Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Formal Specification and Analysis of a Redundancy Management System with TLA$^+$

Jan Täubrich

September 21, 2006

Institute of Computer Science and Applied Mathematics
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Prof. Reinhard v. Hanxleden

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Preface

I want to thank Claus Traulsen for his admirable patience, his helpful comments and for his thoroughness reviewing my thesis. I also want to thank Hagen Peters, who pointed out a lot of linguistic shortcomings in my writings.

Contents

List of Figures

List of Abbreviations

AFDX	Avionics Full Duplex Switched Ethernet
BAG	Bandwidth Allocation Gap
ES	End System
MCFL	Maximum Consecutive Frame Loss
MTF	Maximum Transient Frames
QoS	Quality of Service
RMA	Redundancy Management Algorithm
SNS	Sequence Number Skew
SNO	Sequence Number Offset
TLA	Temporal Logic of Actions
TLC	TLC Model Checker
VL	Virtual Link

1. Introduction

Every day our lives become more dependent on embedded systems, technology that is included in our environment. Most of these embedded systems cannot be recognized as computers in the ordinary sense. Nevertheless, they do not only include safety critical applications for automotive, railway, aircraft or medical devices. More and more mobile communication and also less observable parts of the everyday environment, ranging from intelligent fridges to smart clothes, include multiple embedded systems. About 99 percent of processors applied today are part of an embedded system [10].

The enormous growth of complexity of embedded systems forced the industry to explore new development techniques and tools to handle this complexity and to allow formal reasoning about the operation of these systems. Invented in the early eighties, *Synchronous Languages* like *Esterel* [8], *Signal* [20] and *Lustre* [16] were built on a solid mathematical framework combining perfect synchrony, determinism and concurrency. The synchronous approach divides time in discrete steps, which allows modeling of behavior as a series of states. This approach is suitable for discrete-time dynamical systems and synchronous logic, too, and hence is a good and abstract choice for a wide class of systems. This class includes most systems that occur in engineering disciplines, because it can be assumed that, even though as the system is generally continuous, a reaction to an input from environment can be computed faster than a new input occurs. Ideally a zero reaction delay is assumed. This hypothesis is called the *Synchrony Hypothesis*, first postulated by Berry [14].

While *Lustre* is declarative and well suited for specifying data-flow, *Esterel* is imperative and focuses on describing control-flow. The clean underlying mathematics even allows to develop systems partly written in *Esterel* and partly in *Lustre*, without loosing their soundness. From the early beginning verification was considered central aspect of synchronous languages and several model checking methods for *Esterel* and *Lustre* were developed [5]. Strong industrial support from *Airbus, Schneider Electric* and others led to multiple tools like *SCADE* [12] and *Esterel Studio* [12], which do not solely allow to design complex models, but to simulate and verify in an early design cycle. These models can commonly be translated into software (e.g. C, C++ , Java) [7], hardware (via VHDL, Verilog) [6] and software/hardware co-design [4]. Following recent research results, *Esterel* programs can also be executed on reactive processors [21].

Esterel Studio provides a graphical formalism called *SyncCharts* to model the system's behavior. Though looking quite similar to *Harel Statecharts* [17], its semantics more consequently realizes true synchrony and can be translated directly into *Esterel* [1]. To design a system with *SyncCharts* is similar to describing the system's behavior as a series of state changes modeled by an automaton. Moreover each *SynchChart* and each *Esterel* program can be translated into an equivalent *Mealy Machine*. Designated

properties of a synchronous program can be specified through synchronous observers [24, 22]. Such a synchronous observer itself is a *SyncChart* observing the input and output variables to detect unwanted traces. Formally, an observer is a *Büchi* automaton, which accepts a certain language, more precisely this language describes a set of unwanted system traces. The observer is executed in parallel with the designed system. Nevertheless this kind of verification has its limitations. The model checker included in *Esterel Studio* can only check one property at a time[1] and the expressiveness of the *SyncCharts* limits the range of properties, the model checker can handle to safety properties. Though most properties that an embedded system should satisfy are safety properties, there are problems where liveness and more general temporal properties are necessary to reason about certain system designs. Therefore, new ways must be found that allow specification and verification of system design on a higher level, which on the one hand allow to specify a wider range of properties than the synchronous observer approach does and on the other hand do not require a completely different design approach.

Specifying a system is describing its allowed behaviors, but how can a behavior be formally described? Lamport answers this question in his *Specifying Systems* [18] book:

> For centuries, they [the scientists] have described a system with equations that determine how its state evolves with time, where the state consists of the values of variables. For example, the state of the system comprising the earth and the moon might be described by the values of the four variables *e_pos, m_pos, e_vel*, and *m_vel*, representing the positions and velocities of the two bodies. These values are elements in a 3-dimensional space. The earth-moon system is described by equations expressing the variables' values as functions of time and of certain constants – namely, their masses and initial positions and velocities. A behavior of the earth-moon system consists of a function F from time to states, $F(t)$ representing the state of the system at time t. A computer system differs from the systems traditionally studied by scientists because we can pretend that its state changes in discrete steps. So, we represent the execution of a system as a sequence of states. Formally, we define a behavior to be a sequence of states, where a state is an assignment of values to variables.

In which way can behaviors, represented as a series of state changes, be formally described? Temporal logic was first introduced in computer science by Amir Pnueli [26] in 1977 to describe system behaviors. In principle, a system's behavior could be defined with a single formula using Pnueli's formalism. In Pnueli's logic, however, it can be hard to define certain properties of systems. TLA, the Temporal Logic of Actions, was invented by Leslie Lamport in the late 1980's and is a variant of Pnueli's originally proposed logic. TLA provides a clean mathematical foundation to describe systems in a single formula. Most specifications consist of ordinary mathematics, however, temporal logic is important for describing system properties. The system properties define what

[1] Of course, more than one property can be specified in a single observer, but if a violation occurs, some effort is required to catch the reason for the violation.

a system is supposed to do and the specified automaton describes its real behavior. If the specified behavior implies the conjunction of the properties, the system behaves correctly with regard to the defined properties. Usually, formulas a computer scientist deals with are not longer than 20 lines, however, most system specifications will be far longer. Therefore, TLA and, even more so, TLA$^+$ provide compact notations and styles for writing long formulas.

TLA$^+$ specifies a system as a state transition system containing initial states, guarded state transitions and correctness formulas. Therefore, to write a specification in TLA$^+$ is close to the automaton representation of a synchronous program. This thesis investigates the applicability of TLA$^+$ to a classical safety critical problem: redundancy management. Several redundancy management algorithms are specified in TLA$^+$ and a set of properties is checked using the model checker TLC.

The algorithms that are specified and formally analyzed take two possibly finite streams of messages as input and deliver a single stream to a consuming application. The incoming streams contain redundant copies of messages. The resulting stream should be infinite if at least one input stream is infinite, it should contain no redundant frames, as well as preserve the order of sending. The safety properties are clear, however, an algorithm that satisfies these properties would be quite complicated, which corrupts performance and complicates verification and certification. Thus simple redundancy management algorithms that solve relaxed safety properties are examined in a technical report from *Airbus* [27]. This, however, was done informally, and this thesis redoes the examination in a formal way.

Chapter 2 gives a short introduction to TLA$^+$. Chapter 3 provides information about AFDX, the concept of redundancy management and the redundancy management algorithms that are specified in TLA$^+$. Chapter 4 describes the specification of a proper environment for the redundancy management algorithms, specifies the set of requirements, and finally gives specifications of the redundancy management algorithms. Chapter 5 concludes results about the redundancy management algorithms and the applicability of TLA$^+$ and TLC.

2. A short tour through TLA$^+$

This chapter gives a short overview of TLA$^+$. This is neither a detailed description of the statement's semantics nor a complete description of existing statements. I want to provide information about TLA$^+$ that an unfamiliar reader needs to comprehend the specifications written in this work. A complete description of TLA$^+$ and TLC can be found in *Specifying Systems* [18] and a good introduction to TLA$^+$ is presented in the *Wildfire Challenge* specification [19].

2.1. TLA$^+$ by example

A specification is a mathematical description of a system, more precisely the system's behavior is given as a single formula. Most specification will therefore comprise of a long formula that may occupy multiple pages. TLA$^+$ provides a mathematical foundation and compact notations for writing such long formulas while maintaining the readability. Moreover TLA$^+$ is a complete language that allows the specification of complex systems. To specify a system is to say what the system is supposed to do, more precisely to describe its behavior. The behavior of a computer system is most commonly given as a sequence of states, where a state is just an assignment from the domain of variables to a range of values. Taking this intuitive definition, a specification shall define the set of possible sequences of states, *i. e.*, the set of possible behaviors. Subsequently a short step by step introduction to TLA$^+$ will be given.

Each specification written in TLA$^+$ starts with the definition of a *module*.

MODULE *RMA*1

Operators on natural numbers as well as some basic constructs like sets and sequences are not built into TLA$^+$. They are, however, defined in TLA$^+$ modules themselves and can be incorporated into a specification. For example, operators on natural numbers and sequences become available with

EXTENDS *Naturals*, *Sequences*

TLA$^+$ distinguishes *constants* and *variables*. Constants are not given a certain value in a specification. They will be set when the specification should be model checked with TLC and else left unknown. Variables must have a specific value in each state and changing a variable's value produces a new state during model checking with TLC.[1] Because TLA$^+$ is not typed, one does not have to care about types at declaration time. An example for defining constants and variables would be the following:

[1] See Lamport [18] page 243 for a possibility to exclude variables from state generation.

CONSTANTS
networks, set of networks
SN_CNT, *SN_MAX*, *SN_HALF*, maximum sequence number

VARIABLES
rm Redundancy Management

A good style is now to define a type invariant. As already mentioned, TLA$^+$ is untyped and therefore you may assign a natural number in one step to a variable x and in the next step you may assign a string to it. To maintain control over variables and their structure one can define a formula, say *TypeInvariant*, that expresses which form your variables should have in each state. How this formula can be used to check type invariance of the specification variables will be explained below.

$$\begin{aligned} \mathit{TypeInvRM} \triangleq\ & \mathit{rm} \in [\mathit{rsn} \ : (0\,..\,\mathit{SN\_CNT}),\\ & \qquad\ \mathit{ptn} \ : [A : (0\,..\,\mathit{SN\_CNT}),\ B : (0\,..\,\mathit{SN\_CNT})]] \end{aligned}$$

The example above reveals that the variable *rm* should be a record with two fields named *rsn* and *ptn*, where *rsn* is a natural number in range from zero to *SN_CNT* and *ptn* itself is another record. Note that this definition does not ensure a certain structure of variables, so a syntactically and semantically correct assignment of another action could be $\mathit{rm} \triangleq$ "*hello*".

The next topic to consider is *instantiation*. Lamport mentions instantiation very rarely and only in the context of variable hiding. To me, instantiation is a smart way to keep specifications modular. Instantiation in TLA$^+$ is substitution, and a module can be instanced like this:

INSTANCE *RMA*1

This instantiates the module *RMA1* and allows referring to formulas specified in module *RMA1*. If more than one instance of a module shall be created, each instance must be renamed. Furthermore each constant specified in an instanced module must be mapped to a value. This can be done in two ways: either explicitely like in the example below, or implicitly if a constant with the same name exists in the module that contains the instantiation.

INSTANCE *RMA*4 WITH *SNS_MIN* $\leftarrow$ *MTF*

Behaviors in TLA$^+$ are defined by *actions*, where each action describes a guarded state transition. To enable this approach, initial states must be defined. Hence a new formula *Init* gets defined, which assigns a certain value to each variable.

$$\begin{aligned} \mathit{InitRM} \triangleq\ & \mathit{rm} = [\mathit{rsn} \mapsto \mathit{noVal},\\ & \qquad\ \mathit{ptn} \mapsto [A \mapsto \mathit{noVal},\ B \mapsto \mathit{noVal}]] \end{aligned}$$

In this case we just assign a value to the variable *rm*. Again the assigned value may be of arbitrary type. With this definition of initial values, the behavior of the system can be specified. A system may perform different actions, depending on the actual state. Therefore each action should have the following form:

$$\begin{aligned} action \triangleq\ & \wedge\ guard \\ & \wedge\ assignment \end{aligned}$$

Each clause could either refer to the actual value of a variable or set a variable's value for the next state using the *Prime* operator. A *state function* is an expression that contains no primed variables and if a state function is boolean valued it is called a *state predicate*. The guard consists of all clauses that are state predicates. All other clauses may contain primed variables, where a primed variable denotes the value a variable gets in the next state. TLA$^+$ also allows to write boolean valued functions that contain primed variables and hence guard a transition by the shape of the new state. It is assumed that the user knows what he or she does and uses such expressions with care. A variable that should not change must be explicitly declared *UNCHANGED*. For the assignment of values to complex variables TLA$^+$ provides some nice syntactic sugar that maintains good style and readability. A good example, containing the most important simplifications, is given below.

$$\begin{aligned} \wedge\ rm' = [rm\ \text{EXCEPT}\ &!.rsn = sn, \\ &!.ptn[id] = sn] \end{aligned}$$

Following the intuition the assignment above defines the new value of *rm* to equal the actual value of *rm* except that the field *rsn* equals *sn* and the record field with name *id* of record *ptn* equals *sn*, too. It may seem to be superfluous to assign the actual value of *rm* to the next state and afterwards explicitely give almost all fields new values. I prefer this style, because it maintains readability and allows easily to extend variables without taking care of every appearance of this variable. The meaning of *!.rsn* is an abbreviation for *rm.rsn*. A further expression to shorten specifications is to use the @-statement together with the *EXCEPT* notation for variables. The @ is a substitution for the left side of an assignment. The example below shows the *rm* variable extended with a simple integer valued variable, which alternates between zero and one. Switching this value can be specified by

$$switch \triangleq\ \wedge\ rm' = [rm\ \text{EXCEPT}\ !.bool = 1 - @]$$

Equivalently this could have been specified as

$$switch \triangleq\ \wedge\ rm' = [rm\ \text{EXCEPT}\ !.bool = 1 - rm.bool]$$

Finally a system's specification in TLA$^+$ is a single formula. Hence the so far specified formulas must be assembled together. For a complete system specification, we must specify the set of initial states, a formula that describes the next state relation, and optional formulas to specify fairness and real time.

$$Spec \triangleq Init \wedge \Box[Next]_{\langle var \rangle} \wedge Fairness \wedge RealTime$$

Init specifies the system's initial states and from these states we require every step to be a *Next* step or a *stuttering step*[2]. The *Next* formula is most commonly a simple disjunction of all specified actions, but could be more complicated, too. The *Fairness* and *RealTime* formulas have the obvious meaning and need not necessarily to be specified.

Revisit the type invariance formula specified above. It states that the specified variables shall have a certain type. $\Box\, TypeInvRM$ asserts that this formula is always true, thus for every behavior satisfying *Spec*. These kinds of temporal formulas are called *theorems*. Theorems can be machine checked with TLC. It is a good idea to formulate a type invariance formula and to check it with TLC, because it helps catching a lot of subtle specification gaps made during the specification of actions. To point out that type invariance is claimed, the last line often contains a formula stating that type invariance follows logically from the definitions in this module.

$$\text{THEOREM } Spec \Rightarrow TypeInvRM$$

Each module must end with a further delimiter line.

Most of the TLA$^+$ statements and operators have an equivalent mathematical representative with the same meaning. Only simple predicate logics and mathematical sets are needed to understand the specifications written in this thesis.

2.2. Model checking with TLC

The formal foundation of TLA$^+$ allows other tools to work on TLA$^+$ specifications, like TLC, TLASANY and TLATEX. The most interesting one is TLC, as it enables machine verification of systems specified with TLA$^+$. TLC properties should be defined completely independent from a TLA$^+$ specification, because a specification should be the same, whether it serves as a formal description or gets used for machine verification. Therfore, all subsequently discussed TLC statements should reside in a special specification file that extends the real specification.

MODULE *ENV_TLC*

EXTENDS *ENV*, *TLC*

For a complete description of TLC and its optimization potentials see Lamport [18] pages 221 **ff**. Two possibilities to reduce state space will be introduced. First of all, systems specified with TLA$^+$ may observe infinite behaviors, which cannot be handled with model checkers. Such infinite behaviors can be obtained if a specification uses for example sequences, which are theoretically unbounded. To limit infinite models, TLC provides the possibility to postulate constraints that, for example, bound the length of a sequence to a finite number.

[2] A stuttering step leaves all variables unchanged.

CONSTANT $maxLen$

$constraint \triangleq Len(queue) \leq maxLen$

The second possibility and probably the more fragile one is to take a different *view* than the standard view, which contains all specified variables. This should, however, only be done if the discarded variables have just debugging purpose. Unfortunately variables that were discarded from the actually used *view* are not displayed in counter examples and no variable names get displayed at all.

$$myView \triangleq \langle rm, env, out \rangle$$

TLC needs a separate configuration file that assigns values to all constants defined in the modules, includes constraints and views, declares the specification formula and finally declares which properties TLC shall check. Thereby TLC distinguishes between invariants that must be state predicates and temporal properties that may contain more general formulas. Once the configuration file provides all information TLC needs to check a specification against a certain property, model checking can be started. Everything you can do from that point on is waiting for a result. Especially checking temporal properties, which are not of the form $\Box P$ with state predicate P, may take a long time, depending on the complexity of the checked property and the performance of the model checking hardware up to a few days. Therefore properties and specification should be developed with care.

```
CONSTANTS
  networks = {"A", "B"}
  SN_CNT = 6 SN_MAX = 5 SN_HALF = 3

VIEW
  myView

CONSTRAINT
  constraint

SPECIFICATION
  Spec

INVARIANT
  TypeInv
```

Auflistung 2.1: TLC configuration example

3. Redundancy Management Concept

3.1. AFDX

Reliable communication between avionic subsystems has always been essential, especially as in 1988 with the Airbus A320 the *all-electronic fly-by-wire* technology attained commercial airline service. Since that time it gained such a popularity that currently built airliners only use this technology for avionic subsystem communication.
As the complexity of avionics subsystems has grown since 1988, so has the demand for more bandwidth and reliability. Former avionic data communication protocols like *ARINC 429* and *MIL-STD-1553* did not seem to be able to compete with upcoming demands. A new communication bus-system shall as well serve for subsystem communication as for passenger entertainment. The desire for such a new fast and cheap communication bus forced the industry to explore off-the-shelf-technologies such as IEEE 803.2 Ethernet. Tanenbaum [28] gives a basic introduction about computer networks in general. Advanced topics on *ATM* and switched Ethernet are considered in Goralski [15] and Breyer [9]. Ethernet specification, however, guarantees no maximum latency, as the package collisions are resolved through a *back off* strategy that may lead to an infinite latency in worst case. That is why the next-generation avionics data bus shall on the one hand allow usage of as much cost-efficient, IEEE 803.2 compliant hardware as possible and on the other hand shall guarantee a certain bandwidth and *Quality of Service*, which includes specifying maximal transmission latency. This research resulted in ***A**vionics **F**ull **D**uplex Switched Ethernet* based upon IEEE 803.2 Ethernet technology [3]. See Figure 3.1 for a simple AFDX schematic.

Overview of AFDX

AFDX addresses the shortcomings of Ethernet using concepts of Asynchronous Transfer Mode (ATM). Major aspects of AFDX are:

- AFDX is a profiled network, configuration tables are loaded into switches at start-up.
- It is organized in a star topology with a maximum of 24 *End Systems* (ES) per switch. Larger systems can be realized through cascading.
- AFDX is full duplex. Standard Ethernet suffers the possibility of an infinite chain of frame collisions and hence unpredictable delay of messages. Therefore every

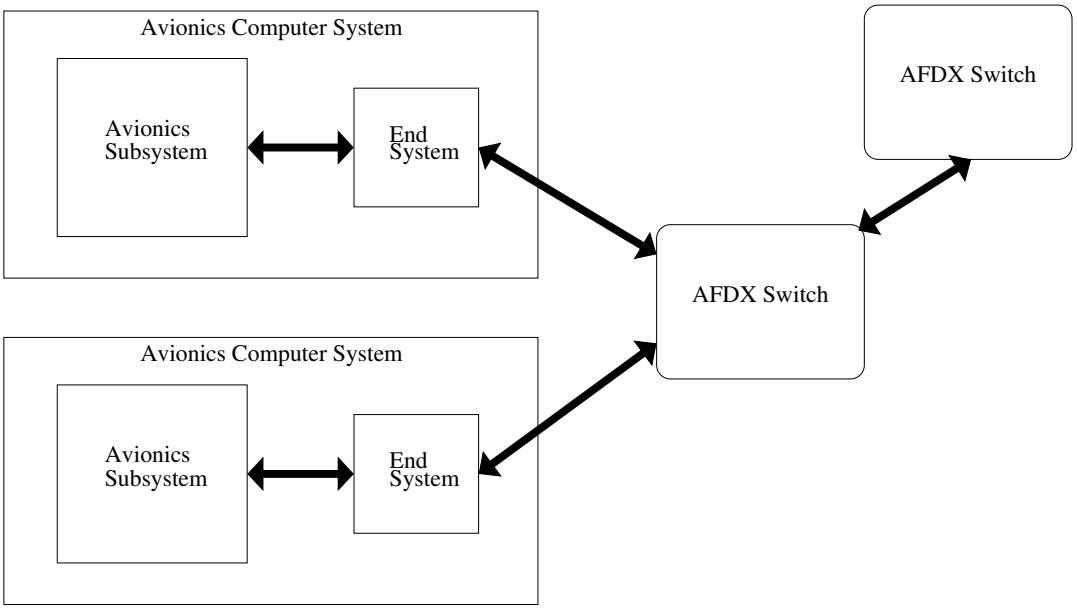

Figure 3.1.: AFDX schematic

End System is connected to a switch with two twisted pair cable, one pair for sending and the other for receiving frames. Each switch has the capability to buffer multiple packages for each ES in each communication direction. Consequently buffer-overflows and message delays due to congestion at the switch may cause erroneous behaviors.

- AFDX emulates a deterministic point-to-point network through the use of *Virtual Links* (VLs). Each VL builds a unidirectional path from a unique ES to one or maybe more ESs. A certain predefined bandwidth is allocated for each VL, ensuring that the sum of allocations does not exceed the maximum available bandwidth of the whole network.

- AFDX guarantees bandwidth and a maximum latency for end-to-end transmission. Bandwidth is provided through *Bandwidth Allocation Gap* (BAG), which defines the minimum time between the sending of two successive frames. It is given that $\text{BAG} = 2^k$ with $k \in \mathbb{N}_{\{0,\dots,7\}}$, which implies a maximum of 128 VLs per ES.

- AFDX provides a highly reliable network scheme. Each frame is transmitted in parallel over two redundant networks and afterwards filtered by redundancy management at the receiving ES.

- AFDX is fast, either 10 Mbps or 100 Mbps.

This short description above shall provide the information needed to comprehend the following. It is meant as background as we almost need no knowledge about how AFDX works to write the specifications for the redundancy management algorithms.

3.2. Redundancy Management Concept

Redundancy is desired to increase reliability of systems by duplicating parts of it. This is often done if the reliability of the duplicated part itself cannot be increased. In case of the AFDX protocol, the weak point is the network. Messages can get lost, or even worse a network can fail completely, therefore a second network is introduced (Figure 3.2) to reduce the probability of loosing frames and enable further operation even in presence of one faulty network. Nevertheless the question arises what to do with redundant frames?

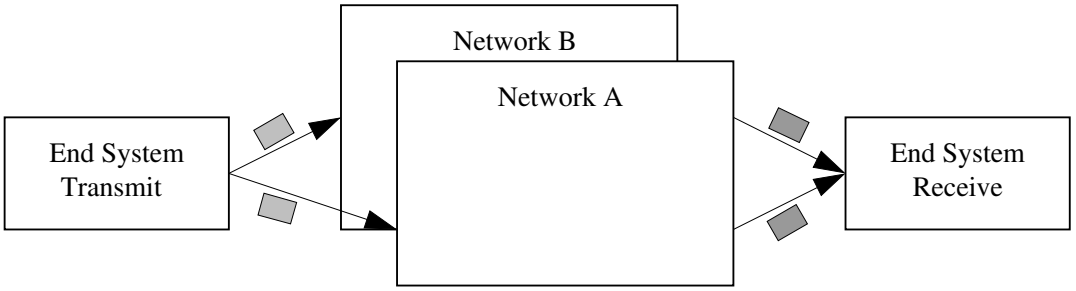

Figure 3.2.: Concept of redundant networks

Shall the receiving application handle them? The redundancy management shall be considered as a part of the receiving *End System* and it shall forward valid frames to the application and discard all others. Hence redundancy shall be transparent to the application beyond the ES. Figure 3.3 shows the principle function of an receiving ES. Each frame has first to pass integrity checking. This integrity checking can be defined in many different ways. A weak one can just checks if a frame is *well-formed* and another one may check if the received frame was expected to be delivered next. Integrity checking, however, is not considered in this work. It is assumed that integrity checking filters frames in a way that all frames reaching the redundancy management are well-formed. No further assumptions about integrity checking are made. Consider two frames with equal content. What is needed to decide whether one frame is the

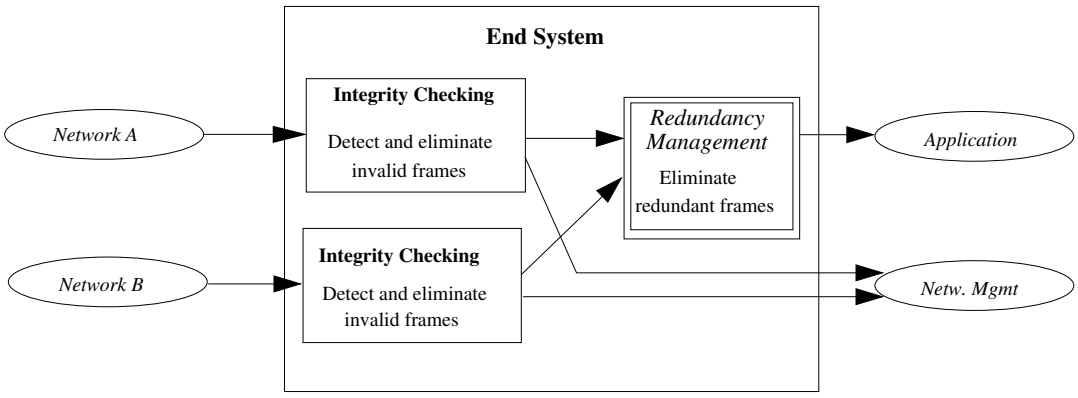

Figure 3.3.: Receiving End System

redundant copy of the other? First we need to know about which network a frame was delivered. Two frames delivered by the same network cannot be redundant copies of each other. Furthermore an order of frames received from a network must be established. Thus every frame contains a "unique" sequence number. Figure 3.4 shows the complete AFDX frame format.

7 bytes	1 byte	6 bytes	6 bytes	2 bytes	20 bytes	8 bytes	17 to 1471 Bytes	1 byte	4 bytes	12 bytes
Preamble	Start Frame Delimiter	Destination Address	Source Address	Type Ipv4	IP Structure	UDP Structure	AFDX Payload	Seq Number	Frame Check Seq	Interframe Gap

Figure 3.4.: AFDX frame format

Sequence Numbers

Subsequently all definitions, corollaries and algorithm definitions are taken from the technical report of von Hanxleden and Gambardella [27]. So if not stated otherwise consider this paper as reference. Sequence numbers are not really unique. Since we may have an arbitrary number of frames and limited resources for sequence numbers, they eventually must wrap around. Figure 3.4 already shows that 8 bits are used for sequential counting. A range of 28 bits was proposed, which increases the range where sequence numbers can be ordered correctly. Finally, for mainly economic reasons, the 8-bit range was chosen. Subsequently some key properties about sequence numbers will be given, which show how sequence numbers can be used to determine redundancy. The number of sequence numbers is $SN\_CNT =_{def} 2^8$. Thus the maximum sequence number is $SN\_MAX =_{def} SN\_CNT - 1$. The mid-point sequence number is denoted as $SN\_HALF =_{def} SN\_CNT/2$. Consecutive frames have a sequence $SN(f_{i+1}) =_{def} SN(f_i) + 1 \bmod SN\_CNT$. The subtraction on sequence number is defined as follows:

Definition 1. *(Sequence Number Subtraction) The subtraction operator $-_{sn}$ is:*

$$s_1 -_{SN} s_2 =_{def} ((s_1 - s_2 + SN\_HALF) \bmod SN\_CNT) - SN\_HALF$$

For example, with the above defined operators we have that:

$$124 -_{SN} 134 = -10$$
$$19 -_{SN} 254 = 21$$
$$238 -_{SN} 78 = -96$$

It can be seen that subtraction of sequence numbers for sequence numbers within a certain bound is equal to common subtraction on natural numbers module SN_CNT. Definition 1 can be used to define the following comparison operators.

Definition 2. *(Comparison Operators)*

$$s_1 <_{SN} s_2 \Leftrightarrow_{def} (s_1 -_{SN} s_2) < 0$$
$$s_1 =_{SN} s_2 \Leftrightarrow_{def} (s_1 -_{SN} s_2) = 0$$
$$s_1 >_{SN} s_2 \Leftrightarrow_{def} (s_1 -_{SN} s_2) > 0$$

Finally, the *unwrapped sequence number* $USN(f)$ is needed to reason about sequence number operations. This number $USN(f)$ is a theoretical number for each frame f, which does not wrap around and thus is unbounded. Therefore a correct ordering of frames using sequence numbers can be established if the unwrapped sequence numbers differ at most by SN_HALF. Formally von Hanxleden and Gambardella [27] state:

Corollary 1. *Let frames f_1 and f_2 be generated without intermediate sequence number reset, and let s_1 and s_2 be their respective sequence numbers. If $\mid USN(f_1) - USN(f_2) < SN\_HALF$, then it is $s_1 -_{SN} s_2 = USN(f_1) - USN(f_2)$*

Subsequently the concept of sequence numbers will be used to postulate algorithms that should determine and discard redundant frames.

3.3. Evolution of Redundancy Management Algorithms

This section introduces and explains the proposed redundancy management algorithms and gives some examples for which they might show faulty behavior. Obviously redundancy management algorithms should hide the redundancy of message transmission to the application beyond the ES. There are, however, constraints that might not allow to perfectly filter redundant frames under all circumstances. For example, the redundancy management may only cause a very small additional message delay, therefore complex buffering strategies could possibly violate these constraints. Moreover the environment may observe strange behaviour that causes difficulties to filter redundant frames. Finally the environment may send arbitrary frames, but only a well defined stream of frames with specific characteristics will reach the redundancy management. The two most important properties of the delivered frames are:

1. The number of frames traveling through a network at a time is bounded by the *Maximum number of Transient Frames* (MTF), which bounds the maximum difference of received sequence numbers from different networks.

2. The maximum number of consecutive frames lost is bounded by the constant $MCFL$, which defines the maximum difference of received sequence numbers on the same network.

Both properties are essential for the correct operation of the proposed algorithms.

What are the requirements for the design of redundancy management algorithms? In general they just should be simple. To be more concrete, they should allow

- easy understanding
- certification
- verification
- cost effective implementation.

Since the redundancy management shall cause as few delay as possible, buffering of frames is considered harmful. Moreover the algorithm shall follow the guideline *first valid wins*, which means that the logically first frame, from a pair of redundant twin frames, that reaches the redundancy management will be submitted. All together this implies that for each frame the redundancy management receives, it decides to accept or reject this frame before it receives the next one. Thirteen algorithms were proposed as a series of refinements from a first simple algorithm. Coming up they will be shortly described together with their acknowledged shortcomings.

RMA1: Accept if Sequence Number Skew is positive

The *Sequence Number* (SN) of frame f is $SN(f) \in \{0 \dots SN_{MAX}\}$. The *Received Sequence Number* of frame f is denoted as $RSN(f)$ and it may be $SN(f) \neq RSN(f)$. However we assume those frames with $SN(f) \neq RSN(f)$ are not well formed and thus discarded by the integrity checking. The *Previous Twin Network Frame* $PTN(f)$ for a frame f received on network N_1 denotes the last frame received on the other network N_2. This enables a definition of the *Sequence Number Skew* as follows:

Definition 3. *(Sequence Number Skew) The Sequence Number Skew of frame f is*

$$SNS(f) =_{def} RSN(f) -_{SN} RSN(PTN(f)).$$

Thus the first proposed algorithm is:

RMA 1. *Accept frame f* ***iff*** $SNS(f) > 0$

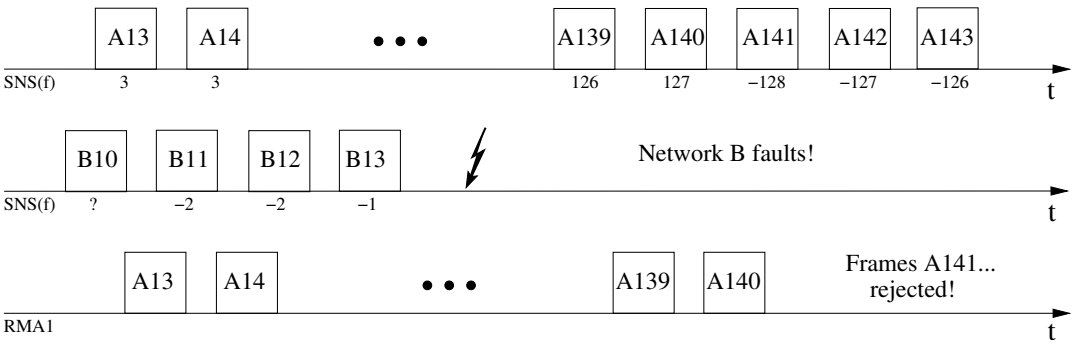

Figure 3.5.: Silent Network Scenario - Using RMA1

Obviously the correct operation of this algorithm assumes operation of both networks. If one network fails, the *Sequence Number Skew* will increase. However, Corollary 1 implies that this may cause erroneous behaviors as the unwrapped sequence differs by values greater than SN_HALF. Figure 3.5 displays a scenario where one network is down and the other in normal operation. It can be seen that valid frames are rejected by **RMA1** in this case.

RMA2: Accept if Sequence Number Offset is positive

This algorithm addresses the shortcoming of **RMA1** and is based on the observation that under normal operation of the remaining network, the difference ($-_{SN}$) between two successive sequence numbers is always positive[1]. For a given RMA, the *Previously Accepted Frame* (PAF) denotes the last frame accepted by that RMA. The *Previous Accepted Sequence Number*($PASN$) for frame f can be defined as:

Definition 4. *(Previously Accepted Sequence Number) Let f be a frame with $SN(f) \in \{0 \ldots SN\_MAX\}$. Then*

$$PASN(f) =_{def} RSN(PAF(f)).$$

With the previously accepted frame defined as above, the *Sequence Number Offset* is given by

Definition 5. *(Sequence Number Offset)*

$$SNO(f) =_{def} RSN(f) -_{SN} PASN(f)$$

Hence the new redundancy management algorithm is:

RMA 2. *Accept frame f **iff** $SNO(f) > 0$*

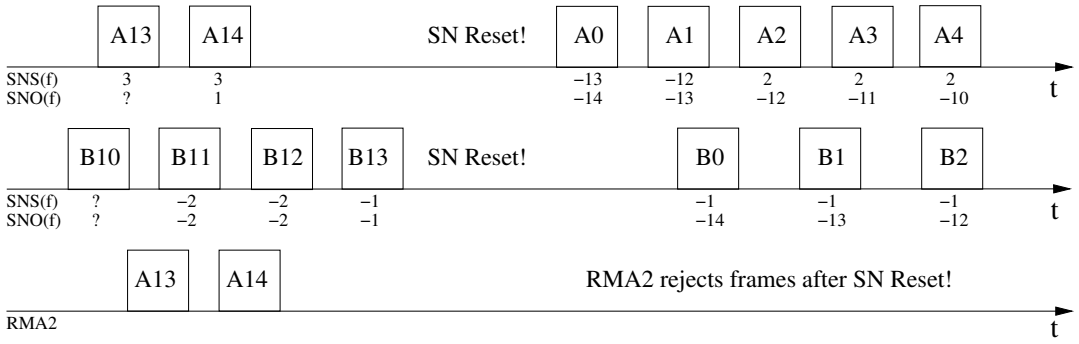

Figure 3.6.: ES reset - Using RMA2

It is easy to see that this RMA has no problem with the scenario shown in Figure 3.5. RMA2, however, fails in case of sequence number resets. Figure 3.6 shows the problematic scenario. It is not that problematic that a sequence number reset leads to rejection of some valid frames, but we can imagine a scenario where the reset occurs periodically whenever the sequence number is close to SN_HALF. This would imply that never again a frame will be accepted. This is problematic for $SN\_HALF = 2^7$ where the required period of consecutive resets is quite small, but it would be absolutely unacceptable with $SN\_HALF = 2^{27}$ as proposed before.

[1] This follows from the knowledge that we have a Maximum Consecutive Frame Loss ($MCFL$) less than SN_HALF

RMA3: Accept if Sequence Number Delta is positive

The next idea is to combine **RMA1** and **RMA2**. Therefore we define

Definition 6. *The Sequence Number Delta of frame f is given by*

$$SND(f) =_{def} max(SNS(f), SNO(f)).$$

Based on that definition we have:

RMA 3. *Accept* ***iff*** $SND > 0$

Obviously this approach corrects the problem of the algorithms above. Figure 3.7 shows that still both networks are needed to handle sequence number resets right. So

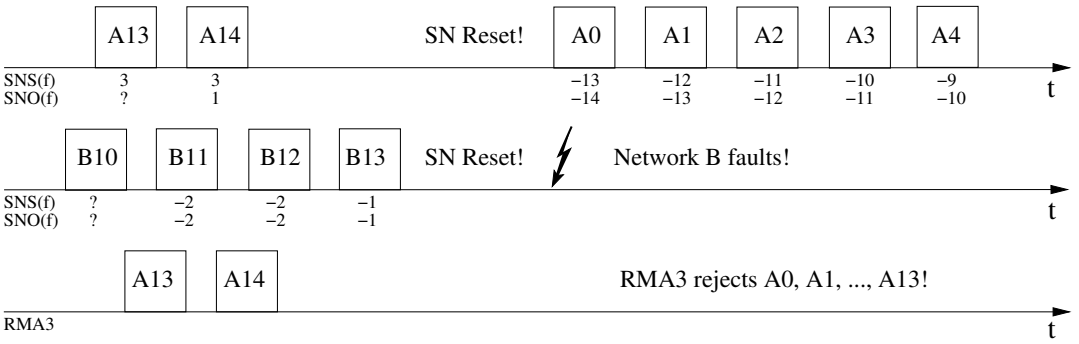

Figure 3.7.: ES reset and networks silence - Using RMA3

far all algorithms defined conditions when to accept frames. This concept proved to be too restrictive in frame acceptance. Often valid frames were rejected. Conversely the next algorithms will ask when to reject a frame. Of course the negation of the afore defined acceptance conditions are as well conditions for rejection of a frame. It is more precise to say that the point of view changes from *which frames must be accepted* to *which frames must be rejected*. Leaving the conservative way, however, may lead to acceptance of redundant or outdated frames. In fact, this is tolerable as it must be ensured that under all circumstances the redundancy management algorithm does not stop forwarding frames as it may happen with the first algorithms.

RMA4: Reject if Sequence Number Skew is redundant

Similar to **RMA1**, **RMA4** is based on the Sequence Number Skew, with Corollary 24, which gives the definition of *SNS_MIN* [27] it can be deduced that

Without sequence number reset, it is $SNS\_MIN \leqslant SNS(f) \leqslant 0$ *iff the twin frame has already arrived on the other network.*

That provides the next RMA:

RMA 4. *Reject frame f* ***iff*** $SNS\_MIN \leqslant SNS(f) \leqslant 0$

Nevertheless both networks are still needed as, similar to **RMA1**, continously frames are lost if one network dies, and the other works alone (Figure 3.8 with assumed $SNS\_MIN = -5$). Nevertheless **RMA4** causes not that much frame loss.

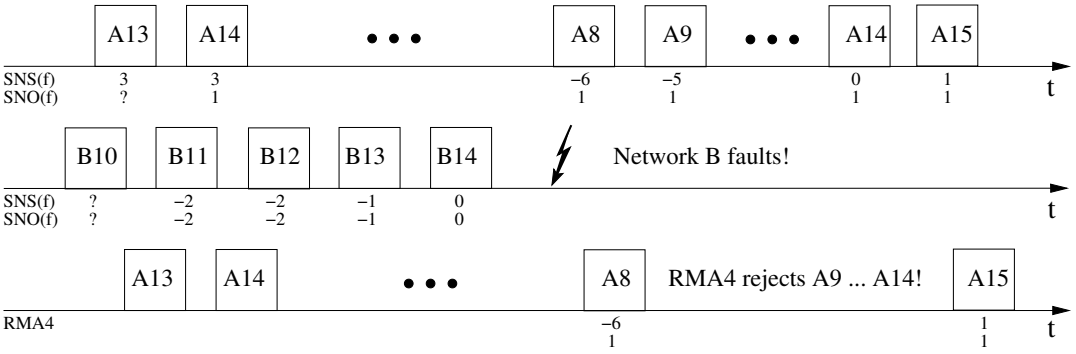

Figure 3.8.: Operation with single network - Using RMA4

RMA5 Reject if Sequence Number Offset is redundant

Analogously to the step from **RMA1** to **RMA2**, now the concept of Sequence Number Offset, defined as above, is used to decide about rejection of frames. Obviously the insight from **RMA4** can be extended to the following observation

Without sequence number reset, it is $SNS\_MIN \leqslant SNS(f) \leqslant SNO(f)) \leqslant 0$ *iff the twin frame has already arrived on the other network.*

Therfore we define:

RMA 5. *Reject frame* f ***iff*** $SNS\_MIN \leqslant SNO(f) \leqslant 0$

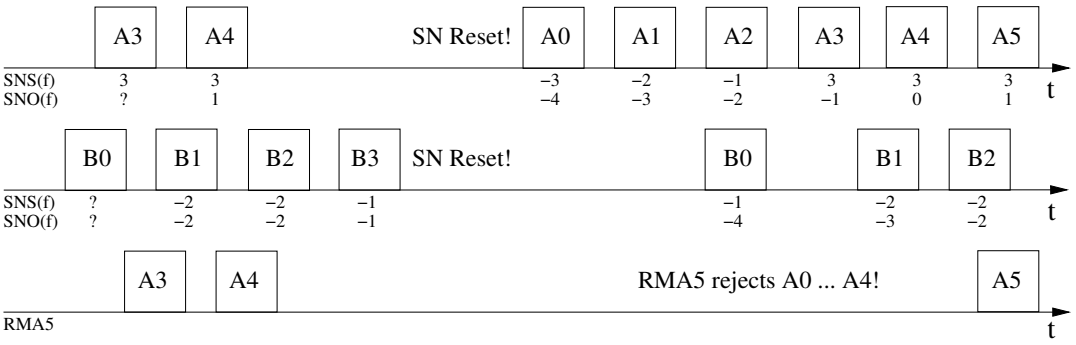

Figure 3.9.: ES reset - Using RMA5

Considering the behavior shown in Figure 3.8, it is clear that this algorithm handles this situation correctly. Nevertheless, Figure 3.9 shows a behavior which is still unacceptable.

RMA6 Reject if SNS and SNO are redundant

Like **RMA3**, **RMA6** combines strategies of both afore mentioned algorithms.

RMA 6. *Reject frame if* ***iff*** $SNS\_MIN \leqslant SNS(f) \leqslant 0$ ***and*** $SNS\_MIN \leqslant SNO(f) \leqslant 0$

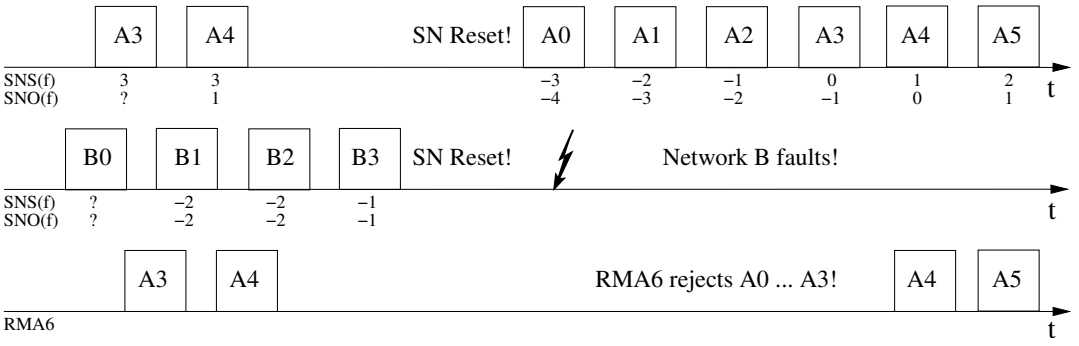

Figure 3.10.: ES reset and low PASN - Using RMA6

This excludes the behaviors that we already considered bad and which lead to frame rejection. It is not that perfect, however; again problematic scenarios with one silent network can occur. See Figure 3.10 for such a behavior.

RMA7 Reject if SNO is redundant and SNI is positive

Close to the definition of *Sequence Number Skew* used before, the *Sequence Number Increment* is defined:

Definition 7. *(Sequence Number Increment) For consecutively received frames* f_1, f_2*, the* Sequence Number Increment *is*

$$SNI(f_2) =_{def} RSN(f_2) -_{\mathrm{SN}} RSN(f_1).$$

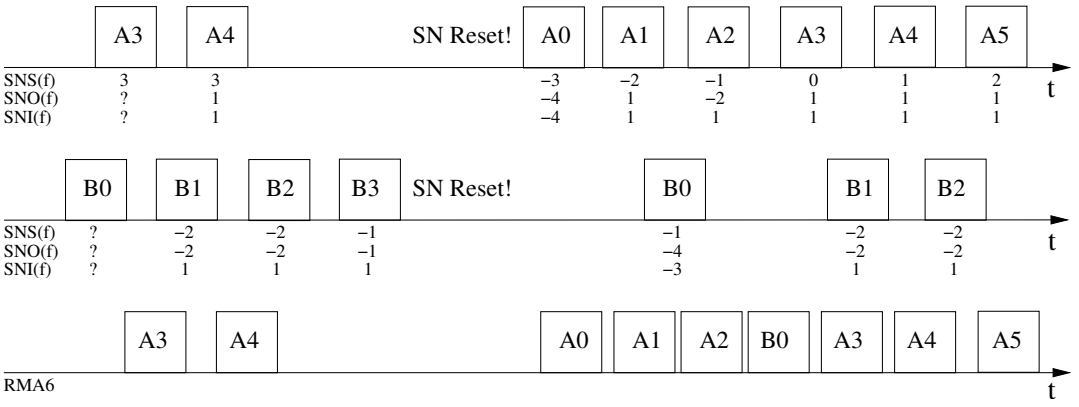

Figure 3.11.: ES reset - Using RMA7

With this definition it is possible to detect reset of an ES and to handle the behavior shown by Figure 3.10 correctly. Thus we define the next algorithm to be:

RMA 7. *Reject frame f* ***iff*** $SNI(f) > 0$ ***and*** $SNS\_MIN \leqslant SNO(f) \leqslant 0$

Nevertheless this leads to the situation that more redundant frames may get accepted and this seems to be an unacceptable trade off. See Figure 3.11 for such a situation.

RMA8 Reject if SNO is redundant and no time-out

All afore described algorithms only use knowledge about recently received and accepted frames. The specification of the environment contains more information usable to design redundancy management algorithms. Under normal network operation, which means that no frames are lost, redundant frame copies arrive at most *Skew Max* time apart. Hence it can be deduced that frames which arrive with a time difference greater than *Skew Max* cannot be redundant. This knowledge combined with the already used concept of redundant Sequence Number Offset yields

RMA 8. *Reject frame f* ***iff*** $SNS\_MIN \leqslant SNO(f) \leqslant 0$ ***and*** $t_{Recv}(f) - t_{Recv}(PAF(f)) \leqslant SkewMax$

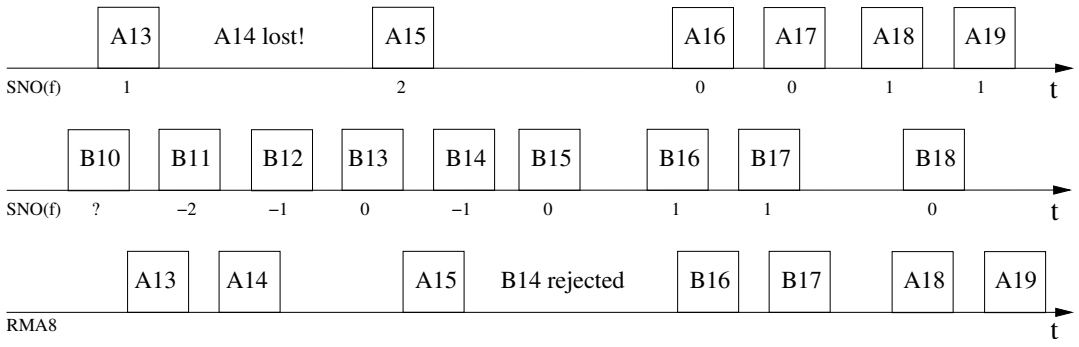

Figure 3.12.: ES reset - Using RMA8

Still problematic are sequence number resets while the last accepted sequence number is lower than SN_HALF. Non redundant frames will be rejected if such a behavior occurs, see Figure 3.12.

RMA9 Accept if successive frame or after time-out

The next proposal is very rigorous in accepting frames. Only successive frames are accepted if no time out occurs. This ensures that even after reset of sequence number no redundant frame will be submitted, but at least after *Skew Max* time elapsed the next frame will pass.

RMA 9. *Accept frame f* ***iff*** $SNO(f) = 1$ ***or*** $t_{Recv}(f) - t_{Recv}(PAF(f)) > SkewMax$

The algorithm of choice, however, should follow a *first valid wins*-strategy and Figure 3.13 shows that **RMA9** does not.

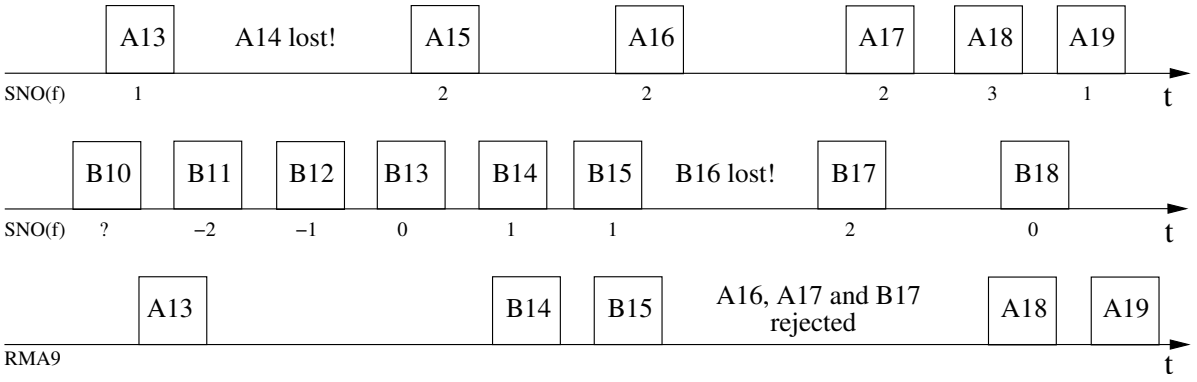

Figure 3.13.: Fast and slow network - Using RMA9

RMA10 Accept if successive frame, buffer non-redundant frames

This algorithm is special as we break with our premise that buffering is harmful, as it may introduce too much additional delay of message delivery. Moreover not every frame is now rejected or accepted at delivery time. Generally all frames that are not considered redundant will be buffered. To decide if a frame is redundant or not, the *Queued Sequence Number Offset* is used, which can be given as follows:

Definition 8. *(Queued Sequence Number) For frame f and buffer b the* Queued Sequence Number Offset *is*

$$QSNO(f) =_{def} \textit{if } b = \langle\rangle \textit{ then } RSN(f) -_{SN} SN(Head(b)) \textit{ else } RSN(f) -_{SN} PASN(f)$$

In case the buffer is not empty, in parallel it is checked whether the first element of the buffer has a sequence number offset equal to 1 or the time since reception of first element has exceeded a certain value, from which it can be deduced that the preceding frame cannot be delivered anymore. Putting it all together we get:

RMA 10.

if $SNO(f) = 1$ **then** accept f
else if $QSNO \notin (-SNS\_MIN \ldots 0)$ **then** enqueue(f)
par
while $b \neq \langle\rangle$ **and** ($SNO(f_q) = 1$ **or** $t > t_{RECV}(f_q) +$ Skew Max - BAG) **do**
dequeue(f);accept(f)

The algorithm turns out to have a minimal per-frame loss and preserves ordering of frames. Nevertheless it is not considered to be a good choice as it introduces further delay and with buffers a new potential source of failure.

RMA11 Reject recently received SNs

The last three redundancy management algorithms already incorporated further ideas to support decisions based upon sequence numbers. This one is based on the idea that

a frame is likely to be redundant if its sequence number was already seen shortly before. This buffer and hence the set of sequence numbers that must be buffered is limited, as we know that each frame must be delivered after a certain time has been elapsed and the capacity of networks bounds number of deliveries of the twin network. Given such a set $PASN_n$ of n recently received sequence numbers the new approach is:

RMA 11. *Reject frame* f ***iff*** $RSN(f) \in PASN_{SNL\_MAX}(f)$.

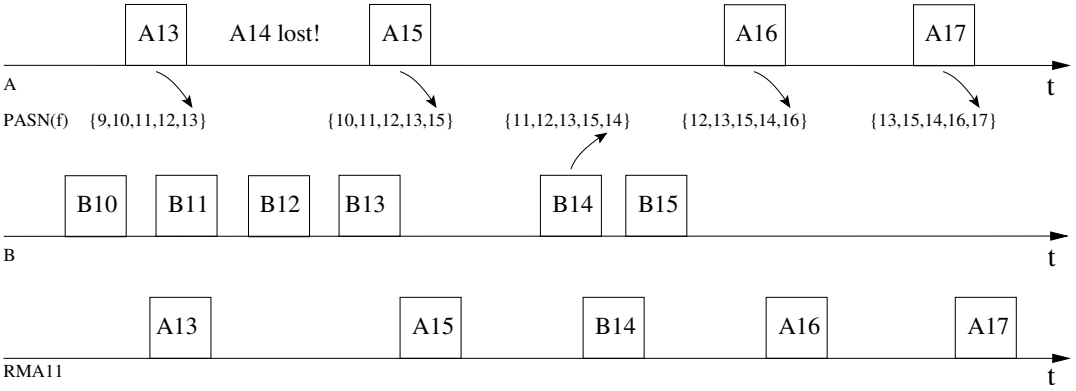

Figure 3.14.: Loss of order - Using RMA11

Obviously this algorithm does not preserve the order of sending. A fast but unreliable network and a slower but more dependable one could lead to the situation shown in Figure 3.14. Even worse things can happen if there is a recurring reset of sequence numbers close to SN_HALF, which may lead to rejection of all subsequent frames.

RMA12 Reject recently received SNs if no time-out

The above described algorithm **RMA11** causes trouble with reordering and unjustified, continuous rejection of frames. The latter is absolutely unacceptable, and therefore it is extended to guarantee resynchronization after a reset of sequence numbers.

RMA 12. *Reject frame* f ***iff*** $t_{Recv}(f) - t_{Recv}(PAF(f)) \leqslant SkewMax$ ***and*** $RSN(f) \in PASN_{SNL\_MAX}(f)$.

RMA13 Accept frame from same network as last frame, or after time-out

The last approach may seem to be the most trivial, but is interesting nonetheless. Initially the redundancy management algorithm just listens on both networks and synchronize to the faster one. Obviously this is not *"first valid wins"* anymore and the overall availability is not optimal, as many non-redundant frames of the "loosing" network are rejected.

RMA 13. *Accept frame* f ***iff*** $N(f) = N(PAF(f))$ ***or*** $t_{Recv}(f) - t_{Recv}(PAF(f)) > SkewMax$.[2]

However this simple approach is very robust against critical sequence number resets with low previously accepted sequence number. Even in the worst case if, after such a critical reset, the preferred network dies, after time-out the redundancy management algorithm switches to the remaining network.

[2] This is contrary to the technical report proposing the algorithms [27] but reflects the description and intuition.

4. Specification

4.1. What to specify?

This chapter gives a detailed explanation of the written specifications as well as for presenting first results of model checking. Nevertheless some major aspects of the built model will be explained to understand the specifications written in TLA$^+$.

1. The first thing to choose is the grain of atomicity for the specification of the receiving ES. Figure 3.3 shows the way each frame has to pass from delivery till submission to application. Each frame must pass an integrity checking, which shall discard *mal-formed* frames. To specify properties that a frame must satisfy to be considered well formed is not in the scope of this work. For the redundancy management algorithm, however, it is undetectable whether a frame got lost during network transmission or failed integrity checking. Hence our model needs not to consider integrity checking. Only the networks together with a fictive sender form the influencing components before the redundancy management algorithm and are therefore included in a first model shown by Figure 4.1. To distinguish frames by

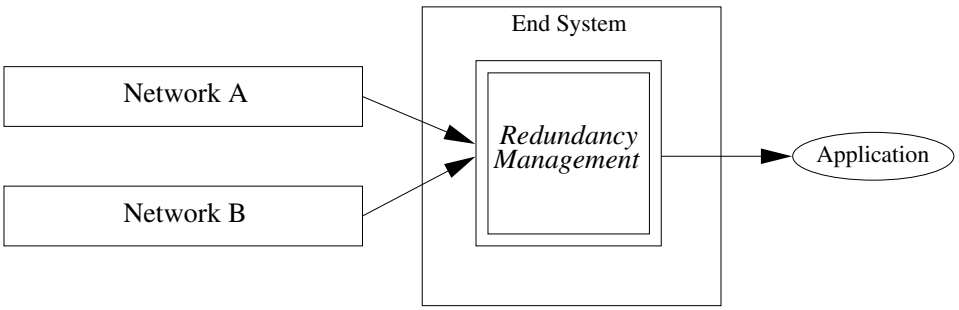

Figure 4.1.: First abstraction of the environment

their delivering network, two different buffers for each network could have been used. Though this is closer to reality, I decided to take frames directly from the network queues to save variables and hence ease model checking. Each accepted frame is delivered to the application set beyond the redundancy management. The model, however, should not contain another pair of producer and consumer, and it is impossible to let the redundancy management forward each frame to buffer without a consuming application, as this would lead to an infinite model. In Section 4.2, an approach how this aspect can be solved will be explained in detail. Finally the complete model just contains a single environment, which includes the

redundant networks, the redundancy management and an output device where submitted frames can get stored in a way.

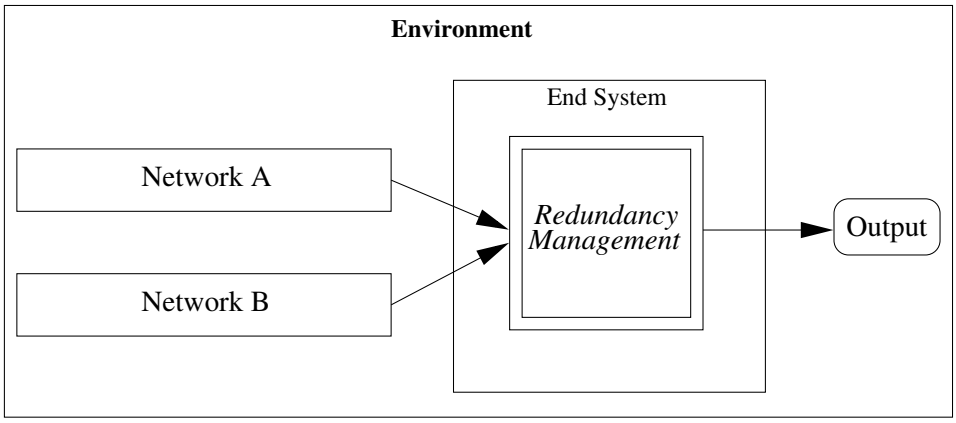

Figure 4.2.: Final Model of the *System under Development*

2. How does the redundancy management gets knowledge about the delivering network of each frame? Figure 3.4 implies that it cannot obtain any information from a frame itself. Thus each network seems to deliver its frames to a different buffer. There is no information provided in the report [27] about how the redundancy management algorithm detects the source of a frame. Subsequently I will assume that information about the delivering network is attached to each frame as it reaches the stage of redundancy management where decision for acceptance or rejection takes place. Furthermore, we abstract away all parts of a frame that is not needed to detect redundancy. Hence the frames used in our model contain only a sequence number and a network identifier (see Figure 4.3). This model will be slightly extended in the next section with some status information.

1 byte	1 byte
Seq Number	Network ID

Figure 4.3.: Frame structure in model

3. The next step is to think about a good structure of the specifications that

   - supports reading and understanding,
   - allows extensions if necessary,
   - follows the *"write things once"* principle and
   - is modular.

TLA$^+$ offers several ways to build up modular specifications. The solution of choice is to write one module for the environment and another for the redundancy management which can then be instantiated in the environment module. This enables a single specification of the environment wherein each version of redundancy management algorithm can be instantiated and model checked.

4.2. Specification of the environment

This section serves the purpose to describe and formally specify an appropriate environment to each of the tested redundancy management algorithms. Such an environment should feed the redundancy management with a *well-formed* stream of frames and provide information to reason about the correctness of the redundancy management's decisions. Figure 4.2 shows the major parts the environment contains and which can be easily found in the specification. These parts are

- two networks that deliver frames to the redundancy management algorithm,
- the redundancy management that submits non-redundant frames and
- an outgoing connection to the application consuming frames.

At a first glance it might look strange that the redundancy management is explicitly considered part of the environment. We want, however, to instantiate specifications of redundancy management algorithms, and instantiation in TLA$^+$ [18] is substitution. Furthermore steps of the redundancy management may have an effect on variables of environment and hence must be specified. The information gathered so far can be realized in a first specification of needed constants and variables shown in Figure 4.4. Furthermore, standard modules for natural numbers and sequences get extended to include their operators and functions. The TLC module is optional, but it allows some more debugging.

EXTENDS *Naturals*, *Sequences*, *TLC*

CONSTANTS

networks,	set of networks
SN_CNT, *SN_MAX*, *SN_HALF*,	maximum sequence number
MCFL,	maximum number of consecutive frame loss
MTF,	maximumb number of transient frames
A, *B*, *SN*, *TAG*	just for convenience

VARIABLES

rm,	Redundancy Management
env,	Environment including the redundant networks
out,	forwarded "frames"
status	debugging

Figure 4.4.: First part of environment's specification

Actions of the Environment

The first step to do is to define the actions the environment can perform. The set of possible actions can be given as follows:

Send a frame :

The environment sends frames to all connected, operating networks with a frequency bounded by the *Bandwidth Allocation Gap* shortly explained in Section 3.1. A certain BAG value tells us about the maximum difference of sequence numbers between deliverance of two twin frames. In detail the maximum *Sequence Number Skew* is defined to be:

$$SNS\_Max =_{def} 1 + \left\lfloor \frac{SkewMax}{BAG} \right\rfloor$$

However, I tried to avoid the usage of the BAG as it would have forced me to introduce further variables and I in any case must use bounded sequences to model the networks, as infinite sequences would cause an infinite and not checkable model with TLC. Hence I could just use these bounds to restrict maximum *Sequence Number Skew*. The drift of frames is bounded through finite capacity of networks, since feeding the faster network with frames requires consumption of frames from both networks, once the slower network reached its capacity. Actually the only thing to do is to choose a suitable capacity for the sequences. Intuitively, from

$$\begin{array}{l}
sendFrame \triangleq \\
\quad \land\ Len(env.frames.A) < MTF \\
\quad \land\ Len(env.frames.B) < MTF \\
\quad \land\ \exists\, id \in networks : isAlive[id] = \text{TRUE} \\
\quad \land\ env' = [env\ \text{EXCEPT} \\
\qquad\qquad !.frames = [\\
\qquad\qquad A \mapsto \text{IF}\ isAlive[A] \\
\qquad\qquad\qquad\qquad \text{THEN}\ Append(@.A, [sn \mapsto env.sn,\ tag \mapsto \text{“n”}]) \\
\qquad\qquad\qquad\qquad \text{ELSE}\ \langle\rangle, \\
\qquad\qquad B \mapsto \text{IF}\ isAlive[B] \\
\qquad\qquad\qquad\qquad \text{THEN}\ Append(@.B, [sn \mapsto env.sn,\ tag \mapsto \text{“n”}]) \\
\qquad\qquad\qquad\qquad \text{ELSE}\ \langle\rangle], \\
\qquad\qquad\quad !.sn = (@ + 1)\%SN_CNT] \\
\quad \land\ \text{UNCHANGED}\ \langle rm,\ out\rangle \\
\quad \land\ status' = \text{“send”}
\end{array}$$

Figure 4.5.: Send frame action of environment

Corollary 1 it can be deduced that the maximum *Sequence Number Skew* and therfore the capacity of each sequence that represents a network must be less or equal to SN_HALF. This is the maximum difference where two sequence number still can be ordered correctly. Figure 4.5 shows how this action is specified in TLA$^+$.

The first two lines of this action check whether both sequences of frames that model the networks still have a length less than the specified maximum MTF. It might be arguable that in the general specification, each network is allowed to buffer infinitely many frames, and that the length of each sequence should only be bounded for model checking. This could be done with a constraint formula defined in the TLC configuration file. I decided, however, to restrict the sequences directly in the specification and give arguments for this decision together with a small example in Chapter 5.2. Naturally a frame is sent only if at least one network is still operating. If these three guards are fulfilled, a frame can be sent and variables are updated accordingly.

Loose a frame:

Unfortunately the networks need not to be completely reliable, and thus it has to be taken into account that each network may loose frames. It is, however, specified that the number of consecutively lost frames is bounded, regardless of how this fact is ensured. At least it can be assumed that frames that violate this boundary, were discarded by integrity checking. Nevertheless, as this boundary is known it is possible to set the capacity of each network larger than the *Maximum Consecutive Frame Loss* which allows the removal of this action. This is easily done by specifying a set of recently deliverable frames (see Figure 4.6), which are all frames of a network that ensure a maximum consecutive frame loss less or equal to the specified constant *MCFL*. Taking such a set of deliverable frames the

$$
\begin{aligned}
\mathit{deliverable} \triangleq\ & \{\langle \mathit{id},\ \mathit{pos}\rangle \in \mathit{networks} \times (1\ ..\ (1 + \mathit{MCFL})) : \\
& \land\ \mathit{env.frames}[\mathit{id}] \neq \langle\rangle \\
& \land\ \mathit{pos} \leq \mathit{Len}(\mathit{env.frames}[\mathit{id}])\}
\end{aligned}
$$

Figure 4.6.: Set of deliverable frames

actually received frame can be chosen from it.

Receive a frame:

This action is actually the disjunction of two steps, namely the accept- and reject-step, where the instantiated actions of the redundancy management get called. Therefore an accept- or reject-step of the environment is just a corresponding step of redundancy management plus updating environment's variables. These actions take a frame from the set of deliverables as parameter. All frames ahead of the chosen one, in that sequence, are considered lost and get removed from the sequence. Updating the networks looks somewhat complicated, but it will be explained in the next section. Both actions specified in TLA$^+$ can be seen in Figure 4.7. Part of the next state specification is therefore a step of the redundancy management, which is existential quantification over the set of deliverable frames and a disjunction of either an accept-step or a reject-step. It can be seen in Figure 4.8.

accept frame:

$$
\begin{array}{l}
extAcceptFrame(id,\ sn,\ pos)\ \triangleq \\
\quad \land\ acceptFrame(id,\ sn) \\
\quad \land\ env' = [env\ \text{EXCEPT} \\
\qquad !.frames[id] = SubSeq(@,\ pos + 1,\ Len(@)), \\
\qquad !.frames[TNid[id]] = \\
\qquad\ \text{IF}\ Len(@) \geq Len(SubSeq(env.frames[id],\ pos,\ Len(env.frames[id]))) \\
\qquad\quad \text{THEN}\ tag[env.frames[TNid[id]], \\
\qquad\qquad\qquad SubSeq(env.frames[id],\ pos,\ Len(env.frames[id])), \\
\qquad\qquad\qquad env.frames[id][pos][SN],\ id] \\
\qquad\quad \text{ELSE}\ @] \\
\quad \land\ out' = out \cup \{env.frames[id][pos]\} \\
\quad \land\ status' = \text{“accept”}
\end{array}
$$

reject frame:

$$
\begin{array}{l}
extRejectFrame(id,\ sn,\ pos)\ \triangleq \\
\quad \land\ rejectFrame(id,\ sn) \\
\quad \land\ env' = [env\ \text{EXCEPT} \\
\qquad !.frames[id] = SubSeq(@,\ pos + 1,\ Len(@)), \\
\qquad !.frames[TNid[id]] = \\
\qquad\ \text{IF}\ Len(@) \geq Len(SubSeq(env.frames[id],\ pos,\ Len(env.frames[id]))) \\
\qquad\quad \text{THEN}\ tag[env.frames[TNid[id]], \\
\qquad\qquad\qquad SubSeq(env.frames[id],\ pos,\ Len(env.frames[id])), \\
\qquad\qquad\qquad env.frames[id][pos][SN],\ id] \\
\qquad\quad \text{ELSE}\ @] \\
\quad \land\ \text{UNCHANGED}\ \langle out \rangle \\
\quad \land\ status' = \text{“reject”}
\end{array}
$$

Figure 4.7.: Reception specified in TLA$^+$

$$
\begin{array}{lll}
SysNext & \triangleq & \exists\, \langle id,\ pos \rangle \in deliverable : \\
 & & \quad \lor\ extAcceptFrame(id,\ env.frames[id][pos][SN],\ pos) \\
 & & \quad \lor\ extRejectFrame(id,\ env.frames[id][pos][SN],\ pos) \\
 & & \quad \lor\ extWait
\end{array}
$$

Figure 4.8.: Definition of a redundancy mangement's step

Reset sequence number:

In case of a failure of the sending ES, it may recover and start sending frames with sequence numbers starting from zero again. This situation is modeled by simply setting the sequence number to zero again. Figure 4.9 shows that everything else is left unchanged. Unfortunately, the technical report of von Hanxleden and Gam-

reset of sequence number

$$
\begin{aligned}
reset \triangleq\ & \wedge \exists\, id \in networks : isAlive[id] = \text{TRUE} \\
& \wedge env' = [env \text{ EXCEPT } !.sn = 0] \\
& \wedge \text{UNCHANGED } \langle rm,\ out \rangle \\
& \wedge status' = \text{“reset”}
\end{aligned}
$$

Figure 4.9.: Sequence number reset

bardella [27] does not specify how often such a reset of sequence numbers may occur. Obviously, in the worst case frames with sequence number zero are sent continuously.

Die:

Specification of the protocol AFDX [3]allows a single network connection to permanently fail or send *mal-formed* frames, though both exceptions are the same to the redundancy management because it will not receive further frames from the faulty network. The redundancy management should then operate with the remaining network. Taking this action (Figure 4.10) disables a single network for good and deletes all messages, which remain on that network.

IF network is alive THEN network goes down

$$
\begin{aligned}
& die(id) \triangleq \\
& \quad \wedge isAlive[id] = \text{TRUE} \\
& \quad \wedge env' = [env \text{ EXCEPT } !.alive[id] = \text{FALSE},\ !.frames[id] = \langle\rangle] \\
& \quad \wedge \text{UNCHANGED } \langle rm,\ out \rangle \\
& \quad \wedge status' = \text{“die”}
\end{aligned}
$$

Figure 4.10.: Disable a single network

Pause:

Certainly the environment is allowed to do just nothing, more precisely it is assumed that the environment cannot send infinitely many frames in finite time. This progress in time can be used to decide about frames. Figure 4.11 shows the specification of this step that is enabled if both sequences have the same length, which models that delivery of twin frames consumes a negligible amount of time. It leaves the environment unchanged, but “calls” the specified reaction of the redundancy management.

```
time exceeds:
extWait ≜
        ∧ Len(env.frames[A]) = Len(env.frames[B])
        ∧ wait
        ∧ UNCHANGED ⟨env, out⟩
        ∧ status′ = "wait"
```

Figure 4.11.: Specification of wait action

Figure 4.8 gives a definition of a redundancy management's step. To complete the specification of possible actions, the behavior of the environment without the redundancy management part must be defined and put together like in Figure 4.12. A not less

```
EnvNext ≜ ∃ id ∈ networks : sendFrame ∨ die(id) ∨ reset
SysNext ≜ ∃ ⟨id, pos⟩ ∈ deliverable :
            ∨ extAcceptFrame(id, env.frames[id][pos][SN], pos)
            ∨ extRejectFrame(id, env.frames[id][pos][SN], pos)
            ∨ extWait
A system-step is a step either of environment or of Redundancy Management
Next ≜ SysNext ∨ EnvNext
```

Figure 4.12.: Step definition of environment without redundancy management

important task is to think about certain fairness conditions needed to reach a proper working model. There are two premises that should be ensured:

1. The environment shall send infinitely many frames.

2. The redundancy management shall decide about every frame.

The above given specification of sending and receiving frames shows that weak fairness is all that is needed. To send a frame there must be remaining capacity in both networks and thus it must be ensured that eventually all frames are taken out of their sequences. Obviously this can be reached with weak fairness, too, as at least one of the receiving steps is enabled if at any of the networks contains a frame. Finally there must be a defined set of initial states, from which the state graph is calculated with help of the

```
InitEnv ≜ env ∈ [
            sn : {0},
            frames : [A : {⟨⟩}, B : {⟨⟩}],
            alive : [A : BOOLEAN , B : {TRUE}]
          ]
```

Figure 4.13.: Defining initial values of variables

next-state function and the specified fairness formula. Specification of initial states can be seen in Figure 4.13 and the final specification formula for the environment is given in Figure 4.14. Remember that the clause *InitRM* is the incorporation of redundancy management's initialization. If one is only interested in the pure specification, one could

$$\begin{aligned} \mathit{Fairness} \;&\triangleq\; \wedge\, \mathrm{WF}_{\langle rm,\, out,\, env,\, status\rangle}(\mathit{sendFrame}) \\ &\quad\; \wedge\, \mathrm{WF}_{\langle rm,\, out,\, env,\, status\rangle}(\mathit{SysNext}) \\ &\quad\; \wedge\, \mathrm{SF}_{\langle rm,\, out,\, env\rangle}(\mathit{extWait}) \\ \mathit{Spec} \;&\triangleq\; \mathit{Init} \wedge \Box[\mathit{Next}]_{\langle rm,\, out,\, env,\, status\rangle} \wedge \mathit{Fairness} \end{aligned}$$

Figure 4.14.: Specification formula for the environment

however, stop at this point or may jump to Section 4.3 for the specification of requirements. In order to enable machine verification of the later on specified requirements, the given specification will be extended by some further information.

Enable Verification

As already stated, we have to face the fact that the proposed algorithms may fail some properties under certain circumstances. To recognize such faulty behaviors, a system must be introduced that, independently of the actual program state, marks pending frames correctly, corresponding to their status as either "normal", "redundant" or "old", with the following denotations:

- A frame is considered to be "normal" **iff**
  - no frame sent later to any network has yet been received and
  - its twin frame has not yet been received.
- A frame is considered to be "redundant" **iff**
  - its twin frame has already been delivered to the redundancy management.
- A frames is considered to be "old" **iff**
  - it is not redundant and
  - another frame, later sent to the networks, has already been delivered to the redundancy management.

Therefore each frame gets another tag with initial tag "normal" resulting in a final frame model shown in Figure 4.15. The next step is to find an easy algorithm to mark all frames correctly with a minimum effort.

1 byte	1 byte	1 byte
Seq Number	Network ID	Status Tag

Figure 4.15.: Final frame model

Description

For better comprehension of the explanation of the algorithm, some facts about the representation of the networks are informally introduced. Each network is modeled as a sequence of frames.

Definition 9. *(Sequence Operations) For each sequence s it is:*

1. *$Len(s)$ the number of elements in s*
2. *$Tail(s)$ the sequence s without its first element*
3. *$Head(s)$ the first value of sequence s, if s is not empty,*
4. *$SubSeq(s, n, m)$ the sequence that contains all elements of s from position n to m*
5. *For sequences s_1, s_2, it is $s_1 \circ s_2$ the concatenated sequence*

The status of a frame, is one of the above described values “normal”, “redundant” or “old”.The environment sends its frames to both networks in parallel and only if both networks still have capacity to buffer another frame. Thus for a given frame f from network N_1 it is

1. decidable whether f's twin frame is still transient on N_2, and
2. possible to find the position of f's twin frame in the sequence where it is located.

Let N_1 be a non-empty network and f a frame in N_1, located at $n \in \{1, \ldots, Len(N_1)\}$. It can be deduced that the twin frame of f is still pending on the second network N_2 if and only if

$$Len(SubSeq(N_1, n, Len(N_1))) \leqslant Len(N_2). \tag{4.1}$$

More precisely, given networks N_1 and N_2 with $Len(N_1) \leqslant Len(N_2)$, we know that, selecting frame at position $0 < k \leqslant Len(N_1)$ from network N_1, its twin frame on network N_2 is located at position l for which holds:

$$Len(SubSeq(N_1, k, Len(N_1))) = Len(SubSeq(N_2, l, Len(N_2))) \tag{4.2}$$

Recall the definition of redundancy management's steps, seen in Figure 4.8. Every frame is given as a tuple of its network descriptor and the position in the network's sequence. That is all one needs to mark frames correctly, which means that all frames sent logically before the actually received frames get marked as “old” and its twin frames, if it is still transient, gets marked as “redundant”. Subsequently Figure 4.16 adheres to a TLA^+-like syntax and gives the marking algorithm in pseudocode.

```
procedure tag(s1, s2 : Sequence)
IF Len(s1) ⩽ Len(s2)
THEN
  IF status(Head(s2)) = r THEN tag(s1,Tail(s2))
  ELSE status(Head(s2)) ← o;
       tag(s1,Tail(s2))
ELSE status(Head(s2)) ← r
```

Figure 4.16.: Marking algorithm in pseudocode

Correctness Proof

Proof. I will give a proof of the described algorithm above using assertional reasoning. Foundations and methods for proving sequential and concurrent programs can be found in books from de Roever [11] and Apt [2].

I will follow *Floyd's Inductive Assertion Method for Transition Diagrams* [13][25], in detail described by de Roever [11]. A transition diagram is a tuple (L, T, s, t), where L denotes the finite set of locations, T is the finite set of transitions, represented by tuples $(l, c \to f, l')$ with $l, l' \in L,\ c : \Sigma \to \mathit{Bool},\ f : \Sigma \to \Sigma$ and $s, t \in L$ as entry- and exit location. Subsequently $(l \xrightarrow{c/f} l')$ will be used as an abbreviation for $(l, c \to f, l')$. Let P be the marking algorithm and build a transition diagram that holds

$$P \equiv (\{s, l, t\}, \{(s \xrightarrow{c_0/f_0} l_0), (s \xrightarrow{c_1/f_1} l_0), (s \xrightarrow{c_3/f_3} t), (l_0 \xrightarrow{c_2/f_2} s), (l_0 \xrightarrow{c_3/f_3} t)\}, s, t), \quad (4.3)$$

where

- $c_0(\sigma) = \mathit{true}\ \mathit{iff}\ (s_1 < s_2) \wedge (\mathit{status}(\mathit{Head}(s_1)) = n)$,
- $c_1(\sigma) = \mathit{true}\ \mathit{iff}\ (s_1 < s_2) \wedge (\mathit{status}(\mathit{Head}(s_1)) \neq n)$,
- $c_2(\sigma) = \mathit{true}\ \mathit{iff}\ s_1 > s_2$,
- $c_3(\sigma) = \mathit{true}\ \mathit{iff}\ s_1 = s_2$,
- $f_0(\sigma) = (\sigma : s_2, s_3, s_4 \mapsto \mathit{Tail}(\sigma(s_2)), \sigma(s_2), \mathit{Append}(\sigma(s_4), (\mathit{sn}(\mathit{Head}(\sigma(s_2))), o))$,
- $f_1(\sigma) = (\sigma : s_2, s_3, s_4 \mapsto \mathit{Tail}(\sigma(s_2)), \sigma(s_2), \mathit{Append}(\sigma(s_4), \mathit{Head}(\sigma(s_2))))$,
- $f_2(\sigma) = (\sigma)$,
- $f_3(\sigma) = (\sigma : s_2 \mapsto \langle(\mathit{sn}(\mathit{Head}(\sigma(s_2))), r)\rangle \circ \mathit{Tail}(\sigma(s_2)))$.

I use $s_1 > s_2$ for sequences s_1, s_2 as an abbreviation for $\mathit{Len}(s_1) > \mathit{Len}(s_2)$ and so on. Figure 4.17 shows the graphical transition diagram that is much better to understand.

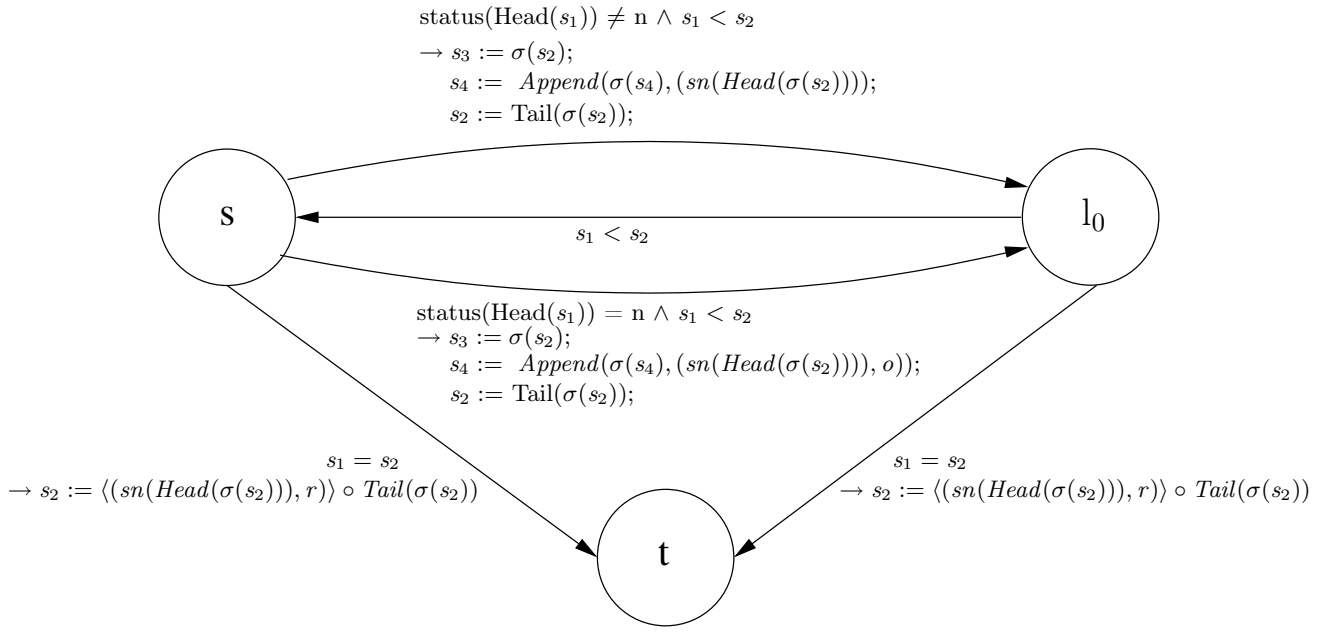

Figure 4.17.: Transition Diagram for marking algorithm

A specification for program P is given by a pair of logical predicates $\langle \phi, \psi \rangle$, where ϕ denotes the precondition and ψ the postcondition. Correctness of P is checked with regard to a this tuple $\langle \phi, \psi \rangle$. The next steps are:

1. find suitable pre- and postconditions,
2. define assertions for every location $l \in L$,
3. show inductiveness of the chosen assertions and
4. finally prove *success* and *convergence*.

It is possible to decide whether the twin frame of the actually received frame is still transient on the other network. This decision is straightforward. With the argumentation above the twin frame of a frame f at position k from a non-empty network N_1 is still not delivered yet if and only if Equation 4.1 is satisfied. Moreover, no frame has to be marked as "old" if Equation 4.1 is not true. Thus we define our precondition as

$$\phi = \{s_1 \leqslant s_2\} \tag{4.4}$$

The postcondition shall state the correctness condition. It is designated that all frames sent before the actually received frame and not already marked must get a new tag. Let us again consider non-empty networks N_1 and N_2. Assume a frame f at position $k < Len(N_1) \leqslant Len(N_2))$ to be from network N_1. The frames sent before f are all frames ahead of f in network N_1 plus all frames ahead of f's twin frame in N_2. Since all frames

ahead of f are considered lost, one has to take care of frames in N_2. The number of frames to be marked can be given by

$$n = Len(SubSeq(N_2, l, Len(N_2))) - Len(SubSeq(N_1, k, Len(N_1))), \tag{4.5}$$

where l is the position of f's twin frame in N_2. Hence the marking algorithm shall tag n frames. None of these frames should be tagged as "normal". Additionally the algorithm must tag the twin frame as "redundant". In summary, ψ is given by

$$\begin{aligned}\psi = \{&s_1 = s_2 \\ &\wedge status(Head(s_2)) = r \\ &\wedge s_4 = Subseq(N_2, 1, l-1) \\ &\wedge \forall\, frame \in s_4 : status(frame) \in \{o, r\}\},\end{aligned} \tag{4.6}$$

where we may assume that if $l - 1$ is less than one, the *SubSeq* operator returns an empty sequence.

In the following I will give assertions for all locations $l \in L$ and show their inductiveness.

$$\begin{aligned}Q_s &= s_1 \leqslant s_2 \\ Q_{l_0} &= s_1 \leqslant s_2 \\ Q_t &= (s_1 = s_2) \wedge (status(Head(s_2)) = r)\end{aligned} \tag{4.7}$$

To ease the correctness proof I want to give an invariant about elements of the s_4, which contains the marked frames. The invariant is

$$inv = \forall\, frame \in s_4 : status(frame) \in \{o, r\}, \tag{4.8}$$

and its proof is trivial, because there are only two transitions that append frames to s_4. The first transition is f_0 guarded by c_0, so the status of first frame from s_2 is "normal" and function f_0 appends this frame to s_4 with setting its status to "old". The second way is f_1 guarded by c_1. Since c_1 ensures that first frame already has a status tag equal to o or r, the invariant holds.

According to our prove steps I will subsequently show inductiveness of the assertions, that is for two assertion Q_{l_0}, Q_{l_1} one has to prove that

$$\models\ Q_{l_0} \wedge c \rightarrow Q_{l_1} \circ f \tag{4.9}$$

is valid. Applying this approach to the pair (Q_s, Q_{l_0}) we see that

$$\begin{aligned}&\models Q_s \wedge c_0 \rightarrow Q_{l_0} \circ f_0 \\ \equiv\ &\models (s_1 \leqslant s_2) \wedge (s_1 < s_2) \wedge (status(Head(s_1)) = n) \\ &\quad \rightarrow\ (s_1 \leqslant s_2)\end{aligned} \tag{4.10}$$

is obviously correct, since c_1 trivially implies $s_1 < s_2$. The second transition from s to l_0 preserves inductiveness as well. Similarly

$$\begin{aligned}&\models Q_s \wedge c_3 \rightarrow Q_t \circ f_3 \\ \equiv\ &\models (s_1 \leqslant s_2) \wedge (s_1 = s_2) \\ &\quad \rightarrow\ (s_1 = s_2) \wedge (status(Head(s_2)) = r)\end{aligned} \tag{4.11}$$

is straightforward. Since trivially $Q_{l_0} \Rightarrow Q_s$ holds and the transition function does nothing to the variables, there is nothing to do for this step. Finally it must be shown that

$$\models \; Q_{l_0} \wedge c_3 \rightarrow Q_t \circ f_3 \tag{4.12}$$

holds. As Q_{l_0} equals Q_s it can be concluded from equation (4.11) that equation (4.12) is valid too.

To complete the proof it must be shown that $\phi \rightarrow Q_s$, which is actually true as $\phi = Q_s$, and that $Q_t \rightarrow \psi$ holds.

$$\begin{aligned} (s_1 = s_2) \wedge (\mathit{status}(\mathit{Head}(s_2)) = r) & \\ \rightarrow \quad (s_1 = s_2) & \\ \wedge\, \mathit{status}(\mathit{Head}(s_2)) = r & \\ \wedge\, s_4 = \mathit{Subseq}(N_2, 1, l-1) & \\ \wedge\, \forall\, \mathit{frame} \in s_4 : \mathit{status}(\mathit{frame}) \in \{o, r\} & \end{aligned} \tag{4.13}$$

To prove Equation 4.13 only the last two clauses must be considered, because the first two are equal to Q_t. Furthermore the invariant defined in Equation 4.8 ensures that the last clause holds and hence I just have to show that all frames located before f's twin frame in sequence s_2 were appended to s_4. However this is quite obvious, since at the time when each of these frame is considered in the algorithm, we have that $s_1 < s_2$ holds. This implies that s is entered and left again through one of its enabled outgoing transitions $(s \xrightarrow{c_0/f_0} l_0), (s \xrightarrow{c_1/f_1} l_0)$. Justified by the definitions of f_0 and f_1 each element of sequence s_2 located before f's twin frame will be appended to sequence s_4.

Convergence in turn is proved straightforwardly, as $(\mathbb{N}, <)$ is known to be a well-founded set and thus no infinitely descending sequence $\ldots < w_2 < w_1 < w_0$ $w_i \in \mathbb{N}$ exists. Consequently the sequence s_2 cannot get shortened infinitely often by one, without getting equal to s_1. This though leads to the termination node. To be completely consistent with *Floyd's Inductive Assertion Method*, ranking functions could have been defined for each location and it could have been shown that they decrease in every step, but this is technical only and does not improve comprehension.

□

4.3. Requirements

So far a specification was built to check nearly every redundancy management algorithm that can be specified in TLA$^+$. The properties that the algorithms should satisfy will be defined and expressed in TLA$^+$, such that it can be checked afterwards if the implication *spec* $\Rightarrow$ *prop* holds for every specification *spec* of a redundancy management algorithm and each property *prop* it should satisfy. First of all, a set of properties must be defined. In a first approach, the informally given requirements from the technical report [27] should be formalized and expressed in TLA$^+$. It gives a large set of requirements concerning all parts of the *End System*. Of course, only the requirements on the

redundancy management and the *High Level Requirements* are of interest. However, it turned out that most of these properties were much too imprecise to be formulated in TLA$^+$. For example requirement 5 states:

> **The redundancy shall increase the macroscopic availability of the avionics network.**

This is much too coarse and would not help to distinguish the candidates, and moreover it is quite challenging to reformulate this requirement in any formalism. Therefore, we decided to build up a set of properties from scratch. The gist of the original requirements shall be preserved but refined enough to enable differentiation of the redundancy management algorithms. This results in a set of 18 properties grouped into 3 parts. Subsequently they are explained informally and formally – written in TLA$^+$.

Safety

Traditionally an algorithm is examined to satisfy safety and liveness properties, stating that it behaves correctly and does not block. This is not appropriate in our situation as we do not expect any of the redundancy management algorithms to be completely safe. That is why in our case safety properties were formulated relative to the behavior of the environment. Subsequently the intention of each property is given textually and afterwards formally in TLA$^+$.
What does safety mean for a redundancy management? First of all, the redundancy management shall not submit any redundant frames to the application layer. Secondly the redundancy management shall preserve the order of frames and hence shall not submit *old* frames to the application layer. Each of these tasks can be weakened accordingly to the benignancy of the environment.

1. ***Redundancy 1*: No redundant frame shall ever be submitted to the application layer.**

$$Redundancy1 \triangleq \forall\, frame \in out : frame[TAG] \neq \text{“r”}$$

   This formula just checks that no frame in the set of submitted frame has a tag that marks it as a redundant frame. It can be checked by TLC very efficiently because it is a state predicate. This is a very strong requirement and it is doubtful whether one algorithm will never submit a redundant frame.

2. ***Redundancy 2*: If the environment does not reset anymore, the redundancy management stabilizes and works properly from that time on.**

$$\begin{aligned} Redundancy2 \triangleq\ & \Diamond\Box\neg[reset]_v \\ & \Rightarrow \Diamond\Box(\forall \langle id, pos\rangle \in deliverable : \\ & \text{ENABLED } \langle extAcceptFrame(id, env.frames[id][pos][SN], pos)\rangle_v \\ & \quad \Rightarrow env.frames[id][pos][TAG] \neq \text{“r”}) \end{aligned}$$

The premise of the TLA$^+$-formula expresses that from some state σ_i on, all consecutive states $\sigma_n \rightarrow \sigma_m$ are neither a *reset* nor a stuttering step. If this holds the redundancy management algorithm shall, after a finite time, only accept frames which are not marked as redundant. This property and all other properties that allow a certain time of bad functional behaviour show that the concept of a set of submitted frames is not strong enough for all properties.

3. ***Redundancy 3*: If the environment does not reset anymore, the redundancy management stabilizes and works properly, except for a finite sequence of tolerated failures.**

$$\begin{aligned} \mathit{Redundancy3} \triangleq\ & \Diamond\Box\neg[\mathit{reset}]_v \\ & \Rightarrow \Box\Diamond(\forall\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \text{ENABLED}\ \langle \mathit{extAcceptFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v \\ & \quad \Rightarrow \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{TAG}] \neq \text{“r”}) \end{aligned}$$

The premise equals the one above, but the implication is weakened. It is now only expected that from time to time frames are accepted correctly. It is a known and intuitive result from logic that $\Diamond\Box p \Rightarrow \Box\Diamond p$ holds for every state predicate p. See [23] for a detailed proof system which enables a straightforward argumentation.

Analogously to the above defined properties concerning redundancy we define the properties concerning order as follows.

4. ***Order 1*: No old frame shall ever be submitted to the application layer.**

$$\mathit{Ordering1} \triangleq \forall\, \mathit{frame} \in \mathit{out} : \mathit{frame}[\mathit{TAG}] \neq \text{“o”}$$

5. ***Order 2*: If the environment does not reset anymore, the redundancy management stabilizes and preserves order from that time on.**

$$\begin{aligned} \mathit{Ordering2} \triangleq\ & \Diamond\Box\neg[\mathit{reset}]_v \\ & \Rightarrow \Diamond\Box(\forall\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \text{ENABLED}\ \langle \mathit{extAcceptFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v \\ & \quad \Rightarrow \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{TAG}] \neq \text{“o”}) \end{aligned}$$

6. ***Order 3*: If the environment does not reset anymore, the redundancy management stabilizes and preserves order excepting for a finite sequence of frames.**

$$\begin{aligned} \mathit{Ordering3} \triangleq\ & \Diamond\Box\neg[\mathit{reset}]_v \\ & \Rightarrow \Box\Diamond(\forall\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \text{ENABLED}\ \langle \mathit{extAcceptFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v \\ & \quad \Rightarrow \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{TAG}] \neq \text{“o”}) \end{aligned}$$

Though it seems that the number of temporal formulas would dramatically increase the time needed for model checking, it is still possible. Properties *Redundancy 1* and *Order 1* are state predicates and hence they can be checked while exploring the state space. It is known that *Redundancy 2* implies *Redundancy 3* as well as *Order 2* implies *Order 3*. With these relations, depending on an initial guess either the one or the other property could be checked first, hoping that the known implications will make a second check superfluous. The last properties about redundancy and ordering can be checked quite fast since behavior of the environment is limited to non-*reset*-step, which clearly reduces the state space.

Liveness

Of course the redundancy management algorithms shall not deadlock as long as it receives frames from its environment. More specifically:

1. ***Liveness*: Each frame that is delivered to the redundancy management will be either accepted or rejected.**

$$\begin{aligned} \mathit{Liveness} \triangleq\ & \forall \langle \mathit{id}, \mathit{pos} \rangle \in \mathit{deliverable} : \\ & \vee \text{ENABLED } \langle \mathit{extAcceptFrame}(\mathit{id}, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}], \mathit{pos}) \rangle_v \\ & \vee \text{ENABLED } \langle \mathit{extRejectFrame}(\mathit{id}, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}], \mathit{pos}) \rangle_v \end{aligned}$$

   It is expected that all algorithms will satisfy this property as they are all of type: IF *condition* = TRUE THEN *accept* ELSE *reject*, which implies that there exists a unique decision for each frame. That is why we do not mind that our formula is stronger than needed as this formula reasons about all frames and not only the set of received frames. It is enough to know that the *Liveness*-formula implies absence of deadlocks.

Quality

Requirement 4 from von Hanxleden and Gambardella [27] states:

> **The redundancy shall not increase the availability of a single network, but it shall maintain it. In other words, it shall not increase the number of frames lost that would be obtained with one of two networks normally running and alone.**

This is a difficult, but important demand. Assume a fast but unreliable, hence lossy network A and a slower, completely reliable network B. It follows that an algorithm needs to use buffering to solve this problem. Buffering is considered harmful since it produces too large delays. So the upcoming gradation shall be a more realistic estimation for the performance of the algorithms.

1. ***Quality 0*: If only one network is connected to the redundancy management, all received frames are forwarded**

$$\begin{aligned} \mathit{Quality0} \triangleq\ & \Box(\forall\, \mathit{id1},\, \mathit{id2} \in \mathit{networks} : \mathit{isAlive}[\mathit{id1}] \wedge \neg \mathit{isAlive}[\mathit{id2}]) \\ & \Rightarrow \Box(\forall\, \langle \mathit{id},\, \mathit{pos} \rangle \in \mathit{deliverable} : \\ & \quad \neg \text{ENABLED } \langle \mathit{extRejectFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos}) \rangle_v) \end{aligned}$$

This is an obvious requirement if there should be a chance to satisfy the requirement from the original paper. If the redundancy management receives only frames from one network, the whole ES shall behave as with only one network running alone.

2. ***Quality 1*: If both networks are alive and at least one member of a *Twin Frame* reaches the redundancy management, one member gets submitted.**

$$\begin{aligned} \mathit{Quality1} \triangleq\ & \forall\, \mathit{id} \in \mathit{networks},\, \mathit{pos} \in (1\,..\,\mathit{MCFL}) : \\ & \wedge\, \mathit{isAlive}[\mathit{id}] \\ & \wedge\, \mathit{isAlive}[\mathit{TNid}[\mathit{id}]] \\ & \wedge\, \langle \mathit{id},\, \mathit{pos} \rangle \in \mathit{deliverable} \\ & \wedge \text{ ENABLED } \langle \mathit{extRejectFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos}) \rangle_v \\ & \Rightarrow \exists\, \mathit{frame} \in \mathit{out} : \mathit{frame}[\mathit{SN}] = \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}] \end{aligned}$$

Quality 1 is equivalent expression to the original requirement. It is easier to formalize the equivalent expression, that if a frame gets rejected, its twin frame has already successfully passed the redundancy management algorithm.

3. ***Quality 2*: If both networks are alive, at least one member of a *Twin Frame* reaches the redundancy management and this frame is neither *redundant* nor *old*, it gets submitted.**

$$\begin{aligned} \mathit{Quality2} \triangleq\ & \forall\, \mathit{id} \in \mathit{networks},\, \mathit{pos} \in (1\,..\,\mathit{MCFL}) : \\ & \wedge\, \mathit{isAlive}[\mathit{id}] \\ & \wedge\, \mathit{isAlive}[\mathit{TNid}[\mathit{id}]] \\ & \wedge\, \langle \mathit{id},\, \mathit{pos} \rangle \in \mathit{deliverable} \\ & \wedge \text{ ENABLED } \langle \mathit{extRejectFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos}) \rangle_v \\ & \wedge\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{TAG}] = \text{“n”} \\ & \Rightarrow \exists\, \mathit{frame} \in \mathit{out} : \mathit{frame}[\mathit{SN}] = \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}] \end{aligned}$$

This weakened version claims that frames that are neither *redundant* nor *old* must be accepted. This is the minimal attribute a good algorithm should have.

4. ***Quality 3*: If the environment does not reset anymore, *Quality 2* holds.**

$$\mathit{Quality3} \triangleq \Diamond\Box\neg[\mathit{reset}]_v \Rightarrow \mathit{Quality2}$$

This property should be satisfied by each algorithm, as we restrict the possible behaviors of the environment such that eventually forever no further *reset*-steps occur.

5. ***Reset*: The redundancy management algorithm is expected to stabilize after a reset of sequence numbers.**

$$\begin{aligned} \mathit{Reset}1 \triangleq\ & \Diamond\Box(\neg[\mathit{reset}]_v \wedge \forall\, \mathit{id} \in \mathit{networks} : \mathit{isAlive}[\mathit{id}]) \\ & \Rightarrow \Box\Diamond(\exists\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \quad \langle \mathit{extAcceptFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v) \end{aligned}$$

We want the redundancy management to work properly after a reset of sequence numbers. Since both networks may be operating, we cannot check for more than acceptance of frames. This does not guarantee that the reset leads not to additional unwanted rejections.

Availability

The goal of redundancy is to raise the availability of a system by duplicating parts of a system. In our case communication is done over multiple networks, since one single network is not reliable enough. The *Quality* requirements state about availability of the system in case of both networks operating. Now the case is considered that one network dies. This is by far the most important part of the properties. Redundancy is used to remain operating in presence of partial failures. These properties tell us how good an algorithm serves this task and finally enables a final decision whether this algorithm is a feasible choice.

1. ***Avail 1*: If one network fails, all consecutive frames of the remaining network are accepted.**

$$\begin{aligned} \mathit{Avail}1 \triangleq\ & \exists\, \mathit{id}1,\, \mathit{id}2 \in \mathit{networks} : (\mathit{isAlive}[\mathit{id}1] \wedge \neg \mathit{isAlive}[\mathit{id}2]) \\ & \Rightarrow (\forall\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \quad \neg\textsc{enabled}\ \langle \mathit{extRejectFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v) \end{aligned}$$

An algorithm that satisfies this property would be a real good choice (unless it fails all other requirements). But it seems very likely that to satisfy this formula the redundancy management must be able to detect whether a network is down. This task, however, was explicitely moved to the *Network Management*.

2. ***Avail 2*: If one network fails and no *reset*-steps occur from that time on, all consecutive frames of the remaining network are accepted.**

$$\begin{aligned} \mathit{Avail}2 \triangleq\ & \Box(\ \wedge\, \Box(\mathit{status} \neq \text{“reject”}\,) \\ & \quad \wedge\, \exists\, \mathit{id}1,\, \mathit{id}2 \in \mathit{networks} : (\mathit{isAlive}[\mathit{id}1] \wedge \neg \mathit{isAlive}[\mathit{id}2]) \\ & \quad \Rightarrow (\forall\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \quad\quad \neg\textsc{enabled}\ \langle \mathit{extRejectFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v)) \end{aligned}$$

The situation is now less complicated due to the restriction of the environment's behavior, but the problem from above remains.

3. ***Avail 3*: If one network fails, after some time the redundancy management stabilizes and all consecutive frames of the remaining network are accepted.**

$$\begin{aligned} \mathit{Avail3} \triangleq\ & \Diamond\Box(\exists\, \mathit{id}1,\, \mathit{id}2 \in \mathit{networks} : (\mathit{isAlive}[\mathit{id}1] \wedge \neg \mathit{isAlive}[\mathit{id}2])) \\ & \Rightarrow \Diamond\Box(\forall\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \quad \neg\text{ENABLED}\ \langle \mathit{extRejectFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v) \end{aligned}$$

This property is much more likely to be fulfilled. Imagine the faster network dies and the slower network delivers the messages. The first frames will be rejected assuming that the faster one has already delivered them. Nevertheless once the slower network delivers a frame with a sequence number higher than the last delivered by the fast network, all consecutive frames can be accepted if the redundancy management works properly.

4. ***Avail 4*: If one network fails and no *reset*-steps occur from that time on, *Avail 3* holds.**

$$\begin{aligned} \mathit{Avail4} \triangleq\ & \Diamond\Box(\neg[\mathit{reset}]_v \wedge \exists\, \mathit{id}1,\, \mathit{id}2 \in \mathit{networks} : (\mathit{isAlive}[\mathit{id}1] \wedge \neg \mathit{isAlive}[\mathit{id}2])) \\ & \Rightarrow \Diamond\Box(\forall\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \quad \neg\text{ENABLED}\ \langle \mathit{extRejectFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v) \end{aligned}$$

Just a simplification of *Avail 3* as it can now be assured that eventually the remaining network sends frames with sequence number equal or higher to the last delivered by the faster network.

5. ***Avail 5*: If one network fails, forever frames of the remaining network gets accepted.**

$$\begin{aligned} \mathit{Avail5} \triangleq\ & \Diamond\Box(\exists\, \mathit{id}1,\, \mathit{id}2 \in \mathit{networks} : (\mathit{isAlive}[\mathit{id}1] \wedge \neg \mathit{isAlive}[\mathit{id}2])) \\ & \Rightarrow \Box\Diamond(\exists\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \quad \text{ENABLED}\ \langle \mathit{extAcceptFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v) \end{aligned}$$

Avail5 allows the redundancy management to reject up to $SN\_CNT - 2$ or $2^n - 2$ consecutive frames, assuming that sequence numbers are in $\{0..2^n - 1\}$ and $n \in \{0, \ldots, 8\}$. Algorithms that fail *avail5* may observe a behaviour where, from a certain point on, no frame will be accepted.

6. ***Avail 6*: If one network fails and no *reset*-steps occur from that time on, *Avail 5* holds.**

$$\begin{aligned} \mathit{Avail6} \triangleq\ & \Diamond\Box(\neg[\mathit{reset}]_v \wedge \exists\, \mathit{id}1,\, \mathit{id}2 \in \mathit{networks} : (\mathit{isAlive}[\mathit{id}1] \wedge \neg \mathit{isAlive}[\mathit{id}2])) \\ & \Rightarrow \Box\Diamond(\exists\, \langle \mathit{id},\, \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \quad \text{ENABLED}\ \langle \mathit{extAcceptFrame}(\mathit{id},\, \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\, \mathit{pos})\rangle_v) \end{aligned}$$

This is again just a relaxation of *Avail 5*.

There are many more properties of the given system specification that could be checked with TLC. To find an optimal set of properties – if such a set exists – is beyond this work.

4.4. Specification of redundancy management

This section gives the specifications of the proposed algorithms and compares the results of model checking. Prior to a first specification, a general description of the redundancy management will be given.

The redundancy management receives continuously frames from two networks after each frame passed integrity checking. This is modeled by extending each frame with a marker for its transmitting network. Figure 3.4 shows a complete frame, and Figure 4.3 shows a reduced version containing all information needed for the redundancy management. Additionally a status tag explained in Section 4.2 is needed for model checking. The resulting model of a frame is shown in Figure 4.15.

The specification of actions is trivial, because all the redundancy management can do is to receive a frame and decide whether it should submit or discard it. Further actions of the redundancy management will be introduced later, as it will be necessary to incorporate behaviours that depend on time.

Some advice concerning model checking

Of course, it is a design goal to write specifications that can be model checked directly. This is, however, sometimes difficult or even impossible. TLA$^+$ provides more than one way to improve model checking by reducing the state space. It is effective to define a different *view* of the model, which is basically defining a new set of variables. Obviously, less variables imply less states. Debugging variables, however, provide often the only way to figure out what is happening. Moreover, using such a user defined *view* causes TLC to generally omit variable names in counter examples. Second best effective method to reduce the state space is to take advantage of the symmetry, and symmetry encounters in the redundancy management algorithm's specification, since it does not have any effect to change networks A and B with each other. Unfortunately, in TLA$^+$ a symmetry function must have model values in domain and range and this is not suitable for various reasons. To make up for the disadvantages I specified constraints to ensure that, *e. g.*, network B is always alive. Of course, this does not save as many states as a complete symmetry specification would, because as long as both networks are alive the model checker permutes the networks for each possible interleaving. Last but not least, though some algorithms differ only a little, I removed unneeded fields in records to reduce the state space.

Next I had to choose concrete values for the specified constants. It turned out that this is maybe the most awkward task. Even a single change could cause additional hours for model checking. Three choices influence the model checking process the most. First

the choice of the set of sequence numbers. Experimental results on a *dual Opteron PC* with a 64-bit *Java Virtual Machine* found $SN\_CNT = 6$ to be the maximum applicable. Corollary 1 implies $SNS\_MIN - 1$ to be the maximum difference between sequence numbers that still allows correct ordering. Therefore the remaining important constants for *MTF*, which determines the length of the network sequences, and for *MCFL*, which specifies the maximum consecutive frame loss, must be chosen in a way that:

- two frames f_1, f_2 both deliverable to the redundancy management from a sequence N have sequence numbers that differ at most by $SN\_HALF - 1$, and
- two frames f_1, f_2, where f_1 is from N_1 and f_2 from N_2, have sequence numbers that have a difference less than SN_HALF.

Finally I chose to reduce the value for *MCFL* to one, which allows a consecutive frame loss of one frame. However it makes no difference as even a consecutive frame loss of two would still guarantee the closeness of sequence numbers that is assumed. Similarly I set the value of *MTF* to 2, which allows to buffer two frames per network sequence and results in a maximum sequence number skew of 2, less than $SN\_Half = 3$. Larger values would cause longer model checking periods or even force me to enlarge the set of sequence numbers, which I experienced to be very difficult.

To slightly automatize the model checking process, I wrote a TLC configuration file for each property to check, regardless of mathematical relations among several properties that might help to reduce the number of model checking processes. Of course, this approach costs much more time than a standard approach of checking several properties together. Nevertheless the standard approach can hardly be automated and requires more human effort as is explained in Section 5.2.

Results of formal analysis

	RMA1	RMA2	RMA3	RMA4	RMA5	RMA6	RMA7	RMA7*	RMA8	RMA9	RMA11	RMA12	RMA13
Availability1	X	X	X	X	X	X	√	X	X	X	X	X	X
Availability2	X	X	X	X	X	X	√	X	X	X	X	X	X
Availability3	X	X	X	X	X	X	√	√	X	X	X	X	√
Availability4	X	√	√	X	√	√	√	√	√	X	X	√	√
Availability5	X	X	X	X	X	X	√	√	√	√	√	X	√
Availability6	√	√	√	√	√	√	√	√	√	√	√	√	√
Liveness	√	√	√	√	√	√	√	√	√	√	√	√	√
Order1	X	X	X	X	X	X	X	X	X	X	X	X	√
Order2	√	√	√	√	√	√	(X)	√	√	X	X	X	√
Order3	√	√	√	√	√	√	(X)	√	√	X	√	√	√
Quality0	√	X	X	√	X	√	√	√	X	X	X	X	√
Quality1	X	X	X	X	X	X	√	X	X	X	√	√	X
Quality2	X	X	X	X	X	X	√	X	X	X	√	√	X
Quality3	√	√	√	√	√	√	√	√	√	X	√	√	X
Redundancy1	X	X	X	X	X	X	X	X	X	X	√	√	√
Redundancy2	√	√	√	√	√	√	X	√	√	√	√	√	√
Redundancy3	√	√	√	√	√	√	X	√	√	√	√	√	√
Reset	√	√	√	√	√	√	√	√	√	√	√	√	√

Figure 4.18.: Model checking results

Subsequently I will give specifications of the redundancy management algorithms as far as possible ordered group wise, since the evolutionary development resulted in al-

gorithms, which often do not differ much in their specification. Figure 4.18 gives an abridgement of the model checking results.

4.4.1. RMA1, RMA2 and RMA3

Section 3.3 gives a description for redundancy management algorithms **RMA1**, **RMA2** and **RMA3**. It is obvious that these three approaches just differ in application of different functions. Therefore, it definitely makes sense to give their specification as a group and to compare their model checking results. As for the environment, I define necessary constants, variables and standard modules that must be extended. Figure 4.19 shows these definitions in TLA$^+$. This first part is almost the same for all subsequent algorithms and therefore I will mention the essential differences without giving the whole specification again. Initially all variables have a special value denoted as *noVal*. This is

```
CONSTANTS
networks,                          set of networks
SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
A, B, SN, TAG                      just for convenience

VARIABLES
rm                                 Redundancy Management
```

Figure 4.19.: Declaration of constants and variables

necessary as with initial values equal to zero for integer values, some properties cannot be satisfied. The design of the proposed algorithms just demands initialization to special values. Hence initialization for **RMA1**, **RMA2** and **RMA3** looks like in Figure 4.20. The next step is to define adequate functions that allow the redundancy management

```
Initially:

noVal ≜ SN_CNT

The redundancy management:
 - has not received any frame
 - each ptn-value equals special value noVal

InitRM ≜ rm = [
           rsn ↦ noVal,
           paf ↦ noVal,
           ptn ↦ [A ↦ noVal, B ↦ noVal]]
```

Figure 4.20.: Definition of set of initial states

to decide about rejection or acceptance. The first three algorithms just use two different functions for their decisions, namely the *Sequence Number Skew* and the *Sequence Number Offset*. Both functions use subtraction, as defined for sequence numbers. So I made subtraction on sequence numbers a function itself to follow the *write things once*

principle. Definitions of these functions can be found in Definitions 1, 3 and 5 and their specification in TLA$^+$ is shown in Figure 4.21. There is another auxiliary function

Some functions:

return the other twin-newtork-*id*

$$TNid[id \in networks] \triangleq \text{IF } id = A \text{ THEN } B \text{ ELSE } A$$

subtract *SN*

$$s_1 -_{SN} s_2 =_{def} ((s_1 - s_2 + SN\_HALF) \bmod SN\_CNT) - SN\_HALF$$

$$subSN[s1, s2, noval, sn\_cnt, sn\_half \in Nat] \triangleq$$
$$\text{IF } s2 = noval \text{ THEN } 1$$
$$\text{ELSE } ((s1 - s2 + (sn\_half))\%sn\_cnt) - (sn\_half)$$

Sequence Number Skew

$$SNS(f) =_{def} RSN(f) -_{SN} RSN(PTN(f))$$

$$snSkew[id \in networks, rsn \in Nat] \triangleq$$
$$subSN[rsn, rm.ptn[TNid[id]], noVal, SN\_CNT, SN\_HALF]$$

Sequence Number Offset

$$SNO(f) =_{def} RSN(f) -_{sn} PASN(f)$$

$$snOffset[rsn \in Nat] \triangleq$$
$$\text{IF } rm.paf = noVal \text{ THEN } SN\_CNT$$
$$\text{ELSE } subSN[rsn, rm.paf, noVal, SN\_CNT, SN\_HALF]$$

Figure 4.21.: SNS, SNO and subtraction on sequence numbers

named *TNid*, standing for *Twin Network identifier*. It just returns for a given network identifier its twin network identifier.

Next the specifications of the accept and reject action will be given. Each of these actions takes the sequence number field and the network tag of a frame as parameter. The guards of each action are specified to equal a decision condition, which ensures that only one action is enabled at the time, but also at least one is enabled. Finally the TLA$^+$-specification of both actions exemplarily for **RMA3** is shown in Figure 4.22. Finally I must define the allowed interleaving of action, the fairness formula and at last give the specification formula. I decided to allow the redundancy management to possibly receive a nearly arbitrary frame, which just has a known network identifier and a sequence number from the possible range. Thus the specified algorithms may be tested in a different environment than the specified one. That is a great advantage of the modularized specification. It is designated that the redundancy management shall decide about every frame it receives. Hence it is specified that if either the rejection or the acceptance of a frames is enabled forever, it is eventually taken. This is equivalent to specifying weak fairness for the *RM_NEXT* formula. Figure 4.23 shows the specification formula in context. It also includes the *wait* action, which the algorithms do not use to decide about the frames. Hence it is continuously disabled.

Figure 4.18 reveals that these three approaches to resolve redundancy do not differ that

accept frame:
IF frames are available AND ($SNS(f) > 0$ OR $SNO(f) > 0$)
THEN forward frame

$$
\begin{array}{l}
acceptFrame(id,\ sn) \triangleq \\
\quad \land\ \lor\ snSkew[id,\ sn] > 0 \\
\qquad \lor\ snOffset[sn] \quad > 0 \\
\quad \land\ rm' = [rm\ \text{EXCEPT}\ !.rsn = sn, \\
\qquad\qquad\qquad\qquad !.paf = sn, \\
\qquad\qquad\qquad\qquad !.ptn[id] = sn]
\end{array}
$$

reject frame:
IF frames are available $SNS(f) <= 0$ THEN reject frame

$$
\begin{array}{l}
rejectFrame(id,\ sn) \triangleq \\
\quad \land\ snSkew[id,\ sn] \leq 0 \\
\quad \land\ snOffset[sn] \leq 0 \\
\quad \land\ rm' = [rm\ \text{EXCEPT}\ !.rsn = sn, \\
\qquad\qquad\qquad\qquad !.ptn[id] = sn]
\end{array}
$$

Figure 4.22.: Specification of accept and reject action

much. However at least there is a difference in property *avail4*, which exactly mirrors the scenario described by Figure 3.5. Both improvements **RMA2** and **RMA3** overcome this problem and guarantee that in case of a faulty network and no further reset of sequence numbers, the redundancy management stabilizes and accepts all consecutively received frames. Though **RMA2** and **RMA3** satisfy *avail4* and therefor are an improvement they both fail the property *qual0*. This concludes that if only one network is connected to the ES the macroscopic availability with the redundancy management is worse than without. The technical report from Hanxleden and Gambardella [27] considers this to be unacceptable. Overall the first three algorithms suffer the same big problem. As long as requirement *avail5* is missed, there exists a worst case scenario, in which no further frames get accepted by the redundancy management. Exemplary for all three algorithms I want to explain the counterexample TLC gives for algorithm **RMA3**, as this algorithm is supposed to be the best one of the first three proposals. TLC produces a textual output that represents an infinite trace if the violated property is a temporal formula and a finite trace to a specific state if the property is of the form $\Box P$, where P is state predicate.

Step of Redundancy Management

$$
\begin{array}{l}
RM\_Next \triangleq \exists\, \langle id,\ sn \rangle \in networks \times (0\ ..\ SN\_MAX) : \\
\qquad\qquad acceptFrame(id,\ sn) \lor rejectFrame(id,\ sn)
\end{array}
$$

The *RM* shall react on each frame

$$RM\_Fairness \triangleq \land \mathrm{WF}_{\langle rm \rangle}(RM\_Next)$$

$$RM\_Spec \triangleq InitRM \land \Box[RM\_Next]_{\langle rm \rangle} \land RM\_Fairness$$

Figure 4.23.: Composed specification formula

The traces in Figures 4.24 and 4.25 give the following information:

- Lines 4 to 70 give a counterexample for the checked temporal property *avail5*. Each counterexample is given as a set of states, where each state is specified as complete definition of all variables.
- Lines 72 to 92 give the coverage statistics that shows the number of times each action was taken. This can help debug a specification, because if an action is never taken, this might indicate an error. In case of the redundancy management this statistic also might give a hint of how many frames get accepted or rejected. Nevertheless this information must be handled with care as the number of overall reachable states may differ from algorithm to algorithm.
- The last line prints the number of generated states and states that are considered to be different. Since TLC uses hash values to compare states, the situation might occur that two states are considered equal, but they in fact are not. Though this is very unlikely to happen if the checked properties are satisfied, TLC prints an estimation of the likelihood that two different states were treated as equal. Finally, if there are states left to check their number is printed.

The scenario shown by the counterexample is quite unlikely as it assumes infinitely many consecutive resets of sequence numbers. Unfortunately, things can get even worse, because if the reset occurs after an accepted sequence number closest to SN_{HALF} a reset at least every 2^{14} milliseconds[1] $= 16,384$ seconds is enough to let the redundancy management reject all subsequent frames. With the originally proposed value of 2^{27} for SN_{HALF}, things would get dramatically worse, because in the worst case a reset step every 37 hours may cause rejection of all frames. To summarize, the model checking results for the first three algorithms, it can be concluded that

- their perception of redundant frames is good, since redundant frames are only submitted to the application in presence of *reset*-steps,
- the preservation of order is as well as the perception of redundant frames,
- the per-frame loss is tolerable, because at least the *first valid wins* property is satisfied, but
- the behaviour in presence of a single, faulted network is bad, as possibly no further frames will be accepted in the worst case.

4.4.2. RMA4, RMA5 and RMA6

The next three algorithms are defined quite similarly to the first three algorithms. However each algorithm does not define when to accept a frame anymore, but when to reject a frame. The justification for rejecting frames based on *Sequence Number Skew* and

[1] for $BAG = 2^7$ and $SN\_HALF = 2^7$

```
--Checking temporal properties for the current state space...
Error: Temporal properties were violated. The following behaviour constitutes a counter-example:

STATE 1: <Initial predicate>
/\ out = {}
/\ rm = [rsn |-> 6, paf |-> 6, ptn |-> [A |-> 6, B |-> 6]]
/\ env = [ sn |-> 0,
  frames |-> [A |-> <<  >>, B |-> <<  >>],
  alive |-> [A |-> TRUE, B |-> TRUE] ]
/\ status = "initial"

STATE 2: <Action line 93, col 3 to line 106, col 65 of module ENV>
/\ out = {}
/\ rm = [rsn |-> 6, paf |-> 6, ptn |-> [A |-> 6, B |-> 6]]
/\ env = [ sn |-> 1,
  frames |->
      [ A |-> << [sn |-> 0, tag |-> "n"] >>,
        B |-> << [sn |-> 0, tag |-> "n"] >> ],
  alive |-> [A |-> TRUE, B |-> TRUE] ]
/\ status = "send"

STATE 3: <Action line 158, col 12 to line 161, col 64 of module ENV>
/\ out = {[sn |-> 0, tag |-> "n"]}
/\ rm = [rsn |-> 0, paf |-> 0, ptn |-> [A |-> 0, B |-> 6]]
/\ env = [ sn |-> 1,
  frames |-> [A |-> <<  >>, B |-> << [sn |-> 0, tag |-> "r"] >>],
  alive |-> [A |-> TRUE, B |-> TRUE] ]
/\ status = "accept"

STATE 4: <Action line 84, col 6 to line 87, col 68 of module ENV>
/\ out = {[sn |-> 0, tag |-> "n"]}
/\ rm = [rsn |-> 0, paf |-> 0, ptn |-> [A |-> 0, B |-> 6]]
/\ env = [ sn |-> 1,
  frames |-> [A |-> <<  >>, B |-> << [sn |-> 0, tag |-> "r"] >>],
  alive |-> [A |-> FALSE, B |-> TRUE] ]
/\ status = "die"

STATE 5: <Action line 75, col 10 to line 78, col 51 of module ENV>
/\ out = {[sn |-> 0, tag |-> "n"]}
/\ rm = [rsn |-> 0, paf |-> 0, ptn |-> [A |-> 0, B |-> 6]]
/\ env = [ sn |-> 0,
  frames |-> [A |-> <<  >>, B |-> << [sn |-> 0, tag |-> "r"] >>],
  alive |-> [A |-> FALSE, B |-> TRUE] ]
/\ status = "reset"
```

Figure 4.24.: Counterexample for property *avail5* checking **RMA3** (beginning)

```
STATE 6: <Action line 158, col 12 to line 161, col 64 of module ENV>
/\ out = {[sn |-> 0, tag |-> "n"]}
/\ rm = [rsn |-> 0, paf |-> 0, ptn |-> [A |-> 0, B |-> 0]]
/\ env = [ sn |-> 0,
  frames |-> [A |-> <<  >>, B |-> <<  >>],
  alive |-> [A |-> FALSE, B |-> TRUE] ]
/\ status = "reject"

STATE 7: <Action line 93, col 3 to line 106, col 65 of module ENV>
/\ out = {[sn |-> 0, tag |-> "n"]}
/\ rm = [rsn |-> 0, paf |-> 0, ptn |-> [A |-> 0, B |-> 0]]
/\ env = [ sn |-> 1,
  frames |-> [A |-> <<  >>, B |-> << [sn |-> 0, tag |-> "n"] >>],
  alive |-> [A |-> FALSE, B |-> TRUE] ]
/\ status = "send"

STATE 8: <Action line 75, col 10 to line 78, col 51 of module ENV>
/\ out = {[sn |-> 0, tag |-> "n"]}
/\ rm = [rsn |-> 0, paf |-> 0, ptn |-> [A |-> 0, B |-> 0]]
/\ env = [ sn |-> 0,
  frames |-> [A |-> <<  >>, B |-> << [sn |-> 0, tag |-> "n"] >>],
  alive |-> [A |-> FALSE, B |-> TRUE] ]
/\ status = "reset"

STATE 9: Back to state 6.

The coverage stats:
  line 105, col 17 to line 105, col 18 of module ENV: 622646
  line 105, col 21 to line 105, col 23 of module ENV: 622646
  line 106, col 6 to line 106, col 21 of module ENV: 622646
  line 128, col 12 to line 135, col 83 of module ENV: 398036
  line 136, col 12 to line 136, col 50 of module ENV: 398036
  line 137, col 12 to line 137, col 29 of module ENV: 398036
  line 143, col 12 to line 150, col 83 of module ENV: 457301
  line 151, col 23 to line 151, col 25 of module ENV: 457301
  line 152, col 12 to line 152, col 29 of module ENV: 457301
  line 76, col 13 to line 76, col 39 of module ENV: 1262380
  line 77, col 24 to line 77, col 25 of module ENV: 1262380
  line 77, col 28 to line 77, col 30 of module ENV: 1262380
  line 78, col 13 to line 78, col 29 of module ENV: 1262380
  line 81, col 6 to line 83, col 38 of module RMAG1: 398036
  line 85, col 9 to line 85, col 68 of module ENV: 631190
  line 86, col 20 to line 86, col 21 of module ENV: 631190
  line 86, col 24 to line 86, col 26 of module ENV: 631190
  line 87, col 9 to line 87, col 23 of module ENV: 631190
  line 91, col 6 to line 92, col 37 of module RMAG1: 457301
  line 96, col 6 to line 104, col 65 of module ENV: 622646
3371555 states generated, 685162 distinct states found, 120816 states left on queue.
```

Figure 4.25.: Counterexample for property *avail5* checking **RMA3** (end)

Sequence Number Offset were given in Section 3.3. Subsequently I will give parts of the new specifications that differ from the shown above specifications.

Frames get now rejected if the decision functions return values in a negative range. The lower bound of this range depends on the maximum difference of sequence numbers and according to Section 4.2 on the constant *MTF*. Moreover, the absolute value of the lower Bound equals *MTF*, which suggests utilization of $-MTF$ as boundary. Therefore all three specifications contain a further constant *SNS_MIN*. Almost all of the rest of

```
CONSTANTS
networks,                          set of networks
SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
SNS_MIN,                           lower bound of saequence number skew
A, B, SN, TAG                      just for convenience
```

Figure 4.26.: Additional constant *SNSMIN*

the specification stays the same. Only the guards of the accept and reject step must be changed accordingly. This is shown in Figure 4.27. Algorithms **RMA4**, **RMA5** and

$$
\begin{array}{l}
acceptFrame(id,\ sn) \triangleq \\
\quad \land \lor snSkew[id,\ sn] \notin (-SNS\_MIN\ ..\ 0) \\
\qquad \lor snOffset[sn] \quad \notin (-SNS\_MIN\ ..\ 0) \\
\quad \land rm' = [rm \text{ EXCEPT } !.rsn = sn, \\
\qquad\qquad\qquad\qquad !.paf = sn, \\
\qquad\qquad\qquad\qquad !.ptn[id] = sn]
\end{array}
$$

$$
\begin{array}{l}
rejectFrame(id,\ sn) \triangleq \\
\quad \land snSkew[id,\ sn] \in (-SNS\_MIN\ ..\ 0) \\
\quad \land snOffset[sn] \quad \in (-SNS\_MIN\ ..\ 0) \\
\quad \land rm' = [rm \text{ EXCEPT } !.rsn = sn, \\
\qquad\qquad\qquad\qquad !.ptn[id] = sn]
\end{array}
$$

Figure 4.27.: Accept und reject steps defined for **RMA6**.

RMA6 use the same functions to decide about frames, with the difference that they take a less conservative approach and define premises for rejection instead for acceptance of frames. So it is not a surprising result that, though each of these redundancy management algorithms handles more problematic scenarios than its counterpart of the first three algorithms, the overall performance of these redundancy management algorithms is almost the same as of **RMA1**, **RMA2** and **RMA3**. The change from defining acceptance conditions to rejection conditions has a single effect on **RMA6** with the property *qual0*. It is now satisfied in contrast to **RMA3** (see Figure 4.18 on page 46).

4.4.3. RMA7

This algorithm is quite special, because though it uses mainly known concepts for its decisions, it is quite efficient. Figure 4.18 shows that it is model checked twice with

different results. Section 3.3, page 20 gives the definition of the *Sequence Number Increment* (Definition 7) that is for consecutively received frames f_1, f_2, the *Sequence Number Increment* is

$$SNI(f_2) =_{def} RSN(f_2) -_{\mathrm{SN}} RSN(f_1).$$

Taking solely this definition, there is no information about the networks where f_1 and f_2 come from, and hence there is no reason to consider them in the definition. Following this approach, once a frame gets accepted, we have that the *Previously Accepted Frame* obviously equals the last received frame used for the *Sequence Number Increment*, which implies that for the next delivered frame f, $SNI(f) = PASN(f)$ holds . Consequently the boolean formula

$$SNI(f) > 0 \textbf{ and } SNS\_MIN \leqslant SNO(f) \leqslant 0$$

is false and stays false for all consecutive frames. If one assumes that initially the first frame gets accepted, it is obvious that no frame will ever be rejected. Of coursse such an redundancy management algorithm is nonsense and Figure 3.11 shows an example bahaviour that justifies to assume the following slightly different definition of the *Sequence Number Increment*:

Definition 10. *(alternative Sequence Number Increment) For consecutively received frames f_1, f_2 on the same network N, the* Sequence Number Increment *is*

$$SNI(f_2) =_{def} RSN(f_2) -_{\mathrm{SN}} RSN(f_1).$$

Considering this definition, without writing an extra function for the alternative *Sequence Number Increment*, the accept and reject actions for **RMA7** are specified below. Figure 4.18 shows the results for both variants of **RMA7**, where **RMA7*** denotes the algorithm with the $SNI(f)$ defined as in Definition 10 on page 54. Two results for the original algorithm are set in brackets, since TLC finds no error, but that is only technical, because if all frames get accepted, then all twin frames will be marked as "redundant" and no frame as "old". If I would have given precedence to the "old" status, both properties would be violated. The original algorithm definition is most probably caused by an imprecise definition of the *Sequence Number Increment*, because the example given for **RMA7** uses the alternative definition given above, and this alternative definition for the *Sequence Number Increment* makes **RMA7*** a very well performing algorithm. Compared to the proposed algorithms **RMA1** to **RMA6**, it satisfies the same properties plus *avail3* and *avail5*. Through satisfaction of *avail5* it is guaranteed that no behaviour of the environment exists such that the redundancy management rejects all subsequent frames, and *avail3* even ensures that after a network failure the redundancy management stabilizes and all consecutive frames of the remaining network will be accepted. This seems to be the best possible result without compromising with other properties. To satisfy *avail1* or *avail2* it needs buffering or violation of ordering properties.

$$
\begin{aligned}
&wait \triangleq \ \land \text{FALSE} \\
&\qquad \land \text{UNCHANGED } \langle rm \rangle
\end{aligned}
$$

accept frame:
IF frames are available AND $SNS(f) > 0$ THEN forward frame

$$
\begin{aligned}
&acceptFrame(id,\ sn) \triangleq \\
&\quad \land \ \lor \ snIncrementAlt[sn,\ id] \leq 0 \\
&\qquad \lor \ snOffset[sn] \ \notin (-SNS\_MIN\ ..\ 0) \\
&\quad \land \ rm' \ = [rm \text{ EXCEPT } !.rsn = sn, \\
&\qquad\qquad\qquad\qquad !.paf = sn, \\
&\qquad\qquad\qquad\qquad !.ptn[id] = sn]
\end{aligned}
$$

reject frame:
IF frames are available $SNS(f) <\ = 0$ THEN reject frame

$$
\begin{aligned}
&rejectFrame(id,\ sn) \triangleq \\
&\quad \land \ snIncrementAlt[sn,\ id] > 0 \\
&\quad \land \ snOffset[sn] \ \in (-SNS\_MIN\ ..\ 0) \\
&\quad \land \ rm' = [rm \text{ EXCEPT } !.rsn = sn, \\
&\qquad\qquad\qquad\qquad !.ptn[id] = sn]
\end{aligned}
$$

Figure 4.28.: Accept and reject action of **RMA7**

4.4.4. RMA8

RMA8 is the first algorithm that introduces time as decision criteria. Justification for that is the knowledge about the maximum delay of message delivery *Skew Max*. Using this concept, **RMA8** combines the aforementioned concept of *Sequence Number Offset*, with the restriction that if more time than *Skew Max* has elapsed, the next frame gets definitely submitted. An naïve approach would be to introduce a further action *wait* in the environment, with the following consequences:

- Each frame must get another boolean tag that indicates whether it will be delivered before or after *Skew Max* time expires.
- The definition of the set of deliverables will be more complicated, because a frame that shall be delivered with an expired time stamp followed by a frame with positive time stamp may cause problems if the second frame gets delivered and the frame afore is considered lost. To handle this the set of deliverables could either give priority to frame with exceeded time stamps or could manipulate frames with positive time stamp if due to their delivery frames with exceeded time stamp get discarded. Nevertheless this seems to be an ugly solution.
- The accept and reject actions must be updated to reset the timer variable each time they process a frame.
- The resulting new environment is no longer compatible with the already available specifications of **RMA1-7**.

These changes would be neither trivial nor would the complexity be approximately the same.
A much better way is to adopt an approach that allows to still obtain a single environment, with some minor restrictions, for all redundancy management algorithms. A *wait* action is added to the specification of **RMA8** and the accept and reject specifications are adapted accordingly. The new wait step can be seen in Figure 4.29, and respectively the changed accept and reject steps are shown by Figure 4.30. The wait actions must

$$
\begin{array}{l}
wait \triangleq \\
\quad \land\ rm.time = \text{TRUE} \\
\quad \land\ rm' = [rm \text{ EXCEPT } !.time = \text{FALSE}]
\end{array}
$$

Figure 4.29.: Specification of wait step for **RMA8**

$$
\begin{array}{l}
acceptFrame(id,\ sn) \triangleq \\
\quad \land\ \lor\ rm.time = \text{FALSE} \\
\qquad \lor\ snOffset[sn] \notin (-SNS\_MIN\ ..\ 0) \\
\quad \land\ rm' = [rm \text{ EXCEPT } !.rsn = sn, \\
\qquad\qquad\qquad\qquad\quad !.paf = sn, \\
\qquad\qquad\qquad\qquad\quad !.time = \text{TRUE}]
\end{array}
$$

$$
\begin{array}{l}
rejectFrame(id,\ sn) \triangleq \\
\quad \land\ rm.time = \text{TRUE} \\
\quad \land\ snOffset[sn] \in (-SNS\_MIN\ ..\ 0) \\
\quad \land\ rm' = [rm \text{ EXCEPT } !.rsn = sn, \\
\qquad\qquad\qquad\qquad\quad !.time = \text{TRUE}]
\end{array}
$$

Figure 4.30.: Adapted specifications of accept and reject actions

finally be added to the environment specification as well as to the next state definition of the redundancy management algorithm.

$$
\begin{array}{rl}
RM\_Next \triangleq & \exists \langle id,\ sn \rangle \in networks \times (0\ ..\ SN\_MAX) : \\
& acceptFrame(id,\ sn) \lor rejectFrame(id,\ sn) \lor wait
\end{array}
$$

An action *extWait* is added to the environment specification that just contains the specified *wait* formula of the redundancy management algorithm's specification and the update information for the environment. But there is still another gap in the specification of the *wait* action. Up to now a wait step may occur at every time even in between the delivery of two twin frames. That would imply the acceptance of redundant frames, but we know that two twin frames must be delivered to the redundancy management within *Skew Max* time has been elapsed. Therfore this action must be guarded by the check that both sequences contain an equal number of frames, which implies that only complete twin pairs remain on the networks. The fairness formula actually guarantees that if frames are deliverable, they eventually will be delivered. Nevertheless we want

$$
\begin{aligned}
&\mathit{extWait} \triangleq \\
&\quad \land \mathit{Len}(\mathit{env.frames}[A]) = \mathit{Len}(\mathit{env.frames}[B]) \\
&\quad \land \mathit{wait} \\
&\quad \land \mathit{status}' = \text{“wait”} \\
&\quad \land \text{UNCHANGED } \langle \mathit{env},\ \mathit{out} \rangle
\end{aligned}
$$

Figure 4.31.: Defintion of extWait for **RMA8**

each decision to represent some advance in time. This implies that there cannot be an infinite chain of rejections, because $t_{RECV}(f) - t_{RECV}(PAF(f))$ will eventually exceed *Skew Max*. In our specification we equivalently want eventually to take the *wait* action, which implies acceptance of the next frame. For the *extWait* action actually strong fairness is required, because it will not be enabled continuously, while for the *wait* action specified for the redundancy management weak fairness is sufficient. The

$$
\begin{aligned}
\mathit{Fairness} \triangleq\ & \land \mathrm{WF}_{\langle \mathit{rm},\, \mathit{out},\, \mathit{env},\, \mathit{status} \rangle}(\mathit{sendFrame}) \\
& \land \mathrm{WF}_{\langle \mathit{rm},\, \mathit{out},\, \mathit{env},\, \mathit{status} \rangle}(\mathit{SysNext}) \\
& \land \mathrm{SF}_{\langle \mathit{rm},\, \mathit{out},\, \mathit{env} \rangle}(\mathit{extWait})
\end{aligned}
$$

Figure 4.32.: New fairness specification for the environment

changes made to the environment specification at a first glance destroy compatibility with redundancy management algorithms **RMA1** to **RMA7**. There are three ways to handle this problem.

1. The easiest but dirty solution is to comment *extWait* out from the next step formula and the fairness formula. This should work properly, however, there is no longer a unique environment for all redundancy management algorithms. This approach has its advantage though, as it does not increment the complexity of model checking for algorithms that do not use time for their decisions.

2. The second possibility is to extend all redundancy management algorithm's specifications with a *wait* action and an accordant record field in the *rm* variable. These changes will not influence any model checking results, however, they will increase the time model checking needs because of more interleaving.

3. A third approach mixes both afore mentioned solutions. Define for all redundancy management algorithms a *wait* action and a new single variable *time*. Furthermore define a new *view* that excludes the *time* variable from model checking. This modified *view* could be used if necessary. This approach is more complex, but combines advantages of solutions one and two.

It is a matter of taste which way one prefers.

4.4.5. RMA9

RMA9 defines even more rigorous acceptance conditions. Since **RMA9** accepts frames only if they a sequence number incremented by one or after time out, **RMA9** does not follow the *first valid wins* strategy. That is, however, not the only shortcoming, because with only one network alive there exists a trace where **RMA9** accepts only one frame every *Skew Max* time. Therefor it is definitely no good choice. The only difference in the specification are the slightly different guards for the accept and reject steps.

4.4.6. RMA11 and RMA12

RMA11 and **RMA12** introduce the concept of the *Previously Accepted Sequence Number Set*[2] based on the observation that a maximum difference between the delivery of both twin frames is bounded, and hence the maximum difference of received sequence numbers is bounded. The new specification therefore just defines the set of previously accepted frames. The actual specification of the acceptance step then appends the currently received sequence number to the *PASN* sequence and deletes its head if the sequence length reached its desired maximum. Figure 4.33 shows a function that checks if a certain sequence number is equal to a sequence number in the *PASN* set and Figure 4.34 shows the accordant specification of the accept and reject action. **RMA12**

$$
\begin{array}{l}
inside[sn \in Nat,\ queue \in Seq(Nat)] \triangleq \\
\quad \text{IF } queue = \langle\rangle \text{ THEN FALSE} \\
\quad \text{ELSE} \\
\quad\quad \text{IF } Head(queue) = sn \text{ THEN TRUE} \\
\quad\quad \text{ELSE } inside[sn,\ Tail(queue)]
\end{array}
$$

Figure 4.33.: Function that checks if sequence number is in the *PASN* set

$$
\begin{array}{l}
acceptFrame(id,\ sn) \triangleq \\
\quad \wedge\ inside[sn,\ rm.pasn] = \text{FALSE} \\
\quad \wedge\ rm' = [rm \text{ EXCEPT } !.pasn = \text{IF } Len(rm.pasn) < SNS\_MIN \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } Append(rm.pasn,\ sn) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } Append(Tail(rm.pasn),\ sn) \\
\qquad\quad] \\
rejectFrame(id,\ sn) \triangleq \\
\quad \wedge\ inside[sn,\ rm.pasn] = \text{TRUE} \\
\quad \wedge\ \text{UNCHANGED } \langle rm \rangle
\end{array}
$$

Figure 4.34.: Adapted accept action for **RMA11**

just uses the improvement first introduced by **RMA8** that after *Skew Max* time has

[2] Actually it is a bounded FIFO queue, but I will stick to the notation used by von Hanxleden and Gambardella [27]

passed, a received frame can neither be the redundant copy of the last accepted frame nor a frame that was sent before the last accepted one. There is no need to give any part of this specification as it would not reveal new insights.

4.4.7. RMA13

In Section 3.3 when the redundancy management algorithms were introduced, **RMA13** was already considered to be a good algorithm, although it is very simple. Maybe the only weakness of **RMA13** is the per-frame availability. Consider an unreliable network that is chosen first and a reliable second network. The algorithm will continuously discard all frames from the second network and keeps accepting the few frames from the first network. Beside this, **RMA13** is the only algorithm that perfectly handles redundancy and order, and it has a good availability in case of a single faulted network. Its specification is quite simple, too. Figure 4.35 shows this specification.

ACTIONS:
exceed time bound:

$$
\begin{aligned}
wait \triangleq\ & \wedge rm.time = \text{TRUE} \\
& \wedge rm' = [rm \text{ EXCEPT } !.time = \text{FALSE}]
\end{aligned}
$$

accept frame:
IF frames are available AND $SNS(f) > 0$ THEN forward frame

$$
\begin{aligned}
&acceptFrame(id,\ sn) \triangleq \\
&\ \wedge\ \vee\ rm.pan = \text{“all”} \\
&\quad\ \vee\ id = rm.pan \\
&\quad\ \vee\ rm.time = \text{FALSE} \\
&\ \wedge\ rm' = [rm \text{ EXCEPT } !.pan = id, \\
&\qquad\qquad\qquad\ !.time = \text{TRUE} \\
&\qquad\ ]
\end{aligned}
$$

reject frame:
IF frames are available $SNS(f) <= 0$ THEN reject frame

$$
\begin{aligned}
&rejectFrame(id,\ sn) \triangleq \\
&\ \wedge\ rm.pan \neq \text{“all”} \\
&\ \wedge\ id \neq rm.pan \\
&\ \wedge\ rm.time = \text{TRUE} \\
&\ \wedge\ rm' = [rm \text{ EXCEPT } !.time = \text{TRUE}]
\end{aligned}
$$

Figure 4.35.: Main part of **RMA13**'s specification

5. Results

This chapter provides the most essential results and conclusions about supporting evolutionary development of algorithms by formal analysis and formal reasoning. Conclusions about specifying systems with TLA$^+$ and about usage of TLC to check a quite large set of properties are provided as well.

5.1. Examining the algorithms

Of course, model checking a large set of algorithms and properties has its limitations. There is no way to give reliable conclusions of how many frames one algorithm accepts and rejects. The algorithms **RMA2** and **RMA3** both satisfy and miss the same properties, however, many scenarios can be thought of where one accepts more frames than the other (see Figure 3.6 on page 17). Each model may exhibit a different number of states and thus the coverage statistics cannot be taken for argumentation. Nevertheless, I tried to give a wide range of property classes, which incorporate many critical behaviors and allow reasonable statements about the quality of the redundancy management algorithms.

The algorithms **RMA1** to **RMA6** differ only little. Each of them, however, is able to treat a certain scenario correctly that the afore mentioned algorithms would fail. Although each of these algorithms is a minor improvement for several scenarios, the big picture is still the same. It seems that solving definite situations is no good approach to solve general problems. If any redundancy management algorithm fails property *avail5* for example, a behaviour of the environment exists that permits acceptance of any further frame. Such a situation would be catastrophic, as the application beyond the redundancy management would not receive any more frames. Scenarios like this must be avoided under all circumstances. Figures 3.5 to 3.10 on pages 16 to 20 show behaviors, where each algorithm reveals an unacceptable behavior, however, none of these bad traces describes an infinite trace that violates *avail5*. Using formal methods and writing a formal specification supports the developer to clearly express what an algorithm must and what it must not do. Nevertheless, checking an redundancy management algorithm manually against a certain property, expressed as a TLA$^+$ formula, is cumbersome. It may be manageable if the property is failed. At least such a proof cannot be established by exhaustive testing like model checkers do.

The first algorithms show perfectly that without formal analysis, algorithms that should be an improvement are in fact not. Though already unacceptable, we can even observe that algorithms were spuriously considered to behave worse or at least not better than others. Take for instance algorithm **RMA7***, which uses the alternative

definition of the *Sequence Number Increment* (Definition 10), and is considered not acceptable as it may submit redundant frames. Having a look at Figure 4.18 obtains that though **RMA7*** maybe accepts more redundant frames than **RMA1-6**, none of these algorithms satisfies a property, concerning redundancy and ordering, that **RMA7*** does not. Therefore based on these properties, it cannot be concluded that **RMA7*** deals worse with redundant and outdated frames than **RMA1-6**. As another example take the evolution from **RMA1** to **RMA2** and **RMA3**. Both improved algorithms only solve certain scenarios correctly that **RMA1** does not. But both fail the property *qual0*, which is satisfied by **RMA1**, and therefore it is not clear whether they are really improvements of the first proposal.

Moreover formal specification and model checking simplifies recognition of an algorithm's weak point. **RMA7*** is considered to accept too many redundant frames. Though this weak point cannot be affirmed by the so far defined properties from **RMA11** and **RMA12**, it can be obtained that the concept of rejecting frames, with a sequence number that was seen a short time before, discards redundant frames properly and does not seem to lead to unwanted rejections. Hence **RMA7*** could be extended in a way that a frame gets rejected if and only if the *SNI* is positive while the *SNO* is redundant and the actual sequence number is in the set of previously accepted sequence numbers.

Finally, the question arises whether the specification of the algorithms and the model checking with TLC allows us to choose a certain algorithm. Obviously the best candidates are **RMA7*** and **RMA13**. Both increase the macroscopic availability to an acceptable level while not failing too many other properties. **RMA13** has the advantage that it never submits redundant or outdated frames. It suffers, however, a poor per frame availability. **RMA7*** instead has an acceptable per frame loss, but does not handle redundancy and order correctly under all circumstances. I tend to take **RMA7*** as the final choice, because it does not relay on an explicit time bound as a fall back criteria and has a much better per frame availability. The properties **RMA7*** fails are not critical, which means that an algorithm need not satisfy these properties to avoid catastrophic behaviors.

5.2. Experiences with TLA$^+$ and TLC

TLA$^+$ is very suitable to specify the redundancy management algorithms. Though the compact notations of TLA$^+$ might be a little bit confusing at a first glance, they are very practical and maintain a good readability. Moreover the concept of untyped variables turned out to be not error-prone as expected and is very flexible. It is desirable to have a little more flexibility in utilization of strings. Figure 5.1 shows an example where the parameter of a function is used to refer to a field of a record. Hence this parameter must be string, as there is no way to convert any other value into a string. Unfortunately, TLC considers strings as non model values, even if they are assigned to constants. This is annoying, because if symmetry functions cannot be used with string valued constants, which leads to an increased complexity for model checking. Beside this, my experiences with TLA$^+$ were very good.

MODULE *module*1

EXTENDS *Naturals*

VARIABLES
sums

$init \triangleq sums \in [sum1 \mapsto 0, sum2 \mapsto 0]$

$add(target) \triangleq sums' = [sums \text{ EXCEPT } ![target] = 1 + @]$

$spec \triangleq init \wedge \Box[\exists\, target \in \{\text{“sum1”}, \text{“sum2”}\} : add(target)]_{\langle sums \rangle}$

Figure 5.1.: Referring to a records field

In contrast, working with TLC was partly more complicated. Subsequently some strange or even faulty aspects is explained.

1. TLC allows to check a wide range of expressible TLA$^+$ properties, but not arbitrary formulas. Temporal formulas that shall be checked with TLC and which contain actions must be of the form $\Box\Diamond A$ or $\Diamond\Box A$, where A denotes the action. Lamport does not explain this decision in his book [18] and I do not have an idea for this choice. The easiest work around for this problem is to introduce another variable that records the actually taken action and to use this variable instead of actions in the formulas. This should be easy to implement in TLC, too.

MODULE *constraint*

EXTENDS *Naturals*, *TLC*

CONSTANT *MAX*

VARIABLE *x*

$Constraint \triangleq x \leq MAX$

$TypeInv \triangleq x \in (0 \,..\, MAX)$

$Init \triangleq x = 0$

$Step \triangleq \wedge x' = x + 1$

$Spec \triangleq Init \wedge \Box[Step]_{\langle x \rangle}$

THEOREM $Spec \Rightarrow TypeInv$

Figure 5.2.: TLC reveals an error while checking the type invariance

2. Several times in this work, the possibility of constraining infinite models was mentioned. Therfore, in Section 4.2 I could, or even should, specify the sending of frames without bounding the capacity[1] of the networks, and use a constraint to restrict the actual length:

$$\begin{aligned} \mathit{SeqConstraint} \triangleq\ & \wedge \mathit{Len}(\mathit{env.frames.A}) \leqslant \mathit{MTF} \\ & \wedge \mathit{Len}(\mathit{env.frames.B}) \leqslant \mathit{MTF} \end{aligned}$$

This looks equivalent, but it turns out to be not. Each time TLC computes a new state, it checks all formulas $\Box P$, when P is a state predicate and afterwards checks if the new state satisfies all constraint formulas. Hence a state that is considered not reachable, because it violates a constraint formula, may lead to a violation of an invariant. Figure 5.2 shows a small example where a single variable is continuously incremented by one. The constraint should limit the value of x to a certain maximum and the invariant shall check whether this maximum is not exceeded. Checking this example with TLC yields an error, because the invariant is violated in a state where x has a value larger than specified by the constraint formula. Lamport explained in personal communication why this decision was made:

> We decided that TLC should do this because, if it has already computed the state, it might as well check that state. [...] Constraints are a method of preventing TLC from running too long on large or infinite-state specifications; they are not part of the specifications.

I totally agree with this opinion, however, recognize some theoretical and practical problems. Theoretically the problem arises that for a state predicate P

$$\Box P \Leftrightarrow (\mathit{true} \Rightarrow \Box P)$$

is no longer valid. The left hand side is considered as an invariant and hence is checked before the reachability of a state is determined. Complementary the right side is considered as a temporal formula, which are only evaluated on reachable states. Practically constraints may cause further complexity in model checking. The specification of the redundancy management allows each sequence to buffer at most MTF frames and this value must be less than $\mathit{SN\_HALF}$. Using constraints, however, TLC would generate states where each sequence buffers $\mathit{MTF}+1$ frames, maybe equal to $\mathit{SN\_HALF}$, which causes trouble with some invariants. Consequently the minimal applicable value of $\mathit{SN\_HALF}$ increases by one and accordingly the value for $\mathit{SN\_CNT}$ by two. Obviously this leads to a significant increment of states and hence model checking time. I suggest to solve this problem by making the order of checking invariants and constraints adjustable.

3. The next problem I encountered seems to be a minor bug of TLC. Figures 5.3 and 5.4 show a first module that defines two simple variables x and y, which get updated

[1] the action is only enabled if both sequences of frames have a length less than MTF

MODULE *moduleA*

EXTENDS *Naturals*, *TLC*

CONSTANTS *FACTOR*

VARIABLES x, y

$$
\begin{aligned}
&\mathit{initA} \triangleq \land x = 0 \\
&\qquad\qquad \land y = 1
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{function1}[n,\, m \in \mathit{Nat}] \triangleq n + (m * \mathit{FACTOR}) \\
&\mathit{stepA} \triangleq \land x' = \mathit{function1}[x,\, y] \\
&\qquad\qquad \land y' = y + 1 \\
&\qquad\qquad \land \mathit{Print}(\mathit{FACTOR},\, \text{TRUE}) \\
&\mathit{spec} \triangleq \mathit{initA} \land \Box[\mathit{stepA}]_{\langle x,\, y \rangle}
\end{aligned}
$$

Figure 5.3.: Module A

by action *stepA*. Thereby the value of x changes to the value of *function1*, which contains a reference to the constant *FACTOR*. A second specification *moduleB* instantiates *moduleA* and in its only action "executes" the action of *moduleA*. A first try to check this with TLC returns the following error message:

```
Error: The identifier FACTOR is either undefined or not an operator.
```

Removing all references to *FACTOR* from *function1* yields a checkable specification, though *FACTOR* is still referenced by the *Print* statement, which correctly prints the value of *FACTOR* each time it is evaluated. This is at least inconsistent and seems to be a minor bug, which I worked around by passing all relevant constants to the functions as additional parameters.

4. TLC allows to define a subset of variables, called a *view*, that is used to explore the state graph. Typically only debugging values are excluded with a manually defined *view*, to reduce the state space. Nevertheless, it is desirable that excluded values are though evaluated and printed in counter examples, because that is what debugging values are for. Furthermore, if a user defined *view* is used, TLC omits *all* names of variables in counter examples. This forces the user to determine, from existing field names or actual values, which value belongs to which variable.

5. Last but not least, it is inconvenient to check more than one temporal property at the same time if it is likely that at least one of them will fail. TLC builds a conjunction of all temporal formulas that shall be checked and if it gets false, it returns a counterexample without any information, which property caused the failure. Taking this approach, the user has to detect the formula, which was violated, delete it from the TLC configuration file and restart TLC at the last

MODULE *moduleB*

EXTENDS *Naturals*

CONSTANTS *FACTOR*

VARIABLES x, y, z

INSTANCE *moduleA* WITH $FACTOR \leftarrow FACTOR$

$constraint \triangleq z < 100$

$initB \triangleq \land z = 0$
$\land\ initA$

$stepB \triangleq \land z' = z + 1$
$\land\ stepA$

$specFinal \triangleq initB \land \Box[stepB]_{\langle x, y \rangle}$

Figure 5.4.: Module B

checkpoint. This is not applicable for large sets of properties and a significant probability for several properties to fail.

I experienced models where TLC produce more than one billion states, however, acceptable performance is reached for few hundred million states. The probability of a state collisions, *i. e.*, the probability that two different states have the same hash value and are mistakenly considered equal, lies between 10^{-6} and 10^{-4} for all checked properties.

5.3. Outlook

Behaviors of systems can be represented as a sequence of states. TLA$^+$ specifications describe what a system is supposed to do and each TLA$^+$ specification can be translated in a straightforward manner to an equivalent *Mealy Machine*. It is therefore possible to translate such a *Mealy Machine* into an equivalent *SyncChart*. For deterministic automaton this would directly allow us to simulate its behavior and to generate either software or hardware to get a first implementation. The challenge of this transformation would be the non determinism of TLA$^+$ specifications and translating expressions of quantified boolean logic. Translating a *SyncChart* into an equivalent TLA$^+$ formula could be even more worthwhile. This step would allow the developer to specify a wide class of system properties, including temporal properties. A *SyncChart* can then be translated into a TLA$^+$ formula and checked against the specified properties. Such an approach combines the flexibility in system design from *SyncCharts* with the ability of TLA$^+$ to specify temporal formulas and machine check them with TLC. I think, however, that to enable this approach, TLC must be optimized to be able to check systems with more than a billion states efficiently.

A. TLA$^+$ specifications

The appendix contains all specifications written in TLA$^+$.

```
MODULE ENV

EXTENDS Naturals, Sequences, TLC

CONSTANTS
networks,                          set of networks
SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
MCFL,                              maximum number of consecutive frame loss
MTF,                               maximumb number of transient frames
A, B, SN, TAG                      just for convenience

VARIABLES
rm,                                Redundancy Management
env,                               Environment including the redundant networks
out,                               forwarded "frames"
status                             debugging

INSTANCE RMA11 WITH SNS_MIN ← MTF  load instance of RMA

TypeInvEnv ≜ env ∈ [
               sn     : (0 .. SN_MAX),
               frames : [A : Seq([sn : (0 .. SN_MAX), tag : {"n", "r", "o"}]),
                         B : Seq([sn : (0 .. SN_MAX), tag : {"n", "r", "o"}])],
               alive  : [A : BOOLEAN , B : BOOLEAN ]
             ]

TypeInvOut ≜ out ⊆ ([sn : (0 .. SN_MAX), tag : {"n", "r", "o"}])

TypeInv ≜ TypeInvEnv ∧ TypeInvOut ∧ TypeInvRM

Initially:

The environment has:
 - sequence numbers equal to zero
 - empty frame queues
 - two networks that can be alive or dead

InitEnv ≜ env ∈ [
            sn : {0},
            frames : [A : {⟨⟩}, B : {⟨⟩}],
            alive : [A : BOOLEAN , B : {TRUE}]
          ]

Init ≜ InitEnv ∧ out = {} ∧ InitRM ∧ status = "initial"

ENVIRONMENT STATEMENTS
```

```
51  The set of frames that could be delivered to the RM
52  deliverable ≜ {⟨id, pos⟩ ∈ networks × (1 .. (1 + MCFL)) :
53                    ∧ env.frames[id] ≠ ⟨⟩
54                    ∧ pos ≤ Len(env.frames[id])}

56  isAlive(id):
57  yields TRUE iff network with identifier 'id' is alive

59  isAlive[id ∈ networks] ≜ IF env.alive[id] THEN TRUE ELSE FALSE

61  reset:
62  reset of sequence number

64  reset ≜ ∧ ∃ id ∈ networks : isAlive[id] = TRUE
65          ∧ env' = [env EXCEPT !.sn = 0]
66          ∧ UNCHANGED ⟨rm, out⟩
67          ∧ status' = "reset"

69  die(id):
70  IF network is alive THEN network goes down

72  die(id) ≜
73        ∧ isAlive[id] = TRUE
74        ∧ env' = [env EXCEPT !.alive[id] = FALSE, !.frames[id] = ⟨⟩]
75        ∧ UNCHANGED ⟨rm, out⟩
76        ∧ status' = "die"

78  sendFrame:
79  delivers a frame to each operating network

81  sendFrame ≜
82    ∧ Len(env.frames.A) < MTF
83    ∧ Len(env.frames.B) < MTF
84    ∧ ∃ id ∈ networks : isAlive[id] = TRUE
85    ∧ env' = [env EXCEPT
86                  !.frames = [
87                  A ↦ IF isAlive[A]
88                        THEN Append(@.A, [sn ↦ env.sn, tag ↦ "n"])
89                        ELSE ⟨⟩,
90                  B ↦ IF isAlive[B]
91                        THEN Append(@.B, [sn ↦ env.sn, tag ↦ "n"])
92                        ELSE ⟨⟩],
93              !.sn = (@ + 1)%SN_CNT]
94    ∧ UNCHANGED ⟨rm, out⟩
95    ∧ status' = "send"

97  ├──────────────────────────────────────────────────────────────────────┤
98  Marks redundant and old frames
```

```
tag[seq1 ∈ Seq([sn : (0 .. SN_MAX), tag : {"n", "r", "o"}]),
    seq2 ∈ Seq([sn : (0 .. SN_MAX), tag : {"n", "r", "o"}]),
    val ∈ (0 .. SN_CNT), id ∈ networks] ≜
      IF Len(seq1) > Len(seq2)
         THEN
          IF Head(seq1)[TAG] = "r" THEN ⟨Head(seq1)⟩ ∘ tag[Tail(seq1), seq2, val, id]
           ELSE ⟨[sn ↦ Head(seq1)[SN], tag ↦ "o"]⟩    ∘ tag[Tail(seq1), seq2, val, id]
       ELSE ⟨[sn ↦ Head(seq1)[SN], tag ↦ "r"]⟩ ∘ Tail(seq1)


 extending instanced actions

 ACTIONS:
 time exceeds:

extWait ≜
        ∧ Len(env.frames[A]) = Len(env.frames[B])
        ∧ wait
        ∧ status′ = "wait"
        ∧ UNCHANGED ⟨env, out⟩

 accept frame:

extAcceptFrame(id, sn, pos) ≜
        ∧ acceptFrame(id, sn)
        ∧ env′ = [env EXCEPT
                !.frames[id] = SubSeq(@, pos + 1, Len(@)),
                !.frames[TNid[id]] =
                 IF Len(@) ≥ Len(SubSeq(env.frames[id], pos, Len(env.frames[id])))
                    THEN tag[env.frames[TNid[id]],
                             SubSeq(env.frames[id], pos, Len(env.frames[id])),
                             env.frames[id][pos][SN], id]
                  ELSE @]
        ∧ out′ = out ∪ {env.frames[id][pos]}
        ∧ status′ = "accept"

 reject frame:

extRejectFrame(id, sn, pos) ≜
        ∧ rejectFrame(id, sn)
        ∧ env′ = [env EXCEPT
                !.frames[id] = SubSeq(@, pos + 1, Len(@)),
                !.frames[TNid[id]] =
                 IF Len(@) ≥ Len(SubSeq(env.frames[id], pos, Len(env.frames[id])))
                    THEN tag[env.frames[TNid[id]],
                             SubSeq(env.frames[id], pos, Len(env.frames[id])),
                             env.frames[id][pos][SN], id]
```

```
                    ELSE  @]
          ∧ UNCHANGED ⟨out⟩
          ∧ status′ = "reject"

 ──────────────────────────────────────────────────────────────────────

  Step of Environment
EnvNext ≜ ∃ id ∈ networks : sendFrame ∨ die(id) ∨ reset

SysNext ≜ ∃ ⟨id, pos⟩ ∈ deliverable :
                ∨ extAcceptFrame(id, env.frames[id][pos][SN], pos)
                ∨ extRejectFrame(id, env.frames[id][pos][SN], pos)
                ∨ extWait

  A system-step is a step either of environment or of Redundancy Management

Next ≜ SysNext ∨ EnvNext

  We want the environment to send infinitely many frames
  and the RM to react on each frame

Fairness ≜ ∧ WF⟨rm, out, env, status⟩(sendFrame)
           ∧ WF⟨rm, out, env, status⟩(SysNext)
           ∧ SF⟨rm, out, env⟩(extWait)

Spec ≜ Init ∧ □[Next]⟨rm, out, env, status⟩ ∧ Fairness

 ──────────────────────────────────────────────────────────────────────
```

```
1 ──────────── MODULE ENV_TLC ────────────

3 EXTENDS Naturals, Sequences, TLC, ENV

5 Use symmetry to reduce number of states
6 Constraint ≜ env.alive[B] = TRUE

8 exclude variable 'status' for model checking
9 NoDebug ≜ ⟨rm, env, out⟩

12 CORRECTNESS SPECIFICATION
13 to shorten the writing

15 v ≜ ⟨rm, env, out⟩

17 Requirements Part 1:
18 The RM hides the redundancy to all applications beyond/above the RM
19 and preserves the order of sending

22 No redundant frames are submitted to application layer

24 Redundancy1 ≜ ∀ frame ∈ out : frame[TAG] ≠ "r"

26 Redundancy2 ≜ ◇□¬[reset]_v
27                  ⇒ ◇□(∀ ⟨id, pos⟩ ∈ deliverable :
28                  ENABLED ⟨extAcceptFrame(id, env.frames[id][pos][SN], pos)⟩_v
29                    ⇒ env.frames[id][pos][TAG] ≠ "r")

31 Redundancy3 ≜ ◇□¬[reset]_v
32                  ⇒ □◇(∀ ⟨id, pos⟩ ∈ deliverable :
33                  ENABLED ⟨extAcceptFrame(id, env.frames[id][pos][SN], pos)⟩_v
34                    ⇒ env.frames[id][pos][TAG] ≠ "r")

36 No old frames w.r.t its sequence numbers are submitted to application layer

38 Ordering1 ≜ ∀ frame ∈ out : frame[TAG] ≠ "o"

40 Ordering2 ≜ ◇□¬[reset]_v
41                  ⇒ ◇□(∀ ⟨id, pos⟩ ∈ deliverable :
42                  ENABLED ⟨extAcceptFrame(id, env.frames[id][pos][SN], pos)⟩_v
43                    ⇒ env.frames[id][pos][TAG] ≠ "o")

45 Ordering3 ≜ ◇□¬[reset]_v
46                  ⇒ □◇(∀ ⟨id, pos⟩ ∈ deliverable :
47                  ENABLED ⟨extAcceptFrame(id, env.frames[id][pos][SN], pos)⟩_v
48                    ⇒ env.frames[id][pos][TAG] ≠ "o")

50 As long as the RM receives frames, each is either accepted or rejected
```

$$Liveness \triangleq \forall \langle id, pos \rangle \in deliverable :$$
$$\quad \vee \text{ENABLED } \langle extAcceptFrame(id, env.frames[id][pos][SN], pos) \rangle_v$$
$$\quad \vee \text{ENABLED } \langle extRejectFrame(id, env.frames[id][pos][SN], pos) \rangle_v$$

The RM shall handle resets of sending ES.

if eventually forever no reset or stuttering step occurs then
forever eventually frames are accepted

$$Reset1 \triangleq \Diamond\Box(\neg[reset]_v \wedge \forall\, id \in networks : isAlive[id])$$
$$\Rightarrow \Box\Diamond(\exists \langle id, pos \rangle \in deliverable :$$
$$\langle extAcceptFrame(id, env.frames[id][pos][SN], pos) \rangle_v)$$

Requirement Part 2:
The redundancy shall increase the availability of the network

Subbpart 1:
The RM does no worse job than a single network would do in
fault-free operation

if only one network is connected to the RM no frame is ever rejected

$$Quality0 \triangleq \Box(\exists\, id1, id2 \in networks : isAlive[id1] \wedge \neg isAlive[id2])$$
$$\Rightarrow \Box(\forall \langle id, pos \rangle \in deliverable :$$
$$\neg\text{ENABLED } \langle extRejectFrame(id, env.frames[id][pos][SN], pos) \rangle_v)$$

If both networks are alive the RM should satisfy:

if at least one member of a Twin Frame reaches the RM
at least one member is submitted to application layer

$$Quality1 \triangleq \forall\, id \in networks, pos \in (1 \,..\, (1 + MCFL)) :$$
$$\wedge\, isAlive[id]$$
$$\wedge\, isAlive[TNid[id]]$$
$$\wedge \langle id, pos \rangle \in deliverable$$
$$\wedge \text{ENABLED } \langle extRejectFrame(id, env.frames[id][pos][SN], pos) \rangle_v$$
$$\Rightarrow \exists\, frame \in out : frame[SN] = env.frames[id][pos][SN]$$

if at least one member of a Twin Frame reaches the RM and is new
$w.r.t$ its sequence number at least one member is submitted to application layer

$$Quality2 \triangleq \forall\, id \in networks, pos \in (1 \,..\, (1 + MCFL)) :$$
$$\wedge\, isAlive[id]$$
$$\wedge\, isAlive[TNid[id]]$$
$$\wedge \langle id, pos \rangle \in deliverable$$
$$\wedge \text{ENABLED } \langle extRejectFrame(id, env.frames[id][pos][SN], pos) \rangle_v$$
$$\wedge\, env.frames[id][pos][TAG] = \text{“n”}$$

$$\Rightarrow \exists\, frame \in out : frame[SN] = env.frames[id][pos][SN]$$

$$Quality3 \triangleq \Diamond(\Box\neg[reset]_v \Rightarrow \Box Quality2)$$

Subpart 2:
The redundancy shall increase availability of the network
three grades:

first grade – no more frames are rejected:

$$\begin{aligned} Avail1 \triangleq\ & \exists\, id1,\, id2 \in networks : (isAlive[id1] \wedge \neg isAlive[id2]) \\ & \Rightarrow (\forall \langle id,\, pos\rangle \in deliverable : \\ & \quad \neg\text{ENABLED } \langle extRejectFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v) \end{aligned}$$

$$\begin{aligned} Avail2 \triangleq\ & \Box(\ \wedge \Box(status \neq \text{“reject”}) \\ & \quad \wedge \exists\, id1,\, id2 \in networks : (isAlive[id1] \wedge \neg isAlive[id2]) \\ & \quad \Rightarrow (\forall \langle id,\, pos\rangle \in deliverable : \\ & \quad\quad \neg\text{ENABLED } \langle extRejectFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v)) \end{aligned}$$

second grade – eventually all successive frames are accepted

$$\begin{aligned} Avail3 \triangleq\ & \Diamond\Box(\exists\, id1,\, id2 \in networks : (isAlive[id1] \wedge \neg isAlive[id2])) \\ & \Rightarrow \Diamond\Box(\forall \langle id,\, pos\rangle \in deliverable : \\ & \quad \neg\text{ENABLED } \langle extRejectFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v) \end{aligned}$$

$$\begin{aligned} Avail4 \triangleq\ & \Diamond\Box(\neg[reset]_v \wedge \exists\, id1,\, id2 \in networks : (isAlive[id1] \wedge \neg isAlive[id2])) \\ & \Rightarrow \Diamond\Box(\forall \langle id,\, pos\rangle \in deliverable : \\ & \quad \neg\text{ENABLED } \langle extRejectFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v) \end{aligned}$$

third grade – continuously frames are accepted

$$\begin{aligned} Avail5 \triangleq\ & \Diamond\Box(\exists\, id1,\, id2 \in networks : (isAlive[id1] \wedge \neg isAlive[id2])) \\ & \Rightarrow \Box\Diamond(\exists \langle id,\, pos\rangle \in deliverable : \\ & \quad \text{ENABLED } \langle extAcceptFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v) \end{aligned}$$

$$\begin{aligned} Avail6 \triangleq\ & \Diamond\Box(\neg[reset]_v \wedge \exists\, id1,\, id2 \in networks : (isAlive[id1] \wedge \neg isAlive[id2])) \\ & \Rightarrow \Box\Diamond(\exists \langle id,\, pos\rangle \in deliverable : \\ & \quad \text{ENABLED } \langle extAcceptFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v) \end{aligned}$$

MODULE $RMA1$

EXTENDS $Naturals$, $Sequences$, TLC

CONSTANTS

$networks$,	set of networks
SN_CNT, SN_MAX, SN_HALF,	maximum sequence number
A, B, SN, TAG	just for convenience

VARIABLES

rm	Redundancy Management

$$TypeInvRM \triangleq rm \in [rsn : (0 \,..\, SN\_CNT),\ ptn : [A : (0 \,..\, SN\_CNT),\ B : (0 \,..\, SN\_CNT)],\ time : \text{BOOLEAN}]$$

Initially:

$$noVal \triangleq SN\_CNT$$

The redundancy management:
- has not received any frame
- each ptn-value equals special value $noVal$

$$InitRM \triangleq rm = [rsn \mapsto noVal,\ ptn \mapsto [A \mapsto noVal,\ B \mapsto noVal],\ time \mapsto \text{TRUE}]$$

REDUNDANCY MANAGEMENT STATEMENTS

Some functions:

return the other twin-newtork-id

$$TNid[id \in networks] \triangleq \text{IF } id = A \text{ THEN } B \text{ ELSE } A$$

subtract SN

$$s_1 -_{SN} s_2 =_{def} ((s_1 - s_2 + SN\_HALF) \bmod SN\_CNT) - SN\_HALF$$

$$subSN[s1,\ s2,\ noval,\ sn\_cnt,\ sn\_half \in Nat] \triangleq \text{IF } s2 = noval \text{ THEN } 1 \text{ ELSE } ((s1 - s2 + (sn\_half))\%sn\_cnt) - (sn\_half)$$

Sequence Number Skew

$$SNS(f) =_{def} RSN(f) -_{SN} RSN(PTN(f))$$

$$snSkew[id \in networks,\ rsn \in Nat] \triangleq subSN[rsn,\ rm.ptn[TNid[id]],\ noVal,\ SN\_CNT,\ SN\_HALF]$$

ACTIONS:
dummy action to obtain single environment

$$wait \triangleq \land \text{FALSE} \land \text{UNCHANGED } \langle rm \rangle$$

accept frame:
IF frames are available AND ($SNS(f) > 0$ OR $SNO(f) > 0$)
THEN forward frame

$$acceptFrame(id,\ sn) \triangleq \land\ snSkew[id,\ sn] > 0 \land\ rm' = [rm \text{ EXCEPT } !.rsn = sn,\ !.ptn[id] = sn]$$

reject frame:
IF frames are available $SNS(f) <= 0$ THEN reject frame

$$rejectFrame(id,\ sn) \triangleq \land\ snSkew[id,\ sn] \leq 0 \land\ rm' = [rm \text{ EXCEPT } !.rsn = sn,\ !.ptn[id] = sn]$$

Step of Redundancy Management

$$RM\_Next \triangleq \exists \langle id,\ sn \rangle \in networks \times (0\ ..\ SN\_MAX) : acceptFrame(id,\ sn) \lor rejectFrame(id,\ sn) \lor wait$$

The *RM* shall react on each frame

$$RM\_Fairness \triangleq \land \text{WF}_{\langle rm \rangle}(RM\_Next) \land \text{WF}_{\langle rm \rangle}(wait)$$

$$RM\_Spec \triangleq InitRM \land \Box[RM\_Next]_{\langle rm \rangle} \land RM\_Fairness$$

THEOREM $RM\_Spec \Rightarrow TypeInvRM$

```
MODULE RMA2

EXTENDS Naturals, Sequences, TLC

CONSTANTS
networks,                       set of networks
SN_CNT, SN_MAX, SN_HALF,        maximum sequence number
A, B, SN, TAG                   just for convenience

VARIABLES
rm                              Redundancy Management

----------------------------------------------------------------------

TypeInvRM ≜ rm ∈ [
                rsn  : (0 .. SN_CNT),
                paf  : (0 .. SN_CNT)
            ]

----------------------------------------------------------------------

 Initially:

noVal ≜ SN_CNT

 The redundancy management:
  - has not received any frame
  - each ptn-value equals special value noVal

InitRM ≜ rm = [
            rsn ↦ noVal,
            paf ↦ noVal
          ]

----------------------------------------------------------------------

 REDUNDANCY MANAGEMENT STATEMENTS

 Some functions:

 return the other twin-newtork-id

TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

 subtract SN
 s1 −SN s2 =def ((s1 − s2 + SN_HALF) mod SN_CNT) − SN_HALF

subSN[s1, s2, noval, sn_cnt, sn_half ∈ Nat] ≜
        IF s2 = noval THEN 1
         ELSE ((s1 − s2 + (sn_half))%sn_cnt) − (sn_half)

 Sequence Number Offset
 SNO(f) =def RSN(f) −sn PASN(f)
```

```
52  snOffset[rsn ∈ Nat] ≜ IF rm.paf = noVal THEN SN_CNT
53                         ELSE subSN[rsn, rm.paf, noVal, SN_CNT, SN_HALF]

55   ACTIONS:
56   dummy action to obtain single environment

58  wait ≜  ∧ FALSE
59          ∧ UNCHANGED ⟨rm⟩

61   accept frame:
62   IF frames are available AND (SNS(f) > 0 OR SNO(f) > 0)
63    THEN forward frame

65  acceptFrame(id, sn) ≜
66    ∧ snOffset[sn] > 0
67    ∧ rm' = [rm EXCEPT !.rsn = sn,
68                       !.paf = sn]

70   reject frame:
71   IF frames are available SNS(f) < = 0 THEN reject frame

73  rejectFrame(id, sn) ≜
74    ∧ snOffset[sn] ≤ 0
75    ∧ rm' = [rm EXCEPT !.rsn = sn]

77 ├──────────────────────────────────────────────────────────────┤
78   Step of Redundancy Management
79  RM_Next ≜ ∃ ⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
80                acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

82   The RM shall react on each frame
83  RM_Fairness ≜ ∧ WF⟨rm⟩(RM_Next)
84                ∧ WF⟨rm⟩(wait)

86  RM_Spec ≜ InitRM ∧ □[RM_Next]⟨rm⟩ ∧ RM_Fairness
87 ├──────────────────────────────────────────────────────────────┤
88  THEOREM RM_Spec ⇒ TypeInvRM

90 └──────────────────────────────────────────────────────────────┘
```

```
1  ---------- MODULE RMA3 ----------

3  EXTENDS Naturals, Sequences, TLC

5  CONSTANTS
6  networks,                          set of networks
7  SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
8  A, B, SN, TAG                      just for convenience

10 VARIABLES
11 rm                                 Redundancy Management

13 ----------

14 TypeInvRM ≜ rm ∈ [
15                  rsn  : (0 .. SN_CNT),
16                  paf  : (0 .. SN_CNT),
17                  ptn : [A : (0 .. SN_CNT), B : (0 .. SN_CNT)]]

19 ----------

20 Initially:

22 noVal ≜ SN_CNT

24 InitRM ≜ rm = [
25                  rsn ↦ noVal,
26                  paf ↦ noVal,
27                  ptn ↦ [A ↦ noVal, B ↦ noVal]]

29 ----------

30 REDUNDANCY MANAGEMENT STATEMENTS

32 Some functions:

34 return the other twin-newtork-id

36 TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

38 subtract SN
39 s1 −SN s2 =def ((s1 − s2 + SN_HALF) mod SN_CNT) − SN_HALF

41 subSN[s1, s2, noval, sn_cnt, sn_half ∈ Nat] ≜
42          IF s2 = noval THEN 1
43           ELSE ((s1 − s2 + (sn_half))%sn_cnt) − (sn_half)

45 Sequence Number Skew
46 SNS(f) =def RSN(f) −SN RSN(PTN(f))

48 snSkew[id ∈ networks, rsn ∈ Nat] ≜
49          subSN[rsn, rm.ptn[TNid[id]], noVal, SN_CNT, SN_HALF]
```

```
51  Sequence Number Offset
52  SNO(f) =_def RSN(f) −_sn PASN(f)

54  snOffset[rsn ∈ Nat] ≜
55          IF rm.paf = noVal THEN SN_CNT
56           ELSE  subSN[rsn, rm.paf, noVal, SN_CNT, SN_HALF]

58  ACTIONS:
59  dummy action to obtain single environment

61  wait ≜  ∧ FALSE
62          ∧ UNCHANGED ⟨rm⟩

64  accept frame:
65  IF  frames are available AND (SNS(f) > 0 OR SNO(f) > 0)
66   THEN  forward frame

68  acceptFrame(id, sn) ≜
69    ∧ ∨ snSkew[id, sn] > 0
70      ∨ snOffset[sn]    > 0
71    ∧ rm′ = [rm EXCEPT !.rsn = sn,
72                       !.paf = sn,
73                       !.ptn[id] = sn]

75  reject frame:
76  IF  frames are available SNS(f) < = 0  THEN  reject frame

78  rejectFrame(id, sn) ≜
79    ∧ snSkew[id, sn] ≤ 0
80    ∧ snOffset[sn] ≤ 0
81    ∧ rm′ = [rm EXCEPT !.rsn = sn,
82                       !.ptn[id] = sn]

84  ├──────────────────────────────────────────────────────────────┤
85  Step of Redundancy Management
86  RM_Next ≜ ∃ ⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
87                acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

89  The RM shall react on each frame
90  RM_Fairness ≜  ∧ WF_⟨rm⟩(RM_Next)
91                 ∧ WF_⟨rm⟩(wait)

93  RM_Spec ≜  InitRM ∧ □[RM_Next]_⟨rm⟩ ∧ RM_Fairness
94  ├──────────────────────────────────────────────────────────────┤
95  THEOREM  RM_Spec ⇒ TypeInvRM
96  └──────────────────────────────────────────────────────────────┘
```

```
1  ---------- MODULE RMA4 ----------

3  EXTENDS Integers, Sequences, TLC

5  CONSTANTS
6  networks,                          set of networks
7  SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
8  SNS_MIN,                           lower bound of saequence number skew
9  A, B, SN, TAG                      just for convenience

11 VARIABLES
12 rm                                 Redundancy Management

14 ---------------------------------------------------------------

15 TypeInvRM ≜ rm ∈ [
16                 rsn  : (0 .. SN_CNT),
17                 ptn : [A : (0 .. SN_CNT), B : (0 .. SN_CNT)]
18               ]

20 ---------------------------------------------------------------

21 Initially:

23 noVal ≜ SN_CNT

25 The redundancy management:
26  - has not received any frame
27  - each ptn-value equals special value noVal

29 InitRM ≜ rm = [
30              rsn ↦ noVal,
31              ptn ↦ [A ↦ noVal, B ↦ noVal]
32            ]

34 ---------------------------------------------------------------

35 REDUNDANCY MANAGEMENT STATEMENTS

37 Some functions:

39 return the other twin-newtork-id

41 TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

43 subtract SN
44 s1 −SN s2 =def ((s1 − s2 + SN_HALF) mod SN_CNT) − SN_HALF

46 subSN[s1, s2, noval, sn_cnt, sn_half ∈ Nat] ≜
47         IF s2 = noval THEN 1
48          ELSE ((s1 − s2 + (sn_half))%sn_cnt) − (sn_half)

50 Sequence Number Skew
```

```
51  SNS(f) =_def RSN(f) -_SN RSN(PTN(f))

53  snSkew[id ∈ networks, rsn ∈ Nat] ≜ subSN[rsn, rm.ptn[TNid[id]], noVal, SN_CNT, SN_HALF]

55  ACTIONS:
56  dummy action to obtain single environment

58  wait ≜  ∧ FALSE
59          ∧ UNCHANGED ⟨rm⟩

61  accept frame:
62  IF frames are available AND SNS(f) > 0 THEN forward frame

64  acceptFrame(id, sn) ≜
65    ∧ snSkew[id, sn] ∉ ( − SNS_MIN .. 0)
66    ∧ rm′ = [rm EXCEPT !.rsn = sn,
67                       !.ptn[id] = sn]

69  reject frame:
70  IF frames are available SNS(f) < = 0 THEN reject frame

72  rejectFrame(id, sn) ≜
73    ∧ snSkew[id, sn] ∈ ( − SNS_MIN .. 0)
74    ∧ rm′ = [rm EXCEPT !.rsn = sn,
75                       !.ptn[id] = sn]

77  ├──────────────────────────────────────────────────────────────┤
78  Step of Redundancy Management
79  RM_Next ≜ ∃⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
80              acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

82  The RM shall react on each frame
83  RM_Fairness ≜  ∧ WF_⟨rm⟩(RM_Next)
84                 ∧ WF_⟨rm⟩(wait)

86  RM_Spec ≜ InitRM ∧ □[RM_Next]_⟨rm⟩ ∧ RM_Fairness
87  ├──────────────────────────────────────────────────────────────┤
88  THEOREM RM_Spec ⇒ TypeInvRM

90  └──────────────────────────────────────────────────────────────┘
```

```
1  ---------- MODULE RMA5 ----------

3  EXTENDS Integers, Sequences, TLC

5  CONSTANTS
6  networks,                          set of networks
7  SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
8  SNS_MIN,                           lower bound of saequence number skew
9  A, B, SN, TAG                      just for convenience

11 VARIABLES
12 rm                                 Redundancy Management

14 ---------------------------------------------------------------

15 TypeInvRM ≜ rm ∈ [
16                  rsn  : (0 .. SN_CNT),
17                  paf  : (0 .. SN_CNT)]

19 ---------------------------------------------------------------

20 Initially:

22 noVal ≜ SN_CNT

24 InitRM ≜ rm = [
25               rsn ↦ noVal,
26               paf ↦ noVal]

28 ---------------------------------------------------------------

29 REDUNDANCY MANAGEMENT STATEMENTS

31 Some functions:

33 return the other twin-newtork-id

35 TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

37 subtract SN
38 s1 −SN s2 =def ((s1 − s2 + SN_HALF) mod SN_CNT) − SN_HALF

40 subSN[s1, s2, noval, sn_cnt, sn_half ∈ Nat] ≜
41         IF s2 = noval THEN 1
42          ELSE ((s1 − s2 + (sn_half))%sn_cnt) − (sn_half)

44 Sequence Number Offset
45 SNO(f) =def RSN(f) −sn PASN(f)

47 snOffset[rsn ∈ Nat] ≜ IF rm.paf = noVal THEN SN_CNT
48                       ELSE subSN[rsn, rm.paf, noVal, SN_CNT, SN_HALF]

50 ACTIONS:
```

```
51  dummy action to obtain single environment

53  wait ≜  ∧ FALSE
54          ∧ UNCHANGED ⟨rm⟩

56  accept frame:
57  IF frames are available AND SNS(f) > 0 THEN forward frame

59  acceptFrame(id, sn) ≜
60    ∧ snOffset[sn] ∉ ( − SNS_MIN .. 0)
61    ∧ rm′ = [rm EXCEPT !.rsn = sn,
62                       !.paf = sn
63            ]

65  reject frame:
66  IF frames are available SNS(f) < = 0 THEN reject frame

68  rejectFrame(id, sn) ≜
69    ∧ snOffset[sn] ∈ ( − SNS_MIN .. 0)
70    ∧ rm′ = [rm EXCEPT !.rsn = sn]

72  ──────────────────────────────────────────────────────────────
73  Step of Redundancy Management
74  RM_Next ≜ ∃ ⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
75                acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

77  The RM shall react on each frame
78  RM_Fairness ≜  ∧ WF⟨rm⟩(RM_Next)

80  RM_Spec ≜ InitRM ∧ □[RM_Next]⟨rm⟩ ∧ RM_Fairness
81  ──────────────────────────────────────────────────────────────
82  THEOREM RM_Spec ⇒ TypeInvRM

84  ──────────────────────────────────────────────────────────────
```

```
1  ---------- MODULE RMA6 ----------

3  EXTENDS Integers, Sequences, TLC

5  CONSTANTS
6  networks,                          set of networks
7  SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
8  SNS_MIN,                           lower bound of saequence number skew
9  A, B, SN, TAG                      just for convenience

11 VARIABLES
12 rm                                 Redundancy Management

14 ----------

15 TypeInvRM ≜ rm ∈ [
16                  rsn  : (0 .. SN_CNT),
17                  paf  : (0 .. SN_CNT),
18                  ptn  : [A : (0 .. SN_CNT), B : (0 .. SN_CNT)]]

20 ----------

21 Initially:

23 noVal ≜ SN_CNT

25 InitRM ≜ rm = [
26                rsn ↦ noVal,
27                paf ↦ noVal,
28                ptn ↦ [A ↦ noVal, B ↦ noVal]]

30 ----------

31 REDUNDANCY MANAGEMENT STATEMENTS

33 Some functions:

35 return the other twin-newtork-id

37 TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

39 subtract SN
40 s1 −SN s2 =def ((s1 − s2 + SN_HALF) mod SN_CNT) − SN_HALF

42 subSN[s1, s2, noval, sn_cnt, sn_half ∈ Nat] ≜
43         IF s2 = noval THEN 1
44          ELSE ((s1 − s2 + (sn_half))%sn_cnt) − (sn_half)

46 Sequence Number Skew
47 SNS(f) =def RSN(f) −SN RSN(PTN(f))

49 snSkew[id ∈ networks, rsn ∈ Nat] ≜ subSN[rsn, rm.ptn[TNid[id]], noVal, SN_CNT, SN_HALF]
```

```
51  Sequence Number Offset
52  SNO(f) =_def RSN(f) −_sn PASN(f)

54  snOffset[rsn ∈ Nat] ≜ IF rm.paf = noVal THEN SN_CNT
55                        ELSE subSN[rsn, rm.paf, noVal, SN_CNT, SN_HALF]

57  ACTIONS:
58  dummy action to obtain single environment

60  wait ≜ ∧ FALSE
61         ∧ UNCHANGED ⟨rm⟩

63  accept frame:
64  IF frames are available AND SNS(f) > 0 THEN forward frame

66  acceptFrame(id, sn) ≜
67    ∧ ∨ snSkew[id, sn] ∉ ( − SNS_MIN .. 0)
68      ∨ snOffset[sn]   ∉ ( − SNS_MIN .. 0)
69    ∧ rm′ = [rm EXCEPT !.rsn = sn,
70                       !.paf = sn,
71                       !.ptn[id] = sn]

73  reject frame:
74  IF frames are available SNS(f) < = 0 THEN reject frame

76  rejectFrame(id, sn) ≜
77    ∧ snSkew[id, sn] ∈ ( − SNS_MIN .. 0)
78    ∧ snOffset[sn]   ∈ ( − SNS_MIN .. 0)
79    ∧ rm′ = [rm EXCEPT !.rsn = sn,
80                       !.ptn[id] = sn]

82  ──────────────────────────────────────────────────────────────────────
83  Step of Redundancy Management
84  RM_Next ≜ ∃ ⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
85              acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

87  The RM shall react on each frame
88  RM_Fairness ≜ ∧ WF_⟨rm⟩(RM_Next)

90  RM_Spec ≜ InitRM ∧ □[RM_Next]_⟨rm⟩ ∧ RM_Fairness
91  ──────────────────────────────────────────────────────────────────────
92  THEOREM RM_Spec ⇒ TypeInvRM

94  ──────────────────────────────────────────────────────────────────────
```

```
1 ---------- MODULE RMA7 ----------

3 EXTENDS Integers, Sequences, TLC

5 CONSTANTS
6 networks,                          set of networks
7 SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
8 SNS_MIN,                           lower bound of saequence number skew
9 A, B, SN, TAG                      just for convenience

11 VARIABLES
12 rm                                Redundancy Management

14 ----------

15 TypeInvRM ≜ rm ∈ [
16                  rsn  : (0 .. SN_CNT),
17                  paf  : (0 .. SN_CNT),
18                  ptn : [A : (0 .. SN_CNT), B : (0 .. SN_CNT)]]

20 ----------

21 Initially:

23 noVal ≜ SN_CNT

25 InitRM ≜ rm = [
26               rsn ↦ noVal,
27               paf ↦ noVal,
28               ptn ↦ [A ↦ noVal, B ↦ noVal]]

30 ----------

31 REDUNDANCY MANAGEMENT STATEMENTS

33 Some functions:

35 return the other twin-newtork-id

37 TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

39 subtract SN
40 s1 −SN s2 =def ((s1 − s2 + SN_HALF) mod SN_CNT) − SN_HALF

42 subSN[s1, s2, noval, sn_cnt, sn_half ∈ Nat] ≜
43         IF s2 = noval THEN 1
44          ELSE ((s1 − s2 + (sn_half))%sn_cnt) − (sn_half)

47 Sequence Number Increment
48 SNI(f) =def RSN(f) −SN PSN(f)

50 snIncrement[sn ∈ Nat] ≜ subSN[sn, rm.rsn, noVal, SN_CNT, SN_HALF]
```

```
52  Sequence Number Skew
53  SNS(f) =_def RSN(f) -_SN RSN(PTN(f))

55  snSkew[id ∈ networks, rsn ∈ Nat] ≜ subSN[rsn, rm.ptn[TNid[id]], noVal, SN_CNT, SN_HALF]

57  Sequence Number Offset
58  SNO(f) =_def RSN(f) -_sn PASN(f)

60  snOffset[rsn ∈ Nat] ≜ IF rm.paf = noVal THEN SN_CNT
61                        ELSE subSN[rsn, rm.paf, noVal, SN_CNT, SN_HALF]

63  ACTIONS:
64  dummy action to obtain single environment

66  wait ≜ ∧ FALSE
67         ∧ UNCHANGED ⟨rm⟩

69  accept frame:
70  IF frames are available AND SNS(f) > 0 THEN forward frame

72  acceptFrame(id, sn) ≜
73    ∧ ∨ snIncrement[sn] ≤ 0
74      ∨ snOffset[sn] ∉ ( − SNS_MIN .. 0)
75    ∧ rm' = [rm EXCEPT !.rsn = sn,
76                       !.paf = sn,
77                       !.ptn[id] = sn]

79  reject frame:
80  IF frames are available SNS(f) < = 0 THEN reject frame

82  rejectFrame(id, sn) ≜
83    ∧ snIncrement[sn] > 0
84    ∧ snOffset[sn] ∈ ( − SNS_MIN .. 0)
85    ∧ rm' = [rm EXCEPT !.rsn = sn,
86                       !.ptn[id] = sn]

88  ──────────────────────────────────────────────────────────────

89  Step of Redundancy Management
90  RM_Next ≜ ∃ ⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
91              acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

93  The RM shall react on each frame
94  RM_Fairness ≜ ∧ WF_⟨rm⟩(RM_Next)

96  RM_Spec ≜ InitRM ∧ □[RM_Next]_⟨rm⟩ ∧ RM_Fairness

97  ──────────────────────────────────────────────────────────────

98  THEOREM RM_Spec ⇒ TypeInvRM

100 ──────────────────────────────────────────────────────────────
```

```
MODULE RMA7_2

EXTENDS Integers, Sequences, TLC

CONSTANTS
networks,                       set of networks
SN_CNT, SN_MAX, SN_HALF,        maximum sequence number
SNS_MIN,                        lower bound of saequence number skew
A, B, SN, TAG                   just for convenience

VARIABLES
rm                              Redundancy Management

TypeInvRM ≜ rm ∈ [
                rsn  : (0 .. SN_CNT),
                paf  : (0 .. SN_CNT),
                ptn : [A : (0 .. SN_CNT), B : (0 .. SN_CNT)]]

Initially:

noVal ≜ SN_CNT

InitRM ≜ rm = [
            rsn ↦ noVal,
            paf ↦ noVal,
            ptn ↦ [A ↦ noVal, B ↦ noVal]]

REDUNDANCY MANAGEMENT STATEMENTS

Some functions:
return the other twin-newtork-id

TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

subtract SN
s1 −SN s2 =def ((s1 − s2 + SN_HALF) mod SN_CNT) − SN_HALF

subSN[s1, s2, noval, sn_cnt, sn_half ∈ Nat] ≜
        IF s2 = noval THEN 1
         ELSE ((s1 − s2 + (sn_half))%sn_cnt) − (sn_half)


Sequence Number Increment
SNI(f) =def RSN(f) −SN PSN(f)

snIncrementAlt[sn ∈ Nat, id ∈ networks] ≜
      subSN[sn, rm.ptn[id], noVal, SN_CNT, SN_HALF]

Sequence Number Skew
```

```
50  SNS(f) =_def RSN(f) -_SN RSN(PTN(f))

52  snSkew[id ∈ networks, rsn ∈ Nat] ≜
53          subSN[rsn, rm.ptn[TNid[id]], noVal, SN_CNT, SN_HALF]

55  Sequence Number Offset
56  SNO(f) =_def RSN(f) -_sn PASN(f)

58  snOffset[rsn ∈ Nat] ≜
59          IF rm.paf = noVal THEN SN_CNT
60           ELSE  subSN[rsn, rm.paf, noVal, SN_CNT, SN_HALF]

62  ACTIONS:
63  dummy action to obtain single environment

65  wait ≜  ∧ FALSE
66          ∧ UNCHANGED ⟨rm⟩

68  accept frame:
69  IF  frames are available AND SNS(f) > 0  THEN  forward frame

71  acceptFrame(id, sn) ≜
72    ∧ ∨ snIncrementAlt[sn, id] ≤ 0
73      ∨ snOffset[sn]  ∉ ( − SNS_MIN .. 0)
74    ∧ rm′ = [rm EXCEPT !.rsn = sn,
75                       !.paf = sn,
76                       !.ptn[id] = sn]

78  reject frame:
79  IF  frames are available SNS(f) < = 0  THEN  reject frame

81  rejectFrame(id, sn) ≜
82    ∧ snIncrementAlt[sn, id] > 0
83    ∧ snOffset[sn]  ∈ ( − SNS_MIN .. 0)
84    ∧ rm′ = [rm EXCEPT !.rsn = sn,
85                       !.ptn[id] = sn]
86  ──────────────────────────────────────────────────────────────────────
87  Step of Redundancy Management
88  RM_Next ≜ ∃ ⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
89                  acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

91  The RM shall react on each frame
92  RM_Fairness ≜  ∧ WF_⟨rm⟩(RM_Next)

94  RM_Spec ≜  InitRM ∧ □[RM_Next]_⟨rm⟩ ∧ RM_Fairness
95  ──────────────────────────────────────────────────────────────────────
96  THEOREM  RM_Spec ⇒ TypeInvRM
97  ──────────────────────────────────────────────────────────────────────
```

```
1 ---------- MODULE RMA8 ----------

3 EXTENDS Integers, Sequences, TLC

5 CONSTANTS
6 networks,                          set of networks
7 SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
8 SNS_MIN,                           lower bound of saequence number skew
9 A, B, SN, TAG                      just for convenience

11 VARIABLES
12 rm                                Redundancy Management

14 ----------

15 TypeInvRM ≜ rm ∈ [
16                  rsn  : (0 .. SN_CNT),
17                  paf  : (0 .. SN_CNT),
18                  time : BOOLEAN ]

20 ----------

21 Initially:

23 noVal ≜ SN_CNT

25 InitRM ≜ rm = [
26               rsn ↦ noVal,
27               paf ↦ noVal,
28               time ↦ TRUE]

30 ----------

31 REDUNDANCY MANAGEMENT STATEMENTS

33 Some functions:

35 return the other twin-newtork-id

37 TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

39 subtract SN
40 s1 −SN s2 =def ((s1 − s2 + SN_HALF) mod SN_CNT) − SN_HALF

42 subSN[s1, s2, noval, sn_cnt, sn_half ∈ Nat] ≜
43         IF s2 = noval THEN 1
44          ELSE ((s1 − s2 + (sn_half))%sn_cnt) − (sn_half)

46 Sequence Number Skew
47 SNS(f) =def RSN(f) −SN RSN(PTN(f))

49 snSkew[id ∈ networks, rsn ∈ Nat] ≜ subSN[rsn, rm.ptn[TNid[id]],
50                                          noVal, SN_CNT, SN_HALF]
```

```
52  Sequence Number Offset
53  SNO(f) =_def RSN(f) −_sn PASN(f)

55  snOffset[rsn ∈ Nat] ≜ IF rm.paf = noVal THEN SN_CNT
56                        ELSE subSN[rsn, rm.paf, noVal, SN_CNT, SN_HALF]

58  ACTIONS:

60  exceed time bound:

62  wait ≜ ∧ rm.time = TRUE
63         ∧ rm′ = [rm EXCEPT !.time = FALSE]

65  accept frame:
66  IF frames are available AND SNS(f) > 0 THEN forward frame

68  acceptFrame(id, sn) ≜
69    ∧ ∨ rm.time = FALSE
70      ∨ snOffset[sn] ∉ ( − SNS_MIN .. 0)
71    ∧ rm′ = [rm EXCEPT !.rsn = sn,
72                       !.paf = sn,
73                       !.time = TRUE]

75  reject frame:
76  IF frames are available SNS(f) < = 0 THEN reject frame

78  rejectFrame(id, sn) ≜
79    ∧ rm.time = TRUE
80    ∧ snOffset[sn] ∈ ( − SNS_MIN .. 0)
81    ∧ rm′ = [rm EXCEPT !.rsn = sn,
82                       !.time = TRUE]

84  ──────────────────────────────────────────────────────────────

85  Step of Redundancy Management
86  RM_Next ≜ ∃ ⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
87              acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

89  The RM shall react on each frame
90  Rec_Fairness ≜  ∧ WF_⟨rm⟩(RM_Next)
91  Wait_Fairness ≜ ∧ WF_⟨rm⟩(wait)

93  RM_Fairness ≜ Rec_Fairness ∧ Wait_Fairness

95  RM_Spec ≜ InitRM ∧ □[RM_Next]_⟨rm⟩ ∧ RM_Fairness

96  ──────────────────────────────────────────────────────────────

97  THEOREM RM_Spec ⇒ TypeInvRM

99  ──────────────────────────────────────────────────────────────
```

```
1  ---------- MODULE RMA9 ----------

3  EXTENDS Integers, Sequences, TLC

5  CONSTANTS
6  networks,                          set of networks
7  SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
8  A, B, SN, TAG                      just for convenience

10 VARIABLES
11 rm                                 Redundancy Management

13 ----------------------------------------------------------------

14 TypeInvRM ≜ rm ∈ [
15                 rsn  : (0 .. SN_CNT),
16                 paf  : (0 .. SN_CNT),
17                 time : BOOLEAN
18               ]

20 ----------------------------------------------------------------

21 Initially:

23 noVal ≜ SN_CNT

25 InitRM ≜ rm = [
26               rsn ↦ noVal,
27               paf ↦ noVal,
28               time ↦ TRUE
29             ]

31 ----------------------------------------------------------------

32 REDUNDANCY MANAGEMENT STATEMENTS

34 Some functions:

36 return the other twin-newtork-id

38 TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

40 subtract SN
41 s1 −SN s2 =def ((s1 − s2 + SN_HALF) mod SN_CNT) − SN_HALF

43 subSN[s1, s2, noval, sn_cnt, sn_half ∈ Nat] ≜
44         IF s2 = noval THEN 1
45          ELSE ((s1 − s2 + (sn_half))%sn_cnt) − (sn_half)

47 Sequence Number Skew
48 SNS(f) =def RSN(f) −SN RSN(PTN(f))

50 snSkew[id ∈ networks, rsn ∈ Nat] ≜ subSN[rsn, rm.ptn[TNid[id]],
```

```
51                                                  noVal, SN_CNT, SN_HALF]

53   Sequence Number Offset
54   SNO(f) =_def RSN(f) −_sn PASN(f)

56  snOffset[rsn ∈ Nat] ≜ IF rm.paf = noVal THEN 1
57                          ELSE subSN[rsn, rm.paf, noVal, SN_CNT, SN_HALF]

59   ACTIONS:

61   exceed time bound:

63  wait ≜  ∧ rm.time = TRUE
64          ∧ rm' = [rm EXCEPT !.time = FALSE]

66   accept frame:
67   IF frames are available AND SNS(f) > 0 THEN forward frame

69  acceptFrame(id, sn) ≜
70    ∧ ∨ rm.time = FALSE
71      ∨ snOffset[sn] = 1
72    ∧ rm' = [rm EXCEPT !.rsn = sn,
73                       !.paf = sn,
74                       !.time = TRUE]

76   reject frame:
77   IF frames are available SNS(f) < = 0 THEN reject frame

79  rejectFrame(id, sn) ≜
80    ∧ rm.time = TRUE
81    ∧ snOffset[sn] ≠ 1
82    ∧ rm' = [rm EXCEPT !.rsn = sn,
83                       !.time = TRUE]

85 ──────────────────────────────────────────────────────────────

86   Step of Redundancy Management
87  RM_Next ≜ ∃ ⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
88              acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

90   The RM shall react on each frame
91  RM_Fairness ≜ ∧ WF_⟨rm⟩(RM_Next)
92                ∧ WF_⟨rm⟩(wait)

94  RM_Spec ≜ InitRM ∧ □[RM_Next]_⟨rm⟩ ∧ RM_Fairness

95 ──────────────────────────────────────────────────────────────

96  THEOREM RM_Spec ⇒ TypeInvRM

98 ──────────────────────────────────────────────────────────────
```

```
1  ---------- MODULE RMA11 ----------

3  EXTENDS Integers, Sequences, TLC

5  CONSTANTS
6  networks,                          set of networks
7  SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
8  SNS_MIN,                           lower bound of saequence number skew
9  A, B, SN, TAG                      just for convenience

11 VARIABLES
12 rm                                 Redundancy Management

14 ---------------------------------------------------------------

15 TypeInvRM ≜ rm ∈ [
16                   pasn : Seq((0 .. SN_MAX))
17                 ]

19 ---------------------------------------------------------------

20  Initially:

22 noVal ≜ SN_CNT

24  The redundancy management:
25   - has not received any frame
26   - each ptn-value equals special value noVal

28 InitRM ≜ rm = [
29                pasn ↦ ⟨⟩
30              ]

32 ---------------------------------------------------------------

33  REDUNDANCY MANAGEMENT STATEMENTS

35  Some functions:

37  return the other twin-newtork-id

39 TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

41  Test wether a certain sequence number is in the queue

43 inside[sn ∈ Nat, queue ∈ Seq(Nat)] ≜
44          IF queue = ⟨⟩ THEN FALSE
45           ELSE
46            IF Head(queue) = sn THEN TRUE
47             ELSE inside[sn, Tail(queue)]

49  ACTIONS:
50  dummy action to obtain single environment
```

```
wait ≜  ∧ FALSE
        ∧ UNCHANGED ⟨rm⟩

 accept frame:
 IF frames are available AND SNS(f) > 0 THEN forward frame

acceptFrame(id, sn) ≜
  ∧ inside[sn, rm.pasn] = FALSE
  ∧ rm′ = [rm EXCEPT !.pasn = IF Len(rm.pasn) < SNS_MIN
                              THEN Append(rm.pasn, sn)
                              ELSE Append(Tail(rm.pasn), sn)
          ]

 reject frame:
 IF frames are available SNS(f) < = 0 THEN reject frame

rejectFrame(id, sn) ≜
  ∧ inside[sn, rm.pasn] = TRUE
  ∧ UNCHANGED ⟨rm⟩

 Step of Redundancy Management
RM_Next ≜ ∃ ⟨id, sn⟩ ∈ networks × (0 .. SN_MAX) :
             acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

 The RM shall react on each frame
RM_Fairness ≜ ∧ WF⟨rm⟩(RM_Next)

RM_Spec ≜ InitRM ∧ □[RM_Next]⟨rm⟩ ∧ RM_Fairness

THEOREM RM_Spec ⇒ TypeInvRM
```

```
1  ---------- MODULE RMA12 ----------

3  EXTENDS Integers, Sequences, TLC

5  CONSTANTS
6  networks,                          set of networks
7  SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
8  SNS_MIN,                           lower bound of saequence number skew
9  A, B, SN, TAG                      just for convenience

11 VARIABLES
12 rm                                 Redundancy Management

14 ---------------------------------------------------------------

15 TypeInvRM ≜ rm ∈ [
16                  pasn  : Seq((0 .. SN_MAX)),
17                  time : BOOLEAN ]

19 ---------------------------------------------------------------

20 Initially:

22 noVal ≜ SN_CNT

24 InitRM ≜ rm = [
25               pasn ↦ ⟨⟩,
26               time ↦ TRUE]

28 ---------------------------------------------------------------

29 REDUNDANCY MANAGEMENT STATEMENTS

31 Some functions:

33 return the other twin-newtork-id

35 TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A

37 Test wether a certain sequence number is in the queue

39 inside[sn ∈ Nat, queue ∈ Seq(Nat)] ≜
40         IF queue = ⟨⟩ THEN FALSE
41          ELSE
42           IF Head(queue) = sn THEN TRUE
43            ELSE inside[sn, Tail(queue)]

45 ACTIONS:
46 exceed time bound:

48 wait ≜ ∧ rm.time = TRUE
49        ∧ rm' = [rm EXCEPT !.time = FALSE]
```

```
51  accept frame:
52  IF frames are available AND SNS(f) > 0 THEN forward frame

54  acceptFrame(id, sn) ≜
55    ∧ ∨ inside[sn, rm.pasn] = FALSE
56      ∨ rm.time = FALSE
57    ∧ rm' = [rm EXCEPT !.pasn = IF Len(rm.pasn) < SNS_MIN
58                                THEN Append(rm.pasn, sn)
59                                ELSE Append(Tail(rm.pasn), sn),
60                        !.time = TRUE
61             ]

63  reject frame:
64  IF frames are available SNS(f) < = 0 THEN reject frame

66  rejectFrame(id, sn) ≜
67    ∧ inside[sn, rm.pasn] = TRUE
68    ∧ rm.time = TRUE
69    ∧ rm' = [rm EXCEPT !.time = TRUE]

71 ├──────────────────────────────────────────────────────────────┤
72  Step of Redundancy Management
73  RM_Next ≜ ∃ ⟨id, sn, time⟩ ∈ networks × (0 .. SN_MAX) :
74              acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

76  The RM shall react on each frame
77  RM_Fairness ≜ ∧ WF⟨rm⟩(RM_Next)
78                ∧ WF⟨rm⟩(wait)

80  RM_Spec ≜ InitRM ∧ □[RM_Next]⟨rm⟩ ∧ RM_Fairness
81 ├──────────────────────────────────────────────────────────────┤
82  THEOREM RM_Spec ⇒ TypeInvRM

84 └──────────────────────────────────────────────────────────────┘
```

```
MODULE RMA13

EXTENDS Integers, Sequences, TLC

CONSTANTS
networks,                          set of networks
SN_CNT, SN_MAX, SN_HALF,           maximum sequence number
A, B, SN, TAG                      just for convenience

VARIABLES
rm                                 Redundancy Management

TypeInvRM ≜ rm ∈ [
                pan  : {A, B, "all"},
                time : BOOLEAN
              ]

Initially:

noVal ≜ SN_CNT

The redundancy management:
 - has not received any frame
 - each ptn-value equals special value noVal

InitRM ≜ rm = [
              pan  ↦ "all",
              time ↦ TRUE
            ]

REDUNDANCY MANAGEMENT STATEMENTS

Some functions:
return the other twin-newtork-id

TNid[id ∈ networks] ≜ IF id = A THEN B ELSE A


ACTIONS:
 exceed time bound:

wait ≜ ∧ rm.time = TRUE
       ∧ rm' = [rm EXCEPT !.time = FALSE]

accept frame:
IF frames are available AND SNS(f) > 0 THEN forward frame
```

```
51  acceptFrame(id, sn) ≜
52    ∧ ∨ rm.pan = "all"
53      ∨ id = rm.pan
54      ∨ rm.time = FALSE
55    ∧ rm' = [rm EXCEPT !.pan = id,
56                       !.time = TRUE
57             ]

59   reject frame:
60   IF frames are available SNS(f) < = 0 THEN reject frame

62  rejectFrame(id, sn) ≜
63    ∧ rm.pan ≠ "all"
64    ∧ id ≠ rm.pan
65    ∧ rm.time = TRUE
66    ∧ rm' = [rm EXCEPT !.time = TRUE]

68 ├──────────────────────────────────────────────────────────────┤
69   Step of Redundancy Management
70  RM_Next ≜ ∃ ⟨id, sn, time⟩ ∈ networks × (0 .. SN_MAX) :
71              acceptFrame(id, sn) ∨ rejectFrame(id, sn) ∨ wait

73   The RM shall react on each frame
74  RM_Fairness ≜ ∧ WF⟨rm⟩(RM_Next)
75                ∧ WF⟨rm⟩(wait)

77  RM_Spec ≜ InitRM ∧ □[RM_Next]⟨rm⟩ ∧ RM_Fairness
78 ├──────────────────────────────────────────────────────────────┤
79  THEOREM RM_Spec ⇒ TypeInvRM

81 └──────────────────────────────────────────────────────────────┘
```

Bibliography

[1] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996. Available from World Wide Web: `http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf`.

[2] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer Verlag, 1991.

[3] ARINC, Annapolis, Maryland, USA. *ARINC 664, Aircraft Data Networks, Part 7 — Deterministic Networks*. Available from World Wide Web: `http://www.arinc.com`.

[4] Felice Balarin, Paolo Giusto, Attila Jurecska, Claudio Passerone, Ellen M. Sentovich, Bassam Tabbara, Massimiliano Chiodo, Harry Hsieh, Luciono Lavagno, Alberto Sangiovanni-Vincentelli, and Kei Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Kluwer Academic Publishers, April 1997.

[5] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.

[6] Gerard Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.

[7] Gerard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999. Available from World Wide Web: `ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps`.

[8] Gerard Berry. The Foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[9] Robert Breyer and Sean Riley. *Switched, Fast and Gigabit Ethernet*. MacMillan Technical Publishing, third edition edition, 1999.

[10] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 1998.

[11] Willem Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification : Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[12] Esterel Technologies. Company homepage. `http://www.esterel-technologies.com`.

[13] Robert W. Floyd. Assigning meaning to programs. In *Proceedings AMS Symposium Applied Mathematics*, pages 19–31, 1967.

[14] S. Moisan G. Berry and J-P. Rigault. Esterel: Towards a synchronous and semantically sound high-level language for real-time applications. In *IEEE Real- Time Systems Symposium*, volume IEEE Catalog 83CH1941-4, pages 30–40, 1983.

[15] W.J. Goralski. *Introduction to ATM Networking*. McGraw-Hill, 1995.

[16] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. Available from World Wide Web: `http://citeseer.nj.nec.com/halbwachs91synchronous.html`.

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

[18] Leslie Lamport. *Specifying Systems – The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[19] Leslie Lamport, Mark Tuttle, and Yuan Yu. The wildfire verification challenge problem. `http://research.microsoft.com/users/lamport/tla/wildfire-challenge.html`.

[20] Paul le Guernic, Thierry Gautier, Michel le Borgne, and Claude le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1335, September 1991. Available from World Wide Web: `http://citeseer.nj.nec.com/context/53115/0`.

[21] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006. `http://www.informatik.uni-kiel.de/inf/von-Hanxleden/downloads/papers/sac06.pdf`.

[22] LNCS. *XEVE: An Esterel verification environment*, volume 1427, 1998.

[23] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[24] M. Nivat, C. Rattray, and G. Scollo T. Rus, editors. *Synchronous observers and the verification of reactive systems.*, June 1993.

[25] P.Naur. Proof algorithms by general snapshots. In *BIT*, pages 310–316, 1966.

[26] Amir Pnueli. The temporal logic of programs. In *In Proceedings of the 18th Symposium on Foundations of Programming Semantics*, pages 46–57, 1977.

[27] Reinhard von Hanxleden, Eddie Gambardella. AFDX Redundancy Management. EADS Airbus Technical report, February 2001.

[28] Andrew S. Tanenbaum. *Computernetzwerke*. Prenctice Hall, 4th edition edition, 2003.