\_NONE}(\mathit{FST}(\mathit{THE}\ (\mathit{inp}\ t)))$ is true. If none of these conditions occur, the output is the disjunction of the subcomponents of the input signal.

In the next section we discuss the specification of a generic system which is capable of computing a certain function, is subject to random failures (due both to the failures in the input signal and the failure model), and has an initialisation delay.

3.2 Specification

We specify a channel as a black-box called *SYSTEM* with inputs and outputs. Every *SYSTEM* computes a function f with a certain initialisation delay d. Occasionally, the *SYSTEM* may 'break' as a consequence of random failures. We model a random error of a *SYSTEM* as a function e from time to boolean. Whenever $e(t)$ is true, it means that the *SYSTEM* presents a random failure at time t. We call e a *failure model*.

Definition 3.1.

$$
\begin{aligned}
&\mathit{SYSTEM}\ d\ e\ f\ (\mathit{inp},\mathit{out})\ = \\
&\forall t.\ \mathit{out}\ (t+d) = \mathit{if}\ (\mathit{IS\_NONE}(\mathit{inp}\ t) \vee e(t))\ \mathit{then} \\
&\qquad\qquad \mathit{NONE} \\
&\qquad\qquad \mathit{else}\ \mathit{SOME}(f\ t\ (\mathit{THE}(\mathit{inp}\ t)))
\end{aligned}
$$

Signals are modelled as functions from time (natural numbers) to α *option*. Note that (i) the time is assumed be discrete and (ii) the data is of type α *option*. As time is discrete, our hardware describes an operation that could be performed through a software

component. The type α *option* models the fact that the input can be of any type. A signal whose value is *NONE* at time t means that the signal in time t comes with an error; and a signal whose value is $SOME(v)$ means that the signal has no errors and carries the value v at time t.

There are two conditions that make the output signal of a *SYSTEM* to be *NONE*. Either the input comes with an error ($IS\_NONE(inp\ t)$), in which case it is impossible to compute anything from *NONE*, or the *SYSTEM* itself breaks ($e(t)$). If none of these conditions happen, then *SYSTEM* outputs the result of the computation of f applied to $THE(inp\ t)$. The computation f takes as input the time t and input value $THE(inp\ t)$. The parameter t is required because some functions need to know the exact time of its application. For example, a function that computes a pseudo-random value based on a seed and the time of the application. More examples of functions that use the time are given below.

3.3 Implementation

This section introduces the components that are used to implement the *SYSTEM* specification and formalise the fault tolerant patterns. The components listed in this section are suitable for the patterns: homogeneous redundancy, heterogeneous redundancy, and triple modular redundancy. (Certainly it will be necessary to define new components in order to formalise patterns not covered in this work.) At the end of this section we provide an implementation of *SYSTEM* that uses only the components defined here and prove the correctness of this implementation.

DEL Component

A *DEL* (Figure 3.1) is a *polymorphic* delay component that introduces a delay d to an input signal. It is polymorphic because the input signal (*inp*) can have any type. The delay refers to the initialisation time of the system.

Definition 3.2.

$$DEL\ d\ (inp : num \to \alpha\ option, out) = \forall t.\ out\ (t+d) = (inp\ t)$$

The output *out* at time $t+d$ is equal to the value of *inp* at time t. Notice that nothing it said about *out* at the first d time units, which means that *DEL* is a partial specification.

Figure 3.1 Polymorphic delay component

This is the only component in our specification that introduces a delay. The notion of delay in a computation is formalised by the sequential composition of the *DEL* component with other components. We discuss the sequential composition of components later.

ERROR Component

An *ERROR* component (Figure 3.2) introduces a random error in a signal at time t based on the failure model e.

Definition 3.3.

$$ERROR\ e\ (inp, out)\ = \forall t.\ out\ t = if\ e(t)\ then\ NONE\ else\ (inp\ t)$$

Figure 3.2 Error component

The *ERROR* component outputs *NONE* whenever $e(t)$ decides that an error should be introduced at time t.

COMB Component

The combinatorial component *COMB* (Figure 3.3) applies a function f to the input at each instant in time.

Definition 3.4.

$$\begin{aligned} COMB\ f\ (inp, out)\ = \forall t.\ out\ t = if\ IS\_NONE(inp\ t)\ then \\ NONE \\ else\ SOME(f\ t\ (THE(inp\ t))) \end{aligned}$$

If the input signal at time t comes with an error (i.e. the input is *NONE*), then the broken signal is propagated, otherwise f is applied to the input.

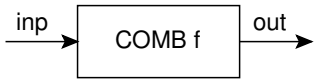


Figure 3.3 Combinatorial component

BUS Component

The *BUS* encapsulates two signals *inp1* and *inp2* into one signal.

Definition 3.5.

$$\begin{aligned} &BUS\ (inp1, inp2, out) = \\ &\forall t.\ out\ t = \mathit{if}\ IS\_NONE(inp1\ t)\ \wedge\ IS\_NONE(inp2\ t) \\ &\qquad\qquad \mathit{then}\ NONE \\ &\qquad\quad \mathit{else}\ SOME(inp1\ t, inp2\ t) \end{aligned}$$

If, at time t, both signals *inp1* and *inp2* are *NONE*, then *BUS* outputs *NONE*. Otherwise it outputs $SOME(inp1\ t, inp2\ t)$. The output is a pair of options whose type is $(\alpha\ option \times\ \beta\ option)\ option$. Note that the *BUS* can output *NONE*, $SOME(NONE,\ SOME(v))$, $SOME(SOME(v),\ NONE)$, or $SOME(SOME(v1), SOME(v2))$. But it never outputs $SOME(NONE, NONE)$ in order to avoid a component that takes it as input to regard it as a valid data.

TBUS Component

TBUS is an extension of *BUS* for three input signals.

Definition 3.6.

$$\begin{aligned} &TBUS\ (inp1, inp2, inp3, out) = \\ &\quad \forall t.\ out\ t = \mathit{if}\ IS\_NONE(inp1\ t) \wedge IS\_NONE(inp2\ t) \wedge \\ &\qquad\qquad IS\_NONE(inp3\ t)\ \mathit{then}\ NONE \\ &\qquad\quad \mathit{else}\ SOME(inp1\ t, inp2\ t, inp3\ t) \end{aligned}$$

MUX Component

The *MUX* component separates a combined signal of type $(\alpha\ option \times \beta\ option)\ option$ into two signals. When $inp\ t$ is *NONE*, it outputs (*NONE*, *NONE*). Otherwise, the input at time t has the form $SOME(x, y)$ and the outputs are $out1\ t = x$ and $out2\ t = y$.

Definition 3.7.

$$
\begin{aligned}
&MUX(inp,(out1,out2)) = \\
&\quad \forall t.\ (out1\ t, out2\ t) = \\
&\qquad\qquad (\mathit{if}\ IS\_NONE(inp\ t)\ \mathit{then}\ NONE \\
&\qquad\qquad\ \mathit{else}\ FST(THE(inp\ t)), \\
&\qquad\qquad\ \mathit{if}\ IS\_NONE(inp\ t)\ \mathit{then}\ NONE \\
&\qquad\qquad\ \mathit{else}\ SND(THE(inp\ t)))
\end{aligned}
$$

The *MUX* component undoes what the *BUS* component does. The *BUS* takes as input two signals and outputs one signal (made of a pair of values), while the *MUX* takes one signal made of a pair of values and outputs two signals.

TMUX Component

TMUX is an extension of *MUX* for three output signals.

Definition 3.8.

$$
\begin{aligned}
&TMUX(inp,(out1,out2,out3)) = \\
&\forall t.\ (out1\ t, out2\ t, out3\ t) = \\
&\quad (\mathit{if}\ IS\_NONE(inp\ t)\ \vee\ IS\_NONE(FST(THE(inp\ t)))\ \mathit{then}\ NONE \\
&\quad\ \mathit{else}\ (FST(THE(inp\ t))), \\
&\quad\ \mathit{if}\ IS\_NONE(inp\ t)\ \vee IS\_NONE(FST(SND(THE(inp\ t))))\ \mathit{then}\ NONE \\
&\quad\ \mathit{else}\ (FST(SND(THE(inp\ t)))), \\
&\quad\ \mathit{if}\ IS\_NONE(inp\ t)\ \vee\ IS\_NONE(SND(SND(THE(inp\ t))))\mathit{then}\ NONE \\
&\quad\ \mathit{else}\ (SND(SND(THE(inp\ t))))
\end{aligned}
$$

SYSTEM Implementation and Correctness

Now we illustrate how we can compose the components shown before to build a more elaborate system. A *BLOCK* implements a SYSTEM (Definition 3.1) that has a certain delay d, may break according to the failure model e, and computes f. A *BLOCK* is constructed as a sequential composition of *ERROR*, *COMB* and *DEL* (see Figure 3.4).

Definition 3.9.

$$
\begin{aligned}
&BLOCK\ d\ e\ f\ (inp\,out) = \\
&\quad \exists out1\ out2.\ ERROR\ e\ (inp, out1) \wedge \\
&\qquad\qquad COMB\ f\ (out1, out2) \wedge \\
&\qquad\qquad DEL\ d\ (out2, out)
\end{aligned}
$$

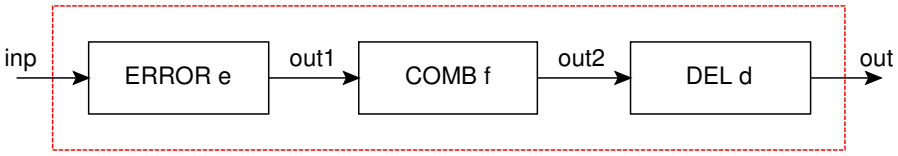


Figure 3.4 The diagram of *BLOCK*.

Internal signals *out1* and *out2* are existentially quantified. This quantification hides from the user of *BLOCK* the values of these signals, exposing only its external interface. Signals with the same name connect two components. For instance, *out1* is the output of *ERROR* and the input of *COMB*.

Theorem 3.10 presented next shows that a $(BLOCK\ d\ e\ f\ (inp, out))$ implements a $(SYSTEM\ d\ e\ f\ (inp, out))$. This theorem states that our definition of *SYSTEM* is implementable. This is not the only way of implementing a *SYSTEM*: in what follows, we prove that if we plug implementations of *SYSTEM*s into the fault tolerant patterns, the resulting circuit is still an implementation of a *SYSTEM*.

Theorem 3.10.

$$\vdash \forall d\ e\ f\ inp\ out.\ BLOCK\ d\ e\ f\ (inp, out) \Rightarrow SYSTEM\ d\ e\ f\ (inp, out)$$

The proof of this theorem in HOL4 is semi-automatic. It is done by expanding all definitions, then eliminating the existential and universal quantifiers (see Camilleri *et al.* [7] for more details on how to do this in HOL4). After that, the proof is subdivided into cases and simplification tactics are conveniently applied to finish the proof. All theorems in this work have been proved mechanically in HOL4. We omit further information on the proofs from now on. The complete proof script of this dissertation is in Appendices A and B.

In what follows we describe how we build fault tolerant patterns using the components described above.

3.4 Homogeneous Redundancy

A homogeneous redundant system *HR* (Figure 3.5) connects two identical systems *S1* and *S2* to a *BLOCK* with delay *dm*[1], failure model *em*[2], and functionality equals to the

[1]This name is an abbreviation for *delay monitor*
[2]This name is an abbreviation for *error monitor*

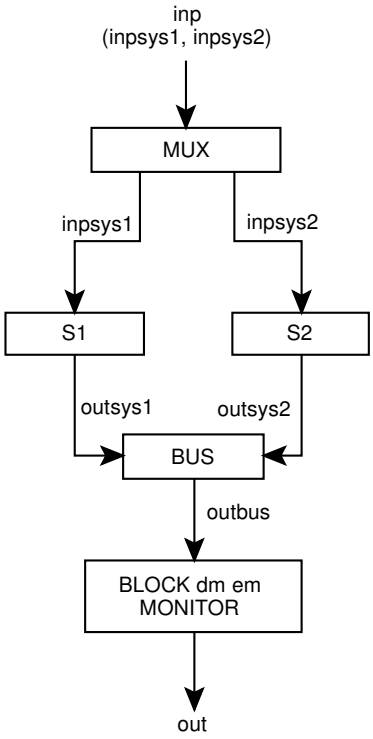


Figure 3.5 Homogeneous and Heterogeneous Redundancy - HOL4 Model.

function *MONITOR* (described next). The input *inp* is split by *MUX* into two signals: *inpsys1* and *inpsys2*. These signals are the inputs of *S1* and *S2*, respectively. The output of both *S1* and *S2* is combined by *BUS* and sent to a monitor *BLOCK*, which decides which system is used. The definition in HOL4 of Figure 3.5 is given next.

Definition 3.11.

$$
\begin{aligned}
&HR\ dm\ em\ S1\ S2\ (inp : num \rightarrow (\alpha\ option \times \alpha\ option)\ option, out) \ = \\
&\quad \exists inpsys1\ inpsys2\ outsys1\ outsys2\ outbus. \\
&\qquad MUX\ (inp, (inpsys1, inpsys2)) \wedge \\
&\qquad S1\ (inpsys1, outsys1) \wedge \\
&\qquad S2\ (inpsys2, outsys2) \wedge \\
&\qquad BUS\ ((outsys1, outsys2), outbus) \wedge \\
&\qquad BLOCK\ dm\ em\ MONITOR\ (outbus, out)
\end{aligned}
$$

Note that the definition of *HR* does not force *S1* and *S2* to be duplicates of the same system. This will be done in the correctness theorem of *HR* in which we instantiate *S1* and *S2*. Another important thing to note is that the inputs are not necessarily the same.

In fact, it is common that different replicas acquire input data from different sensors in order to increase the fault tolerance. We do not restrict the inputs *inpsys1* and *inpsys2* to be replicas. It means that our formalisation also admits applying HR to two *SYSTEM*s whose inputs are different.

The function *MONITOR* takes as input the current time and a pair of type α *option* $\times$ α *option* and decides which output to choose: the one from *S1* or the one from *S2*. Note that the parameter t (time) is not used by *MONITOR*. It is present in this definition because in Section 3.2 we define that any function computed by a *SYSTEM* receives at least two parameters: the time of its application and the data carried by the signal at this time. The *MONITOR* is a function computed by a *BLOCK* that implements a *SYSTEM*, thus the interface of the function computed by a *SYSTEM* must be satisfied.

Definition 3.12.

$$\begin{aligned} MONITOR\ t\ inp = if\ (IS\_SOME(FST(inp)))\ then \\ THE(FST(inp)) \\ else\ THE(SND(inp)) \end{aligned}$$

If the output from *S1* is valid ($IS\_SOME(FST(inp))$) then *MONITOR* returns the first element. Otherwise, it returns the second element. Note that *MONITOR* is not a component. It instantiates the function f of a *BLOCK*.

The correctness theorem for *HR* is shown next. We assume that *I1* and *I2* are two implementations of a *SYSTEM* that has delay d and computes the function f. The implementations *I1* and *I2* differ only in their failure models *e1* and *e2*, i.e. they do not necessarily synchronise on their random failures.[3] Given these two implementations of a *SYSTEM* that compute f, the theorem states that if we plug them into the *HR*, the resulting system also implements a *SYSTEM*. Moreover, such *SYSTEM* has a delay $d + dm$ (sum of the delays of the subsystems and the monitor), a failure model ($E\ e1\ e2\ em\ d\ inp$) and computes the function ($FHR\ f\ e1$). The definitions of E and *FHR* are given next.

[3]This assumption is in accordance to real duplicate systems: they do the same thing and they have the same fail *rates*, but they do not necessarily fail together. That is the whole point of duplicating them for fault tolerance.

Theorem 3.13.

$$
\begin{aligned}
&\vdash \forall I1\ I2\ d\ e1\ e2\ f\ dm\ em\ inp\ out.\\
&\quad (\forall inp\ out.\ I1(inp,out) \Rightarrow SYSTEM\ d\ e1\ f\ (inp,out)) \wedge\\
&\quad (\forall inp\ out.\ I2(inp,out) \Rightarrow SYSTEM\ d\ e2\ f\ (inp,out))\\
&\qquad \Rightarrow (HR\ dm\ em\ I1\ I2\ (inp,out)\\
&\qquad\quad \Rightarrow SYSTEM\ (d+dm)\ (E\ e1\ e2\ em\ d\ inp)\ (FHR\ f\ e1)\ (inp,out))
\end{aligned}
$$

The failure model of an *HR* that comprises two channels *I1* and *I2* with failure models *e1* and *e2*, respectively, is given next.

Definition 3.14.

$$
\begin{aligned}
&E\ e1\ e2\ em\ d\ inp\ t =\\
&\qquad em(t+d) \vee\\
&\qquad (e1(t) \wedge e2(t)) \vee\\
&\qquad (e1(t)\ \wedge\ IS\_NONE(SND(THE(inp\ t)))) \vee\\
&\qquad (e2(t)\ \wedge\ IS\_NONE(FST(THE(inp\ t)))) \vee\\
&\qquad (IS\_NONE(SND(THE(inp\ t)))\ \wedge\ IS\_NONE(FST(THE(inp\ t))))
\end{aligned}
$$

The *HR* can fail under five different conditions: (i) the monitor block fails; or (ii) both duplicated channels *I1* and *I2* fail simultaneously; or (iii) *I1* fails and the input for *S2* is *NONE*; or (iv) *I2* fails and the input for *I1* is *NONE*; or (v) the input for both *I1* and *I2* are *NONE* simultaneously. This failure model was discovered during the process of proof of the Theorem 3.13.

The *HR* functionality is described by the function $(FHR\ f\ e1)$, where f is the functionality of both *I1* and *I2*, *e1* is the failure model of primary channel *I1* and t is the current time.

Definition 3.15.

$$
\begin{aligned}
&FHR\ f\ e1\ t\ inp =\\
&\qquad if\ IS\_SOME(FST(inp)) \wedge \neg e1(t)\\
&\qquad then\ (f\ t\ (THE(FST(inp))))\\
&\qquad else\ (f\ t\ (THE(SND(inp))))
\end{aligned}
$$

If the input for *I1* is valid ($IS\_SOME(FST(inp))$) and *I1* itself does not break at time t ($\neg e1(t)$), then *HR* behaves like *I1* (it applies f to the input of *I1*). Otherwise, it behaves

like *I2* by applying f to the input of *I2*. As we need to check if there is no failure of *I1* at time t, we take both the time and the failure model *e1* as argument. Notice that, in both cases, the same function f is applied in both branches of the *if-then-else*. This suggests informally that *HR* preserves the behaviour of its subsystems *I1* and *I2*. The formalisation of behavioural preservation is presented in Chapter 4. Similar to the case of the failure model, the functionality of the HR was also discovered during the process of proof of the Theorem 3.13. In general, the failure model and the functionality of the patterns were not known *a priori*, they were discovered during the process of proof of the respective theorems.

Theorem 3.13 is intuitively depicted in Figure 3.6. It shows that if *I1* and *I2* implement *SYSTEM*s that compute f, then an *HR* built from them implements a *SYSTEM* that computes f applied to the input of *I1* or the input of *I2*. Note that this theorem is compositional: if we use two implementations I_1 and I_2 that are *SYSTEMs*, then an *HR* built from them also implements a *SYSTEM*.

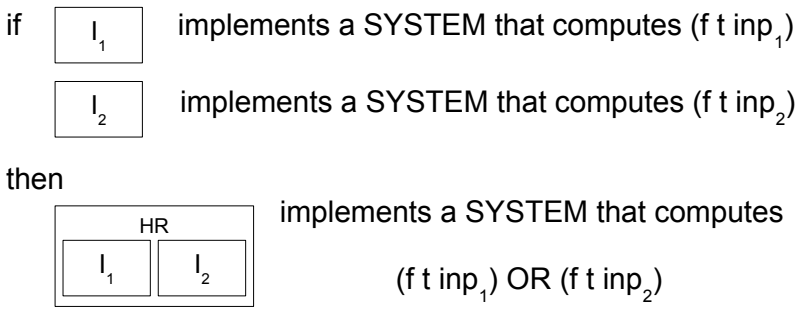


Figure 3.6 Theorem 3.13 intuitively.

3.5 Heterogeneous Redundancy

The Heterogeneous Redundant pattern *HetR* specification is quite similar to *HR*, except that the inputs have different types as the replicated systems have different implementations. Their outputs, however, have the same types as they must compute the same thing. As in *HR*, the *HetR* connects two systems *S1* and *S2* to a *BLOCK* with delay *dm*, failure model *em* and that implements the function *MONITOR*. The monitor function is the same of *HR*.

Definition 3.16.

$$\begin{aligned}
&\mathit{HetR\ dm\ em\ S1\ S2}\ (\mathit{inp} : \mathit{num} \to (\alpha\ \mathit{option} \times \beta\ \mathit{option})\ \mathit{option}, \mathit{out}) = \\
&\quad \exists \mathit{inpsys1\ inpsys2\ outsys1\ outsys2\ outbus} \\
&\quad\quad \mathit{MUX}\ (\mathit{inp}, (\mathit{inpsys1}, \mathit{inpsys2})) \wedge \\
&\quad\quad \mathit{S1}\ (\mathit{inpsys1}, \mathit{outsys1}) \wedge \\
&\quad\quad \mathit{S2}\ (\mathit{inpsys2}, \mathit{outsys2}) \wedge \\
&\quad\quad \mathit{BUS}\ ((\mathit{outsys1}, \mathit{outsys2}), \mathit{outbus}) \wedge \\
&\quad\quad \mathit{BLOCK\ dm\ em\ MONITOR}\ (\mathit{outbus}, \mathit{out})
\end{aligned}$$

Note that the input is now of type $\mathit{num} \to (\alpha\ \mathit{option} \times \beta\ \mathit{option})\ \mathit{option}$. For *HR*, the input is of type $\mathit{num} \to (\alpha\ \mathit{option} \times \alpha\ \mathit{option})$. This is what captures heterogeneity in our model. As the difference between homogeneous and heterogeneous depends only on the type of the input, their architectures are the same (Figure 3.5).

The correctness theorem is also very similar to the *HR*. If *I1* and *I2* are correct implementations of a *SYSTEM*, then an *HetR* system that contains *I1* and *I2* is also an implementation of a *SYSTEM*. Such *SYSTEM*, however, computes the function *FHetR* (shown next).

Theorem 3.17.

$$\begin{aligned}
&\vdash \forall \mathit{I1\ I2\ d\ e1\ e2\ f1\ f2\ dm\ em\ inp\ out}. \\
&\quad (\forall \mathit{inp\ out}.\mathit{I1}(\mathit{inp}, \mathit{out}) \Rightarrow \mathit{SYSTEM\ d\ e1\ f1}\ (\mathit{inp}, \mathit{out})) \wedge \\
&\quad (\forall \mathit{inp\ out}.\mathit{I2}(\mathit{inp}, \mathit{out}) \Rightarrow \mathit{SYSTEM\ d\ e2\ f2}\ (\mathit{inp}, \mathit{out})) \\
&\quad\quad \Rightarrow (\mathit{HetR\ dm\ em\ I1\ I2}\ (\mathit{inp}, \mathit{out}) \\
&\quad\quad \Rightarrow \mathit{SYSTEM}\ (d + dm)\ (\mathit{E\ e1\ e2\ em\ d\ inp})\ (\mathit{FHetR\ f1\ f2\ e1})\ (\mathit{inp}, \mathit{out}))
\end{aligned}$$

The function *FHetR* chooses between the functionalities provided by the replicas. Note that we do not make assumptions about *f1* and *f2*. It means that if we build a system by combining two systems with different functionalities (but having the same output type) using *HetR*, the functionality of this system is also described by *FHetR*.

Definition 3.18.

$$\begin{aligned}
&\mathit{FHetR\ f1\ f2\ e1\ t}\ (\mathit{inp} : (\alpha\ \mathit{option} \times \beta\ \mathit{option})) = \\
&\quad \mathit{if\ IS_SOME}(\mathit{FST}(\mathit{inp})) \wedge \neg \mathit{e1}(t) \\
&\quad \mathit{then}\ (\mathit{f1\ t\ THE}(\mathit{FST}(\mathit{inp}))) \\
&\quad \mathit{else}\ (\mathit{f2\ t\ THE}(\mathit{SND}(\mathit{inp})))
\end{aligned}$$

If the input for *I1* has no error and this system does not break at time t, then the function *f1* of *I1* is used. Otherwise, *f2* of *I2* is computed.

Theorem 3.17 is intuitively represented in Figure 3.7. As Theorem 3.13, this theorem is also compositional in the sense that if the parts I_1 and I_2 implement a *SYSTEM*, then an *HetR* built from them also implements a *SYSTEM*. The proof of this theorem reuses the effort spent in proof of the Theorem 3.13.

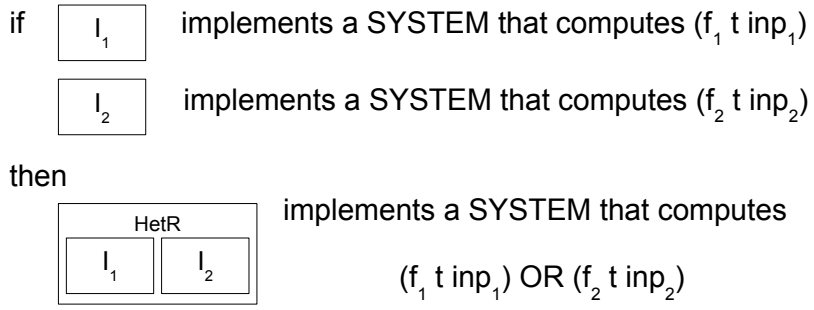


Figure 3.7 Theorem 3.17 intuitively.

3.6 Triple Modular Redundancy

A Triple Modular Redundant system increases the reliability of a channel by triplicating it and submitting the outputs to a voter, which produces a measure of central tendency like average, median or mode [38]. We assume that the channel's outputs can diverge slightly, i.e. two *SOME* outputs do not need to be exactly the same. This assumption reflects the fact that every channel receives its input from independent sources (typically, distinct sensors), and that these sources can produce slightly different values even using the same technology. This happens specially when the input sources produce values of type real.

The *VOTER* shown next is a function that averages the valid outputs from the channels in order to minimise deviations. We assume that the type of the function computed by the channels is $num \rightarrow \alpha \rightarrow real$ as *real* arithmetic is needed in order to compute the average. For simplicity, the input *inp* is subdivided in three components: *in1*, *in2* and *in3*. These components refer to the output of the three channel replicas. If all signals are valid, the *VOTER* outputs the arithmetic average of all them; in case of just one signal is invalid, this signal is disregarded and the average of the other two replicas is given as result. Finally, if just one signal is valid, this signal is the output itself.

Definition 3.19.

$$
\begin{array}{l}
VOTER\ t\ inp = \\
\quad let\ in1 = FST(inp)\ in \\
\quad let\ in2 = FST(SND(inp))\ in \\
\quad let\ in3 = SND(SND(inp))\ in \\
\qquad if\ IS\_SOME(in1)\ \wedge\ IS\_SOME(in2)\ \wedge\ IS\_SOME(in3)\ then \\
\qquad\quad (1/3) * (THE(in1) + THE(in2) + THE(in3)) \\
\qquad else\ if\ IS\_SOME(in1)\ \wedge\ IS\_SOME(in2)\ then \\
\qquad\quad (1/2) * (THE(in1) + THE(in2)) \\
\qquad else\ if\ IS\_SOME(in1)\ \wedge\ IS\_SOME(in3)\ then \\
\qquad\quad (1/2) * (THE(in1) + THE(in3)) \\
\qquad else\ if\ IS\_SOME(in2)\ \wedge\ IS\_SOME(in3)\ then \\
\qquad\quad (1/2) * (THE(in2) + THE(in3)) \\
\qquad else\ if\ IS\_SOME(in1)\ then\ THE(in1) \\
\qquad else\ if\ IS\_SOME(in2)\ then\ THE(in2) \\
\qquad else\ THE(in3)
\end{array}
$$

The *TMR* pattern connects three systems *S1*, *S2* and *S3* to a *BLOCK* with delay *dv*[4], failure model *ev*[5] and that implements the function *VOTER* (see Figure 3.8). Notice that *TBUS* and *TMUX* play the role of *BUS* and *MUX* used with *HR* and *HetR*. The HOL4 definition of a *TMR* is given next.

Definition 3.20.

$$
\begin{array}{l}
TMR\ dv\ ev\ S1\ S2\ S3\ (inp : num \to (\alpha\ option \times (\alpha\ option \times \alpha\ option))\ option, out) = \\
\qquad \exists inpsys1\ inpsys2\ inpsys3\ outsys1\ outsys2\ outsys3\ outbus. \\
\qquad\quad TMUX\ (inp, (inpsys1, inpsys2, inpsys3)) \wedge \\
\qquad\quad S1\ (inpsys1, outsys1) \wedge \\
\qquad\quad S2\ (inpsys2, outsys2) \wedge \\
\qquad\quad S3\ (inpsys3, outsys3) \wedge \\
\qquad\quad TBUS\ (outsys1, outsys2, outsys3, outbus) \wedge \\
\qquad\quad BLOCK\ dv\ ev\ VOTER\ (outbus, out)
\end{array}
$$

The correctness theorem states that if *I1*, *I2* and *I3* are correct implementations of a *SYSTEM*, then a *TMR* system that contains *I1*, *I2* and *I3* is also an implementation of a

[4]This name is an abbreviation for *delay voter*
[5]This name is an abbreviation for *error voter*

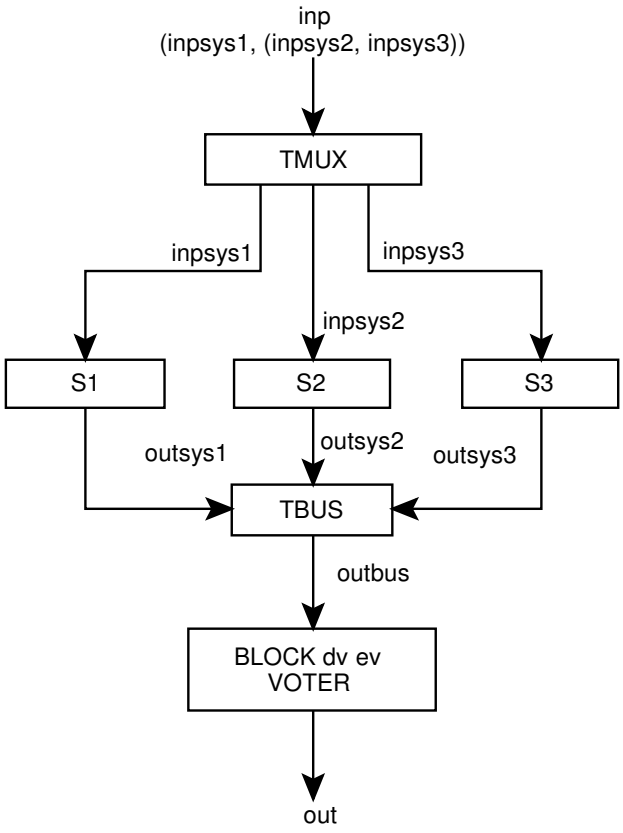


Figure 3.8 Triple Modular Redundancy - HOL4 Model.

SYSTEM. Such *SYSTEM* has a delay $d+dv$, a failure model (*ETMR e1 e2 e3 ev d inp*) and computes the function (*FTMR e1 e2 e3 f*). The definitions of *ETMR* and *FTMR* are explained next. These definitions were discovered during the process of proof.

Theorem 3.21.

$$
\begin{aligned}
&\vdash \forall I1\ I2\ I3\ dv\ ev\ d\ e1 e2\ e3\ f inp\ out.\\
&\quad (\forall inp\ out.I1(inp,out) \Rightarrow SYSTEM\ d\ e1\ f\ (inp,out)) \wedge\\
&\quad (\forall inp\ out.I2(inp,out) \Rightarrow SYSTEM\ d\ e2\ f\ (inp,out)) \wedge\\
&\quad (\forall inp\ out.I3(inp,out) \Rightarrow SYSTEM\ d\ e3\ f\ (inp,out))\\
&\qquad \Rightarrow (TMR\ dv\ ev\ I1\ I2\ I3\ (inp,out)\\
&\qquad\quad \Rightarrow SYSTEM\ (d+dv)\ (ETMR\ e1\ e2\ e3\ ev\ d\ inp)\\
&\qquad\qquad (FTMR\ e1\ e2\ e3\ f)\ (inp,out))
\end{aligned}
$$

The *TMR* can fail in one of these cases: either i) all channels present a random failure simultaneously; or ii) the voter presents a random failure; or iii) there is a combination of *NONE* inputs and channel failures that involves all channel replicas; or iv) all inputs are *NONE*.

Definition 3.22.

$$
\begin{aligned}
&ETMR\ e1\ e2\ e3\ ev\ dv\ inp\ t =\\
&\quad let\ in1 = FST(THE(inp))\ in\\
&\quad let\ in2 = FST(SND(THE(inp)))\ in\\
&\quad let\ in3 = SND(SND(THE(inp)))\ in\\
&\qquad (e1(t)\ \wedge\ e2(t)\ \wedge\ e3(t)) \vee\\
&\qquad (ev(t+dv)) \vee\\
&\qquad (IS\_NONE(in1)\ \wedge\ e2(t)\ \wedge\ e3(t)) \vee\\
&\qquad (IS\_NONE(in2)\ \wedge\ e1(t)\ \wedge\ e3(t)) \vee\\
&\qquad (IS\_NONE(in3)\ \wedge\ e1(t)\ \wedge\ e2(t)) \vee\\
&\qquad (IS\_NONE(in1)\ \wedge\ IS\_NONE(in2)\ \wedge\ e3(t)) \vee\\
&\qquad (IS\_NONE(in1)\ \wedge\ IS\_NONE(in3)\ \wedge\ e2(t)) \vee\\
&\qquad (IS\_NONE(in2)\ \wedge\ IS\_NONE(in3)\ \wedge\ e1(t)) \vee\\
&\qquad (IS\_NONE(in1)\ \wedge\ IS\_NONE(in2)\ \wedge\ IS\_NONE(in3))
\end{aligned}
$$

The function *FTMR* (shown below) takes 6 arguments: the failure model of each replica, the computing function of the replicas (which is the same function for all replicas), the current time, and the input. Based on the input and the failure models, *FTMR*

dismisses *NONE* outputs of each channel and outputs the average of the valid outputs. This function assumes that the input *inp* has at least one valid signal. This assumption is always satisfied as *inp* comes from *TMUX*, which never outputs the value $SOME(NONE,\ (NONE,\ NONE))$.

Definition 3.23.

$$
\begin{array}{l}
FTMR\ e1\ e2\ e3\ f\ t\ inp\ = \\
\quad let\ in1 = FST(inp)\ in \\
\quad let\ in2 = FST(SND(inp))\ in \\
\quad let\ in3 = SND(SND(inp))\ in \\
\quad\quad if\ IS\_SOME(in1)\ \wedge\ IS\_SOME(in2)\ \wedge\ IS\_SOME(in3)\ \wedge \\
\quad\quad\quad \neg e1(t)\ \wedge\ \neg e2(t)\ \wedge\ \neg e3(t) \\
\quad\quad\quad then\ (1/3) * (f\ t\ (THE\ in1) + f\ t\ (THE\ in2) + f\ t\ (THE\ in3)) \\
\quad\quad else\ if\ IS\_SOME(in1)\ \wedge\ IS\_SOME(in2)\ \wedge\ \neg e1(t)\ \wedge\ \neg e2(t) \\
\quad\quad\quad\quad then\ (1/2) * (f\ t\ (THE\ in1) + f\ t\ (THE\ in2)) \\
\quad\quad else\ if\ IS\_SOME(in1)\ \wedge\ IS\_SOME(in3)\ \wedge\ \neg e1(t)\ \wedge\ \neg e3(t) \\
\quad\quad\quad\quad then\ (1/2) * (f\ t\ (THE\ in1) + f\ t\ (THE\ in3)) \\
\quad\quad else\ if\ IS\_SOME(in2)\ \wedge\ IS\_SOME(in3)\ \wedge\ \neg e2(t)\ \wedge\ \neg e3(t) \\
\quad\quad\quad\quad then\ (1/2) * (f\ t\ (THE\ in2) + f\ t\ (THE\ in3)) \\
\quad\quad else\ if\ IS\_SOME(in1)\ \wedge\ \neg e1(t)\ then\ (f\ t\ (THE\ in1)) \\
\quad\quad else\ if\ IS\_SOME(in2)\ \wedge\ \neg e2(t)\ then\ (f\ t\ (THE\ in2)) \\
\quad\quad else\ (f\ t\ (THE\ in3))
\end{array}
$$

An intuitive version of Theorem 3.21 is shown in Figure 3.9. It states that if we build a *TMR* from three implementations of a *SYSTEM*, then the whole *TMR* is also a *SYSTEM* (compositionality). Each implementation I_1, I_2 and I_3 is used in the computation of the average (provided their outputs are valid). In this figure, $E_{I_j}(t) = 1$ means that the implementation I_j produces a valid output at time t and $E_{I_j}(t) = 0$ means that I_j failures at time t.

Differently from *HR* and *HetR*, it is hard to see that triple modular redundancy preserves the essence of behaviour of the channels. By looking at Definition 3.23, it is not so clear that *TMR* behaves like its subsystems. In Chapter 4 we define new notions of refinement in order to formalise a relation between each pattern and its subsystems.

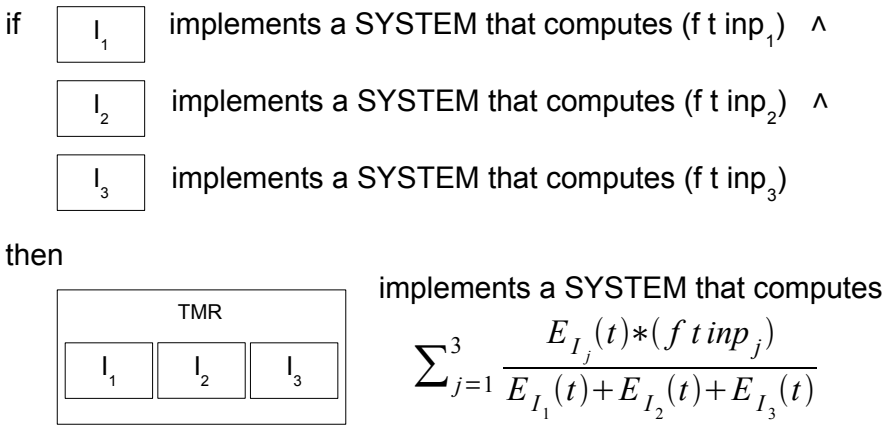


Figure 3.9 Theorem 3.21 intuitively.

3.7 Concluding Remarks

In this chapter we specified the nominal and faulty behaviour of a generic hardware *SYSTEM*, presented components that make up the implementation of a *SYSTEM*, and formalised three fault tolerant patterns: Homogeneous Redundancy (HR), Heterogeneous Redundancy (HetR) and Triple Modular Redundancy (TMR).

For each pattern, we proved a theorem that states that the pattern implements a *SYSTEM* with: (i) a specific delay (which depends on the delay of its subsystems and the delay of the redundancy manager); (ii) a specific failure model (which depends on the failure model of its subsystems and the failure model of the redundancy manager); and (iii) a functionality that depends on the functionality of its subsystems.

All theorems are compositional: if the fault tolerant patterns are built from subsystems that implement a *SYSTEM*, then they become *SYSTEMs* themselves. Compositionality allows us to compose patterns ad infinitum and prove (almost effortlessly) that the whole system is still a *SYSTEM*. For instance, it is easy to prove in HOL4 that if we apply *HR* to some *SYSTEMs*, followed by *HetR*, followed by *TMR*, followed by *HR* again, the result is still a *SYSTEM*. Actually, we can even mechanise such proofs completely in HOL4. Such mechanisation has not been done for this work yet, but is a potential future work.

Each fault tolerant pattern presented here behaves “similarly” to its subsystems. In Chapter 4 we present a refinement calculus that formalises the notion of “similarly” and that allows us conclude that the patterns preserve the behaviour of its subsystems.

As an alternative model for the Heterogeneous Redundancy pattern, we could relax the requirement for the subsystems outputs to have the same type. In order to do that, we could extend the interface of *HetR* to accept a retrieve relation [44] between the output

values of its subsystems. The retrieve relation would be used to unify the types of the output.

The voter presented in this chapter is only suitable to systems whose output is of type *real*. However, we could formalise other voting algorithms in order to remove this restriction. For example, the specification of the Redundant Management System (RMS) of the X33 [38] uses different voting algorithms for different data-types, namely majority voting for finite discrete data; mid-value selection for integer or floating-point numbers; and mean of medial extreme values for time. The majority voting chooses the value that has the higher occurrence among the input values and the mid-value selection chooses the value of the middle for sorted list. The mean of the medial extreme values requires use of *real* types as uses real arithmetic to compute the a central tendency value then would not be suitable to remove the type restriction of the outputs. We leave such extensions as future work.

The entire HOL4 model presented in this chapter contains 580 lines of specification (see Appendix A) and takes about 5 minutes to load and prove all theorems.[6] This work was the first contact of the author with the HOL notation and the HOL4 tool. He had previous knowledge with the Z/EVES theorem prover [36], which shortened significantly his learning curve in order to guide proofs with HOL4. The specification discussed in this chapter took approximately three months to be formalised and the proofs discharged in HOL4. Most of the proof steps in this work required human intervention to provide the appropriate parameters (typically a list of theorems, and simplification sets) to guide the proof.

The specification and components described in the Chapter 3 are generic in the sense that they could be used to model other patterns not described in Section 2.4, as for example the M-out-of-N pattern [2]. This pattern consists of N identical channels that operate in parallel until N-M random faults occur simultaneously. However, perhaps it may be necessary define new components in order to formalise other patterns as the Watchdog Pattern [2].

[6]These figures were obtained by running the HOL4 system on Linux Ubuntu 9.4 in a Intel T7250 with 2 GB of RAM, 2.0 GHz and 2 MB of cache.

4

New Notions of Refinement

In this chapter we formalise new notions of refinement and prove that the fault tolerant patterns *HR*, *HetR* and *TMR* preserve the behaviour of their subsystems. In the previous chapter we said that the function computed by a replicated system behaves similarly to the function of its non-replicated subsystems. For example, we proved that, by replicating a system that behaves like f using *TMR*, we implement a *SYSTEM* that behaves like $(FTMR\ e1\ e2\ e3\ f)$. In this chapter we define a refinement relation that allows us to state, for example, that $f \sqsubseteq (FTMR\ e1\ e2\ e3\ f)$. In order to achieve that, we define new notions of refinement to capture the common assumption taken by all systems engineers: fault tolerant patterns preserve the behaviour of their subsystems. This behaviour preservation occurs in a non-traditional way. We postulate that: (i) the introduction of replicas; and (ii) the computation of the average of the replicas preserve functionality. Although these notions seem controversial and different from traditional refinement notions [3], these postulates simply capture what is a common practise for systems engineers. By postulating such practise we could prove that the behaviour of *HR*, *HetR*, and *TMR* are a refinement of the behaviour of their subsystems.

In this chapter, we formalise systems behaviour as mathematical *expressions* that depend on time, input signal and are amenable to present random failures. Such expressions are simply the *body*[1] of the functions used to compute the functionality of a *SYSTEM*. We also propose a transitive, reflexive and anti-symmetric relation to ordering expressions under behavioural preserving. The chapter is organised as follows: Section 4.1 presents a model of expressions without detailing the generation of the conditional in $if-then-else$ expressions. Section 4.2 presents the definition of behavioural preserving and introduces a refinement calculus for relating expressions. Section 4.3 extends the ideal expressions by

[1] We use the term *body* to refer to the definition of a function, e.g. in $f\ x = x + x - 1$ the body of the function f is $x + x - 1$.

$$
\begin{array}{ll}
\mathit{Expression} \rightarrow & \mathit{Application} \\
& \mid \mathit{BinaryExpr} \\
& \mid \mathit{IfExpr} \\
\mathit{Application} \rightarrow & \mathit{Function\ Time}\ (\mathit{THE\ Input}) \\
\mathit{BinaryExpr} \rightarrow & \digamma\,(\mathit{Expression}, \mathit{Expression}) \\
& \mid \mathit{Expression} + \mathit{Expression} \\
& \mid \mathit{Expression}/n \\
& \mid \ldots \\
\mathit{IfExpr} \rightarrow & \mathit{if\ Condition\ then\ Expression\ else\ Expression}
\end{array}
$$

Figure 4.1 Expressions – BNF Grammar

detailing the generation of the conditional of *if−then−else* expressions, and introduces a explicit failure model. Section 4.4 extends the refinement calculus to support the failure model and the generation of the conditional of *if−then−else* expressions. Section 4.5 concludes.

4.1 Ideal Expressions

The expressions discussed in this section are generated by the grammar presented in Figure 4.1. Typically, these expressions represent the body of functions like *FHR*, *FHetR* and *FTMR* defined in Chapter 3.

The initial symbol of this grammar is *Expression*. Each expression is either an application (*Application*), or a binary expression (*BinaryExpr*) or an *if−then−else* expression (*IfExpr*). Applications have the form *Function Time* (*THE Input*), where *Function* is an identifier of the function and *Input* is the value carried out by the input signal at time *Time*. Binary expressions are formed by the non-deterministic composition of expressions $\digamma$, and arithmetic expressions like sum, division and so on. The non-deterministic composition $\digamma$ is discussed in details later. Finally, *if−then−else* expressions comprise two expressions and a conditional (*Condition*). When the conditional evaluates to *true*, the *if−then−else* expression reduces to the *then* branch, otherwise the expression is reduced to the *else* branch. Conditionals are boolean sentences and are omitted for simplicity. In Section 4.3 these conditionals are discussed in details.

We assume that all expressions have the type real. This assumption is justified by the fact that expressions represent information manipulated in digital circuits and these can be abstracted as manipulators of signals of real numbers. An example of an expression

generated by this grammar is presented next:

$$\begin{array}{l} \mathit{if}\ c_1 \\ \quad \mathit{then}\ f\ t\ (\mathit{THE}\ \mathit{FST}(\mathit{inp})) \\ \quad \mathit{else}\ f\ t\ (\mathit{THE}\ \mathit{SND}(\mathit{inp})) \end{array}$$

This expression is produced by the derivation

$$\mathit{Expression} \rightarrow \mathit{IfExpr} \rightarrow \mathit{if}\ \mathit{Condition}\ \mathit{then}\ \mathit{Expression}\ \mathit{else}\ \mathit{Expression}.$$

The production *Condition* generates the boolean sentence c_1 and each production *Expression* generates an *Application*. The *Application* in the *then* branch generates $f\ t\ (\mathit{THE}\ \mathit{FST}(\mathit{inp}))$. The *Application* in the *else* branch produces the expression $f\ t\ (\mathit{THE}\ \mathit{SND}(\mathit{inp}))$.

The non-deterministic composition F is a binary expression inspired by the non-deterministic choice operator $\sqcap$ of the process algebra CSP[2] [19]. The expression $F(f_1, f_2)$ represents a system that behaves non-deterministically as f_1 or f_2. The non-deterministic composition, F, takes two expressions and produces an expression that satisfies the following axioms (where f, g, and h are expressions):

Axiom 4.1.

$$F(f, F(g,h)) = F(F(f,g),h)$$

Axiom 4.2.

$$F(f,g) = F(g,f)$$

Axiom 4.3.

$$F(f,f) = f$$

Axiom 4.1 establishes the associativity of F, Axiom 4.2 establishes commutativity and Axiom 4.3 states that F is idempotent. For example, these axioms allow us to show that the expressions $F(F(f_1, F(f_2,f_3)), F(f_1,f_2))$ and $F(f_1, F(f_2,f_3))$ are equal. It means that it does not matter the order, repetition or nesting of subexpressions, both behaves as $F(f_1, F(f_2,f_3))$. To show this equivalence, we start from $F(F(f_1, F(f_2,f_3)), F(f_1,f_2))$ and apply Axiom 4.1 to the most external F, followed by the application of Axiom

[2]Communicating Sequential Processes (CSP) is an algebra for describing the event flow and interactions between concurrent systems. The process $P \sqcap Q$ of CSP refers the non-deterministic composition of the processes P and Q and behaves as P, or Q. This process has the same event flow of P, or Q, but not both.

4.2 to $F(F(f_1,F(f_2,f_3)),f_1)$, Axiom 4.1 to $F(f_1,F(f_1,F(f_2,f_3)))$ and Axiom 4.3 to $F(f_1,f_1)$. These steps reduce the original expression to $F(F(f_1,F(f_2,f_3)),f_2)$. The next step is to eliminate one of the duplicated f_2 from this expression. This is done applying Axiom 4.2 to the most external F, followed the application of the axioms 4.1 to $F(f_2,F(f_1,F(f_2,f_3)))$, 4.1 to $F(F(f_2,f_1),F(f_2,f_3))$, 4.2 to $F(F(f_2,f_1),f_2)$, 4.1 to $F(f_2,F(f_2,f_1))$ and 4.3 to $F(f_2,f_2)$. The expression obtained after this steps is $F(f_2,F(f_1,f_3))$. To finish the proof, the order of subexpressions is changed by applying the Axioms 4.1 and 4.2 in a convenient order. The entire proof is presented next.

1.	$F(F(f_1,F(f_2,f_3)),F(f_1,f_2))$	
2.	$F(F(F(f_1,F(f_2,f_3)),f_1),f_2)$	4.1 Associativity of F
3.	$F(F(f_1,F(f_1,F(f_2,f_3))),f_2)$	4.2 Commutativity of F
4.	$F(F(F(f_1,f_1),F(f_2,f_3)),f_2)$	4.1 Associativity of F
5.	$F(F(f_1,F(f_2,f_3)),f_2)$	4.3 Idempotence of F
6.	$F(f_2,F(f_1,F(f_2,f_3)))$	4.2 Commutativity of F
7.	$F(F(f_2,f_1),F(f_2,f_3))$	4.1 Associativity of F
8.	$F(F(F(f_2,f_1),f_2),f_3)$	4.1 Associativity of F
9.	$F(F(f_2,F(f_2,f_1)),f_3)$	4.2 Commutativity of F
10.	$F(F(F(f_2,f_2),f_1),f_3)$	4.1 Associativity of F
11.	$F(F(f_2,f_1),f_3)$	4.3 Idempotence of F
12.	$F(f_3,F(f_2,f_1))$	4.2 Commutativity of F
13.	$F(F(f_3,f_2),f_1)$	4.1 Associativity of F
14.	$F(f_1,F(f_3,f_2))$	4.2 Commutativity of F
15.	$F(f_1,F(f_2,f_3))$	4.2 Commutativity of F

Notational Conventions

As F is associative and commutative, we omit the internal parentheses of F in what follows. For example, $F(f_1,F(f_2,f_3))$ is written as $F(f_1,f_2,f_3)$.

4.2 Refinement

The refinement relation proposed here is inspired by the traditional refinement [3] and defines a partial order on the set of the expressions. This relation establishes our notion of behavioural preservation: given expressions S and S' we say that S' preserves the behaviour of S' if $S \sqsubseteq S'$. By definition, the three basic axioms presented below must be satisfied by the refinement relation, since it defines a partial order. In the following

equations, e, f and g are expressions.

Axiom 4.4.

$$e \sqsubseteq e$$

Axiom 4.5.

$$\frac{e \sqsubseteq f \qquad f \sqsubseteq g}{e \sqsubseteq g}$$

Axiom 4.6.

$$\frac{e \sqsubseteq f \qquad f \sqsubseteq e}{e = f}$$

Axiom 4.4 states that every system is refined by itself and Axiom 4.5 and 4.6 establish transitivity and anti-symmetry, respectively. In addition to these axioms, two common practises in engineering that results from the introduction of replicas of a component are introduced to the refinement axioms. The first practise is to consider that a system is refined by another system in which a backup replica is present. The second practise assumes that any system is refined by another system where replicas are present and whose output is the average of all replicas outputs.

Axiom 4.7 is inspired by the first practise. It states that given two refinements $e_1 \sqsubseteq f_1$ and $e_2 \sqsubseteq f_2$ of expressions e_1 and e_2 (replicas), then $F(e_1, e_2)$ is refined by $\mathit{if}\ c\ \mathit{then}\ f_1\ \mathit{else}\ f_2$. The condition c depends of the failure model and input signal used in f_1. How to build this condition is discussed in the next section. For now, this axiom is used to develop refinements that construct $\mathit{if}-\mathit{then}-\mathit{else}$ expressions that choose which replica of a system is executed, i.e. a switch-to-backup policy.

Axiom 4.7.

$$\frac{e_1 \sqsubseteq f_1 \qquad e_2 \sqsubseteq f_2}{F(e_1, e_2) \sqsubseteq \mathit{if}\ c\ \mathit{then}\ f_1\ \mathit{else}\ f_2}$$

To exemplify how this law is applied, we derive an expression with structure similar to the body of *FHetR*. The proof is presented below. For the sake of simplicity, we define constants by using *let* expressions.

1. $let\ f = f_1\ t\ (THE\ FST(inp))$
2. $let\ g = f_2\ t\ (THE\ SND(inp))$
3. $f \sqsubseteq f$ — 4.4 $[f/e]$
4. $g \sqsubseteq g$ — 4.4 $[g/e]$
5. $F(f,g) \sqsubseteq if\ c_1\ then\ g\ else\ f$ — 4.7 in 3, 4

We omit the construction of the condition c_1 for now. In Section 4.4 we define a strategy to build such conditions of the $if-then-else$ expressions.

Axiom 4.8 defines that a non-deterministic expression is refined by the average of its operands. An example of how this axiom is used is presented below.

Axiom 4.8.

$$F(e_1,\ldots,e_n) \sqsubseteq \frac{(e_1+\ldots+e_n)}{n}$$

We derive below an expression with structure similar to *FTMR*.

1. $let\ f_1 = f\ t\ (THE\ FST(inp))$
2. $let\ f_2 = f\ t\ (THE\ FST(SND(inp)))$
3. $let\ f_3 = f\ t\ (THE\ SND(SND(inp)))$
4. $f_1 \sqsubseteq f_1$ — 4.4 $[f_1/e]$
5. $f_2 \sqsubseteq f_2$ — 4.4 $[f_2/e]$
6. $f_3 \sqsubseteq f_3$ — 4.4 $[f_3/e]$
7. $F(f_2,f_3) \sqsubseteq if\ c_1\ then\ f_2\ else\ f_3$ — 4.7 in 5, 6
8. $let\ \kappa_1 = if\ c_1\ then\ f_2\ else\ f_3$
9. $F(f_1,f_2,f_3) \sqsubseteq if\ c_2\ then\ f_1\ else\ \kappa_1$ — 4.7 in 4, 7
10. $let\ \kappa_2 = if\ c_2\ then\ f_1\ else\ \kappa_1$
11. $F(f_2,f_3) \sqsubseteq (f_2+f_3)/2$ — 4.8 to $F(f_2,f_3)$
12. $F(f_1,f_2,f_3) \sqsubseteq if\ c_3\ then\ (f_2+f_3)/2\ else\ \kappa_2$ — 4.7 in 11, 9; 4.1 - 4.3
13. $let\ \kappa_3 = if\ c_3\ then\ (f_2+f_3)/2\ else\ \kappa_2$
14. $F(f_1,f_3) \sqsubseteq (f_1+f_3)/2$ — 4.8 to $F(f_1,f_3)$
15. $F(f_1,f_2,f_3) \sqsubseteq if\ c_4\ then\ (f_1+f_3)/2\ else\ \kappa_3$ — 4.7 in 14, 12; 4.1 - 4.3
16. $let\ \kappa_4 = if\ c_4\ then\ (f_1+f_3)/2\ else\ \kappa_3$
17. $F(f_1,f_2) \sqsubseteq (f_1+f_2)/2$ — 4.8 to $F(f_1,f_2)$
18. $F(f_1,f_2,f_3) \sqsubseteq if\ c_5\ then\ (f_1+f_2)/2\ else\ \kappa_4$ — 4.7 in 17, 15; 4.1 - 4.3
19. $let\ \kappa_5 = if\ c_5\ then\ (f_1+f_2)/2\ else\ \kappa_4$

20.	$F(f_1,f_2,f_3) \sqsubseteq (f_1+f_2+f_3)/3$	4.8 to $F(f_1,f_2,f_3)$
21.	$F(f_1,f_2,f_3) \sqsubseteq$ *if* c_6 *then* $(f_1+f_2+f_3)/3$ *else* κ_5	4.7 in 20, 18; 4.1 - 4.3

We start by defining aliases in the Steps 1-3 to shorten the expressions. The first axiom used is Axiom 4.4 in Steps 4-6 to state that the expressions f_1, f_2 and f_3 are refined by themselves. Step 7 applies Axiom 4.7 to Steps 5 and 6 to produce an *if−then−else* expression. Note that c is renamed to c_1. Such renaming is applied in the remaining steps of the proof. Step 9 applies Axiom 4.7 to Steps 4 and 7. Note that one of the *if−then−else* branches is the *if−then−else* obtained in Step 7. Step 11 applies Axiom 4.8 to $F(f_2,f_3)$. Step 12 applies Axiom 4.7 to Steps 11 and 9 to produce an *if−then−else* expression. Notice that the expression $F(F(f_2,f_3),F(f_1,f_2,f_3))$ is written as $F(f_1,f_2,f_3)$. We apply such a simplification whenever possible. Step 14 applies Axiom 4.8 to $F(f_1,f_3)$. Step 15 applies Axiom 4.7 to Steps 14 and 12 to produce an *if−then−else* expression. Step 17 applies Axiom 4.8 to $F(f_1,f_2)$. Step 18 applies Axiom 4.7 to Steps 17 and 15 to produce an *if−then−else* expression. Step 20 uses Axiom 4.8, and Step 21 concludes the proof.

After expanding $\kappa_1,...,\kappa_5$ in the equation obtained in Step 21 it is easy to confirm that the right side of this equation has the same structure of Definition 3.23 (on page 37).

Equation 4.9.

$$
\begin{aligned}
F(f_1,f_2,f_3) \sqsubseteq\ & \mathit{let}\ f_1 = f\ t\ (\mathit{THE}\ \mathit{FST}(\mathit{inp}))\ \mathit{in}\\
& \mathit{let}\ f_2 = f\ t\ (\mathit{THE}(\mathit{FST}\ \mathit{SND}(\mathit{inp})))\ \mathit{in}\\
& \mathit{let}\ f_3 = f\ t\ (\mathit{THE}(\mathit{SND}\ \mathit{SND}(\mathit{inp})))\ \mathit{in}\\
& \mathit{if}\ c_6\ \mathit{then}\ (1/3)*(f_1+f_2+f_3)\\
& \qquad \mathit{else\ if}\ c_5\ \mathit{then}\ (1/2)*(f_1+f_2)\\
& \qquad \mathit{else\ if}\ c_4\ \mathit{then}\ (1/2)*(f_1+f_3)\\
& \qquad \mathit{else\ if}\ c_3\ \mathit{then}\ (1/2)*(f_2+f_3)\\
& \qquad \mathit{else\ if}\ c_2\ \mathit{then}\ f_1\\
& \qquad \mathit{else\ if}\ c_1\ \mathit{then}\ f_2\\
& \qquad \mathit{else}\ f_3
\end{aligned}
$$

Intuitively, this equation states that f_1 is refined by the body of *FTMR*, f_2 is refined by the body of *FTMR* and f_3 is refined by the body of *FTMR*. As F chooses an expression non-deterministically, each expression f_1, f_2 and f_3 (non-replicated systems) is refined by a replicated system in the form of (the body of) *FTMR*.

In the next section we introduce a failure model into expressions in order to allow us to build the conditions c_1, c_2, ..., c_6.

4.3 Real Expressions

This section extends the notation for expressions presented in Section 4.1 to include a failure model for expressions. We annotate the expressions with failure information in order to generate the conditions $c_1, c_2, \ldots$ that remained undefined in the previous section. The annotations do not store all information needed to build the conditions, but they play a crucial role in generating them.

An extended grammar is presented in Figure 4.2. Notice in *Application* that *Function* is labelled with a superscript *FailureID*. This label establishes a failure condition for the associated *Application* at time *Time*. *FailureID* is a function from time to boolean. Whenever $FailureID(Time)$ returns *true*, the result of the associated *Application* is undefined.

$$
\begin{array}{ll}
Expression \rightarrow & Application \\
 & \mid BinaryExpr \\
 & \mid IfExpr \\
Application \rightarrow & Function^{FailureID}\ Time\ (THE\ Input) \\
BinaryExpr \rightarrow & F\ (Expression,\ Expression) \\
 & \mid\ Expression + Expression \\
 & \mid\ Expression / n \\
 & \mid \ldots \\
IfExpr \rightarrow & if\ Conditional\ then\ Expression\ else\ Expression \\
Conditional \rightarrow & (Conditional \wedge Conditional) \\
 & \mid\ (Conditional \vee Conditional) \\
 & \mid\ \neg Conditional \\
 & \mid\ IS\_SOME(Input) \\
 & \mid\ IS\_NONE(Input) \\
 & \mid\ FailureID(Time)
\end{array}
$$

Figure 4.2 Expressions – BNF Grammar

Some examples of the *FailureIDs* that can be associated to applications are: $(\lambda t.\ t > 100)$ — permanent failure after 100 time units, $(\lambda t.\ F)$ — never fails, $(\lambda t.\ (t\ mod\ 2 = 0))$ — failure at even times, $(\lambda t.\ (\exists k : \mathbb{Z}.\ sin(t) = k * cos(t)))$ — failures in periodic time intervals. We can also specify the function in terms of other predefined functions, as in $(\lambda t.\ f1(t)\ \vee\ f2(t))$ for example. Any function from time to boolean can be used. These functions are used to build the conditions of $if-then-else$ expressions. Note that these functions capture the failure rate of the expressions. For example, the function $(\lambda t.\ (t\ mod\ 2 = 0))$ represents a system with failure rate equals to 50%. In our

particular case of our fault tolerant patterns, the *FailureIDs* will look like parts of the conditions of the *FHR*, *FHetR* and *FTMR* (see definitions 3.15, 3.18 and 3.23). Detailed examples are given shortly.

In this grammar, we detail the conditions of $if-then-else$ expressions. Conditions are defined recursively as conjunction, disjunction, negation or one of the three base cases: $IS\_SOME(Input)$, $IS\_NONE(Input)$, which determine if the input signal comes with an error or not, respectively, and $FailureID(Time)$, which determines if a random failure occurs at time *Time*.

4.4 Extended Refinement

In this section we discuss the concept of equivalence between expressions and redefine Axiom 4.7 to construct the conditional c.

Axiom 4.10 postulates that given two equivalent expressions, each one of them is refined by the non-deterministic composition of both.

The concept of equivalent expressions (denoted by $\sim$) is informal and states that two expressions are equivalent whenever they are (homogeneous or heterogeneous) replicas of each other. In the real world, this assumption is supposed to be introduced by engineers when they decide whether two systems are replicas of each other or not, even when the systems are dissimilar. Alternatively, two expressions are equivalents if one can replace the other in case of a failure. Due to the informal nature of this statement, the equivalence between expressions is always introduced in proofs as an assumption. Let f and g be expressions. Then,

Axiom 4.10.

$$(f \sim g) \;\Rightarrow\; (f \sqsubseteq F(f,g)) \wedge (g \sqsubseteq F(f,g))$$

This theorem states that if two expressions f and g are equivalent then each one separately is refined by the non-deterministic choice of both. Intuitively, this axiom states that the non-deterministic composition of equivalent expressions generates results that are better or as good as the expression alone.

Expression equivalence ($\sim$) presented in Axiom 4.10 is an equivalence relation (reflexive, transitive and symmetric). Axioms 4.11, 4.12 and 4.13 formalise these properties (where f, g and h are expressions):

Axiom 4.11.

$$f \sim f$$

Axiom 4.12.

$$(f \sim g \wedge g \sim h) \Rightarrow f \sim h$$

Axiom 4.13.

$$f \sim g \Rightarrow g \sim f$$

To illustrate how to use the Axiom 4.10, we show that the equivalent expressions $f^{e1}\ t\ (THE\ inp)$ and $f^{e2}\ t\ (THE\ inp)$ are refined by $F(f^{e1}\ t\ (THE\ inp), f^{e2}\ t\ (THE\ inp))$.

1.	$f^{e1}\ t\ (THE\ inp) \sim f^{e2}\ t\ (THE\ inp)$	Assumption
2.	$f^{e1}\ t\ (THE\ inp) \sqsubseteq F(f^{e1}\ t\ (THE\ inp), f^{e2}\ t\ (THE\ inp))$	4.10 to 1
3.	$f^{e2}\ t\ (THE\ inp) \sqsubseteq F(f^{e1}\ t\ (THE\ inp), f^{e2}\ t\ (THE\ inp))$	4.10 to 1

Axiom 4.14 redefines Axiom 4.7. It states that given two refinements $e \sqsubseteq g$ and $f \sqsubseteq h$, the expression $F(e,f)$ is refined by $(if\ WELL\_DEF(g)\ then\ g\ else\ h)$.

Axiom 4.14.

$$\frac{e \sqsubseteq g \qquad f \sqsubseteq h}{F(e,f) \sqsubseteq if\ WELL\_DEF(g)\ then\ g\ else\ h}$$

The function *WELL_DEF* takes an expression as argument and generates an expression that checks whether the expression given as argument does not fail. This function is responsible for generating the conditions of the *if−then−else* expressions. *WELL_DEF* is defined below in Equation 4.15.

Definition 4.15.

$$WELL\_DEF(f^{FailureID}\ t\ (THE\ inp)) = IS\_SOME(inp) \wedge \neg FailureID(t)$$
$$WELL\_DEF(f \odot g) = WELL\_DEF(f) \wedge WELL\_DEF(g)$$
$$WELL\_DEF(f/n) = WELL\_DEF(f)$$

The symbol $\odot$ denotes any binary arithmetic operator like addition, subtraction, multiplication, etc. Note that *WELL_DEF* builds the conditions for each *if−then−else* expression using the labels (annotations) *FailureID*.

4.4.1 Behavioural Preservation of the HR, HetR and TMR

In this section, we prove that the Homogeneous Redundancy pattern, the Heterogeneous Redundancy pattern and the Triple Modular Redundancy pattern preserve the behaviour of their subsystems. The proof is done showing that the body of the function computed by these patterns, namely *FHR* (Definition 3.15), *FHetR* (Definition 3.18) and *FTMR* (Definition 3.23), is a refinement of their subsystems. Each subsystem behaviour is represented by a function application.

Homogeneous Redundancy

The proof of behavioural preservation of the Homogeneous Redundancy pattern is presented below. Step 3 adds to the proof the assumption that f_1 and f_2 are equivalent expressions, which, in this case, means homogeneous replicas. Steps 4 and 5 apply Axiom 4.4 to state that each expression f_1 and f_2 is refined by itself. Step 6 applies Axiom 4.14 to steps 4 and 5 to show that $F(f_1, f_2) \sqsubseteq \mathit{if}\ \mathit{WELL\_DEF}(f_1)\ \mathit{then}\ f_1\ \mathit{else}\ f_2$. Steps 7, 8 apply axioms 4.10 and 4.5 to show that each one of applications f_1 and f_2 is refined by $\mathit{if}\ \mathit{WELL\_DEF}(f_1)\ \mathit{then}\ f_1\ \mathit{else}\ f_2$.

1.	$\mathit{let}\ f_1 = f^{e1}\ t\ (\mathit{THE\ FST}(\mathit{inp}))$	
2.	$\mathit{let}\ f_2 = f^{e2}\ t\ (\mathit{THE\ SND}(\mathit{inp}))$	
3.	$f_1 \sim f_2$	Assumption
4.	$f_1 \sqsubseteq f_1$	4.4 $[f_1/e]$
5.	$f_2 \sqsubseteq f_2$	4.4 $[f_2/e]$
6.	$F(f_1, f_2) \sqsubseteq \mathit{if}\ \mathit{WELL\_DEF}(f_1)\ \mathit{then}\ f_1\ \mathit{else}\ f_2$	4.14 in 4, 5
7.	$f_1 \sqsubseteq \mathit{if}\ \mathit{WELL\_DEF}(f_1)\ \mathit{then}\ f_1\ \mathit{else}\ f_2$	4.10 in 3; 4.5
8.	$f_2 \sqsubseteq \mathit{if}\ \mathit{WELL\_DEF}(f_1)\ \mathit{then}\ f_1\ \mathit{else}\ f_2$	4.10 in 3; 4.5

Finally, $\mathit{WELL\_DEF}(f_1)$ is expanded with Definition 4.15. It results in two equations that state that *FHR* refines f_1 and f_2 alone (Equations 4.16 and 4.17). Compare the right side of these equations with the Definition 3.15 (on page 30). If we ignore the label annotations, the body of *FHR* is the refinement of $f^{e1}\ t\ (\mathit{THE\ FST}(\mathit{inp}))$ (the primary channel) and $f^{e2}\ t\ (\mathit{THE\ SND}(\mathit{inp}))$ (the backup channel).

Equation 4.16.

$$f^{e1}\ t\ (THE\ FST(inp)) \sqsubseteq \begin{array}{l} if\ IS\_SOME(FST(inp)) \wedge \neg\ e1(t) \\ then\ f^{e1}\ t\ (THE\ FST(inp)) \\ else\ f^{e2}\ t\ (THE\ SND(inp)) \end{array}$$

Equation 4.17.

$$f^{e2}\ t\ (THE\ SND(inp)) \sqsubseteq \begin{array}{l} if\ IS\_SOME(FST(inp)) \wedge \neg\ e1(t) \\ then\ f^{e1}\ t\ (THE\ FST(inp)) \\ else\ f^{e2}\ t\ (THE\ SND(inp)) \end{array}$$

Heterogeneous Redundancy

The proof of behavioural preservation of Heterogeneous Redundancy pattern is done as follows. First, assume that f and g are equivalent expressions (in this case, heterogeneous replicas). Steps 4 and 5 apply Axiom 4.4 to establish the fact that each one of expressions f and g is refined by itself. Step 6 applies Axiom 4.14 to steps 4 and 5 to derive an *if−then−else* expression. Steps 7 and 8 use the Axioms 4.10 and 4.5 to conclude that each application is refined by the $if\ WELL\_DEF(f)\ then\ f\ else\ g$.

1.	$let\ f\ =\ (f_1)^{e1}\ t\ (THE\ FST(inp))$	
2.	$let\ g\ =\ (f_2)^{e2}\ t\ (THE\ SND(inp))$	
3.	$f\ \sim\ g$	Assumption
4.	$f\ \sqsubseteq\ f$	4.4 $[f/e]$
5.	$g\ \sqsubseteq\ g$	4.4 $[g/e]$
6.	$F(f,g)\ \sqsubseteq\ if\ WELL\_DEF(f)\ then\ f\ else\ g$	4.14 in 4, 5
7.	$f\ \sqsubseteq\ if\ WELL\_DEF(f)\ then\ f\ else\ g$	4.10 in 3; 4.5
8.	$g\ \sqsubseteq\ if\ WELL\_DEF(f)\ then\ f\ else\ g$	4.10 in 3; 4.5

By expanding the Definition 4.15 in the last two equations, we obtain the Equations 4.18 and 4.19. Compare the right side of these equations with the Definition 3.18 (on page 33). If we ignore the label annotations, the right side of the refinement is exactly the body of *FHetR*.

Equation 4.18.

$$f_1^{e1}\ t\ (THE\ FST(inp)) \sqsubseteq \begin{array}{l} if\ IS\_SOME(FST(inp)) \wedge \neg\ e1(t) \\ then\ (f_1)^{e1}\ t\ (THE\ FST(inp)) \\ else\ (f_2)^{e2}\ t\ (THE\ SND(inp)) \end{array}$$

Equation 4.19.

$$f_2^{e2}\ t\ (THE\ SND(inp)) \sqsubseteq \begin{array}{l} if\ IS\_SOME(FST(inp)) \wedge \neg\ e1(t) \\ then\ (f_1)^{e1}\ t\ (THE\ FST(inp)) \\ else\ (f_2)^{e2}\ t\ (THE\ SND(inp)) \end{array}$$

Triple Modular Redundancy

The proof of behavioural preservation of the Triple Modular Redundancy pattern is longer than the proofs for *FHR* and *FHetR*. It starts by adding three assumptions stating that the applications f_1, f_2 and f_3 are equivalent. In this case, these expression are homogeneous replicas.

1. $let\ f_1 = f^{e1}\ t\ (THE\ FST(inp)))$
2. $let\ f_2 = f^{e2}\ t\ (THE\ FST(SND(inp))))$
3. $let\ f_3 = f^{e3}\ t\ (THE\ SND(SND(inp))))$
4. $f_1 \sim f_2$ Assumption
5. $f_1 \sim f_3$ Assumption
6. $f_2 \sim f_3$ Assumption

Steps 7, 8 and 9 use Axiom 4.4 to add the fact that each one of applications is refined by itself. In Step 11, Axiom 4.14 is applied to steps 8 and 9 to produce an $if-then-else$. Step 14 apply Axiom 4.14 to steps 7 and 11. Step 16 infers that $F(f_2, f_3) \sqsubseteq (f_2 + f_3)/2$ using Axiom 4.8. Step 18 applies Axiom 4.14 to steps 16 and 14 and the Axioms 4.1 - 4.3 to unify expressions.

7.	$f_1 \sqsubseteq f_1$	4.4 $[(f_1)/e]$
8.	$f_2 \sqsubseteq f_2$	4.4 $[(f_2)/e]$
9.	$f_3 \sqsubseteq f_3$	4.4 $[(f_3)/e]$
10.	$let\ c_1 = WELL\_DEF(f_2)$	
11.	$F(f_2,f_3) \sqsubseteq if\ c_1\ then\ f_2\ else\ f_3$	4.14 in 8, 9
12.	$let\ \kappa_1 = if\ c_1\ then\ f_2\ else\ f_3$	
13.	$let\ c_2 = WELL\_DEF(f_1)$	
14.	$F(f_1,f_2,f_3) \sqsubseteq if\ c_2\ then\ f_1\ else\ \kappa_1$	4.14 in 7, 11
15.	$let\ \kappa_2 = if\ c_2\ then\ f_1\ else\ (\kappa_1)$	
16.	$F(f_2,f_3) \sqsubseteq (f_2+f_3)/2$	4.8 to $F(f_2,f_3)$
17.	$let\ c_3 = WELL\_DEF((f_2+f_3)/2)$	
18.	$F(f_1,f_2,f_3) \sqsubseteq if\ c_3\ then\ (f_2+f_3)/2\ else\ \kappa_2$	4.14 in 16, 14; 4.1 - 4.3
19.	$let\ \kappa_3 = if\ c_3\ then\ (f_2+f_3)/2\ else\ \kappa_2$	

Step 20 is similar to Step 16, but infers that $F(f_1,f_3) \sqsubseteq (f_1+f_3)/2$. Step 22 applies Axiom 4.14 to steps 20 and 18 to achieve an *if−then−else* expression and the Axioms 4.1 - 4.3 to unify expressions. Steps 24 and 28 follow the same structure of Step 20, but infer refinements from different expressions. Step 26 applies Axiom 4.14 to steps 24 and 22 and the Axioms 4.1 - 4.3 to unify expressions.

20.	$F(f_1,f_3) \sqsubseteq (f_1+f_3)/2$	4.8 to $F(f_1,f_3)$
21.	$let\ c_4 = WELL\_DEF((f_1+f_3)/2)$	
22.	$F(f_1,f_2,f_3) \sqsubseteq if\ c_4\ then\ (f_1+f_3)/2\ else\ \kappa_3$	4.14 in 20, 18; 4.1 - 4.3
23.	$let\ \kappa_4 = if\ c_4\ then\ ((f_1+f_3)/2)\ else\ (\kappa_3)$	
24.	$F(f_1,f_2) \sqsubseteq (f_1+f_2)/2$	4.8 to $F(f_1,f_2)$
25.	$let\ c_5 = WELL\_DEF((f_1+f_2)/2)$	
26.	$F(f_1,f_2,f_3) \sqsubseteq if\ c_5\ then\ (f_1+f_2)/2\ else\ \kappa_4$	4.14 in 24, 22; 4.1 - 4.3
27.	$let\ \kappa_5 = if\ c_5\ then\ (f_1+f_2/2)\ else\ \kappa_4$	
28.	$F(f_1,f_2,f_3) \sqsubseteq (f_1+f_2+f_3)/3$	4.8 to $F(f_1,f_2,f_3)$

Step 30 applies Axiom 4.14 to steps 28 and 26 to produce the most external *if−then−else* and the Axioms 4.1 - 4.3 to unify expressions. Steps 31 - 39 manipulate equivalences to infer that each one of applications is equivalent to $F(f_1,f_2,f_3)$. Steps 31, 32 and 33 finish the proof applying the Axioms 4.10 and 4.5.

29.	$let\ c_6 = WELL\_DEF((f_1+f_2+f_3)/3)$	
30.	$F(f_1,f_2,f_3) \sqsubseteq if\ c_6\ then\ (f_1+f_2+f_3)/3\ else\ \kappa_5$	4.14 in 28, 26; 4.1 - 4.3
31.	$f_1 \sim F(f_1,f_2)$	4.10 in 4, 5
32.	$f_2 \sim F(f_1,f_2)$	4.10 in 4, 5
33.	$f_1 \sim F(f_1,f_3)$	4.10 in 4, 6
34.	$f_3 \sim F(f_1,f_3)$	4.10 in 4, 6
35.	$F(f_1,f_2) \sim F(f_1,f_2,f_3)$	4.13, 4.12 in 31, 32
36.	$F(f_1,f_3) \sim F(f_1,f_2,f_3)$	4.13, 4.12 in 33, 34
37.	$f_1 \sim F(f_1,f_2,f_3)$	4.12 in 31, 35
38.	$f_2 \sim F(f_1,f_2,f_3)$	4.12 in 32, 35
39.	$f_3 \sim F(f_1,f_2,f_3)$	4.12 in 33, 36
40.	$f_1 \sqsubseteq if\ c_6\ then\ (f_1+f_2+f_3)/3\ else\ \kappa_5$	4.10 in 37; 4.5
41.	$f_2 \sqsubseteq if\ c_6\ then\ (f_1+f_2+f_3)/3\ else\ \kappa_5$	4.10 in 38; 4.5
42.	$f_3 \sqsubseteq if\ c_6\ then\ (f_1+f_2+f_3)/3\ else\ \kappa_5$	4.10 in 39; 4.5

Before comparing the right side of the equations obtained in Steps 40, 41 and 42 with Definition 3.23 (on page 37), we need to expand the conditions using the definition of *WELL_DEF* (Definition 4.15). Below we expand each condition separately.

1. $c_1 = WELL\_DEF(f_2) = IS\_SOME(FST(SND(inp))) \wedge \neg e2(t)$
2. $c_2 = WELL\_DEF(f_1) = IS\_SOME(FST(inp)) \wedge \neg e1(t)$
3. $c_3 = WELL\_DEF((f_2+f_3)/2) = IS\_SOME(FST(SND(inp))) \wedge$
   $IS\_SOME(SND(SND(inp))) \wedge \neg e2(t) \wedge \neg e3(t)$
4. $c_4 = WELL\_DEF((f_1+f_3)/2)) = IS\_SOME(FST(inp)) \wedge$
   $IS\_SOME(SND(SND(inp))) \wedge \neg e1(t) \wedge \neg e3(t)$
5. $c_5 = WELL\_DEF(f_1+f_2)/2) = IS\_SOME(FST(inp)) \wedge$
   $IS\_SOME(FST(SND(inp))) \wedge \neg e1(t) \wedge \neg e2(t)$
6. $c_6 = WELL\_DEF((f_1+f_2+f_3)/3) = IS\_SOME(FST(inp)) \wedge$
   $IS\_SOME(FST(SND(inp))) \wedge IS\_SOME(SND(SND(inp))) \wedge$
   $\neg e1(t) \wedge \neg e2(t) \wedge \neg e3(t)$

After expanding all definitions, we can conclude:

Equation 4.20.

$$
\begin{aligned}
&f^{e1}\ t\ (THE\ FST(inp)) \sqsubseteq \\
&\quad let\ in1 = FST(inp)\ in \\
&\quad let\ in2 = FST(SND(inp))\ in \\
&\quad let\ in3 = SND(SND(inp))\ in \\
&\qquad if\ IS\_SOME(in1) \wedge IS\_SOME(in2) \wedge IS\_SOME(in3) \wedge \\
&\qquad\quad \neg e1(t) \wedge \neg e2(t) \wedge \neg e3(t) \\
&\qquad\ then\ (1/3) * (f^{e1}\ t\ (THE\ in1) + f^{e2}\ t\ (THE\ in2) + f^{e3}\ t\ (THE\ in3)) \\
&\qquad else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in2) \wedge \neg e1(t) \wedge \neg e2(t) \\
&\qquad\quad then\ (1/2) * (f^{e1}\ t\ (THE\ in1) + f^{e2}\ t\ (THE\ in2)) \\
&\qquad else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in3) \wedge \neg e1(t) \wedge \neg e3(t) \\
&\qquad\quad then\ (1/2) * (f^{e1}\ t\ (THE\ in1) + f^{e3}\ t\ (THE\ in3)) \\
&\qquad else\ if\ IS\_SOME(in2) \wedge IS\_SOME(in3) \wedge \neg e2(t) \wedge \neg e3(t) \\
&\qquad\quad then\ (1/2) * (f^{e2}\ t\ (THE\ in2) + f^{e3}\ t\ (THE\ in3)) \\
&\qquad else\ if\ IS\_SOME(in1) \wedge \neg e1(t)\ then\ (f^{e1}\ t\ (THE\ in1)) \\
&\qquad else\ if\ IS\_SOME(in2) \wedge \neg e2(t)\ then\ (f^{e2}\ t\ (THE\ in2)) \\
&\qquad else\ (f^{e3}\ t\ (THE\ in3))
\end{aligned}
$$

Equation 4.21.

$$
\begin{aligned}
&f^{e2}\ t\ (THE\ FST(SND(inp))) \sqsubseteq \\
&\quad let\ in1 = FST(inp)\ in \\
&\quad let\ in2 = FST(SND(inp))\ in \\
&\quad let\ in3 = SND(SND(inp))\ in \\
&\qquad if\ IS\_SOME(in1) \wedge IS\_SOME(in2) \wedge IS\_SOME(in3) \wedge \\
&\qquad\quad \neg e1(t) \wedge \neg e2(t) \wedge \neg e3(t) \\
&\qquad\ then\ (1/3) * (f^{e1}\ t\ (THE\ in1) + f^{e2}\ t\ (THE\ in2) + f^{e3}\ t\ (THE\ in3)) \\
&\qquad else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in2) \wedge \neg e1(t) \wedge \neg e2(t) \\
&\qquad\quad then\ (1/2) * (f^{e1}\ t\ (THE\ in1) + f^{e2}\ t\ (THE\ in2)) \\
&\qquad else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in3) \wedge \neg e1(t) \wedge \neg e3(t) \\
&\qquad\quad then\ (1/2) * (f^{e1}\ t\ (THE\ in1) + f^{e3}\ t\ (THE\ in3)) \\
&\qquad else\ if\ IS\_SOME(in2) \wedge IS\_SOME(in3) \wedge \neg e2(t) \wedge \neg e3(t) \\
&\qquad\quad then\ (1/2) * (f^{e2}\ t\ (THE\ in2) + f^{e3}\ t\ (THE\ in3)) \\
&\qquad else\ if\ IS\_SOME(in1) \wedge \neg e1(t)\ then\ (f^{e1}\ t\ (THE\ in1)) \\
&\qquad else\ if\ IS\_SOME(in2) \wedge \neg e2(t)\ then\ (f^{e2}\ t\ (THE\ in2)) \\
&\qquad else\ (f^{e3}\ t\ (THE\ in3))
\end{aligned}
$$

Equation 4.22.

$$
\begin{aligned}
&f^{e3}\ t\ (THE\ FST(SND(SND(inp)))) \sqsubseteq \\
&\quad let\ in1 = FST(inp)\ in \\
&\quad let\ in2 = FST(SND(inp))\ in \\
&\quad let\ in3 = SND(SND(inp))\ in \\
&\qquad if\ IS\_SOME(in1) \wedge IS\_SOME(in2) \wedge IS\_SOME(in3) \wedge \\
&\qquad\quad \neg e1(t) \wedge \neg e2(t) \wedge \neg e3(t) \\
&\qquad\ then\ (1/3) * (f^{e1}\ t\ (THE\ in1) + f^{e2}\ t\ (THE\ in2) + f^{e3}\ t\ (THE\ in3)) \\
&\qquad else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in2) \wedge \neg e1(t) \wedge \neg e2(t) \\
&\qquad\quad then\ (1/2) * (f^{e1}\ t\ (THE\ in1) + f^{e2}\ t\ (THE\ in2)) \\
&\qquad else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in3) \wedge \neg e1(t) \wedge \neg e3(t) \\
&\qquad\quad then\ (1/2) * (f^{e1}\ t\ (THE\ in1) + f^{e3}\ t\ (THE\ in3)) \\
&\qquad else\ if\ IS\_SOME(in2) \wedge IS\_SOME(in3) \wedge \neg e2(t) \wedge \neg e3(t) \\
&\qquad\quad then\ (1/2) * (f^{e2}\ t\ (THE\ in2) + f^{e3}\ t\ (THE\ in3)) \\
&\qquad else\ if\ IS\_SOME(in1) \wedge \neg e1(t)\ then\ (f^{e1}\ t\ (THE\ in1)) \\
&\qquad else\ if\ IS\_SOME(in2) \wedge \neg e2(t)\ then\ (f^{e2}\ t\ (THE\ in2)) \\
&\qquad else\ (f^{e3}\ t\ (THE\ in3))
\end{aligned}
$$

Notice that the right side of these equations are exactly the body of *FTMR* (Definition 3.23 on page 37).

4.5 Concluding Remarks

Systems engineers assume that redundancy patterns improve the system. They regard that these patterns improve the safety and preserve the functionality of the system (although they might worsen timing, power consumption, weigh, etc.).

In this work we note that the behavioural preservation does not occur according to the classical notions of refinement. In this chapter we axiomatised new refinement notions that capture the industry practise: a system preserves the behaviour when replicated (Axiom 4.14); and a system also preserves the behaviour if it outputs the average of its subsystems (Axiom 4.8). By using these two axioms, we are able to prove that an individual system is refined by *HR*, *HetR* and *TMR*.

We have also experimented other approaches. For example, interval arithmetic could also be a possible formalism. However, by using interval arithmetic we could not prove that an isolated system is refined by the average of its replicas. This only happens when the isolated system has the worst precision among all replicas. This does not capture exactly what is common practise in the industry.

The refinement calculus proposed in this chapter is suitable to prove the behavioural preservation of *HR*, *HetR* and *TMR*. In order to prove the behavioural preservation of other patterns that use other mechanisms to manage the redundant systems, it may be necessary extend our refinement calculus. We have not analysed our refinement calculus regarding completeness.

The approach presented in the Section 4.2 was mechanised in HOL4 using the command `new_axiom`. The theorems of Section 4.2 were proved mechanically using HOL4. However, axiom postulation is not an acceptable practise in the theorem proving community, as it is not a model built as a conservative extension. Therefore, we decided to mechanise only Section 4.2 as proof of concept. Nevertheless, we believe that we could have implemented labels (Section 4.3) in a similar way to our mechanisation of Section 4.2 easily.

5 Case Study

This chapter presents a case study that describes the introduction of a fault tolerant pattern to a simplified model of an aircraft *Elevator Control System* (ECS). Elevator surfaces control the aircraft's orientation by changing the up-and-down movement of the aircraft's nose. The original model of the ECS used in this case study was developed by Embraer (Empresa Brasileira de Aeronáutica) using Simulink [26], one of the main modelling environments used to simulate and validate mathematical/physical models. The original model consists of block diagrams representing the architecture and operation of an ECS and employs several fault tolerant patterns to obtain reliable information from different sensors. In this work, we used an ECS inspired by that from Embraer and published by Jesus [9].

Here, we aim to show the applicability of the results described in chapters 3 and 4 in the development of aeronautical systems. We informally translate the Simulink diagram to a HOL4 function and show how to introduce more redundancy in order to increase the fault tolerance to the component that runs the control algorithm of the ECS.

This chapter is organised in four sections. Section 5.1 provides an overview of the ECS. Section 5.2 details the translation of the control algorithm of the ECS to a HOL4 function. Section 5.3 applies the triple modular redundancy pattern to the ECS, and Section 5.4 concludes.

5.1 Elevator Control System

In most aircrafts, the pitching movement (the up-and-down movement of the aircraft's nose) is controlled by two elevator surfaces at the rear of the fuselage. The ECS is the main system responsible for controlling these two surfaces [9]. When the elevator's surfaces are in the up position (upward deflection) the nose of the aeroplane is forced to

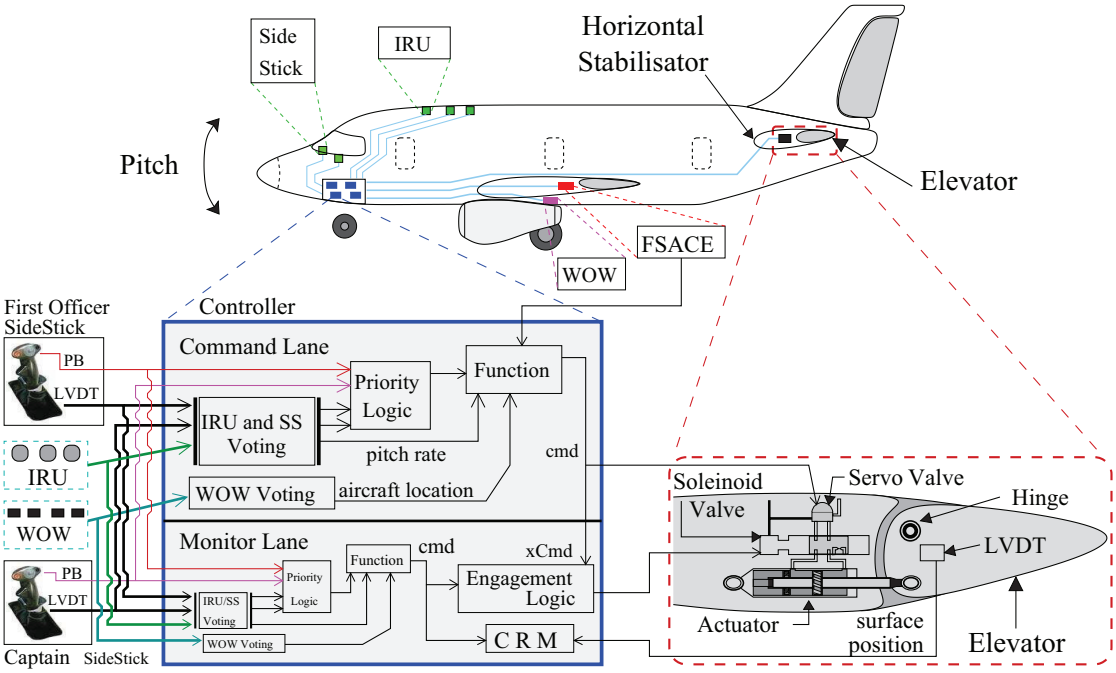


Figure 5.1 The Elevator Control System - Overview [9]

point upward. And when the elevators are deflected downward, then the nose is forced downward.

Two typical mechanisms are used by aeronautical industry to control the surfaces responsible for changing the aircraft's direction. The conventional mechanism consists of employing mechanical devices, which limit the force used to steer and manoeuvre the aeroplane by the pilots' physical capabilities. The second mechanism is called *Fly-By-Wire* (FBW), which employs electronic devices for weight savings and better performance and handling qualities. Our case study uses FBW technology to control the elevator surfaces. An overview of this model is presented in the Figure 5.1.

The ECS receives commands from *inceptors*, which are side-stick joysticks that capture the commands from the pilots. Each pilot has a private side-stick, which is composed of a priority button (PB) and three longitudinal side-stick deflections, captured from three Linear Variable Differential Transformers (LVDT). The deflections are measured in degrees and the priority button is a push-button "normally closed", i.e. a button that, whenever pressed, interrupts the electrical current in a circuit. Inceptors are depicted in Figure 5.2. The priority button installed in each side-stick sets the respective stick as the only one in control of the system (by keeping the button pressed). Each LVDT is connected to an exclusive digital data bus and sends signals to a voter equipment. In

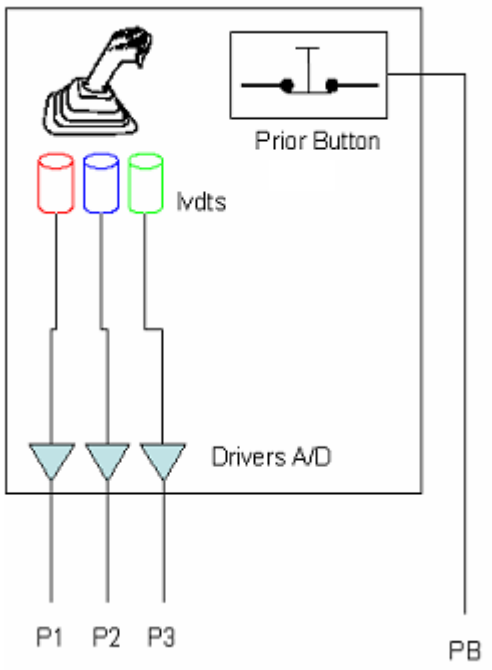


Figure 5.2 Inceptors (Side-stick)

addition, signals captured from three external sensors are sent to other voters. These sensors capture: the pitch rate in degrees per second, the aircraft location ("on ground" or "in air"), and the flap position (retracted or deployed). Flaps are hinged surfaces on the trailing edge of the wings of a fixed-wing aircraft. As the flaps are extended, the stalling speed of the aircraft is reduced. These signals are provided by three Inertial Reference Units (IRUs), four Weight-On-Wheels (WOW) sensors and the Flap-Slat Actuator Control Electronics (FS-ACE), respectively. All information obtained from inceptors and sensors can be seen as electronic signals. After processing these signals according to specific control algorithm the controllers send command signals to the PCUs (actuator) via electrical wiring, which in turn will drive the surface movements. Actuators are the devices that really move the surfaces. The actuator depicted in Figure 5.1 is composed of a solenoid valve and a electrical-hydraulic servo valve. These valves are used to controls the engagement of the actuator from power pumps that supply the hydraulic pressure used to displace the actuator ram (piston).

In this case study we are interested in increasing the level of fault tolerance of the Elevator Control System by applying the triple modular redundancy pattern to the control algorithm of ECS. The original model [9] does not have this replication, thus in this sense this case study is a contribution to the original model.

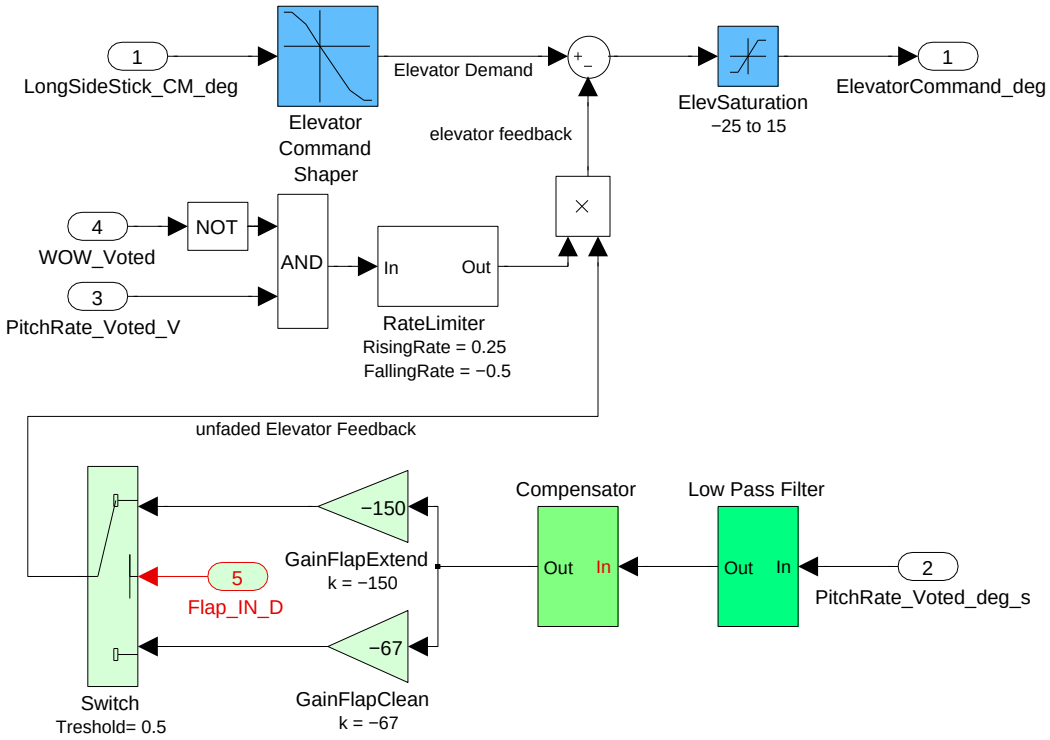


Figure 5.3 Control Algorithm of the ECS in Simulink

5.2 Translation

This section describes how we manually translated the Simulink block diagrams [26] for the ECS into a HOL4 function. Such a function was later embedded into a *BLOCK*.

Figure 5.3 depicts the Simulink diagram for the block "Function" depicted in Figure 5.1. Except for the block *Elevator Command Shaper*, each block could be easily translated into a transfer function. A transfer function is a mathematical representation of the relation between input and output of a linear time-invariant system [32]; it captures the behaviour of the blocks and is used as a specification. (Some blocks like *Compensator* and *Low Pass Filter* had to be expanded before we could infer their transfer function.) The *Elevator Command Shaper* did not have an explicit transfer function available in the original Simulink files, although the parameters of this block were provided. In this case, we calculated an approximate transfer function based on the parameters of this block and the Lagrange interpolation method [45] (this block performs a linear interpolation).

The translated HOL4 function was obtained by mapping the components of the control algorithm (Figure 5.3) to the corresponding transfer function and then composing these

functions according to the connections among the blocks. At the end of the translation, the main function of the ECS (control algorithm) was embedded into a *BLOCK* as we defined in Chapter 3. We assume the translation is correct. This assumption does not compromise our case study as our verification concerns the redundancy of the ECS instead of verifying the ECS itself. The next paragraphs describe some components of the elevator control algorithm, present their Simulink blocks and show their respective transfer function.

The block *Gain* amplifies the input by a certain amount given by the first argument. Simulink describes this component as a signal multiplier. Figure 5.4 presents the HOL4 representation of this block at the left and its corresponding block at the right. This convention is followed in the next figures.

$$Gain(K : real, inp) = K * inp$$

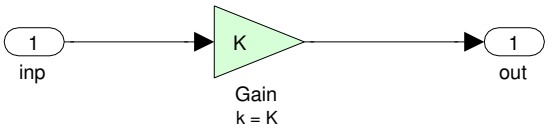


Figure 5.4 Gain block

The *SwitchThreshold* (Figure 5.5) chooses between two inputs (*inpA* and *inpB*). The decision is made according to the value of a threshold k and the switch input t.

$$SwitchTreshold\ (k, t, inpA, inpB) = \\ if\ (t >= k)\ then\ inpB \\ else\ inpA$$

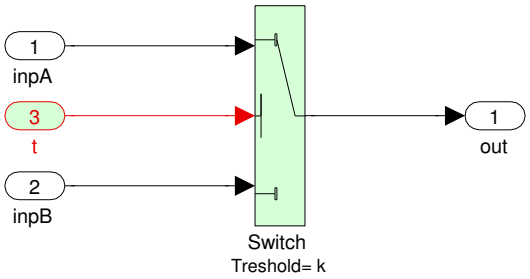


Figure 5.5 Switch Threshold

The *ElevSaturation* (Figure 5.6) imposes upper and lower bounds on a value. When the input value z is within the range *kmin* and *kmax*, the output signal is equal to the input signal. Otherwise, it restricts the signal to the upper and lower bounds.

The *Low_Pass_Filter* (Figure 5.7) filters low-frequency signals and reduces the amplitude of signals with frequencies higher than a certain cut-off frequency. It is used to generate the "filtered pitch rate" signal.

$$ElevSaturation(kmin, kmax, z) =$$
$$\quad if\ (z > kmax)\ then\ kmax$$
$$\quad else\ if\ (v < kmin)\ then\ kmin$$
$$\quad else\ z$$

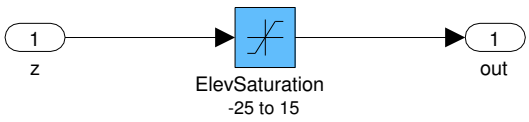


Figure 5.6 Elevator Saturation

$$Low\_Pass\_Filter(z) =$$
$$\frac{0.0725z + 0.0725}{z - 0.8551}$$

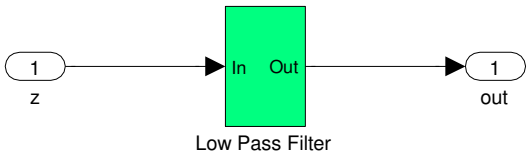


Figure 5.7 Low Pass Filter

The *compensator* (Figure 5.8) increases the stability of the system response by improving the undesirable frequency response in a feedback and control system. It is done by decreasing the gain of the system at frequencies above the location of the pole, and decreasing the phase of the system near the pole and zero locations. It is a second order filter used in the ECS to generate the "compensated pitch rate" signal.

$$Compensator(z) =$$
$$\frac{0.0058z^2 - 0.0115z + 0.0057}{(88.4z^2 + 680.13z + 1)}$$

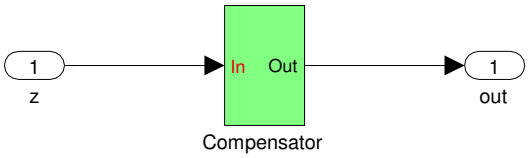


Figure 5.8 Compensator

The *RateLimiter* limits the first derivative of the signal passing through it. The output changes no faster than the specified limit. Unfortunately, for this particular case, the original Simulink model gives no information about the computation of the *RateLimiter*. We abstracted this function by defining it as a simple identity function. This decision does not compromise the case study, as our model of the elevator is not been verified here. We are concerned here with a correct replication of the elevator via our fault tolerant patterns.

The *ElevatorCommandShaper* performs a linear interpolation among the values shown in Figure 5.9. It is used to generate the"elevator demand" signal. In the ECS documentation, the transfer function is not given explicitly, but the parameters of this block are present. With these parameters, we could calculate the transfer function using the Lagrange Interpolation method [45].

$$
\begin{aligned}
&ElevatorCommandShaper(x) = \\
&\quad (12314146553082069706217045714300 \cdot x^7 - \\
&\quad 37509402239037932550883038920300 0 \cdot x^6 - \\
&\quad 16735880068062678550941672637144643 \cdot x^5 + \\
&\quad 40735593241606482111614587802571703 0 \cdot x^4 + \\
&\quad 72563638646860501175465404091486668 75 \cdot x^3 - \\
&\quad 95695167884615315360532825481007218250 \cdot x^2 - \\
&\quad 72561965071167841460692379630440000 \cdot x + \\
&\quad 95691094362800556951359596573110000 0)/ \\
&\quad 11575286279507525925931077239861718600 00
\end{aligned}
$$

Stick Shaper Table

Longitudinal Side Stick Command[deg] (z)	Elevator Demmand[deg] (out)
-25	15
-20	15
-10	10
-0.1	0
0.1	0
20	-20
30	-25
35	-25

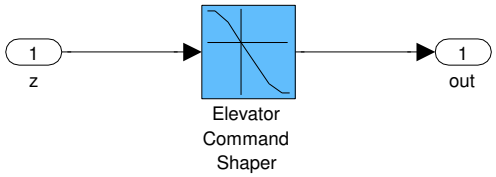


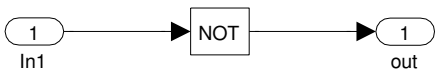
Figure 5.9 Elevator Command Shaper

The functions *NOT*, *AND*, *SUM*, *MULT* represent boolean negation, boolean conjunction, arithmetic addition and arithmetic multiplication, respectively. The translation is completely straightforward (see Figure 5.10).

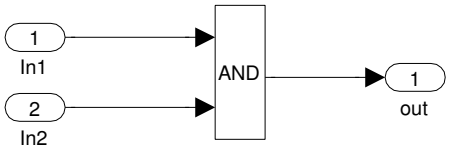
Finally, as Simulink carries out implicit data type conversions, the original model does not need any block to perform conversion between types. However, in HOL4 this conversion needs to be done explicitly. The function *B2REAL* is used to convert a boolean input to real.

$$B2REAL(a : bool) = if\ a\ then\ 1\ else\ 0$$

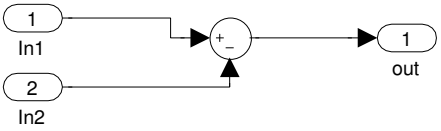
$$NOT(In1 : bool) = \neg\, In1$$



$$AND(In1, In2) = In1 \wedge In2$$



$$SUM(In1 : real, In2) = In1 + In2$$



$$MULT(In1 : real, In2) = In1 * In2$$

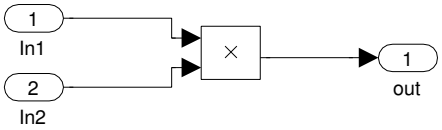


Figure 5.10 $NOT, AND, SUM, MULT$

Now we present the elevator control function itself (Figure 5.11). This function is built based on the connections among the blocks depicted in Figure 5.3. The elevator controller takes as input the angular momentum of the aeroplane (*PitchRate*), the flap position (*Flap*), the weight on wheels (*WOW*), the longitudinal side-stick deflection (*LongSideStick*) and a signal that validates the pitch rate (*PitchRate_Voted*).

We embedded the *elevator* function into a *BLOCK*. It is easy to build such a *BLOCK*. Let d be the delay and e be an error function of the *elevator*. Then

$$BLOCK\ d\ e\ elevator\ (inp, out)$$

is a block whose input has type $num \rightarrow (real\ \#\ boolean\ \#\ boolean\ \#\ real\ \#\ boolean)option$ and whose output has type $num \rightarrow real\ option$.

5.3 Applying Triple Modular Redundancy to ECS

We applied the Triple Modular Redundancy to ECS. However we could have chosen any one of our verified patterns to illustrate this case study. The first step in order to apply a pattern is to build an implementation that computes the function *elevator*. It is easy to

$$
\begin{array}{l}
elevator\ t\ (PitchRate, Flap, WOW, LongSideStick, PitchRate\_Voted) = \\
\quad let\ out\_lpf = Low\_Pass\_Filter(PitchRate)\ in \\
\quad let\ out\_cpt = compensator(out\_lpf)\ in \\
\quad let\ out\_gfe = Gain(-150, out\_cpt)\ in \\
\quad let\ out\_gfc = Gain(-67, out\_cpt)\ in \\
\quad let\ out\_sth = SwitchThreshold(1/2, B2REAL(Flap), out\_gfe, out\_gfc)\ in \\
\quad let\ out\_not = NOT(WOW)\ in \\
\quad let\ out\_and = AND(out\_not, PitchRate\_Voted)\ in \\
\quad let\ out\_rtl = RateLimiter(B2REAL(out\_and))\ in \\
\quad let\ out\_mul = MULT(out\_rtl, out\_sth)\ in \\
\quad let\ out\_ecs = ElevatorCommandShaper(LongSideStick\_CM\_deg)\ in \\
\quad let\ out\_sum = SUM(out\_ecs, out\_mul)\ in \\
\quad let\ out\_str = ElevSaturation(-25, 25, out\_sum) \\
\quad in\ out\_str
\end{array}
$$

Figure 5.11 The elevator control function.

prove that a *BLOCK* that computes the *elevator* is a *SYSTEM*.

$$
\begin{array}{l}
\vdash \forall d\ e\ inp\ out.\ BLOCK\ d\ e\ elevator(inp, out) \\
\qquad\qquad \Rightarrow SYSTEM\ d\ e\ elevator(inp, out)
\end{array}
$$

The proof of this theorem is a simple instantiation of the function f of the Theorem 3.10 (on page 27).

In a similar way, we can instantiate the correctness theorem for the Triple Modular Redundancy in order to prove that the redundant system implements an *FTMR* that averages the computation of the *elevator*.

$$
\begin{array}{l}
\vdash \forall d\ e1\ e2\ e3\ ev\ dv\ inp\ out. \\
\quad TMR\ dv\ ev\ (BLOCK\ d\ e1\ elevator)\ (BLOCK\ d\ e2\ elevator) \\
\qquad\qquad (BLOCK\ d\ e3\ elevator)\ (inp, out) \\
\quad \Rightarrow\ SYSTEM\ (d + dv)\ (ETMR\ e1\ e2\ e3\ ev\ d\ inp) \\
\qquad\qquad (FTMR\ e1\ e2\ e3\ elevator)\ (inp, out)
\end{array}
$$

The correctness theorem for the TMR is easily derived from the theorems shown in Section 2.4. The proofs for these theorems are trivial. We only have to instantiate the function to be *elevator*. The proof effort was entirely on the proof of the theorems of Section 2.4. It is possible to make this particular proof of the *elevator* completely automatic, although we have not done it yet.

Note we do not need to instantiate the failure model in the theorem above. This is because the correctness theorem for *TMR* (and the other patterns) universally quantify the failure model. This is an advantage of our model, since our theorems can be applied to systems with any failure model (our main concern is behavioural preservation, not safety improvement).

To prove that the TMR pattern applied to *elevator* preserves the behaviour of *elevator* we need to prove that

$$\begin{aligned} &elevator^{e1}\ t\ (THE\ FST(inp)) \sqsubseteq \\ &\quad (FTMR\ e1\ e2\ e3\ elevator)^{(ETMR\ e1\ e2\ e3\ ev\ d\ inp)}\ t\ (THE\ inp) \end{aligned}$$

By rewriting the application $(FTMR\ e1\ e2\ e3\ elevator)^{(ETMR\ e1\ e2\ e3\ ev\ d\ inp)}\ t\ (THE\ inp)$ using the *FTMR* (Definition 3.23) we obtain the equation next, which is essentially the Equation 4.20 with f replaced by *elevator*. Recall that, in the TMR, all replicas are equivalent as they are identical copies of each other. This refinement establishes the behavioural preservation of *TMR* applied to *elevator*. Similarly, we could prove refinements for $elevator^{e2}\ t\ (THE(FST(SND(inp))))$ and $elevator^{e3}\ t\ (THE(SND(SND(inp))))$ (the other replicas of the elevator).

$$\begin{aligned} &elevator^{e1}\ t\ (THE\ FST(inp)) \sqsubseteq \\ &\quad let\ in1 = FST(inp)\ in \\ &\quad let\ in2 = FST(SND(inp))\ in \\ &\quad let\ in3 = SND(SND(inp))\ in \\ &\quad\quad if\ IS\_SOME(in1) \wedge IS\_SOME(in2) \wedge IS\_SOME(in3) \wedge \\ &\quad\quad\quad \neg e1(t) \wedge \neg e2(t) \wedge \neg e3(t) \\ &\quad\quad\quad then\ (1/3) * (elevator^{e1}\ t\ (THE\ in1) + elevator^{e2}\ t\ (THE\ in2) \\ &\quad\quad\quad\quad + elevator^{e3}\ t\ (THE\ in3)) \\ &\quad\quad else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in2) \wedge \neg e1(t) \wedge \neg e2(t) \\ &\quad\quad\quad\quad then\ (1/2) * (elevator^{e1}\ t\ (THE\ in1) + elevator^{e2}\ t\ (THE\ in2)) \\ &\quad\quad else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in3) \wedge \neg e1(t) \wedge \neg e3(t) \\ &\quad\quad\quad\quad then\ (1/2) * (elevator^{e1}\ t\ (THE\ in1) + elevator^{e3}\ t\ (THE\ in3)) \\ &\quad\quad else\ if\ IS\_SOME(in2) \wedge IS\_SOME(in3) \wedge \neg e2(t) \wedge \neg e3(t) \\ &\quad\quad\quad\quad then\ (1/2) * (elevator^{e2}\ t\ (THE\ in2) + elevator^{e3}\ t\ (THE\ in3)) \\ &\quad\quad else\ if\ IS\_SOME(in1) \wedge \neg e1(t)\ then\ (elevator^{e1}\ t\ (THE\ in1)) \\ &\quad\quad else\ if\ IS\_SOME(in2) \wedge \neg e2(t)\ then\ (elevator^{e2}\ t\ (THE\ in2)) \\ &\quad\quad else\ (elevator^{e3}\ t\ (THE\ in3)) \end{aligned}$$

5.4 Concluding Remarks

This chapter illustrated how the theorems of the Chapters 3 and 4 can be used in practise. We have shown how a Simulink model of a simplified elevator control system (ECS) can take advantage of the triple modular redundancy pattern to improve the safety and reliability of the original model. We translated manually the Simulink model to a HOL4 function called *elevator*. We have shown that by plugging into the *TMR* three *BLOCKs* that implement a *SYSTEM* that computes the *elevator* function, the new system behaves as one that computes *FTMR e1 e2 e3 elevator*, where *e1*, *e2* and *e3* are the failure models of each *BLOCK*. By applying the refinement notions developed at Chapter 4 we easily proved that the body of the function *FTMR e1 e2 e3 elevator* refines $elevator^{e1}t(THE\ FST(inp))$. Therefore, we showed that the *elevator* system is improved by one that computes *FTMR e1 e2 e3 elevator*.

The ECS case study uses 116 lines of specification in HOL4 (see Appendix A). To prove the case study described in this chapter, the HOL4 system takes less than 2 seconds to discharge the proofs.[1]

Thanks to the compositionality of our theorems, we could continue to apply more fault redundant patterns if we wished. For instance, we could replicate the entire *TMR* system and plug the replicas into an *HR* (see Figure 5.12). As the *TMR* also implements a *SYSTEM*, the entire replication with *HR* could be easily proved to implement a *SYSTEM* too. This *SYSTEM* would compute the function $(FHR\ (FTMR\ e1\ e2\ e3\ elevator))\ e1)$. With a bit of an effort we could prove that the correspondent expression to this *SYSTEM* refines $elevator^{e1}\ t\ (THE\ (FST\ inp))$.

[1] These figures were obtained by running the HOL4 system on Linux Ubuntu 9.4 in a Intel T7250 with 2 GB of RAM, 2.0 GHz and 2 MB (cache).

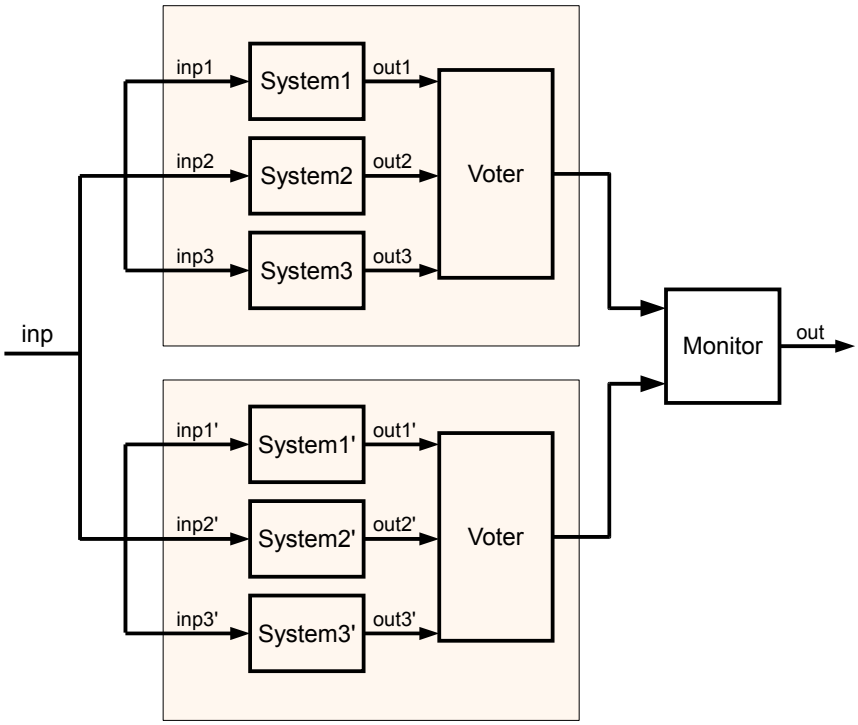


Figure 5.12 Compositionality of patterns.

6 Related Work

This chapter presents an overview of previous work on the formal verification of fault tolerant systems and of block diagrams. We start by discussing related works on the formalisation of block diagrams in Section 6.1 and then focus on works that specifically model fault tolerant patterns (Section 6.2). At the end of this chapter, in Section 6.3 we evaluate each approach with respect to ours.

6.1 Formal Model of Block Diagrams

IEC 61131-3 [20] is a standard that specifies Function Block Diagrams (FBD) as a programming language. FBD is a notation usually applied to model programmable controllers and are composed of blocks that compute functions and can present a failure during its execution. Each block has two input signals: one carries data and the other is a control signal that enables or disables the execution of the block. Disabling does not necessarily happens due to a failure in the data signal. It is the environment responsibility to set that signal. The model also presents two output signals: one for validation and another for data. The output validation signal indicates when a failure occurs during the block execution.

Orlarey *et al.* [33] proposed an algebraic approach to block diagram construction. The proposed algebra is based on three binary operators: sequential, parallel and recursive composition. These operations are used in order to connect two block diagrams to generate a new one. The main aim of Orlarey *et al.* [33] is to provide a denotational semantics for a block diagram language that describes temporal and functional aspects of the operators. The authors assume that basic blocks are pre-defined. However, two operations on basic blocks are defined: $ins(b)$, which gives the number of input ports of a block b, and $outs(b)$, which gives the number of output ports of b. The operations

defined on block diagrams are constrained by a type system that limits the composition to cases where the number of input and output ports satisfy specific rules. Signals are modelled as functions from natural numbers to real and the block functionality is given by functions from tuple of signals to tuple of signals.

6.2 Verification of Fault Tolerant Systems

Pioneering work on the verification of fault tolerant patterns were done by Owre *et al.* [34], Butler *et al.* [6] and Sokolsky *et al.* [38] in the nineties. They verified a model for a fault tolerant architecture for distributed processors called the Reliable Computing Platform (RCP) and a redundancy management system (RMS) for a Space Launch Vehicle. More recently, Dajani-Brown *et al.* applied SCADE [1] (a commercial language and tool similar to Lustre [17]) to verify a triple modular redundancy with SCADE's model checker [8], and Jesus [9] verified properties of a model of an Elevator Control System (ECS) using Communicating Sequential Processes (CSP) [19]. In what follows we describe each work in more detail.

6.2.1 Formal verification of fault-tolerant architectures: Prolegomena to the design of PVS

The formalisation of Owre *et al.* [34] was done in EHDM [1] [27] (the predecessor of PVS [35]). The authors point out that the redundancy management is a challenging problem that, if not handled correctly, can itself become the primary source of system failure. The Reliable Computing Platform (RCP) [34] is a real-time computing architecture designed to run in aerospace missions. Therefore it must be able to recover from the effects of transient failures, which are the most common failures in this environment. Transient failures are temporary failures caused by single-event upsets such as cosmic radiation, electromagnetic interference and others passing hazards. In the case of a hardware running in an aerospace environment, these failures occur naturally when an equipment cross high intensity radiated fields (HIRF). After some random time or after resetting the hardware component, the transient failures disappears. The RCP architecture has several independent computing channels (processors) operating synchronously. All channels run the same application on the same data at approximately the same time and the results are submitted to majority voting before being sent to the actuators.

[1] Enhanced Hierarchical Development Methodology

Faults are classified in three categories: *byzantine* (non-detectable faults in all channels), *symmetric* faults (systematic faults in all channels) and *manifest* faults (that can be detected by all non-faulty receivers). It is assumed that all faults are transient and that damages to a data caused by one subsystem do not propagate to cause another working subsystem to failure. Recovering from *manifest* faults typically requires less resources than recovering from *byzantine* faults. For example, in order to withstand two byzantine failures, the RCP architecture requires at least five channels operating in parallel. In contrast, five channels are enough to withstand up to four manifest faults. Owre *et al.* proved that a replicated synchronous system using majority voting presents the same behaviour of a single system with no failures.

The tool used to perform the verification of behavioural preservation of the RPC was EHDM. It is a system for the development, management and analysis of formal specifications. The proofs in EHDM are structured in a similar way to HOL4, with proof strategies that looks like *tactics*. During the verification of the RCP architecture, several other verifications were conducted by the same team in cooperation with the National Aeronautics and Space Administration (NASA). Besides the mechanisation effort to build verified hardware and software, these verifications fostered improvements in EHDM that guided the authors to developed the "next generation" of EHDM, which became PVS [35].

6.2.2 Formal design and verification of a reliable computing platform for real-time control (phase 3 results)

A more ambitious and detailed verification of the RCP was carried out by Butler *et al* [6]. Their verification is divided into 5 levels of specifications as depicted in Figure 6.1. The topmost level of hierarchy, called Uniprocessor Synchronous (US), consists of an operating system that sequentially invokes the tasks. This level behaves as a single processor that never fails: it is used as a correctness criterion in comparison with the lower levels. In comparison to Owre's verification, this level is equivalent to a single ideal system.

The level below US is the replicated synchronous system (RS). It extends the operating system to support multiple processors that execute the same application. At this level, it is assumed the existence of a global time base (i.e. synchronised time) and a reliable voting mechanism that does not fail itself. In this level the fault tolerance is achieved by

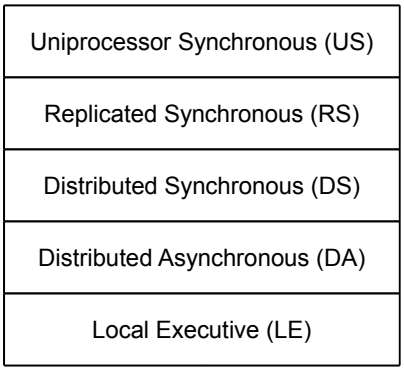


Figure 6.1 Five-level of RCP Hierarchy of Butler *et al.* [6]

applying an exact match voting[2] mechanism to the results of each processor result. The voting and the data exchange are regarded as atomic actions as they do not take time to occur.

The next level is the distributed replicated synchronous system (DS). It models the interprocessor communication mechanism and breaks the voting and data exchange transitions of the RS level into four sequential sub-transitions (compute, broadcast, compute and synchronisation). This break is done in order to separate the concerns related to real-time computation from those related to interprocessor communication, and the time involved in the computation, communication and voting. At this level all processors still share a common clock time.

The fourth level is the distributed asynchronous replicated system (DA). At this level the assumptions of the ideal synchronisation among channels is discharged and each process is associated to a different clock time. The DA machine implements a DS machine provided an underlying clock synchronisation mechanism has been established.

The last level of RCP architecture is the local executive (LE), which takes into account details of the operating system memory management, task management, and inter-processor communication. The verification of the behavioural preservation between the layers was done in EHDM. The verification between US and RS layers is quite similar to verification of RCP carried out by Owre *et al.* [34].

Each level is described by at least one state and a set of operations. These operations are functions that change the state. Axioms were introduced to relate the operations and the components of the states at the same level. This style of specification reminds the Z [39] specification style, which uses state schemes to model a system and operations

[2]Exact match voting uses a single source input data that must be shared by all redundant processors to ensure that the same input had been processed by all channels [5].

schemes to model possible changes in the state. The correctness proof between levels is established by defining a mapping that relates the state of these levels. Besides this mapping, two theorems are proved for each correctness proof: the *frame_commutes*, which states that any state changing in one level has an equivalent state changing in the other level; and the *initial_maps*, which states that any initial state in the lower level can be mapped into an initial state in the higher level.

The overall specification and proofs are described along a series of technical reports and papers published by Butler *et a.l* [41, 5, 6, 40, 42].

6.2.3 Verification of the redundancy management system for space launch vehicle: A case study

Sokolsky *et al.* [38] verified the Multiple computer Architecture for Fault Tolerance (MAFT), a redundancy management system (RMS) designed for a Space Launch Vehicle . MAFT is a modular architecture that consists of multiple processing nodes (*channels*), called application processors, that compute exactly the same function simultaneously. Every node is connected to an RMS processor, which detects faulty nodes and exclude them from the voting process. All RMS processors are mutually connected through exclusive links. An overview of this architecture for four application processes is depicted in Figure 6.2. Note that there is no central node that synchronises the computation between the application processors. Each RMS node receives the outputs from other RMS nodes and locally performs the voting process and sends the result back to the application node.

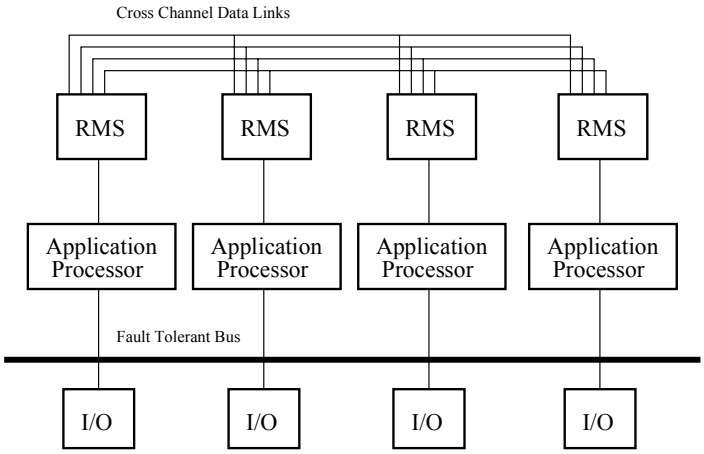


Figure 6.2 Four channel RMS based fault-tolerant system [38]

The partition between application processors and the RMS processor can be physical, in case where each one is a dedicated hardware, or logical, in case where they share the same hardware. Independently of how this partition is done, the processor nodes operate independently of each other in the sense that a fault in an application node must not influence a fault in the RMS associated. The RMS node is subject to several operational and functional requirements, being the most of these related to temporal constraints inherent to real time systems.

The fault tolerance provided by MAFT can be realised in four stages: fault detection, fault containment, fault diagnosis and recovering. Faults can be transient, intermittent and permanent. After detecting a fault, it is contained in their originated nodes. The purpose of this action is to avoid that a fault in one application process node propagates to other nodes during the voting process.

An important point of MAFT is a separation of two concerns: the redundancy management system and the application. This separation of concerns was also done in the RCP and reduces the development complexity of designing new application.

The verification of the MAFT was carried out in PARAGON [37], a tool-set for visual specification and formal verification of distributed real-time systems, which is based on the Algebra of Communicating Shared Resources (ACSR) [4]. PARAGON formalises temporal properties easily and checks for absence of zenoness[3]. The goal of formal analysis of MAFT was to ensure the compliance of the RMS design with the operational and functional requirements of this component. Not all requirements were verified by formal analysis due to the abstractions and assumptions made to represent the model. Actually, the verified requirements refer to interactions among RMS nodes and temporal constraints that the MAFT must satisfy, as maximum time to complete the functional computation. One of the requirements that were not verified concerns the verification of different voting algorithms specific for particular datatypes. The authors assumed that whatever algorithm used for voting would produce correct results regardless the datatypes.

The formal model consisted of three parallel processes representing three RMS nodes. Each one of these processes is described by the composition of internal processes. The most important processes are: the Fault Tolerant Executive (FTE), which performs the redundancy management functions, the Cross Channel Data Link (CCDL), which performs the cross-channel data communication and the *Timer*, which represents the

[3]Zenoness refers to the situation that infinite actions may take place in a finite amount of time. It is a behaviour often considered as undesirable since it violates a fundamental requirement for timed systems for they cannot be infinitely fast [16].

hardware timer included in each node. For every property to be verified, an observer process was created to run in parallel with the system and to detect violation of the property in question. The verification was carried out by the PARAGON model checker. Whenever an illegal behaviour was observed, the observer induced a deadlock in order to stop the checker.

At the end of the verification, the authors could verify 11 of 32 requirements of MAFT. Several violations were detected in the original specification. Besides the verified properties, properties as behavioural preservation were also formalised. However, these properties were not verified due the state explosion problem.

6.2.4 Formal Verification of an Avionics Sensor Voter Using SCADE

More recently, Dajani-Brown *et al.* [8] applied SCADE (Safety Critical Application Development Environment) [1], a commercial language and tool based on the Lustre language [17], to verify a triple modular redundancy with SCADE's model checker.

The authors used as a starting point the Simulink diagram of a triplex sensor voter that takes inputs from three redundant sensors. The voter assumes that each input comes with data and a self-check bit (validity flag) and outputs a single reliable output. The inputs that do not differ more than a threshold are used in order to produce the output, which is the average of them. Valid outputs are always produced when there are one or three valid inputs. However, if there are two valid inputs and they differ more than a threshold, the voter cannot determine which one is the faulty. In this case, the voter produces an output signalling the data as invalid. Faults are classified in two categories: hardware faults and signal faults. The former comprises faults that are identified by the validity flag. These faults are detected by the voter, which eliminates the sensor after 2 voting cycles. Signals faults are only detectable during the voting process when two valid signals differ more than a threshold. All faults regarded in this work are permanent faults.

In the SCADE model checker, called Design Verifier, safety properties are modelled as an observer node, which is a diagram that receives as input the variables involved in the property and produces an output that should always be true. If the output is always true, the model checker validates the property, otherwise a counter-example that violates the property is generated. The counter-example is a set of input values that makes the assertion to be false.

Although the voter supports up to two sensors faults without producing an invalid output, the model does not capture simultaneous fault injection. Two or three faults can

occur subsequently, i.e. in cascade. This assumption was done due to the probability of two simultaneous faults be sufficiently low. The possible states transitions of the voter are depicted in Figure 6.3. Note that sensor recovering is not allowed by this model as there is no transition from a state with more faulty sensors to a state with less faulty sensors. The initial state of the diagram is S0.

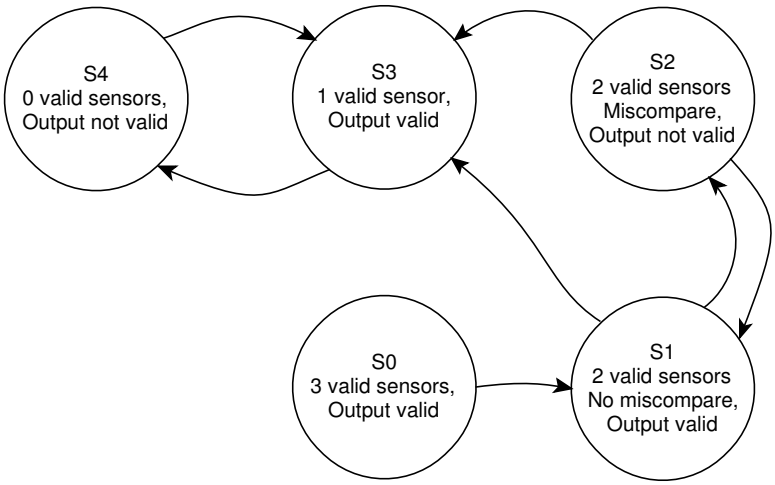


Figure 6.3 Fault states of the sensor voter [17]

The properties that were verified state that, after a specific number of cycles of voting, the voter detects a fault. All transitions depicted in Figure 6.3 were verified to validate that these expected effects. It was also checked that faulty sensors eliminated by the voting algorithm do not become available again.

The whole point of this work was to show the significance of modelling the environment were the sensors act. The environment model limits the range of values the model checker should explorer, thus making the verification feasible.

6.2.5 Formal Design and Validation of Fly-by-Wire Control Systems

Jesus [9] verified safety properties of a model of an Elevator Control System (ECS) developed by Embraer. In his dissertation he proposed a strategy to translate Simulink block diagrams to CSP [19] using a set of algebraic rules. The original ECS model from Embraer employs redundancy in the actuators to move two elevator surfaces. According to the ECS specification, the elevator system must control only the up and down movement

of the aircraft. Other movements as changing the rolling of the aircraft (rotate the aircraft body right and left) should not be caused by the elevator system.

The ECS is composed of two elevators surface and four actuators, with two actuators per surface. The actuators receive commands from different controllers. The analysis in CSP revealed that if the controllers are not synchronised, there is the possibility of each elevator surface to reflect the intention of one of the pilots of the aircraft. In this case, if the side-sticks of the captain and the first-officer are in opposite directions, the controllers can drive the elevators surfaces to opposite directions, causing the rolling of the aircraft. This behaviour was revealed by an observer process that induces a deadlock in the system if the surfaces are driven in opposite directions.

By using the refinement checker FDR2 [12], it was possible check that the alphabetised parallel composition of the system and the observer processes was not deadlock free. In order the solve this problem, the original architecture was changed to use only two controllers and force them to read the priorities of the pilots side-sticks synchronously. These changes solved the synchronisation problem and reduced the time required to perform the verification using FDR2.

6.3 Concluding Remarks

Function Block Diagrams (FBDs) differ from our block diagram model in HOL4 as they capture failures caused by abnormal termination (runtime errors, as division by zero, for example), while our model captures random failures. Besides this, the standard IEC 61131-3 [20] formalises FBDs as a programming language and not as a formal specification. In order to verify properties of FBDs, the first step should be to provide a formal semantics for them.

The work by Orlarey *et al.* [33] on an algebra for block diagrams has influenced ours by inspiring us to check if the definitions of *BLOCK* and *SYSTEM* are compositional, i.e. if the sequential and the parallel composition of these definitions still generate *BLOCKs* and *SYSTEMs*, respectively. Compositionality for the fault tolerant patterns were proved in Chapter 3. In addition to that, we proved that the sequential and the parallel composition of *BLOCKs* are well defined if the interfaces of the *BLOCKs* match (the same was proved for *SYSTEM*). We did not present these proofs in this dissertation since they are not directly related to the verification of fault tolerant patterns. Notwithstanding, these proofs can be used in future works in the definition of a semantic for block diagrams. Finally, Orlarey *et al.* did not formalise faults (neither random nor systematic).

Regarding the related works on formal verification of fault tolerant systems, they all differ from ours mainly on the compositionality of the theorems and on the model of errors. Our work separated the concept of errors from the functional behaviour of the system. Owre *et al.* [34] and Butler *et al.* [6] compared the replicated system with an original channel that does not fail, which plays the role of an oracle. Different from their models, our model takes into account the possibility of a failure on every system: both the original system and the redundancy managers (monitors, voters, etc.) can fail. The replicated system may fail less often than the original system; but that depends simply on an appropriate instantiation of the failure model and is concerned with an orthogonal issue (safety). Different from the work of Owre *et al.* [34], in our work all faults are detectable as we use a signal error embedded in the data signal.

Sokolsky *et al.* [38] verified a redundancy management system with respect to operational and functional requirements. Their verification assumed that the voter component is correct in the sense that it never introduces errors in the output when the inputs are valid. Their model deals with three kinds of voters: majority voting, mid-value selection and mean of medial extreme values. None of these voters is equivalent to the voter that we presented in this work. Similar to our work, they assumed that the same function is computed at same time on replicated systems. Due to our notion of equivalent systems, we allow dissimilar systems to be used as replicas.

We also capture the situation where a simultaneous failure occurs, differently from Dajani-Brown *et al.* [8]. They assumed that all faults are permanent and that the voter itself does not break, i.e. it always produces valid outputs if the inputs sensors are non-faulty. The main goal of the Dajani-Brown is not to check the behavioural preservation of the voter, but to check if temporal restrictions are satisfied by the model. Their model is very specific with respect to the timing of the failure. Similar to our model, noise is not formalised.

The work from Jesus [9] can also be used to check the properties of fault tolerant patterns, mainly those related to synchronisation among the replicated systems. Although this work does not focus on checking fault tolerant patterns, it was able to reveal that redundancy management used in the ECS of Embraer could lead to an undesired behaviour. Different from ours, Jesus' work does not separate the notions of functionality, delay and failures.

None of the related works relates the refinement notions with the application of the fault tolerant patterns. To our knowledge, this is the first work that proposes an axiomatic basis for reasoning about fault tolerant patterns.

7 Conclusions

In this dissertation we proposed a formal model using the HOL4 system to describe the behaviour of fault tolerant patterns regarding functionality, delay and failures as separate entities. Thus, a non-replicated channel is a system capable of: computing a certain function; subject to failure; and having an initialisation delay. The fault tolerant patterns put together non-replicated channels and pieces of redundancy management systems like monitors and voters. The separation of concerns (functionality, failure model and the initialisation delay) distinguishes our model from other related works, which omit the delay and the failure model.

When we say that the computation performed by the fault tolerant patterns is essentially the same of its subsystems, we mean that its behaviour is a refinement of the behaviour of its subsystems with respect to our new notions of refinement (Chapter 4). The traditional refinement notions do not capture the systems engineer practise. Whenever engineers introduce redundancy, they assume the behaviour is preserved in two cases: (i) when a replica is introduced as a backup system (Axiom 4.14); (ii) when the output is produced from the average of the output of the replicas (Axiom 4.8). By postulating these two scenarios as axioms, we could capture the system engineers practises and could prove that the behaviour of the patterns *HR*, *HetR* and *TMR* are, in fact, derivable from these axioms and the functionality of their subsystems.

We also proved that if the fault tolerant patterns are built from subsystems that implement a *SYSTEM*, then they become *SYSTEMs* themselves. Such compositionality allows us to easily compose patterns and prove that the whole system is still a *SYSTEM*.

Finally, the process of proving the correctness theorems also revealed us the failure model of the fault tolerant patterns: it is by proving the correctness theorem that we found out under which conditions the patterns can fail: i.e. the final definitions of the functions *E* and *ETMR* were discovered during the proof process.

7.1 Future Work

Here we point out possible directions for future works. Some of them has been undertaken, while others just illustrate possible extensions to this work.

- Formalisation of the notion of *cold* redundancy. In this work the patterns use *hot* redundancy, which requires that all channels to be active at the same time and all the time. Due to it, the formalised patterns have no notion of intermediate states during the recovering from a failure: selecting a non-faulty replica is instantaneous. *Cold* redundancy uses replicas that are activated just on demand, when the current active channel has a persistent failure. In this case, the component that manages the redundancy takes some time to recover from a failure because of the initialisation delay of the secondary channel. To formalise this notion we should change the definition of *MONITOR* and define how to classify a fault as permanent. We could also introduce delay in the *MONITOR* to reflect the fact that switching between channels takes time to occur.

- Investigate how to extend our model to represent a generic block diagram. We proved theorems stating that the sequential and parallel composition of *BLOCKs* result in a *BLOCK*, and that the sequential and parallel composition of *SYSTEMs* result in a *SYSTEM*. We also intend to investigate if recursive composition of *BLOCKs* and *SYSTEMs* are compositional.

- Extend the *BLOCK* and the *SYSTEM* definitions to allow inter-system communication. The communication protocol should allow a system be depowered or enter in a fail-safe state. This extension would turn feasible to model lightweight fault tolerant patterns, such as the watchdog pattern, where a channel is monitored by another and, in case of a permanent fault, it is depowered or sent to a fail-safe state.

- Extend the *SYSTEM* definition to support non-deterministic computation. This extension allows us to specify the behaviour of state in the *SYSTEMs*. We realise this extension could allow us to model a monitor with state that records what is the current active channel. It reflects the fact that the monitor is a physical component that switches to the alternative channel when a fails occur and, after the switch, remains in the alternative channel until a new failure be detected.

- Finally, we plan to verify the correctness of other fault tolerant patterns used in real projects in industry. Especially, those patterns used by our partner, Embraer. In

particular, there are several alternatives for the implementation of the homogeneous redundancy and triple modular redundancy.

Bibliography

[1] The SCADE suite. http://www.esterel-technologies.com/products/scade-suite [Online; accessed on 25th February 2012].

[2] Ashraf Armoush. *Design Patterns for Safety-Critical Embedded Systems*. Dissertation, Embedded Software Laboratory - RWTH Aachen University, 2010.

[3] Ralph J. Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction (Texts in Computer Science)*. Springer, 1998.

[4] Patrice Bremond-Gregoire, Insup Lee, and Richard Gerber. ACSR: An algebra of communicating shared resources with dense time and priorities. *Lecture Notes in Computer Science*, 715, 1993.

[5] Ricky W. Butler and Ben L. Di Vito. Formal design and verification of a reliable computing platform for real-time control (phase 2 results). NASA Technical Memorandum 104196, 1992.

[6] Ricky W. Butler, Ben L. Di Vito, and C. Michael Holloway. Formal design and verification of a reliable computing platform for real-time control (phase 3 results). NASA Technical Memorandum 109140, 1994.

[7] Albert Camilleri, Michael J. C. Gordon, and Tom Melham. Hardware verification using higher-order logic. In Dominique Borrione, editor, *Proceedings of the IFIP WG 10.2 Working Conference on From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. North-Holland, 1987.

[8] Samar Dajani-Brown, Darren Cofer, and Amar Bouali. Formal verification of an avionics sensor voter using scade. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 381–386. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30206-3_3.

[9] Joabe Bezerra de Jesus Júnior. Design e Validação Formal de Sistemas de Controle de Voo Fly-By-Wire. Master's thesis, Universidade Federal de Pernambuco, Recife, 2009.

[10] Diego Machado Dias and Juliano Manabu Iyoda. Behavioural preservation in fault tolerant patterns. In *Proceedings of the 14th Brazilian conference on Formal*

Methods: foundations and Applications, SBMF'11, pages 156–171, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Bruce Powel Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Professional, 2002.

[12] M. Goldsmith. *FDR: User Manual and Tutorial, version 2.77*. Formal Systems (Europe) Ltd, August 2001.

[13] Michael J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, The Computer Laboratory, University of Cambridge, 1985.

[14] Michael J. C. Gordon. Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North-Holland, 1986.

[15] Michael J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[16] Rachid Hadjidj, Hanifa Boucheneb, and Drifa Hadjidj. Zenoness detection and timed model checking for real time systems. In *Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS)*, 2007.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[18] Keith Hanna and Neil Daeche. Specification and Verification using Higher-Order Logic: A Case Study. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, 1986.

[19] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[20] International Electrotechnical Commission. *IEC 61131-3 Ed. 1.0 en:1993: Programmable controllers — Part 3: Programming languages*. International Electrotechnical Commission, pub-IEC:adr, 1993.

[21] Jeffrey J. Joyce. More reasons why higher-order logic is a good formalism for specifying and verifying hardware. Technical report, Vancouver, BC, Canada, Canada, 1990.

[22] N.M. Karayanakis. *Advanced System Modelling and Simulation With Block Diagram Languages*. CRC Press, 1995.

[23] Leroy Keith. Advisory Circular - System Design and Analysis, 25.1309-1A, 1988.

[24] Kathryn Kemp. Formal methods specification and verification guidebook for software and computer systems. volume i: Planning and technology insertion. Technical Report NASA/TP-98-208193, National Aeronautics and Space Administration, NASA Office of Safety and Mission Assurance, Washington D.C., December 1998.

[25] Israel Koren and C. Mani Krishna. *Fault Tolerant Systems*. Morgan Kaufmann Pblishers Inc., San Francisco, CA, USA, 2007.

[26] Mathworks. Matlab/simulink, 2011. http://www.mathworks.com [Online; accessed on 25th February 2012].

[27] P. Michael Melliar-Smith and John Rushby. The Enhanced HDM system for specification and verification. In *VerkShop III*, pages 41–43, Watsonville, CA, 1985.

[28] Michael Norrish and Konrad Slind. *The HOL System DESCRIPTION*. HOL4 Manual, 1988.

[29] Michael Norrish and Konrad Slind. *The HOL System LOGIC*. HOL4 Manual, 1988.

[30] Michael Norrish and Konrad Slind. *The HOL System REFERENCE*. HOL4 Manual, 1988.

[31] Michael Norrish and Konrad Slind. *The HOL System TUTORIAL*. HOL4 Manual, 1988.

[32] Katsuhiko Ogata. *Modern Control Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2001.

[33] Y. Orlarey, D. Fober, and S. Letz. An algebraic approach to block diagram constructions. In GMEM, editor, *Actes des Journées d'Informatique Musicale JIM2002, Marseille*, pages 151–158, 2002.

[34] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification of fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[35] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[36] Mark Saaltink. The Z/EVES System. In *ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, pages 72–85, London, UK, 1997. Springer-Verlag.

[37] Oleg Sokolsky, Insup Lee, and Hanêne Ben-Abdallah. Specification and analysis of real-time systems with PARAGON. *Annals of Software Engineering 7(1), 211-234*, 1999.

[38] Oleg Sokolsky, Mohamed F. Younis, Insup Lee, Hee-Hwan Kwak, and Jeffrey X. Zhou. Verification of the redundancy management system for space launch vehicle: A case study. In *IEEE Real Time Technology and Applications Symposium*, pages 220–229, 1998.

[39] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[40] Ben L. Di Vito and Ricky W. Butler. Formal Techniques for Synchronized Fault-Tolerant Systems. In C. E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications—3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 163–188. Springer-Verlag, Vienna, Austria, September 1992.

[41] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell II. Formal design and verification of a reliable computing platform for real-time control (phase 1 results). NASA Technical Memorandum 102716, 1990.

[42] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell II. High level design proof of a reliable computing platform. In *In Dependable Computing for Critical Applications 2, Dependable Computing and Fault-Tolerant Systems*, pages 279–306. Springer Verlag, 1992.

[43] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, 1990.

[44] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[45] C.J. Zarowski. *An Introduction to Numerical Analysis for Electrical and Computer Engineers*. John Wiley & Sons, 2004.

Appendices

A

Specification and Case Study: Proof Scripts

This chapter contains all proof scripts in HOL4, including the case study.

```
(*-------------------------------------------------------------------------
Definitions and theorems to support composition of blocks and realiability
patterns
-------------------------------------------------------------------------*)


quietdec := true;
show_assums := true;

map load ["metisLib", "optionLib", "combinTheory", "optionTheory",
          "realLib", "realTheory", "realSimps", "RealArith"];
open HolKernel
     Parse
     boolLib
     bossLib
     metisLib
     optionLib
     arithmeticTheory
     combinTheory
     optionTheory
     pairTheory
     realLib
     realTheory
     realSimps
     RealArith
     PairRules;
quietdec := false;


(*-------------------------------------------------------------------------
 Start new theory "block"
-------------------------------------------------------------------------*)
```

```
val _ = new_theory "block";

fun t() = (show_types := not(!show_types));
val kill = (fn theorem => K ALL_TAC theorem);


(*---------------------------------------------------------------------------
In this model, we assume that:

1. Designs are correct
2. Systems may break from fatigue: a random error
3. There are 2 possible sources of random errors: the environment and
   our own system


In this setting, signals are modeled as functions from time (natural number)
to 'a option. The type 'a option is a lift of 'a with the value NONE (bottom)
that represents an error. A regular value v is represented by SOME(v).
Example. An num option value can be NONE, SOME(0), SOME(200), etc.
Either the environment can provide NONE signals or our own system.

Once a signal carries NONE, it is carried over to the rest of
the system, i.e. a NONE signal is never transformed into SOME(_).

---------------------------------------------------------------------------*)

(*---------------------------------------------------------------------------
COMB
Combinational (i.e. zero-delay) component with input inp and output out
which computes a function f.

If the input is NONE, then the output is also NONE.
Otherwise, it outputs SOME(f t THE(inp t)) --- this prevents f to
introduce NONE in a valid input as f:'a -> 'b (not 'a option -> 'b option).
---------------------------------------------------------------------------*)
val COMB_def =
   Define `COMB f (inp:num->'a option,out:num->'b option) =
             !t:num. out t = if IS_NONE(inp t) then NONE
                             else SOME(f t (THE(inp t)))`;


(*---------------------------------------------------------------------------
DEL
Polymorphic delay component
---------------------------------------------------------------------------*)
val DEL_def = Define `DEL d (inp:num->'a option,out) = !t. out(t+d) = inp t`;


(*---------------------------------------------------------------------------
ERROR
The ERROR box takes as input a boolean function e that decides if
an error is to be introduced or not. If the input is NONE, it returns NONE
regardless e.
```

```
The decision function e takes as input the current time.
-----------------------------------------------------------------------------*)
val ERROR_def =
   Define `ERROR (e:num->bool) (inp:num->'a option,out) =
              !t:num. out t = if e(t) then NONE else (inp t)`;


(*-----------------------------------------------------------------------------
BLOCK
A block is a Simulink system (it represents a network of subsystems).
A block computes f with a certain delay and with possible random errors.
-----------------------------------------------------------------------------*)
val BLOCK_def =
   Define `BLOCK d e f (inp:num->'a option,out) =
     ?out1 out2. ERROR e (inp,out1) /\
                 COMB f (out1,out2) /\
                 DEL d (out2,out)`;


(*-----------------------------------------------------------------------------
SYSTEM
A system computes f with a certain delay and possible error.
A system is not necessarily atomic. It may be the composition of systems
and blocks. This is our specification.
-----------------------------------------------------------------------------*)
val SYSTEM_def =
   Define `SYSTEM (d:num) e f (inp: num->'a option, out: num->'b option) =
        !t. out (t+d) = if (IS_NONE (inp t) \/ e(t))
                        then NONE
                        else SOME(f t (THE(inp t)))`;


(*-----------------------------------------------------------------------------
BLOCK_IMPL_SYSTEM
A block is an implementation of a system.
-----------------------------------------------------------------------------*)
val BLOCK_IMPL_SYSTEM =
   Q.store_thm(
      "BLOCK_IMP_SYSTEM",
  ` !d e f inp out. BLOCK d e f (inp,out) ==> SYSTEM d e f (inp,out)`,
       PURE_REWRITE_TAC [BLOCK_def, SYSTEM_def, ERROR_def, COMB_def, DEL_def]
       THEN REPEAT STRIP_TAC
       THEN Cases_on `e t`
       THEN ASM_SIMP_TAC bool_ss [IS_NONE_DEF]);



(*-----------------------------------------------------------------------------
BUS
A bus convert a pair ('a option, 'b option) into a pair
('a option, b' option) option.

MUX undoes what BUS does.
-----------------------------------------------------------------------------*)
```

```
val BUS_def =
   Define `BUS ((inp1: num->'a option,inp2 : num-> 'b option),out) =
             !t. out t = if (IS_NONE(inp1 t) /\ IS_NONE(inp2 t)) then NONE
                         else SOME(inp1 t, inp2 t)`;

val MUX_def = Define
    `MUX (inp:num->('a option # 'b option) option,(out1,out2)) =
         !t. (out1 t =
             if IS_NONE (inp t) then NONE
                else FST (THE (inp t))) /\
             (out2 t =
             if IS_NONE (inp t) then NONE
                else SND (THE (inp t)))`;


(*---------------------------------------------------------------------------
MONITOR
This function instantiates "f" on the definition of HR.
This function takes as input a type ('a option # 'b option), and assumes
that (NONE,NONE) does not happen. This assumption is OK, since this
function is used in a COMB just after BUS, which eliminates this case.
---------------------------------------------------------------------------*)
val MONITOR_def =
    Define `MONITOR (t: num) (inp:('a option # 'a option)) =
                if IS_SOME(FST(inp)) then THE(FST(inp))
                else THE(SND(inp))`;



(*---------------------------------------------------------------------------
HR
Homogeneous Redundancy (HR) Pattern
Version 1.0: a monitor that fails
---------------------------------------------------------------------------*)

val HR_def =
    Define `HR d e s1 s2 (inp:num->('a option# 'a option) option, out) =
             ?outsys1 outsys2 outbus inpsys1 inpsys2.
                               MUX(inp,(inpsys1,inpsys2)) /\
                               s1(inpsys1, outsys1) /\
                               s2(inpsys2, outsys2) /\
                               BUS((outsys1,outsys2),outbus) /\
                               BLOCK d e MONITOR (outbus,out)`;



(*---------------------------------------------------------------------------
Few IS_NONE_IS_SOME lemmas
---------------------------------------------------------------------------*)
val IS_NONE_IS_SOME = Q.store_thm("IS_NONE_IS_SOME",
              `!x. IS_NONE(x) = ~(IS_SOME(x))`,
              RW_TAC arith_ss [IS_NONE_DEF,IS_SOME_DEF]);
```

```
val IS_NONE_IS_SOME2 = Q.store_thm("IS_NONE_IS_SOME2",
             `!x. (IS_SOME(x) = ~(IS_NONE(x)))`,
             METIS_TAC [IS_NONE_IS_SOME]);

val IS_NONE_IS_SOME3 = Q.store_thm("IS_NONE_IS_SOME3",
         `!x. (IS_SOME(x) ==> ~(x = NONE))`,
         METIS_TAC [IS_NONE_DEF,IS_SOME_DEF]);



(*---------------------------------------------------------------------------
EHR
The error of an HR system: it fails whenever both (e1 and e2 fails) or
if the monitor fails at t+d.

FHR
The functionality of an HR system implements a switch-to-backup policy.
---------------------------------------------------------------------------*)
val EHR_def = Define `EHR e1 e2 em (d:num) inp t =
            em(t+d) \/
            e1(t) /\ e2(t) \/
            e1(t) /\ IS_NONE(SND(THE(inp t))) \/
            e2(t) /\ IS_NONE(FST(THE(inp t))) \/
            IS_NONE(SND(THE(inp t))) /\ IS_NONE(FST(THE(inp t)))`;

(* first version of FHR *)
val FHR_def = Define `
    FHR f e (t:num) (inp:('a option # 'a option)) =
        if IS_NONE(FST(inp)) \/ e(t) then (f t)(THE(SND(inp)))
                                     else (f t)(THE(FST(inp)))`;
(* dissertation version of FHR *)
val FHR_dissertation_def = Define `
    FHR_dissertation f e (t:num) (inp:('a option # 'a option)) =
        if IS_SOME(FST(inp)) /\ ~e(t) then (f t)(THE(FST(inp)))
                                      else (f t)(THE(SND(inp)))`;

val FHR_EQUALS_FHR_dissertation = Q.store_thm(
      "FHR_EQUALS_FHR_dissertation",
      `FHR_dissertation f e t i = FHR f e t i`,
      RW_TAC arith_ss [FHR_def,FHR_dissertation_def,IS_NONE_IS_SOME,
                       IS_NONE_IS_SOME2, IS_NONE_IS_SOME3]
      THEN METIS_TAC []);



(*---------------------------------------------------------------------------
HR_IMPL_SYSTEM
The homogeneous redundancy for the function f implements a system
that computes f.
The delay and functionality of each replica is the same.
The error functions are different.
---------------------------------------------------------------------------*)
val HR_IMPL_SYSTEM = Q.store_thm("HR_IMPL_SYSTEM",`
```

```
    !dm em d e1 e2 f inp out.
    HR dm em (SYSTEM d e1 f) (SYSTEM d e2 f) (inp ,out) ==>
     SYSTEM (d+dm) (EHR e1 e2 em d inp) (FHR f e1) (inp,out)`,
     PURE_REWRITE_TAC [HR_def, SYSTEM_def, BLOCK_def,
                       BUS_def, MONITOR_def, EHR_def, FHR_def,
                       ERROR_def, COMB_def, DEL_def, MUX_def]
     THEN REPEAT STRIP_TAC THEN
     ASM_SIMP_TAC bool_ss [] THEN
     Cases_on `IS_NONE(inp t)` THEN
     Cases_on `em (t+d)` THEN
     Cases_on `e1 t` THEN
     Cases_on `e2 t` THEN
     Cases_on `IS_NONE (FST(THE(inp t)))` THEN
     Cases_on `IS_NONE (SND(THE(inp t)))` THEN
     ASM_SIMP_TAC bool_ss [IS_SOME_DEF, FST, SND, THE_DEF, IS_NONE_DEF, ADD_ASSOC]);


(*---------------------------------------------------------------------------
HetR
Heterogeneous Redundancy (HetR) Pattern
---------------------------------------------------------------------------*)
val EHetR_def = Define `EHetR e1 e2 em (d:num) inp t =
       em(t+d) \/
       e1(t) /\ e2(t) \/
       e1(t) /\ IS_NONE(SND(THE(inp t))) \/
       e2(t) /\ IS_NONE(FST(THE(inp t))) \/
       IS_NONE(SND(THE(inp t))) /\ IS_NONE(FST(THE(inp t)))`;

(* the dissertation uses E for EHR and EHetR. It is ok because they are the same *)
val EHR_EQUALS_EHetR =
    Q.store_thm("EHR_EQUALS_EHetR",
         `!e1 e2 em d inp t. ((EHR e1 e2 em d inp t) = (EHetR e1 e2 em d inp t))`,
         METIS_TAC [EHR_def,EHetR_def]);


val FHetR_def = Define
   `FHetR f1 f2 e1 (t:num) inp =  if IS_NONE(FST(inp)) \/ e1(t)
                                  then (f2 t)(THE(SND(inp)))
                                  else (f1 t)(THE(FST(inp)))`;

(* dissertation version of FHetR *)
val FHetR_dissertation_def = Define
   `FHetR_dissertation f1 f2 e1 (t:num) inp =  if IS_SOME(FST(inp)) /\ ~e1(t)
                                  then (f1 t)(THE(FST(inp)))
                                  else (f2 t)(THE(SND(inp)))`;

val FHetR_EQUALS_FHetR_dissertation = Q.store_thm(
      "FHetR_EQUALS_FHR_dissertation",
      `!f1 f2 e1 t i. ((FHetR_dissertation f1 f2 e1 t i) = (FHetR f1 f2 e1 t i))`,
      RW_TAC arith_ss [FHetR_def,FHetR_dissertation_def,IS_NONE_IS_SOME,
                       IS_NONE_IS_SOME2, IS_NONE_IS_SOME3]
      THEN METIS_TAC []);
```

```
val HetR_def = Define `
    HetR d e sys1 sys2 (inp:num->('a option # 'b option) option,out) =
       ?inpsys1 inpsys2 outsys1 outsys2 outbus.
          MUX(inp,(inpsys1,inpsys2)) /\
          sys1(inpsys1,outsys1) /\
          sys2(inpsys2,outsys2) /\
          BUS((outsys1,outsys2),outbus) /\
          BLOCK d e MONITOR (outbus, out)`;


val HETR_IMPL_SYSTEM = Q.store_thm("HETR_IMPL_SYSTEM",`
    !f1 f2 e1 e2 d dm em inp out.
    HetR dm em (SYSTEM d e1 f1) (SYSTEM d e2 f2) (inp, out) ==>
         SYSTEM (d+dm)
         (EHetR e1 e2 em d inp)
         (FHetR f1 f2 e1)
         (inp, out)`,
     PURE_REWRITE_TAC [SYSTEM_def, BLOCK_def, BUS_def, MONITOR_def, EHetR_def,
                       ERROR_def, COMB_def, DEL_def, HetR_def, FHetR_def,
                       MUX_def ] THEN
     REPEAT STRIP_TAC THEN
     ASM_SIMP_TAC bool_ss [] THEN
     Cases_on `IS_NONE(inp t)` THEN
     Cases_on `em (t+d)` THEN
     Cases_on `e1 t` THEN
     Cases_on `e2 t` THEN
     Cases_on `IS_NONE (FST(THE(inp t)))` THEN
     Cases_on `IS_NONE (SND(THE(inp t)))` THEN
     ASM_SIMP_TAC bool_ss [IS_SOME_DEF, FST, SND, THE_DEF, IS_NONE_DEF, ADD_ASSOC]
     THEN RES_TAC);


(*-----------------------------------------------------------------------------
 TMR
 Triple Modular Redundancy Pattern
-----------------------------------------------------------------------------*)
val TBUS_def = Define `
    TBUS (inp1: num->'a option, inp2: num->'b option, inp3: num->'c option,out) =
            !t. out t = if (IS_NONE(inp1 t) /\
                            IS_NONE(inp2 t) /\
                            IS_NONE(inp3 t))
                            then NONE
                            else SOME(inp1 t, inp2 t, inp3 t)`;


val TMUX_def = Define
   `TMUX (inp:num->('a option # 'b option #'c option) option,(out1,out2,out3)) =
        !t. (out1 t =
            if IS_NONE (inp t) \/ IS_NONE (FST(THE(inp t)))
               then NONE
               else FST (THE (inp t))) /\
            (out2 t =
            if IS_NONE (inp t) \/ IS_NONE (FST(SND(THE(inp t))))
               then NONE
```

```
              else FST(SND(THE (inp t)))) /\
            (out3 t =
            if IS_NONE (inp t) \/ IS_NONE (SND(SND(THE(inp t))))
              then NONE
              else SND(SND(THE(inp t))))`;

val VOTER_def = Define `
    VOTER (t:num) inp =
     if ~IS_NONE(FST(inp)) /\ ~IS_NONE(FST(SND(inp))) /\ ~IS_NONE(SND(SND(inp)))
        then (1/3)*(THE(FST(inp)) + THE(FST(SND(inp))) + THE(SND(SND(inp))))
     else if ~IS_NONE(FST(inp)) /\ ~IS_NONE(FST(SND(inp)))
        then (1/2)*(THE(FST(inp)) + THE(FST(SND(inp))))
     else if ~IS_NONE(FST(inp)) /\ ~IS_NONE(SND(SND(inp)))
        then (1/2)*(THE(FST(inp)) + THE(SND(SND(inp))))
     else if ~IS_NONE(FST(SND(inp))) /\ ~IS_NONE(SND(SND(inp)))
        then (1/2)*(THE(FST(SND(inp))) + THE(SND(SND(inp))))
     else if ~IS_NONE(FST(inp))
        then (THE(FST(inp)))
     else if ~IS_NONE(FST(SND(inp)))
        then (THE(FST(SND(inp))))
     else THE(SND(SND(inp)))`;

val TMR_def =  Define `
    TMR d e S1 S2 S3
        (inp:num->('a option # 'a option # 'a option) option, out) =
        ? inpsys1 inpsys2 inpsys3 outsys1 outsys2 outsys3
          outbus.
                      TMUX(inp,(inpsys1, inpsys2, inpsys3)) /\
                      S1 (inpsys1, outsys1) /\
                      S2 (inpsys2, outsys2) /\
                      S3 (inpsys3, outsys3) /\
                      TBUS (outsys1,outsys2,outsys3,outbus) /\
                      BLOCK d e VOTER (outbus,out)`;

val ETMR_def = Define `
    ETMR e1 e2 e3 ev (d:num) inp t = ev (t+d) \/
                            e1(t) /\ e2(t) /\ e3(t) \/
         IS_NONE(FST(THE(inp t))) /\ e2(t) /\ e3(t) \/
    IS_NONE(FST(SND(THE(inp t)))) /\ e1(t) /\ e3(t) \/
    IS_NONE(SND(SND(THE(inp t)))) /\ e1(t) /\ e2(t) \/
         IS_NONE(FST(THE(inp t))) /\
    IS_NONE(FST(SND(THE(inp t)))) /\ e3(t)          \/
         IS_NONE(FST(THE(inp t))) /\
    IS_NONE(SND(SND(THE(inp t)))) /\ e2(t)          \/
    IS_NONE(FST(SND(THE(inp t)))) /\
    IS_NONE(SND(SND(THE(inp t)))) /\ e1(t)          \/
        IS_NONE(FST(THE(inp t)))  /\
   IS_NONE(FST(SND(THE(inp t))))  /\
   IS_NONE(SND(SND(THE(inp t))))`;


val FTMR_def = Define `
    FTMR e1 e2 e3 f (t:num) inp =
    if ~IS_NONE(FST(inp)) /\ ~IS_NONE(FST(SND(inp))) /\ ~IS_NONE(SND(SND(inp)))
```

```
        /\ ~e1(t) /\ ~e2(t) /\ ~e3(t)
       then (1/3)*((f t (THE(FST(inp)))) + (f t (THE(FST(SND(inp))))) +
                   (f t (THE(SND(SND(inp))))))
    else if ~IS_NONE(FST(inp)) /\ ~IS_NONE(FST(SND(inp))) /\
               ~e1(t) /\ ~e2(t)
               then (1/2)*((f t (THE(FST(inp)))) + (f t (THE(FST(SND(inp))))))
    else if ~IS_NONE(FST(inp)) /\ ~IS_NONE(SND(SND(inp))) /\
               ~e1(t) /\ ~e3(t)
               then (1/2)*((f t (THE(FST(inp)))) + (f t (THE(SND(SND(inp))))))
    else if ~IS_NONE(FST(SND(inp))) /\ ~IS_NONE(SND(SND(inp))) /\
               ~e2(t) /\ ~e3(t)
               then (1/2)*((f t (THE(FST(SND(inp))))) + (f t (THE(SND(SND(inp))))))
    else if ~IS_NONE(FST(inp)) /\ ~e1(t)
               then (f t (THE(FST(inp))))
    else if ~IS_NONE(FST(SND(inp))) /\ ~e2(t)
               then (f t (THE(FST(SND(inp)))))
    else (f t (THE(SND(SND(inp)))))`;

(* FTMR as published in the dissertation *)
val FTMR_dissertation_def = Define `
    FTMR_dissertation e1 e2 e3 f t inp =
    if IS_SOME(FST(inp)) /\ IS_SOME(FST(SND(inp))) /\ IS_SOME(SND(SND(inp)))
        /\ ~e1(t) /\ ~e2(t) /\ ~e3(t)
       then (1/3)*((f t (THE(FST(inp)))) + (f t (THE(FST(SND(inp))))) +
                   (f t (THE(SND(SND(inp))))))
    else if IS_SOME(FST(inp)) /\ IS_SOME(FST(SND(inp))) /\
               ~e1(t) /\ ~e2(t)
               then (1/2)*((f t (THE(FST(inp)))) + (f t (THE(FST(SND(inp))))))
    else if IS_SOME(FST(inp)) /\ IS_SOME(SND(SND(inp))) /\
               ~e1(t) /\ ~e3(t)
               then (1/2)*((f t (THE(FST(inp)))) + (f t (THE(SND(SND(inp))))))
    else if IS_SOME(FST(SND(inp))) /\ IS_SOME(SND(SND(inp))) /\
               ~e2(t) /\ ~e3(t)
               then (1/2)*((f t (THE(FST(SND(inp))))) + (f t (THE(SND(SND(inp))))))
    else if IS_SOME(FST(inp)) /\ ~e1(t)
               then (f t (THE(FST(inp))))
    else if IS_SOME(FST(SND(inp))) /\ ~e2(t)
               then (f t (THE(FST(SND(inp)))))
    else (f t (THE(SND(SND(inp)))))`;

val FTMR_EQUALS_FTMR_dissertation = Q.store_thm(
   "FTMR_EQUALS_FTMR_dissertation",
   `!e1 e2 e3 f t i. ((FTMR_dissertation e1 e2 e3 f t i) = (FTMR e1 e2 e3 f t i))`,
      RW_TAC arith_ss [FTMR_def,FTMR_dissertation_def,IS_NONE_IS_SOME,
                       IS_NONE_IS_SOME2, IS_NONE_IS_SOME3]
      THEN METIS_TAC []);


val TMR_IMPL_SYSTEM = Q.store_thm("TMR_IMPL_SYSTEM",`
  !dv ev d e1 e2 e3 f inp out.
  TMR dv ev (SYSTEM d e1 f) (SYSTEM d e2 f) (SYSTEM d e3 f) (inp, out) ==>
  SYSTEM (dv+d) (ETMR e1 e2 e3 ev d inp) (FTMR e1 e2 e3 f) (inp,out)`,
  PURE_REWRITE_TAC [TMR_def, SYSTEM_def, VOTER_def, ETMR_def,
                    BLOCK_def, ERROR_def, DEL_def, COMB_def, TBUS_def,
```

```
                    TMUX_def, FTMR_def ] THEN
  REPEAT STRIP_TAC THEN
  `!t d dv. out (t+dv+d) = out(t+d+dv)` by RW_TAC arith_ss [] THEN
  ASM_SIMP_TAC bool_ss [] THEN
  Cases_on `IS_NONE(inp t)` THEN
  Cases_on `e1 t` THEN
  Cases_on `e2 t` THEN
  Cases_on `e3 t` THEN
  Cases_on `ev (t+d)` THEN
  Cases_on ` IS_NONE(FST(THE(inp t)))` THEN
  Cases_on `IS_NONE(FST(SND(THE(inp t))))` THEN
  Cases_on `IS_NONE(SND(SND(THE(inp t))))` THEN
  ASM_SIMP_TAC bool_ss [FST, SND, THE_DEF, IS_NONE_DEF, IS_SOME_DEF, ADD_ASSOC]);



(*------------------ Compositionality of the theorems ------------------------*)

(*---------------------------- HOMOGENEOUS REDUNDANCY -------------------------*)
val LEMMA_HR_COMPOSITIONAL = Q.store_thm("LEMMA_HR_COMPOSITIONAL",
   `!I1 I2 d e1 e2 f dm em inp out.
    (!inp out. I1 (inp,out) ==> SYSTEM d e1 f (inp,out)) /\
    (!inp out. I2 (inp,out) ==> SYSTEM d e2 f (inp,out))
   ==> (HR dm em I1 I2 (inp,out)
       ==> (HR dm em (SYSTEM d e1 f) (SYSTEM d e2 f) (inp,out)))`,
  PURE_REWRITE_TAC [HR_def,SYSTEM_def] THEN REPEAT STRIP_TAC THEN
  Q.EXISTS_TAC `outsys1` THEN Q.EXISTS_TAC `outsys2` THEN Q.EXISTS_TAC `outbus`
  THEN Q.EXISTS_TAC `inpsys1` THEN Q.EXISTS_TAC `inpsys2` THEN
  ASM_SIMP_TAC bool_ss [IS_NONE_DEF]);

val HR_COMPOSITIONAL = Q.store_thm("HR_COMPOSITIONAL",
   `!I1 I2 d e1 e2 f dm em inp out.
    (!inp out. I1 (inp,out) ==> SYSTEM d e1 f (inp,out)) /\
    (!inp out. I2 (inp,out) ==> SYSTEM d e2 f (inp,out))
    ==> (HR dm em I1 I2 (inp,out)
        ==> SYSTEM (d+dm) (EHR e1 e2 em d inp) (FHR f e1) (inp,out))`,
   PROVE_TAC [LEMMA_HR_COMPOSITIONAL, HR_IMPL_SYSTEM]);

(* as published in the dissertation *)
val HR_COMPOSITIONAL2 = Q.store_thm("HR_COMPOSITIONAL2",
   `!I1 I2 d e1 e2 f dm em inp out.
    (!inp out. I1 (inp,out) ==> SYSTEM d e1 f (inp,out)) /\
    (!inp out. I2 (inp,out) ==> SYSTEM d e2 f (inp,out))
    ==> (HR dm em I1 I2 (inp,out)
        ==> SYSTEM (d+dm) (EHR e1 e2 em d inp) (FHR_dissertation f e1) (inp,out))`,
   METIS_TAC [HR_COMPOSITIONAL, FHR_EQUALS_FHR_dissertation]);


(*---------------------- TRIPLE MODULAR REDUNDANCY --------------------------*)
val LEMMA_TMR_COMPOSITIONAL = Q.store_thm("LEMMA_TMR_COMPOSITIONAL",
  ` !I1 I2 I3 e1 e2 e3 f d dv ev inp out.
    (!inp out. I1 (inp,out) ==> SYSTEM d e1 f (inp,out)) /\
    (!inp out. I2 (inp,out) ==> SYSTEM d e2 f (inp,out)) /\
```

```
    (!inp out. I3 (inp,out) ==> SYSTEM d e3 f (inp,out))
   ==> (TMR dv ev I1 I2 I3 (inp,out)
   ==> (TMR dv ev (SYSTEM d e1 f) (SYSTEM d e2 f) (SYSTEM d e3 f) (inp,out)))`,
  PURE_REWRITE_TAC [TMR_def,SYSTEM_def] THEN REPEAT STRIP_TAC THEN
  Q.EXISTS_TAC `inpsys1` THEN Q.EXISTS_TAC `inpsys2` THEN
  Q.EXISTS_TAC `inpsys3` THEN Q.EXISTS_TAC `outsys1` THEN
  Q.EXISTS_TAC `outsys2` THEN Q.EXISTS_TAC `outsys3` THEN
  Q.EXISTS_TAC `outbus` THEN ASM_SIMP_TAC bool_ss [IS_NONE_DEF]);

val TMR_COMPOSITIONAL = Q.store_thm("TMR_COMPOSITIONAL",
   `!f I1 I2 I3 e1 e2 e3 d dv ev inp out.
    (!inp out. I1 (inp,out) ==> SYSTEM d e1 f (inp,out)) /\
    (!inp out. I2 (inp,out) ==> SYSTEM d e2 f (inp,out)) /\
    (!inp out. I3 (inp,out) ==> SYSTEM d e3 f (inp,out))
     ==> (TMR dv ev I1 I2 I3 (inp,out)
       ==> SYSTEM (dv+d) (ETMR e1 e2 e3 ev d inp) (FTMR e1 e2 e3 f) (inp,out))`,
     REPEAT STRIP_TAC THEN
     ASSUME_TAC LEMMA_TMR_COMPOSITIONAL THEN ASSUME_TAC TMR_IMPL_SYSTEM THEN
     RES_TAC THEN ASM_SIMP_TAC bool_ss []);

(* as published in the dissertation *)
val TMR_COMPOSITIONAL2 = Q.store_thm("TMR_COMPOSITIONAL2",
   `!f I1 I2 I3 e1 e2 e3 d dv ev inp out.
    (!inp out. I1 (inp,out) ==> SYSTEM d e1 f (inp,out)) /\
    (!inp out. I2 (inp,out) ==> SYSTEM d e2 f (inp,out)) /\
    (!inp out. I3 (inp,out) ==> SYSTEM d e3 f (inp,out))
     ==> (TMR dv ev I1 I2 I3 (inp,out)
       ==> SYSTEM (dv+d) (ETMR e1 e2 e3 ev d inp) (FTMR_dissertation e1 e2 e3 f) (inp,out))`,
     METIS_TAC [TMR_COMPOSITIONAL, FTMR_EQUALS_FTMR_dissertation]);


(*---------------------------- HETEROGENOUS REDUNDANCY -----------------------*)
val LEMMA_HETR_COMPOSITIONAL = Q.store_thm("LEMMA_HETR_COMPOSITIONAL",
  ` !I1 I2 d e1 e2 f1 f2 dm em inp out.
    (!inp out. I1 (inp,out) ==> SYSTEM d e1 f1 (inp,out)) /\
    (!inp out. I2 (inp,out) ==> SYSTEM d e2 f2 (inp,out))
   ==> (HetR dm em I1 I2 (inp,out)
       ==> (HetR dm em (SYSTEM d e1 f1) (SYSTEM d e2 f2) (inp,out)))`,
  PURE_REWRITE_TAC [HetR_def,SYSTEM_def] THEN REPEAT STRIP_TAC THEN
  Q.EXISTS_TAC `inpsys1` THEN Q.EXISTS_TAC `inpsys2` THEN
  Q.EXISTS_TAC `outsys1` THEN Q.EXISTS_TAC `outsys2` THEN
  Q.EXISTS_TAC `outbus` THEN
  PROVE_TAC []);

val HETR_COMPOSITIONAL = Q.store_thm("HETR_COMPOSITIONAL",
   `!I1 I2 d e1 e2 f1 f2 dm em inp out.
    (!inp out. I1 (inp,out) ==> SYSTEM d e1 f1 (inp,out)) /\
    (!inp out. I2 (inp,out) ==> SYSTEM d e2 f2 (inp,out))
     ==> (HetR dm em I1 I2 (inp,out)
     ==> SYSTEM (d+dm) (EHetR e1 e2 em d inp) (FHetR f1 f2 e1) (inp,out))`,
     PROVE_TAC [ LEMMA_HETR_COMPOSITIONAL, HETR_IMPL_SYSTEM]);


(* as published in the dissertation *)
```

```
val HETR_COMPOSITIONAL2 = Q.store_thm("HETR_COMPOSITIONAL2",
   `!I1 I2 d e1 e2 f1 f2 dm em inp out.
    (!inp out. I1 (inp,out) ==> SYSTEM d e1 f1 (inp,out)) /\
    (!inp out. I2 (inp,out) ==> SYSTEM d e2 f2 (inp,out))
     ==> (HetR dm em I1 I2 (inp,out)
     ==> SYSTEM (d+dm) (EHetR e1 e2 em d inp) (FHetR_dissertation f1 f2 e1) (inp,out))`,
     METIS_TAC [HETR_COMPOSITIONAL, FHetR_EQUALS_FHetR_dissertation]);


(*---------------------- Case Study: Elevator Controlling --------------------------*)

val LOW_PASS_FILTER_def = Define
    `low_pass_filter(z) = ((725/10000)*z+(725/10000)) / (z-8551/10000)`;
val GAIN_def = Define `Gain (k:real,inp) = k*inp`;
val SWITCH_TRESHOLD_def = Define
    `SwitchTreshold (k,t,inpA,inpB) =
    if (t >= k) then inpB else inpA`;
val NOT_def = Define `NOT (a:bool) = ~a`;
val SUM_def = Define `SUM (a:real,b) = a+b`;
val MULT_def = Define `MULT (a:real,b) = a*b`;
val B2REAL_def = Define `B2REAL(a) = if a then 1 else 0`;
val AND_def = Define `AND (x,y) = x /\ y`;
val COMPENSATOR_def =
    Define `compensator(z) =
     ((58/10000)*(z pow 2) - (115/10000)*z + (57/10000)) /
      ((884/10)*(z pow 2) + (68013/100)*z + 1)`;
val ELEVATOR_COMMAND_SHAPER_def =
    Define `ElevatorCommandShaper(x) =
             (123141465530820697062170457143 00 * (x pow 7) -
              375094022390379325508830389203000 * (x pow 6) -
              167358800680626785509416726371446 43 * (x pow 5) +
              407355932416064821116145878025717030 * (x pow 4) +
              725636386468605011754654040914866 6875 * (x pow 3) -
              956951678846153153605328254810072 18250 * (x pow 2) -
              725619650711678414606923796304400 00 * x +
              956910943628005569513595965731100000)/
              115752862795075259259310772398617186 0000`;
val ELEV_STATURATION_def = Define ` ElevSaturation (kmin,kmax,z) =
    if z > kmax then kmax else if z < kmin then kmin else z`;
val RATE_LIMITER_def =
    Define `RateLimiter(z:real) = z`;


val elevator_def = Define `
    elevator (t:num) (PitchRate, Flap, WOW,
                          LongSideStick, PitchRate_Voted) =
    let out_lpf = low_pass_filter(PitchRate) in
    let out_cpt = compensator(out_lpf) in
    let out_gfe = Gain(-150, out_cpt) in
    let out_gfc = Gain(-67, out_cpt) in
    let out_sth = SwitchTreshold (1/2, B2REAL(Flap),out_gfe,out_gfc) in
    let out_not = NOT(WOW) in
    let out_and = AND(out_not, PitchRate_Voted) in
    let out_rtl = RateLimiter(B2REAL(out_and)) in
```

```
    let out_mul = MULT(out_rtl, out_sth) in
    let out_ecs = ElevatorCommandShaper(LongSideStick) in
    let out_sum = SUM(out_ecs,out_mul) in
    let out_str = ElevSaturation(-25,25,out_sum) in out_str`;


val ELEVATOR_IMPL_SYSTEM = Q.store_thm ("ELEVATOR_IMPL_SYSTEM",
     `!d e inp out.
      BLOCK d e elevator (inp,out) ==>
      SYSTEM d e elevator (inp, out)`,
     PROVE_TAC [BLOCK_IMPL_SYSTEM]);

val ELEVATOR_HR_COMPLIANCE = Q.store_thm ("ELEVATOR_HR_COMPLIANCE",
   `!d dm e1 e2 em inp out.
    HR dm em (BLOCK d e1 elevator)
             (BLOCK d e2 elevator) (inp,out) ==>
             SYSTEM (d+dm) (EHR e1 e2 em d inp)
                           (FHR elevator e1) (inp,out)`,
   PROVE_TAC [ELEVATOR_IMPL_SYSTEM, HR_COMPOSITIONAL]);


(* A generic theorem for Elevator *)

val ELEVATOR_TMR_COMPLIANCE = Q.store_thm ("ELEVATOR_TMR_COMPLIANCE",
   `!d e1 e2 e3 ev dv inp out.
    TMR dv ev (BLOCK d e1 elevator)
              (BLOCK d e2 elevator)
              (BLOCK d e3 elevator) (inp,out) ==>
     SYSTEM (dv+d) (ETMR e1 e2 e3 ev d inp)
                   (FTMR e1 e2 e3 elevator) (inp,out)`,
    REPEAT STRIP_TAC THEN
    UNDISCH_TAC `` TMR dv ev (BLOCK d e1 elevator)
    (BLOCK d e2 elevator) (BLOCK d e3 elevator)(inp,out)``
    THEN
    ASSUME_TAC (IPSPECL [``elevator``,
                     ``(BLOCK d e1 elevator)``,
                     ``(BLOCK d e2 elevator)``,
                     ``(BLOCK d e3 elevator)``,
                     ``e1:num->bool``, ``e2:num->bool``,
                     ``e3:num->bool``, ``d:num``, ``dv:num``,
                     ``ev:num->bool``,
                     `` inp: num ->
                        ((real # bool # bool # real # bool) option #
                         (real # bool # bool # real # bool) option #
                         (real # bool # bool # real # bool) option) option ``,
                        ``out :num -> real option`` ] TMR_COMPOSITIONAL) THEN
     ASSUME_TAC ELEVATOR_IMPL_SYSTEM THEN
    ASM_SIMP_TAC bool_ss []);



(* A theorem for Elevator fully instantiated *)

val ELEVATOR_TMR_COMPLIANCE_INST = Q.store_thm ("ELEVATOR_TMR_COMPLIANCE_INST",
```

```
   `TMR dv ev (BLOCK d e1 elevator)
              (BLOCK d e2 elevator)
              (BLOCK d e3 elevator) (inp,out) ==>
     SYSTEM (dv+d) (ETMR e1 e2 e3 ev d inp)
                   (FTMR e1 e2 e3 elevator) (inp,out)`,
    ASSUME_TAC (IPSPECL [``elevator``,
                     ``(BLOCK d e1 elevator)``,
                     ``(BLOCK d e2 elevator)``,
                     ``(BLOCK d e3 elevator)``,
                     ``e1:num->bool``, ``e2:num->bool``,
                     ``e3:num->bool``, ``d:num``, ``dv:num``,
                     ``ev:num->bool``,
                     `` inp: num ->
                        ((real # bool # bool # real # bool) option #
                        (real # bool # bool # real # bool) option #
                        (real # bool # bool # real # bool) option) option ``,
                        ``out :num -> real option`` ] TMR_COMPOSITIONAL) THEN
     ASSUME_TAC ELEVATOR_IMPL_SYSTEM THEN
    ASM_SIMP_TAC bool_ss []);


val _ = export_theory();
```

B

Refinement Calculus: Proof Scripts

This chapter contains the proofs of the Chapter 4 discussed in Section refsec:Refinement.

```
(*-------------------------------------------------------------------
-------------------------------------------------------------------*)
quietdec := true;
show_assums := true;

map load ["metisLib", "optionLib", "combinTheory", "optionTheory"];
open HolKernel
     Parse
     boolLib
     bossLib
     metisLib
     optionLib
     arithmeticTheory
     combinTheory
     optionTheory
     pairTheory;

quietdec := false;

Hol_datatype `some_exp = COND | E01 | E02 | E03 | E04 | E05
                              | E06 | E07 | E08 | E09 | E10
                              | E11 | E12 | E13 | E14 | E15`;


Hol_datatype `exp = SOME_EXP of some_exp
                  | FF of exp list
                  | AVG of exp list
                  | ITE of exp => exp => exp`;

val AX1_def = new_axiom ("AX1",
```

```
              ``!f g h. FF [f ; FF [g;h]] = FF [FF [f;g]; h]``);

val AX2_def = new_axiom ("AX2",
              ``!f g. FF [f ; g] = FF [g ; f]``);

val AX3_def = new_axiom ("AX3",
              ``!f. FF [f ; f] = f``);

val AX4_def = new_axiom ("AX4",
              ``!(e:exp). IS_REF e e``);

val AX5_def = new_axiom ("AX5",
              ``!(e:exp) (f:exp) (g:exp).
                 (IS_REF e f) /\ (IS_REF f g) ==> IS_REF e g``);

val AX6_def = new_axiom ("AX6",
              ``!(e:exp) (f:exp).
                 (IS_REF e f) /\ (IS_REF f e) ==> (e=e)``);

val AX7_def = new_axiom ("AX7",
              ``!(e1:exp) (f1:exp) (e2:exp) (f2:exp).
                 (IS_REF e1 f1) /\ (IS_REF e2 f2)
                 ==> IS_REF (FF [e1;e2]) (ITE (SOME_EXP COND) f1 f2)``);

val AX6_def = new_axiom ("Ax5",
              ``!(es:exp list). IS_REF (FF es) (AVG es)``);

(* HetR *)
g `(IS_REF: exp -> exp -> bool)
   (FF [SOME_EXP E01; SOME_EXP E02])
   (ITE (SOME_EXP COND)
        (SOME_EXP E01)
        (SOME_EXP E02))`;
e(METIS_TAC [AX1_def,AX2_def,AX3_def,AX4_def,AX5_def,AX6_def,AX7_def]);


(* TMR passo 9 *)
g `(IS_REF: exp -> exp -> bool)
   (FF [SOME_EXP E01; FF [SOME_EXP E02; SOME_EXP E03]])
   (ITE (SOME_EXP COND)
        (SOME_EXP E01)
        (ITE (SOME_EXP COND)
             (SOME_EXP E02)
             (SOME_EXP E03)
        )
   )`;
e(METIS_TAC [AX1_def,AX2_def,AX3_def,AX4_def,AX5_def,AX6_def,AX7_def]);
val p9 = top_thm();


(* TMR passo 9 *)
g `(IS_REF: exp -> exp -> bool)
   (FF [FF[SOME_EXP E02; SOME_EXP E03]; FF [SOME_EXP E01;
    FF[SOME_EXP E02; SOME_EXP E03]]])
```

```
   (ITE (SOME_EXP COND)
        (AVG [SOME_EXP E02; SOME_EXP E03])
        (ITE (SOME_EXP COND)
             (SOME_EXP E01)
             (ITE (SOME_EXP COND)
                  (SOME_EXP E02)
                  (SOME_EXP E03))))`;
e(METIS_TAC [AX4_def,AX5_def,AX6_def,AX7_def,p9]);
val p12 = top_thm();


(* TMR passo 15 *)
g `(IS_REF: exp -> exp -> bool)
   (FF [FF [SOME_EXP E01; SOME_EXP E03];
    FF [FF[SOME_EXP E02; SOME_EXP E03]; FF [SOME_EXP E01;
    FF[SOME_EXP E02; SOME_EXP E03]]]])
   (ITE (SOME_EXP COND)
        (AVG [SOME_EXP E01; SOME_EXP E03])
        (ITE (SOME_EXP COND)
             (AVG [SOME_EXP E02; SOME_EXP E03])
             (ITE (SOME_EXP COND)
                  (SOME_EXP E01)
                  (ITE (SOME_EXP COND)
                       (SOME_EXP E02)
                       (SOME_EXP E03)))))`;
e(METIS_TAC [AX4_def,AX5_def,AX6_def,AX7_def,p12]);
val p15 = top_thm();


(* TMR passo 18 *)
g `(IS_REF: exp -> exp -> bool)
   (FF [FF [SOME_EXP E01; SOME_EXP E02];
    FF [FF [SOME_EXP E01; SOME_EXP E03];
    FF [FF[SOME_EXP E02; SOME_EXP E03];
    FF [SOME_EXP E01; FF[SOME_EXP E02; SOME_EXP E03]]]])
   (ITE (SOME_EXP COND)
        (AVG [SOME_EXP E01; SOME_EXP E02])
        (ITE (SOME_EXP COND)
             (AVG [SOME_EXP E01; SOME_EXP E03])
             (ITE (SOME_EXP COND)
                  (AVG [SOME_EXP E02; SOME_EXP E03])
                  (ITE (SOME_EXP COND)
                       (SOME_EXP E01)
                       (ITE (SOME_EXP COND)
                            (SOME_EXP E02)
                            (SOME_EXP E03))))))`;
e(METIS_TAC [AX4_def,AX5_def,AX6_def,AX7_def,p15]);
val p18 = top_thm();


(* TMR passo 21 *)
g `(IS_REF: exp -> exp -> bool)
   (FF [FF [SOME_EXP E01; FF [SOME_EXP E02; SOME_EXP E03]];
    FF [FF [SOME_EXP E01; SOME_EXP E02];
```

```
    FF [FF [SOME_EXP E01; SOME_EXP E03];
    FF [FF[SOME_EXP E02; SOME_EXP E03];
    FF [SOME_EXP E01; FF[SOME_EXP E02; SOME_EXP E03]]]]]])
   (ITE (SOME_EXP COND)
        (AVG [SOME_EXP E01; FF [SOME_EXP E02; SOME_EXP E03]])
   (ITE (SOME_EXP COND)
        (AVG [SOME_EXP E01; SOME_EXP E02])
        (ITE (SOME_EXP COND)
             (AVG [SOME_EXP E01; SOME_EXP E03])
             (ITE (SOME_EXP COND)
                  (AVG [SOME_EXP E02; SOME_EXP E03])
                  (ITE (SOME_EXP COND)
                       (SOME_EXP E01)
                       (ITE (SOME_EXP COND)
                            (SOME_EXP E02)
                            (SOME_EXP E03)))))))`;
e(METIS_TAC [AX4_def,AX5_def,AX6_def,AX7_def,p18]);
val p21 = top_thm();
```