Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science (120 credits) in Computer Science with Specialization in Software Engineering - DVA501

USING SAFETY ANALYSIS TECHNIQUES TO DERIVE SAFETY PROPERTIES FOR FORMAL VERIFICATION OF SAFETY-CRITICAL SYSTEMS

Ermia Hassanpour
ehr20003@student.mdh.se

Examiner: Antonio Cicchetti
Mälardalen University, Västerås, Sweden

Supervisors: Sara Abbaspour Asadollah
Barbara Gallina
Marjan Sirjani
Mälardalen University, Västerås, Sweden

Industrial supervisor:
Roger Dahlgren ABB, Västerås, Sweden

October 7, 2022

Abstract

Failure of safety-critical systems may cause severe injury to humans or the environment. In order to ensure the safety of such systems, rigorous methods must be used. Formal verification is one of these methods that helps in ensuring the system safety based on robust mathematical methods. Formal verification helps to prove certain properties of the model. This mathematical proof can strengthen the evidence for the safety claims of the system and help in standardization and receiving certificates. Defining safety properties for formal verification, more specifically for model checking, is very challenging. Finding the right set of properties, and writing these properties as temporal logic formulas, are difficult tasks for safety experts, software engineers, and developers. Model-driven development utilizes the concept of abstraction to deal with the complexity of systems and allow for earlier safety analysis. In this thesis, we intend to use model-driven methods and their supporting safety analysis tools to help in deriving the safety properties for model checking.

We use Timed Rebeca language and Afra model checking tool for formal verification, and CHESS tools for modelling the system and for safety analysis. As part of CHESS, ConcertoFLA technique includes the combination and automation of traditional safety analysis techniques that provide a qualitative assessment of the dependability of component-based systems. Failure Propagation Transform Calculus (FPTC) rules are the result of ConcertoFLA analysis. We extract safety contracts from FPTC rules. Then, we introduce a mapping between safety contracts and temporal logic properties in Afra. We also map internal block diagrams and sequence diagrams from CHESS models to Rebeca models to be able to perform model checking. Furthermore, we applied our approach to two case studies and used safety analysis results to more effectively derive the safety properties, and model check and debug the models.

Acknowledgements

I want to extend my sincere thanks to my supervisors, Marjan Sirjani, Sara Abbaspour Asadollah and Barbara Gallina, for the support and guidance during the process of creating and writing the thesis and allowing us to participate in the research work at Mälardalen University. Secondly, we would like to express our gratitude to Roger Dahlgren for helping us find and address fundamental issues in the provided system. Last but not least, we would like to thank my family for supporting me throughout the entire process of our studies.

Thank you very much!

Ermia Hassanpour, Västerås, September 2022

Table of Contents

List of Figures

List of Codes

1 Introduction

Technology increases the risk of harm as it becomes more prevalent [2]. Safety is often characterized as a system element that enables the system to function correctly [3]. In order to ensure the safety of safety-critical systems, rigorous methods must be used to verify the safety criteria. Formal verification can be used to ensure the system is safe based on robust mathematical methods. For formal verification, models are created for a given system, and then these models are checked to see if they satisfy the rigorous specifications of desired behaviour [4]. Formal verification helps to prove certain properties of the model. This mathematical proof can strengthen the evidence for the safety claims of the system. Hence, they can help in standardization and receiving certificates.

In order to apply formal verification, most specifically model checking, we need to model the system, specify the properties, and then check if the system is safe. One of the most challenging tasks in model checking is finding the set of required properties, including safety properties. In order to obtain safety properties, it is possible to extract them from the system specification documents. However, this may not be the most effective approach. Moreover, safety properties for model checking are formulated in temporal logic [5] and it is not easy for safety experts to specify properties in temporal logic. In this thesis, our goal is to create an approach to derive safety properties for model checking from a systematic safety analysis technique. This is a more effective and efficient method for identifying safety properties rather than relying solely on system specifications. In addition, using this approach, the safety experts do not need to specify the safety properties directly in temporal logic. They start from safety analysis using a model-driven approach. Model-driven approaches are generally more usable for safety experts.

There are different model-checking tools with different modelling languages; also, different methodologies and tools for safety analysis. We choose Afra [6] model checking tool, and Timed Rebeca modelling language [7, 8, 9]. The reason for this is that Timed Rebeca is an actor-based [10] language, suitable for modelling modern systems which are mostly networked asynchronous system. For safety analysis, we selected ConcertoFLA [11] to conduct the analysis. ConcertoFLA enables system architects and dependability engineers to customize component-based architectural models (specified in CHESSML [12]) with dependability information, analyse failure logic, and propagate the results back to the original model using Failure Logic Analysis (FLA) techniques [13].

The ConcertoFLA technique not only helps in finding the interest set of properties, but also helps engineers to create safety properties from ConcertoFLA results instead of trying to specify them in temporal logic directly. In ConcertoFLA, users specify the behaviour of individual components and possible failures, and then they can derive the possible failures at the system level. ConcertoFLA uses Failure Propagation Transform Calculus (FPTC). We produce FPTC rules, and from FPTC rules safety contracts are generated based on the method outlined in [14]. An integral part of ConcertoFLA is the combination and automation of traditional safety analysis techniques that provide a qualitative assessment of the dependability of component-based systems.

In this thesis, we show how to extract a set of safety properties from ConcertoFLA results. However, these safety properties cannot be used for model checking without the model of the system. Traditionally, similar to the safety properties, we create the model from system specifications. In this thesis, we extract the model from CHESSML diagrams. When the industry uses CHESS to run safety analyses, CHESSML diagrams are created. In CHESSML, Internal Block Diagrams (IBDs) model the system's structures. ConcertoFLA is applied on IBDs for failure transformation and propagation analysis. In this thesis, we introduce a method to map Internal Block Diagrams of CHESSML to Rebeca code. IBDs do not capture the behaviour of the system. The behavioural models include sequence diagrams. We apply methods presented in [15] to map sequence diagrams to Rebeca codes. The generated Rebeca codes are then used for model checking. We formally verify the model against the safety properties derived from safety contracts to identify potential hazards.

We apply our method to two case studies. We examine the first case study of a traffic light. The traffic light contains two lights and a controller that controls the lights. It is important to ensure that the light will not cause a traffic accident in order to ensure the safety of the traffic. The second case study is about train doors. The train contains a door, passengers, and a controller. An accident may occur if the door opens while the train is moving.

In the thesis, we first introduce all the concepts necessary for understanding our approach. After explaining all concepts, we present related work. Then, we introduce the research question and describe how the problem is formulated, and the research is conducted. As a next step, we propose a method to answer the research question, and apply our methodology on two case studies. Finally, we present the conclusion and potential future work.

2 Background

An introduction to safety analysis approaches, notations, and formal verification approaches to be used in the Rebeca modelling language and the Afra model checker is provided in this chapter.

2.1 Model-Driven Engineering

"Model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively." [16] Using models as fundamental artefacts in software engineering is known as model-driven engineering (MDE). In software development, model-driven engineering encompasses a wide range of model-driven techniques, including model-driven architecture, domain-specific modelling, model-driven integration, and component-based modelling [17].

Using component-based approaches, it is possible to construct systems from pre-existing components; component-based systems are designed to identify and develop reusable entities and their relationships based on the requirements for the system and the availability of pre-existing components as the starting point. A significant amount of implementation work will be eliminated during system development; however, the effort required to manage the components will increase: identifying them, choosing the most relevant ones, testing them, etc [18].

Safety Engineering Life-cycle: Safety engineering emphasizes the identification of failures that can result in hazardous situations and the presentation of compelling arguments proving that the system is safe. As part of the system engineering and assessment process, evidence was collected to support this argumentation. The first phase of the safety engineering life cycle is item definition. This phase outlines the item to be considered by the safety engineering process and the item's dependencies on its surroundings. Following a clear definition of the system, the Hazards and Risks assessment is conducted [19].

Unified Modelling Language (UML) Modern safety-critical system have become increasingly reliant on modelling. Modelling software systems has become standardized using the Unified Modelling Language (UML) since the mid-1990s. There are many phases involved in the development of software, including requirement phases, implementation phases, and maintenance phases. Modelling in software development has been empirically proven to be effective, but empirical evidence is scarce and limited. The UML is designed to provide a way for modelling applications across a broad range of fields in software engineering. UML has been designed in order to incorporate various existing modelling languages [20].

SysML *"OMG Systems Modelling Language (OMG SysML) is the official name of the language but it is referred as SysML."* [21] OMG SysML is a set of diagrams for specifying system requirements, behaviour, structure, and parametric connections. Block definition diagrams and internal block diagrams depict the system's structure. Figure 1, depicts the link between UML and OMG SysML using a Venn diagram [22].

Internal Block Diagram *"The internal block diagram describes the internal structure of a system in terms of its parts, ports, and connectors."*[22] In SysML, the Internal Block Diagram represents a block's internal structure in terms of attributes and connections between attributes. A block may have characteristics that define the block's values, components, and references to other blocks. Ports are a subclass of attributes used to indicate the sorts of interactions between blocks that are permitted.

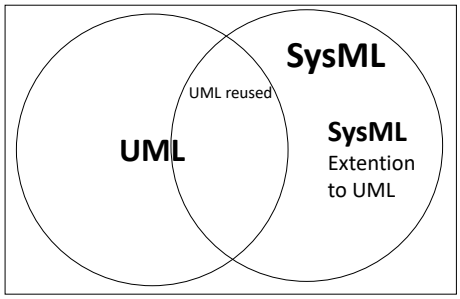


Figure 1: UML/OMG SysML Relationships

2.1.1 CHESS

An open-source initiative developed to improve visibility, usability, and standardization, CHESS is an open-source project that was developed with this goal in mind. It intends to improve model-driven engineering practices and technologies by using the CHESS methodology and supporting toolset (initially developed in the CHESS project) to ensure that components are developed and incorporated correctly in embedded systems to address safety, reliability, performance, robustness, and other non-functional concerns. CHESS is a project provides a selection of modelling tools for safety-critical systems. Also, CHESS is an a methodological framework for Eclipse based on MDT Papyrus. We can design, analyse, and model for essential systems using CHESS. CHESS study does not evaluate any functional properties for dependability. All modelling in CHESS is separated into viewpoints. These views will have distinctive nomenclature and constants [23, 24]. Figure 2 below gives a conceptual view about the CHESSML dependencies.

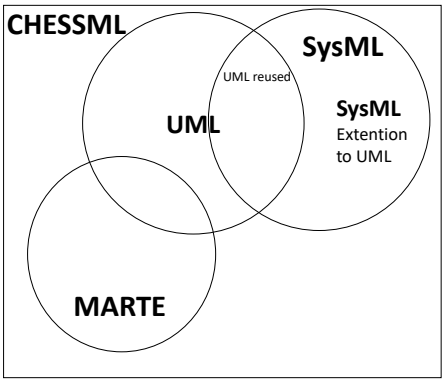


Figure 2: CHESSML dependencies

These are some of the many viewpoints available inside the CHESS tool environment, each of which is dedicated to a certain facet of the system under construction.

Requirement View. It is used to represent software requirements and link them to other features of the model, such as the implementations of particular components. SysML Requirement Diagrams are brought in to accompany the imported requirement view [23, 24].

Component View. It is a collection of perspectives that dissect the system into its component elements from several viewpoints, including functional and non-functional views. Included are both the functional and extra-functional aspects. In the functional viewpoint, state machines and other standard UML diagrams are used to show functional activity, and the system is modelled using a component-based design approach in which each component is supplied and needed interfaces built through ports [23, 24].

Deployment View. Parallel to the construction of the software architecture, the CHESS development process demands that the user model the target execution platform. The Deployment view enables the modelling of the target execution platform and software component allocations to hardware components [23, 24].

Analysis View. It is a collection of views via which the user may model the analysis contexts that will be utilized as input for many of the analysis tools provided. This perspective is divided into two different sections, each devoted to a certain sort of analysis [23, 24].

Figure 3 illustrates many of the key distinguishing characteristics of the CHESS tool environment's enforcement of separations of concerns.

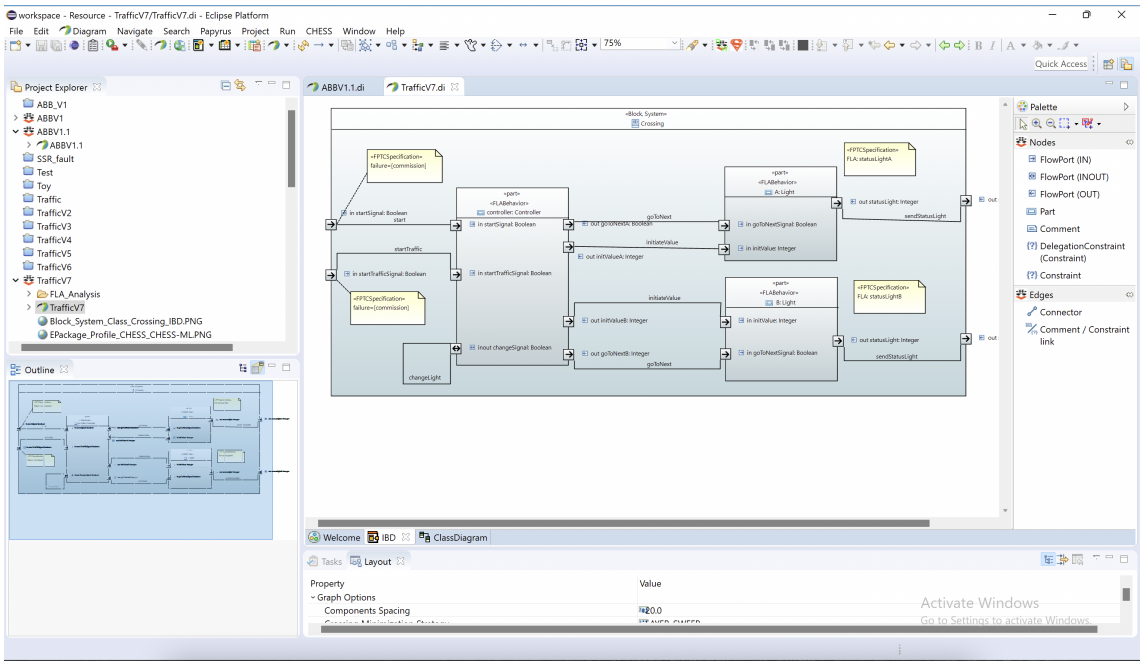


Figure 3: A screenshot of the CHESS application IDE with its available features

CHESS-FLA As part of the CHESS toolbox, CHESSFLA provides safety engineers with the opportunity to review high-level models for early based safety analysis. As a result, component-based architectural models can be enriched with dependability information, thereby serving as a foundation for Failure Logic Analysis (FLA) approaches, such as FPTC and FI4FA. A CHESSFLA investigation can be conducted on both hardware and software architectures. An analysis technique called FPTC (Failure Propagation Transformation Calculus) is explored in this thesis that allows for qualitative failure propagation analysis and transformation calculus analysis of systems consisting of components [25].

ConcertoFLA A component-based architectural model (specified in CHESSML) can be decorated with dependability related information by ConcertoFLA, which extends CHESSFLA, and failure logic analysis (FLA) results can then be back propagated to the original model by the user (system architects and dependability engineers). As the primary foundation for ConcertoFLA, Failure Propagation Transform Logic (FPTC) is utilized. A compositional approach, such as ConcertoFLA, is used to analyse the dependability of systems using established safety analysis techniques that are integrated and automated. Using ConcertoFLA, users are able to compute system-level behaviour based on the definition of the behaviour of each individual component. Four types of components can be used in ConcertoFLA: source of failure, which generates a failure due to a fault within the component; sink of failure, which prevents the external fault from propagating through fault tolerance; propagator of failure; and transformer of failure. ConcertoFLA rules are logical expressions that describe the input/output connection and determine the component's behaviour [11].

2.1.2 Failure Propagation and Transformation Calculus (FPTC)

As a result of the FPTC method, a system's failure behaviour is automatically deduced from the failure behaviours of its constituent components. A system's performance must be improved by understanding how it behaves during a failure [26].

Failure Behaviour of a single component Each component of the system must be analysed separately using the FPTC methodology. In order to determine whether a component is prone to failure, it must be analysed how it responds to potential failure stimuli. As a result of receiving input, a component will respond as follows:

- **Sink**:It implies that the component is capable of detecting and correcting failures that are produced elsewhere in the system
- **Source**: Components behave as sources when they generate failures on their own without receiving any failures as inputs.
- **Propagation**: A component receives a failure input and the type of failure on the output remains the same.
- **Transformation**: Transformation behaviour means that a component changes the nature of failure from one type to another [27].

Types of failure A system's failure modes provide an overview of the possible failure scenarios a system may face, and these scenarios can take four different perspectives. There are four perspectives from which failure modes can be viewed. These perspectives are: domains, consistency, detect ability, and environmental impacts. FPTC complies with the failure types defined by HAZOP/SHARD. Through a set of guide words, they identified types of failure [28].

- **Value failure**: Value failure means that the value of the result provided by a component is deviated from the expected range; value Subtle and value Coarse are subtypes of type value failure.
  - **Value Subtle** User cannot detect the deviation from expected values because the output falls outside the expected range of values.
  - **Value Coarse** This deviation can be detected by the user by noticing MORE/LESS in the output values.
- **Timing failure**: Timing failures refer to failures that occur outside the requested delivery window. Timing failure is divided into two subtypes named as early and late failure.
  - **Late** Components that provide results beyond the expected time-frame are considered late.
  - **Early** Components that provide results before their expected time frame are considered early.
- **Sequence / Provision failure**: This type of failure mode represents an assertion that a component delivered a sequence of failures that could be infinite, as well as having a relationship with the timing pattern. In sequence failures, there are two subtypes, namely, omissions and commissions.
  - **Omission** Describes a situation in which a component generates a result that is infinitely late (time interval) in comparison to what is expected.
  - **Commission** Supplying when it is not necessary

SyntaxExpressions are composed of two parts: the left-hand side (LHS) and the right-hand side (RHS). These expressions contain propagation and transformation instructions. FPTC syntax (Figure 4) enables users to specify component behaviours. A component's input behaviour is indicated on the left-hand side of the expression, and its output behaviour is indicated on the right-hand side of the expression [27]. As a means of reducing specification burden, wildcards

behaviour = expression + expression = LHS '→' RHS
LHS = portname'.' bL | portname '.' bL (',' portname '.' bL) +
RHS = portname'.' bR | portname '.' bR (',' portname '.' bR) +
failure = 'early' | 'late' | 'commission' | 'omission' | 'valueSubtle' | 'valueCoarse'
bL = 'wildcard' | bR
bR = 'noFailure' | failure

Figure 4: FPTC syntax supported in CHESS-FLA

and variables are used in FPTC. By using wildcards in an expression, the user does not need to care about failure types in specific positions or input behaviours that can play a decisive role in generating output behaviour. Input tokens that are matched by wildcards are denoted by "-" Variables should appear on the right-hand side of an expression in order to propagate them. Wildcards are denoted by "*". FPTC defines the following four possible behaviours of a component as a variable: normally, variables cannot be failure types, and any values other than failure types are treated as variables [27].

late → *	(*Sink*)
* → early	(*Source*)
late→ late	(*Propagate*)
omission→ value Subtle	(*Transform*)

Benefits of FPTC There are several benefits associated with the FTPC technique, including:

- FPTC is a qualitative analysis technique.
- In contrast to computing the failure behaviour of an entire system, analysing the failure behaviour of a single component (or component from bottom to top) is more straightforward.
- In the event that failure behaviour is calculated from the building blocks of the system, it is much easier to determine the impact of change of one component on the entire system and hence the process of improving it will be more cost-effective.

2.1.3 Safety Contracts

Assumptions A are made about the surroundings of a component, and assurances G are provided in exchange for those assumptions being met; this is a contract based on historical assumptions and guarantees. The semantics of contracts are determined by their implementation contexts. As long as an environment satisfies all the assumptions A of a contract,

$$C = \{A, G\}$$

it is considered to have fulfilled the contract. The contract is satisfied by implementations that fulfil both the assumption A and the guarantee G [27].

2.2 Formal Verification

In software and hardware systems, mathematical methodologies are used to define and verify their properties, often using computer-based tools to facilitate the process. Formal verification consists of developing mathematical models for a given system, and verifying that these models adhere to rigorous specifications of desired behavior [4].

Reactive Objects Language (Rebeca) Rebeca is a modeling language based on actors [7, 8, 9]. In concurrent computing, actors are commonly used as the universal primitives. It is a non-blocking asynchronous call to a message server associated with the actor that is used to communicate with the actor by way of message passing. Actors in Rebeca are self-contained units of concurrent executing programs. In Rebeca's Java-like syntax, actors are defined by reactive class definitions, which are analogous to Java's concept of classes. A message server is an actor method that defines the actor's response to a message. It is typically sufficient to inform them that each actor is a separate thread of operation and that message servers operate atomically and without preemption. Rebeca is being improved in a number of ways to ensure that it is suitable for use in various areas and analysis methods. A time-critical system can be modelled and verified using Timed Rebeca since it incorporates time-related properties [29].

The Rebeca model combines some concepts from actor-based modeling with some concepts from object-based modeling, thereby implying that Rebeca is utilized for designing distributed object-based systems that can be formally verified using a model checking tool. To make selected system model more appropriate for model checking, Rebeca uses abstraction techniques to the state space of the model.

The Structure of a Rebeca Model is briefly presented in thesis [15]:

1. Definition of reactive class
   (a) Generic: reactiveclass ClassName(queue size) {class body}
   (b) Example: reactiveclass Test(2) {...}
   (c) Rule: Class name should start with a capital letter.
2. Definition of constructor
   (a) Generic (without arguments): ClassName() {constructor body}
   (b) Generic (with arguments): ClassName(ExtType argument,...) {constructor body}
   (c) Example (without arguments): Test() { self.move(); testVar = true; }
   (d) Example (with arguments): Test(boolean value) { testVar = value; self.move(); }
   (e) Rule: Constructor is not preceded by any keyword other than the name of the class and it is used for initializing state variables and calling appropriate message servers. Notably, it is the first message that is executed by each Rebeca.
3. Definition of known rebecs or known reactive objects
   (a) Generic: knownrebecs {specification of known rebecs}
   (b) Example: knownrebecs { Test knowntest; }
   (c) Rule: Each known rebec is defined by a class name followed by the name of the object that is instantiated.
4. Definition of state variables
   (a) Generic: statevars {specification of state variables}
   (b) Example: statevars { boolean testVar; }
   (c) Rule: Each state variable is defined by a type (Type excluding ExtType) followed by the name of the variable.
5. Definition of message servers
   (a) Generic (without arguments): msgsrv MessageServerName() message server body
   (b) Generic (with arguments): msgsrv MessageServerName(ExtType argu- ment,...) message server body
   (c) Example (without arguments): msgsrv move() ...
   (d) Example (with arguments): msgsrv move(int value) ...

(e) Rule: Message servers accept ExtType for its arguments and, in compare with local methods, message servers can be accessed from other reactive classes as well.

6. Definition of local methods
   (a) Generic (without arguments): MethodName() {local method body}
   (b) Generic (with arguments): MethodName(ExtType argument,...) {local method body}
   (c) Example (without arguments): test() {...}
   (d) Example (with arguments): test(int numberOfUnits) {...}
   (e) Rule: Local methods accept ExtType for its arguments and in compare with message servers, local methods can not be accessed from other re- active classes. Local methods can be void and return methods.
7. Definition of main
   (a) Generic: main {main body}
   (b) Example: main Test test(otherTest):(); OtherTest optherTest(test):();
   (c) Rule: After the reactive classes are defined, they use main for instantiating reactive classes and passing required arguments and known rebecs which enables the execution.

[15]

The Model Checking Tool Afra Rebeca and Timed Rebeca models can be modelled and checked using Afra, a tool suite. Afra is a model checking integrated development environment (IDE) that facilitates the modeling and testing of Rebeca and Timed Rebeca models. Initially, it is intended to be a standalone Eclipse application. Afra provides a suite of Eclipse views and editors along with Java components for the construction and analysis of models, which is similar to many other Eclipse plugins [29].

2.3 CHESSML to Rebeca

When all the considered Rebeca concepts have been identified and documented they can proceed by providing a corresponding CHESSML concepts that can be used as mapping pairs in the process of creation of the mapping procedure. This process is iterative as they iterate both through CHESSML and identified Rebeca concepts and perform detection of mapping pairs.

Mapping Sequence Diagram to Rebeca Code A sequence diagram represents methods (message servers, local methods, or constructors) defined as part of the local method or message server. In addition to defining sequences of messages exchanged via message servers and local methods, it also allows us to define internal circular messages. The left-hand side of the diagram displays a source lifeline containing only the class name, while the right-hand side displays the object name and its corresponding class name. The sequence diagrams model a single method (attributed to class-level logic), which can be invoked from objects within other classes defined within them (which is typical for Rebeca and definitions of known Rebecas). The following notation represents the expected modeling notation in this thesis, and these semantic additions will be incorporated into the conversion validation. Sequence diagram and its semantics include in the thesis [15].

3 Related Work

In this thesis, one of the main objectives is to identify the hazards and risks in an early stage of the development of a system.

In previous works, Sirjani et al. [30] tried to close the remaining gap between high-level needs and the actor model; they used a structuring strategy based on the GIVEN-WHEN-THEN syntax to remove ambiguity and smooth the transfer from requirements to the formal model [30].

According to Gallina et al. [31] it has become common to make multi-concern statements about, for instance, safety and security, and how these issues interact in safety-critical domains such as automotive, rail, and aviation, and to justify these statements with evidence. The validity of such claims is largely explained utilizing process-and-product-related types of evidence. It may be necessary to combine mono-concern and multi-concern analysis results at the level of system design, along with a justification of whether the chosen analysis techniques are compliant with relevant standards. Also, they said engineers can use FLA as a tool to add dependability-related behavioural specifications to component-based system specifications. Through FLA, traditional safety analysis techniques (FTA and failure mode and effects analysis) are combined and automated, resulting in a calculation of failure behaviour at the system level based on failure behaviour at the component level.

According to Sljivo et al. [32], a variety of contexts restricts the reuse of accompanying safety artefacts. Their method involves generating reusable arguments-fragments related to compositional safety analysis that include supporting evidence. The generation process takes place based on safety contracts that represent the behaviour of components in assumption/guarantee pairs, accompanied by supporting evidence. In relation to our work, the authors attempted to apply failure logic analysis results to safety case fragments and artefacts. In the FLAR part, they attempted to model the component architecture in CHESS-FLA, then specify the failure behaviour of a component isolated using FPTC rules; translate the FPTC rules into corresponding safety contracts. The next step was to formally verify the safety of contracts so that they could be used as evidence. He verified safety contracts by OCRA tools. A component-based system specific with a temporal logic is supported by OCRA for the first time, which allows the verification of refined contracts. In conjunction with the SafeCer project, OCRA was developed as a component-based certification tool that is currently being used by industrial partners. The tool is also integrated with a UML-based modelling environment for aerospace systems [33]. We used the Afra model checker instead of OCRA for our formal verification.

According to Fatemeh et al. [34] introduced a new tool that facilitates the modelling and verification of distributed systems composed of reactive objects communicating using asynchronous messages. The tool includes all features of our previous studies and introduces new features specific to the development process. By using Rebeca tools and theory, this approach allows users to develop their software systems using UML while still benefiting from formal verification support. The addition of verification to the software development lifecycle combines two separate approaches to modelling. However, they introduced new tools which do not include any safety analysis, which does not convince industries to use them. Also, these approaches did not help with the main problem we noted, which is the property part. As part of this study, extensive conceptual mappings are provided for all of Rebeca's structural and behavioural ideas on a syntactic level, for which short semantical mappings are provided for structural concepts, and incomplete behavioural concepts are included without a description of their mappings.

According to Djukanovic et al. [15] a UML profile did not create for this work as they wish to maintain the UML in its original form. Further, they need to specify certain rules that will be used to verify the correctness of source UML models before they are converted to Rebeca, as well as ensuring compliance with the mapping technique. As a result of the combination of these principles, a modelling pattern appears to be recommended for the UML diagrams being considered. They also consider both structural and behavioural diagrams of UML models, namely the Class diagrams and Object diagrams for structural diagrams and Sequence diagrams and State-machine diagrams for behavioural diagrams. They emphasize primarily three of the UML diagrams listed (a class diagram, an object diagram, and a behavioural diagram) in an effort to maximize the available information and reduce the number of diagrams. In this thesis, they did not use an internal block diagram, which is one of the essential diagrams needed for safety analysis, but their approach can

be used to complete part of our Rebeca models.

To summarize related work, there are studies that have been done over safety analyses by ConcertoFLA technique. Additionally, those studies attempted to formally verify techniques, first of all, there is no bridge between CHESS and Afra. Also, we could not find any work about extracting safety properties.

4 Problem Formulation

Model-Driven Development (MDD) is a software development methodology that stresses abstraction to manage system complexity. This technique allows for analysis to occur early in the development phase [30]. MDD applies UML profiling and separation of concerns by specifying well-defined design views that address specific aspects of the problem. The system can then be described in terms of its non-functional, functional, and deployment aspects in a focused manner, thereby avoiding interference between issues pertaining to various domains of the system [35].

The use of MDD in safety-critical systems requires (semi-)formal methods of designing and simulating the system at various stages of development. Using modelling language standards such as UML has become widespread among engineers in various fields. The essential step to verify the modelled system in UML is to verify them throw safety requirements. Identifying and verifying safety requirements in safety-critical systems requires stringent solutions. Therefore, we need a modelling framework that supports formal verification. It is possible to check models in different languages using a variety of model-checking tools. In order to conduct formal verification of the system, the Rebeca modelling language was chosen. In order to build a model that is faithful to the system to be modelled, Rebeca models are constructed based on the safety requirements as well as the system architecture. The model is also used to generate executable code for the system as a result [30]. An industry safety document which is designed for the particular system is used to describe all safety requirements that have to be addressed.

Various types of documents are supplied by industries. Among the most essential are safety documents. Businesses may use safety documents to guarantee that their designs adhere to safety regulations. These documents include the design of the safety component of a system. This safety layer aims to resolve problems and identify methods for making the system safer. They strive to create evidence to guarantee the safety of their systems. Explanations and evidence are supported by lucid reasoning and rational arguments. According to Šljivo et al., the standard safety method starts with hazard identification. This identification may be achieved by analysing the system's operation and response to failures. After identifying potentially hazardous components, they may seek to build safety contracts to assure the safe construction of their system [14]. ConcertoFLA is a qualitative failure logic analysis. This technique attempts to analyse each component separately by calculating its failure propagation and transformation (FPTC). The end of FPTC rules may assist us comprehend each component's failure-related behaviour.

A model of the safety-critical system is created and uses model checkers to ensure that the approach is accurate. If the modelled system can be formally verified, its safety requirements will be backed by substantial evidence. For formal verification, we use Rebeca, an actor-based modelling language, and its model-checking tool, Afra. As a first step, the specified system should be modelled using the Rebeca modelling language. Afra tools will use the modelled system to check the safety properties that are also provided to Afra as part of the verification process. It will formally verify that all safety requirements have been examined and satisfied. Locating these properties is one of the primary concerns in this thesis.

4.1 Research Goal and Question

As a result, the key research question is presented below:

RQ: How to derive safety properties from qualitative failure logic analysis (FLA) results and use them for formal verification? More specifically, how to derive safety properties from safety contracts and use them as properties for model checking Rebeca models? [1]

[1]Note: Safety contract are derived from FPTC rules (Failure Propagation and Transformation Calculus), and

There are two failure logic analyses, qualitative and quantitative. As part of its qualitative failure logic analysis (FLA), ConcertoFLA employs the Failure Propagation and Transformation Calculus (FPTC) formalism while utilizing the State-based Analysis (SBA). Engineers may enhance component-based system specifications with dependability-related behavioural specifications, perform the analysis, and incorporate the analysis results into the original system specifications in both cases. In FLA, local component failure propagation channels are defined in accordance with FPTC rules to specify qualitative dependency-related behaviour. As a result of component failures and injected behaviour, FLA automates safety analysis (Fault tree analysis and failure mode and effects analysis) and calculates qualitative system failure.

In this thesis, using the method presented in [14], we aim to extract safety properties. In order to create safety contracts, they used the results of a ConcertoFLA technique based on the failure propagation and transformation calculus (FPTC). Secondly, the use of ConcertoFLA findings throughout the model verification process will assist in the identification of safety properties that are used in model checking. For answering the second question, we investigate how safety analysis (ConcertoFLA technique) helps in improving the process of specifying temporal logic properties used in model checking, and how it helps in building a more complete set of properties.

Also, one of the outcomes of this thesis is a mapping from the internal block diagram to the Rebeca code. This will enable us to have a bridge from CHESS to Afra tools.

4.2 Limitations

The following section aims to provide a summary of the limitations of this dissertation over a broad range of topics:

- One of the most significant constraints in the project is that we need the safety documents of a company to perform the analysis. It is important to note that without these documents, we would not be able to apply our method.
- Another limitation is that the properties derived using this analysis are not necessarily the most interesting safety properties of the system that we are looking for in model checking, properties that show different relations of components in a system.

4.3 System development research methodology

According to Nunamaker and Chen [1], system development research methodologies are multi-dimensional and multi-methodological approaches that can be applied to engineering and technical science research. System development research involves the *development of systems*, *experimentation*, *observation* and *performance testing* as part of the overall research contribution. The system development research method follows the theory of system development and other complementary methods. In addition to investigate the different facets of the research questions, these methods have the additional benefit of making them applicable in areas where it may be difficult to formulate research questions and determine the direction of a project in the early stages.

As part of the systems development research methodology, four phases are involved in the actual development of the system which are shown in Figure 5. In the iterative nature of the chosen research methodology, each phase might be conducted more than once with the knowledge and information gained from previously conducted iterations. Based on the scientific contribution contained in this thesis, the following steps are considered:

Construct a Conceptual Method. During this phase, the research question and the scientific purpose of the thesis is defined. Also, the system's requirements are investigated and the method-building process is clarified.

FPTC rules are derived from the system specification using ConcertoFLA, based on the approach presented in [14], [32].

Analyse and Design the Solution. The purpose of this phase is defining a method to derive safety properties. In addition, to meet the objective of this thesis, we considered reviewing the relevant studies in order to develop approaches and ideas for proposing solutions. This phase begins with the specification of the system.

Case Studies. This thesis runs the design method (discussed in the previous phase) over two case studies. Case studies typically are the most suitable methodological approaches to conducting an intensive study of selected examples.

Observe and Evaluate System. Performance evaluation is carried out in this phase by including a proof of concept assessment and an evaluation of performance.

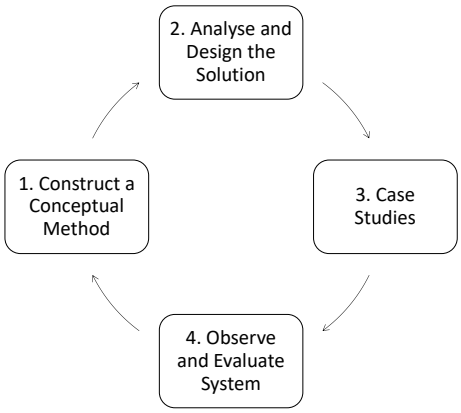


Figure 5: System development research process [1]

5 Methods to Solve the Problem

This chapter presents methods to solve the problem explained in Section 4, Sections 5.2 presents the mapping rules for transforming IBD to Rebeca code . Sections 5.2, 5.4 addresses the integration-related characteristics of FPTC and Rebeca code. Figure 6 depicts the steps we followed to answer our the research question. In this methodology, The first step (given in gray background) is inspired by the introduced method in [27]. We use the results of the first three steps in the next steps to answer our research question.

Rationale Safety-critical systems that are developed using a model-driven approach will attempt to model their specifications. There are two types of modelling tools: formal and informal. We use CHESS, a modelling tool that allows us to perform failure logic analysis, as one of the informal tools. On the formal side, Afra Tools is a formal verification tool that requires a system model in Rebeca modelling language along with properties to verify the model. A significant challenge in the formal verification tools of Afra was how to derive properties when using them for formal verification. One way to determine Afra's safety properties is to connect CHESS and Afra. By utilizing the method described in the thesis [14], we extracted safety contracts from ConcertoFLA results, one of the CHESS tools for safety analyses.

In order to create a safety contract, we first need to perform the steps proposed in [14]. The safety contracts typically specify that each component may fail in a certain situation. Using the failure information, we determine safety properties that can be used in the model checking step. A safety property is a model for verifying the system in the formal verification language of Rebeca.

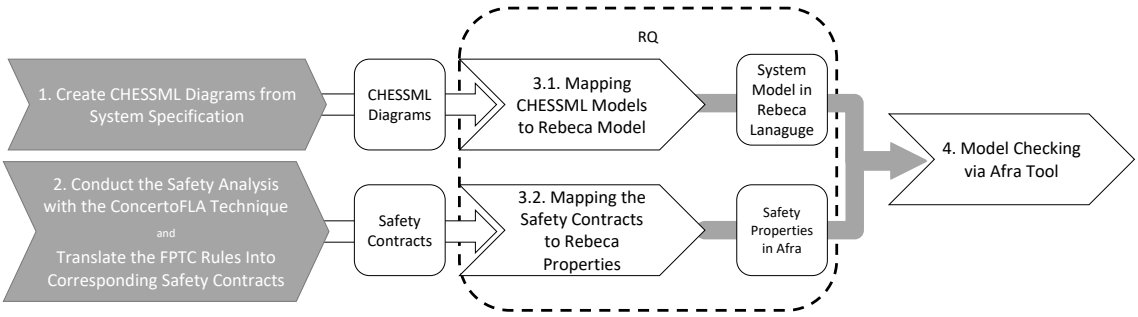


Figure 6: An overview of our approach.

5.1 Create CHESSML Diagrams from System Specification

The minimum CHESSML diagrams necessary for establishing a detailed enough mapping procedure that ensures a meaningful Rebeca information, include Class diagram and Object diagram as structural diagrams and Sequence diagram as behavioural [15]. As part of this step, we create the minimum number of diagrams necessary to model the system. Diagrams are constructed to represent both the behaviour and the structure of the diagram. The thesis we try to use Sequence diagram to model the behaviour of the system. Also, we try to create a mapping between Internal Block Diagrams and Rebeca code to model the structure of the system.

5.2 Conduct the Safety Analysis with ConcertoFLA Technique

In this step, we tried to reproduce the steps done in the thesis. We try to define all system specifications in the CHESS tools. Then, for each component created in the internal block diagram, we write FPTC rules for all of its ports. These FPTC rules are the results of ConcertoFLA analysis.

Translate the FPTC Rules into Corresponding Safety Contracts In this step, we translate the FPTC rules to safety contracts with help of two rules (Rule 1 and Rule 2) introduced in [36]. The rules and an example for each rule are listed as follows:

- ***Rule 1***: For each input, identify the different failures and assign them as assumptions connected by the AND (&&) operator. Figure 7 shows an example of rule 1 when we have a component with three inputs and two outputs. Two of the three inputs are defective with a failure. After mapping them to assumptions, there are two inputs that have a failure in the FPTC rule.

FPTC rules	Safety contracts
Inputs: Each FPTC rule contains all inputs, some inputs may have NoFailure, while others may have a type A failure. *ComponentInputAPort.noFailure ,* *ComponentInputBPort.FailureType ,* *ComponentInputCPort.FailureType* *->* *ComponentOutputAPort.FailureType ,* *ComponentOutputBPort. noFailure ;*	For each input, identify the different failures and assign them as assumptions connected by the AND operator. A: not (*ComponentInputBPort.FailureType &&* *ComponentInputCPort.FailureType)*

Figure 7: An example of Rule 1

- ***Rule 2***: For each output, identify the different failures and assign them as guarantee connected by the AND (&&) operator. Figure 8 shows an example of rule 2 when we have a component with three inputs and two outputs. Two of the three output are defective with a failure. After mapping them to guarantee, there are two outputs translated to safety contract.

FPTC rules	Safety contracts
Outputs: Each FPTC rule contains all outputs, some outputs may have NoFailure, while others may have a type A failure. *ComponentInputAPort.noFailure ,* *ComponentInputBPort.FailureType ,* *->* *ComponentOutputAPort. noFailure ,* *ComponentOutputBPort. FailureType ,* *ComponentOutputCPort.FailureType ;*	For each output, identify the different failures and assign them as guarantee connected by the AND operator. G: not (*ComponentOutputBPort.FailureType &&* *ComponentOutputCPort.FailureType)*

Figure 8: An example of Rule 2

5.3 Mapping Internal Block Diagram to Rebeca Code

In this section, we describe how we map internal block diagrams to Rebeca codes. This is presented as arrow 2.1 in Figure 6. Each internal block diagram is divided to three primary sections: blocks, ports, and connections. Ports are assigned to blocks, and connections are established between blocks via the use of ports. Blocks are classified into two types: system blocks and regular blocks. System blocks are the primary constituents of each internal block diagrams. Multiple system blocks can be incorporated into a single system block. The following list shows our rules to map an internal block diagram to a Rebeca code model:

1. ***Rule 1:*** This rule is for mapping the main internal diagram to Rebeca. To achieve this, we add *main*{} into the Rebeca code to map a primary internal block diagram to the Rebeca code. In Rebeca, *main* shows the entry point for defining reactive classes. As it is shown in

Figure 9, since we have an internal block diagram for the whole system; therefore we create a *main* in Rebeca code to use it to instantiate new the components.

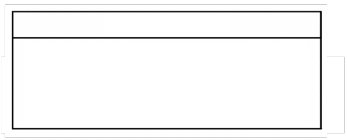
Internal Block Diagram in CHESS	Rebeca Modeling Language
Main File 	Map to Main The Main includes instantiation of rebecs from Reactive classes. `main {}`

Figure 9: Mapping system Internal Block Diagram to Rebeca

2. ***Rule 2:*** Each internal block diagrams block corresponds to a reactive class in Rebeca. The name of the reactive class is mapped to the name of the block in internal block diagrams. After generating a reactive class in Rebeca code, we also need to create an instance of the reactive class in the main{}. Each component inside the system may have its own block diagram. There is an internal block diagram hierarchy that starts with the system's internal block diagrams and extends to each component; If that component contains more components, we need to create an internal block diagram for them too. Recursively, internal block diagrams must be created until there is no component left without a diagram. Figure 10 shows an example of Mapper rule 2. It shows when we have a block, Test, in CHESS and after applying the Mapper Rule 2 we have a *reactiveclass* $Test()\{\}$, and an instance of this reactive class in the $main\{\}$ code.

Internal Block Diagram in CHESS	Rebeca Modeling Language

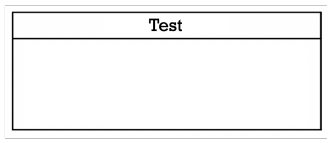
Block 	Map to Reactiveclass
	
	The name of the block is the name of the Reactive class. One instance of this class is added to the main part of the code.
	
	`reactiveclass Test() {}`
	
	`main {`
	`  Test reactiveclassInstanceName():();`
	`}`

Figure 10: Mapping a block to Rebeca

3. ***Rule 3:*** After applying the Mapper Rule 2, we need to map each system block in CHESS to reactive class constructor, as well as adding the *Msgsrvs*. Figure 11 shows an example of Mapper Rule 3. It shows when we have a block, Test, in CHESS and after applying the Mapper Rule 3 we have a constructor, Test(){}, and a *Msgsrvs*, msgsrv testMsgsrv(){} , in the *reactiveclass* code.

4. ***Rule 4:*** This rule is for mapping the block's ports. When we have a port, then we add *Statevar* in the Rebeca code, while the types of variables in *Statevar* are as the same as the

Internal Block Diagram in CHESS	Rebeca Modeling Language

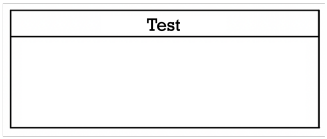
System Block
 	
It is a system-wide block that will contain all of the system's blocks.
 	
 
	Reactive class
	
	System block is like other blocks, and it calls all its Msgsrvs in its constructor.
	
	`reactiveclass Test() {`
	`  Test() {`
	`  self.testMsgSrv();`
	`  }`
	
	`  msgsrv testMsgSrv(){}`
	`}`

Figure 11: Mapping a system block to Reactiveclass

types of the ports in the block. Figure 12 presents an example of mapping port in Rebeca code.

Internal Block Diagram in CHESS	Rebeca Modeling Language

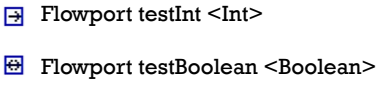
Ports
 	
Ports are points at which external entities can connect to and interact with a block.
 	
CHESS has Input, InOut, and Output port types.
 	
 
	Map to Statevars
	
	Each state variable is defined by the name of the variable.
	
	Port name map to the state variable name, port type map to the state variable type.
	
	`statevars {`
	`  int testInt;`
	`  boolean testBoolean;`
	`}`

Figure 12: Map Ports to Statevars

5. ***Rule 5:*** The previous Mapper Rules show how we map IBD connections to message servers (*Msgsrv*) in Rebeca. All connections have a source block which starts from that block and a destination block which ends on that block. For each connection, a message server (*Msgsrv*) is produced and added to Rebeca for both the source and destination blocks. The transmitting value on the output port of the mapped-to-Statevar block will be transmitted as an argument to the destination message server in the source block message server. Moreover, the Reactiveclass of the destination block contains the Knownrebecs of the source block. We add an instance of the *Reactiveclass* to the *Knownrebecs* of the source block.

   Figure 13 shows an example of Mapper rule 5. When we have 2 blocks, BlockA and BlockB, which are linked with their ports. So we map it to a message server (*msgsrv*) in *reactiveclass*. Figure 13 shows an example of Mapper Rule 5 when we have two blocks (BlockA and BlockB) with a connection between these two blocks. We call the message server (*BMsgSRV()*) of BlockB by adding ***b.BMsgSrv();*** in a message server(*test()*), of BlockA to map the connection between these to block.

6. ***Rule 6:*** When a connector is connected to the InOut port (the source port is the same as

Internal Block Diagram in CHESS	Rebeca Modeling Language

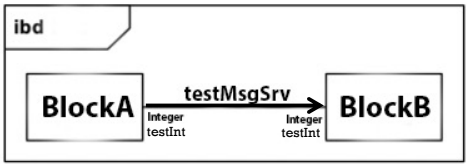
Connectors
 	
A port of Block A is connected to a port of Block B by a connection.
 	
Each connector has a source block and a destination block.
 	
Block A is the source and Block B is the destination.
 	
 
	Map to Msgsrv
	
	Msgsrvs are added to both Reactiveclasses that are mapped to the source and destination blocks.
	There are input parameters for both Msgsrvs that have the same type of port of connection on the block.
	The Reactiveclass assigned to the source block has a reactive class of destination block as Knownrebecs.
	
	`reactiveclass BlockA() { knownrebecs{ BlockB b; } statevars { int testInt; } msgsrv testMsgSrv(){ b.testMsgSrv(testInt); } }`
	
	`reactiveclass BlockB() { statevars { int testInt; } msgsrv testMsgSrv(int param){ testInt = param; } } main { BlockA a(b):(); BlockB b():(); }`

Figure 13: Map Connections to Rebeca code

the destination port) and has the same source block and destination, the connector will map according to this rule. This connection will map to a Msgsrv, which will call itself within its body.

As an example, Figure 14 shows a component with one port. The connector selfCall will map to the Msgsrv selfCall, and the self.selfCall will be added to its body as well.

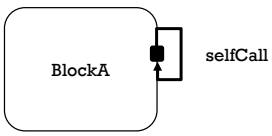
Internal Block Diagram in CHESS	Rebeca Modeling Language
Self connectors A self connector is a connection to the InOut port (the source port is as same as the destination port) and has the same source block and destination. 	Map to Msgsrv This connection map to a Msgsrv, which calls itself within its body. `msgsrv selfCall() { self.selfCall(); }`

Figure 14: Map Self connections to Rebeca code

5.4 Mapping the Safety Contracts to Rebeca Properties

The FPTC rules are mapped into safety contracts. The safety contracts have two elements: *assumptions* and *guarantees*. Assumptions are conditional statements over input ports (p statement) and guarantees are conditional statements over output ports (q statement). Here is what a safety contract looks like:

$$\begin{cases} \textbf{Assumption} : not(p) \\ \\ \textbf{Guarantee} : not(q) \end{cases}$$

A safety contract implicates that if the assumptions on the inputs of the component do not

hold, then the guarantees on the outputs of the component will not hold. A safety contract can be present as an implication:

$$not(p) \rightarrow not(q)$$

We present this implication as an assertion in the property file in Afra. We write the implication as a disjunction:

$$\neg p \rightarrow \neg q \equiv \neg(\neg p)|(\neg q)$$

The above disjunction is mapped to an assertion in Afra as:

$$\neg\ (\ \neg p\ \&\ q\ )$$

Each port may have a different type of failure. In the Rebeca model, the failure of a port is captured as specific values of certain variables. In step 4 of mapping the internal block diagram to the Rebeca model, we mapped each port to a Statevar (variables) of Rebeca. Mapped Statevar has the same type as the Port type. The Statevar value will check in a conditional statement. The created statement will add to a variable in the definition part Afra. The statement shows a port value and compares it with the expected value when it failed. After all ports of safety contract added to definition part of Afra. We can create the assertion. We used a logical approach to map "safety contracts" to assertions. "Safety contracts" can be mapped to "assertions" in a straightforward way. An assertion contains two parts; assertions and guarantees defined variables. For each assertion, in the last step, we defined a variable. The defined variable is negated and added to the assertion and logically and (&&) to what was previously added. Also, in the last step, we defined a variable for each guarantee. The defined variable will add to the assertion logically, and (&&) to what was added previously.

A safety contract for ComponentA would be:

$$\begin{cases} \textbf{Assumption} : not(InputA.comission) \\ \\ \textbf{Guarantee} : not(OutputA.valueCourse) \end{cases}$$

The assumption part of safety contact is $not(InputA.comission)$, this assumption is added to the definition part of the Afra as follows:

```
define {
    inputAValue = ComponentA.InputA;
}
```

The guarantee part of safety contact is $not(outputName.comission)$, this assumption is added to the definition part of the Afra as follows:

```
define {
    OutputAValue = ComponentA.OutputA;
}
```

The assertion of this safety contract is:

```
define {
    inputAValue = ComponentA.InputA;
    OutputAValue = ComponentA.OutputA;
}

Assertion{
    Assertion1: !(!inputAValue && OutputAValue )
}
```

Figure 15 shows an example of safety contract mapped to properties in Afra.

Safety Contracts	Properties in Afra
Every safety contract has two parts: assumptions and guarantees. Assumptions are statements that are made based on the input ports. guarantees are statements that are made based on the output ports. A: not (ComponentInputPortName.FailureType) G: not (ComponentOutputPortName.FailureType)	The safety contracts are turned into assertions and moved to the property file in Afra. The file has two parts: the definitions and the assertions. The clauses are written based on the value of a variable for that actor. These clauses are used in assertion part. `property { define{ // Boolean type inputValueFailed = component.ComponentInputPortName; // Int type outputValue = component.ComponentOutputPortName == 1; } Assertion{ Assertion1: !(inputValueFailed && outputValue); } }`

Figure 15: Mapping safety contracts to Afra properties example

5.5 Model Checking via Afra tool

In this step, we should try the assertions to ensure that the system will not fail, and we can ensure that some of the combinations of the assertions defined in the model are not true. As a result of the mapping, we generate a Rebeca code that represents the system. We must complete the logic of some $Msgsrv$ before we start the model-checking process. The behaviour of the system is extracted from the documents. There are some gaps in the logic used in some $Msgsrvs$. Before beginning the verification process, we add the $Msgsrv$ missing logic from the documents to be able to perform model checking.

After completing the model, Afro Tools tries to verify the model by finding a counter-example on one of its assertions that gets true. This counter-example tells us that this model cannot satisfy its properties. It is essential to remember that Rebeca does not fail the assertions unless it finds a state where an assertion is true. If none of the assertions become true, the model satisfies the properties. At the end of this step, we generate a model that satisfies all the properties, or we find assertions failed in this model.

6 Traffic Light Case Study

Traffic lights, traffic signals, or stop lights are signalling devices used to manage traffic flow at road junctions, pedestrian crossings, and other sites. Typically, traffic lights consist of three signals that convey information to cars and cyclists through colours and symbols, including arrows and bicycles. Red, yellow, and green are the standard traffic signal colours, set vertically or horizontally in that sequence. We performed all methodological procedures, as shown in Figure 6, for the traffic light case study.

System description Traffic lights are designed to manage traffic flow and make driving safer. Always approach the drivers slowly enough that the signal does not change before the drivers reach the intersection. The colour of the traffic light in front of drivers as they approach a junction dictates their course of action.

This case study consists of two traffic lights ($LightA$, $LightB$) and it is equipped with a controller that is responsible for adjusting traffic lights as needed. Each traffic signal operates in three unique modes: *Red*, *Yellow*, and *Green*. These statuses will not occur simultaneously at a single traffic signal. Traffic lights will alternate between red and green, green and yellow, and yellow and red (Figure 16).

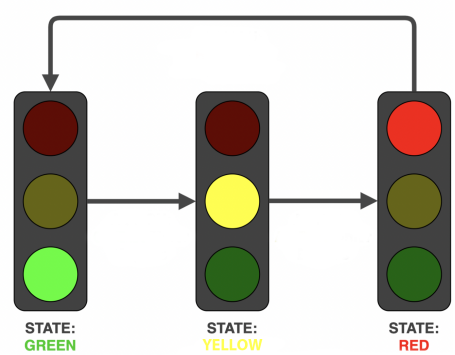


Figure 16: Traffic Lights states

The crossing system is consisted of two traffic lights that should not be in the following states concurrently and if they get the following values, an accident will occur at crossings:

- **Green, Green**: If both lights turn green, both vehicles on those routes will be able to proceed through the crossing, resulting in a collision.
- **Yellow, Yellow**: If both lights turn yellow, vehicles on both routes may pass cautiously, yet an accident will still occur.
- **Red, Red**: If both traffic lights turn red, all vehicles on both routes have to stop; thus, this status is undesirable since it would result in a deadlock. Here, the crossing is in a deadlock state, meaning no vehicle can move through.

These are the following safety-critical states of this system. If the lights get the colour in this list, it causes a hazard or harm.

- Green, Green
- Yellow, Yellow
- Green, Yellow

After 10 unit of time it will send the $startTrafficSignal$. This signal is responsible for starting running traffic lights, which includes changing lights by sending $goToNext$ signals. Each light supposed be active for 5 unit of time.

Controller A signal will be transmitted to the controller as a start signal from outside the system. When the controller receives the start signal, it sets the starting settings; this controller works with two traffic lights. The system also includes a *StartTraffic* signal as an input. This signal initiates the light sequence. When the *StartTraffic* signal engaged, the controller may deliver the following signal. The controller call go to next signal every five time cycles.

- **Inputs**:
  - **startSignal**: Is considered for initiating the system at initial state.
  - **startTrafficSignal**: Is considered for changing the lights' cycle.
  - **changeSignal**: The controller starts to send GoToNext signals (goToNextA and goToNextB) every 5 units of time when this signal become active. This signal causes the lights to change every 5 units of time.
- **Outputs**:
  - **goToNextA, goToNextB**: These signals can change the sequence of lights' function.
  - **initValueA, initValueB**: These outputs set initial value of lights A and B.
  - **changeSignal**: Is considered for activating the self-changing light feature to transmit signals for changing lighting.

Traffic Light The light changes its colour in response to the input signal. It will have an output that shows the colour of the traffic light at the junction where it is positioned. Red stands for the first level of light. It will be set up for the first time when a signal comes in. This signal has the values 3, 4, and 5 as its values. (Red, Yellow, Green) Right away, the lights will be changed.

Conduct the Safety Analysis with ConcertoFLA Technique This system consists of two components (light and controller components). We considered two light components and one controller component in this case study.

Model the Component Architecture in Internal Block Diagram The traffic controller document provides the specifications of the system, which are then modelled in CHESS, and specifically in the internal block diagrams. A system's internal block diagram has been shown inside the CHESS tools.

Controller Block Diagram Figure 17 depicts an internal block diagram of a traffic light controller. There are three inputs and five outputs on this controller. One of the inputs also serves as an output. *startSignal* is considered as one of the input signals. The *startSignal* is responsible for the system's startup settings. The second input is the system's *startTrafficSignal* for turning on the traffic light system. The *ChangeSignal* is the last input. The controller will alter the traffic lights whenever it receives a signal from *ChangeSignal* input.

Light Block Diagram Figure 18 It shows a block diagram of the internal parts of the traffic system. This component has two inputs and one output. When the *GoToNext* signal is true, the red light turns yellow. If the *GoToNext* signal is true again, the yellow light turns green. *statusLight* is what comes out, and it shows what colour of the light is. The initial value, which is the colour of the lights when they first turn on, comes from another input of component. The initial value can be 3, 4, or 5;

Crossing Internal Block Diagram An internal block diagram of the Crossing represents the system's internal components. Its inputs are two traffic signals and a controller. The system output consists of the lighting status as well as the initValue and start signal. Figure 19 illustrates how each component is connected.

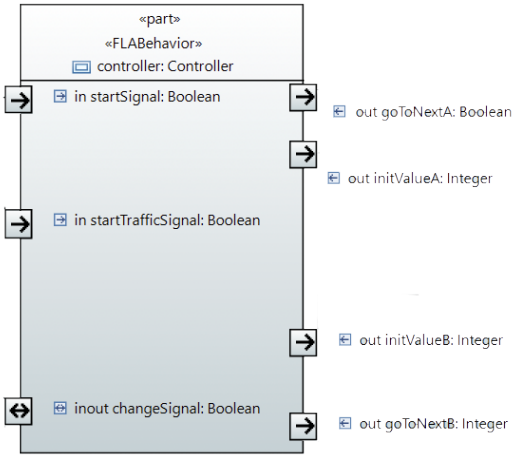


Figure 17: Internal block diagram of Controller component

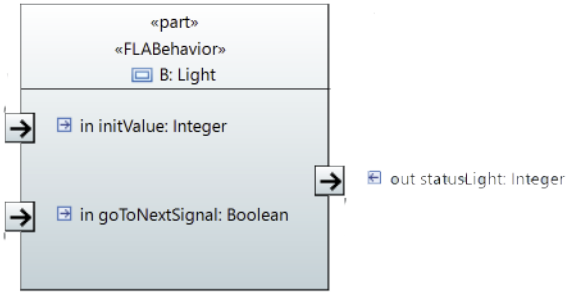


Figure 18: Internal block diagram of Traffic light component

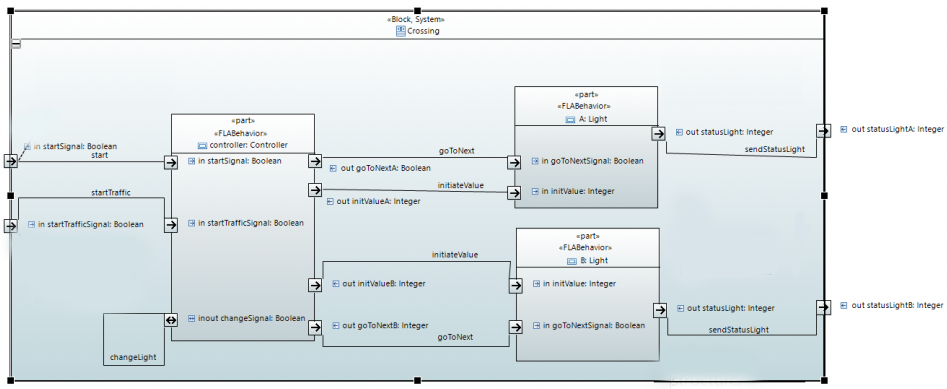


Figure 19: Internal block diagram of Traffic light system

Model the Component Architecture in Sequence Diagram In the previous steps, we created all actors and their message servers. Unfortunately, the internal block diagram does not indicate how these message servers will be called. We have therefore created a sequence diagram to indicate the order in which the message servers will be called.

Inside the sequence diagram, it also illustrates the conditional statement of the traffic light. It shows that when a *Controller* sends *goToNext* to the lights, a conditional logic is executed inside the message server of *goToNext*. The sequence diagram in Figure 20 illustrates how this occurs.

FPTC Rules for Each Component In order to compose the FPTC rules, we first need to understand the rationale behind them.

Controller FPTC rules The controller in this system is subject to the following failure scenarios:

- The controller transmits the initial value to the lights several times if the system developer sets the configuration multiple times. Therefore, the startSignal is active when it is not supposed to be and the commission faults typically happen. As a result, the outputs become active when they are not supposed to be. There is a commission caused by initlValueA and initlValueB activation at the wrong time (Code 1 rule 1).
- startSignal follows the same pattern. startSignal has failure of Commission, and it means supposed to when they are not intended to. This situation is shown in Code 1 rule 2.

```
Rule1: startSignal.commission, startTrafficSignal.noFailure ->
goToNextA.noFailure, goToNextB.noFailure,
initValueA.commission, initValueB.commission;

Rule2: startSignal.noFailure, startTrafficSignal.commission ->
goToNextA.commission, goToNextB.commission,
initValueA.noFailure, initValueB.noFailure;
```

Code 1: Controller FPTC rules

Light FPTC rules Light FPTC rules for the following scenario are presented in Code2.

- The traffic light component gets the *goToNEXTSignal* signal when it is not supposed to get. This indicates a failure on an input (a commission). The traffic light component has the incorrect colour at the wrong time, so it shows an incorrect colour to users/drivers. This result is undetectable, indicating that its value is *ValueSubtle* (Code 2 Rule1).
- Similar to the last item, the *initValue* input becomes active unintentionally and causes the output of the traffic light instruction. The output becomes incorrect from the beginning (*ValueCourse* failure). Code 2 Rule 2 shows the Light FPTC rules for *ValueCourse* failure.

```
Rule1: goToNextSignal.commission, initValue.noFailure -> statusLight.valueSubtle;
Rule2: goToNextSignal.noFailure, initValue.commission -> statusLight.valueCoarse;
```

Code 2: Light FPTC rules

6.1 Translate the FPTC Rules into Corresponding Safety Contracts

It is intended that all extracted FPTC rules be transferred to safety contracts for each component. First FPTC rule defined for Controller, Code 1.Rule 1. Code 3 represents the safety contract created from the FPTC rule Code 1.

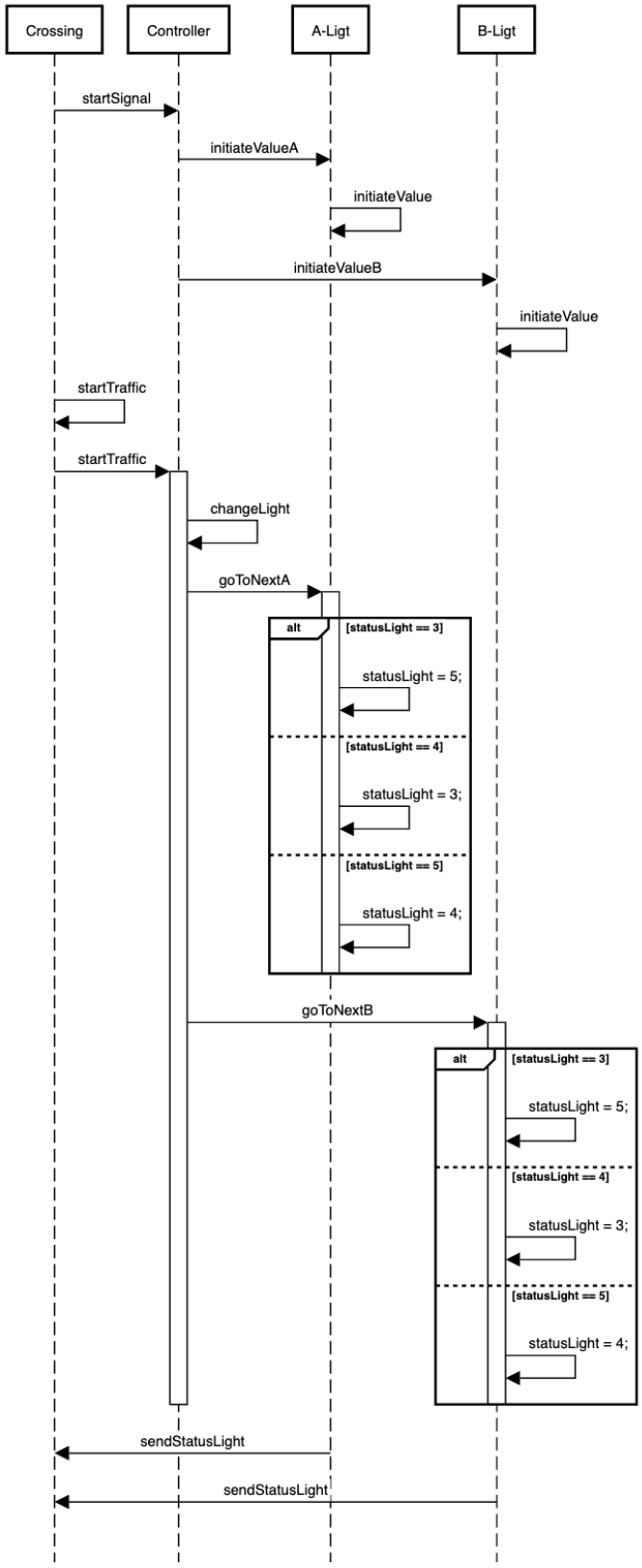


Figure 20: Sequence diagram of Traffic light system

```
-
1:
1.A: not (startSignal.commission)
1.G: not ( {initValueA , initValueB}.commission)
2:
2.A: not (startTrafficSignal.commission )
2.G: not ({ goToNextA , goToNextB}.commission)
-
```

Code 3: Generated safety contracts for Controller component

The same approach from Section 2.1.3 will apply to creating safety contracts from the FPTC rule of Light component and the safety contract presented in Code 4.

```
-
1:
1.A: not (goToNextSignal.commission)
1.G: not ( {statusLight}.valueSubtle)
2:
2.A: not (initValue.commission )
2.G: not ({ statusLight }.valueCoarse)
-
```

Code 4: Light FPTC rules

To finalize the safety contracts for the whole system, four FPTC rules are used to generate four safety contracts.

6.2 Mapping Internal Block Diagram to Rebeca

All the information was added to CHESS-FLA using the proposed mapper and the Rebeca code was generated.

- First, we apply the Mapper Rule 1. Code 5.

```
main {
}
```

Code 5: Start point of mapper first step

- We use the Mapper Rule 2 for all of the blocks in the system. We have three components, and we map them to three reactive classes (one Controller and two lights). The system consists of two traffic lights and the Mapper Rule 2 map them to two reactive classes (Code 6).

```
reactiveclass Light(){
}

reactiveclass Controller(){
}

reactiveclass Crossing(){
}

main {
  Light A():();
  Light B():();
  Controller controller():();
  Crossing crossing():();
}
```

Code 6: Rebeca code of all blocks rule 2

- We have one system IBD in this case study (Crossing). When we map it to a *reactiveclass*, the system blocks will have a constructor and a *msgsrv* as it is shown in Code 7

```
reactiveclass Crossing(){
    Crossing() {
        self.start();
    }

    msgsrv start() {

    }
}
main {
    Crossing crossing():();
}
```

Code 7: Crossing System block Rebeca code

- Rule 4 utilizes the available ports for each block and transmits them to a Rebeca code. *InitValue* and *goToNextSignal* are the inputs for the Light components with Integer and Boolean types, respectively. *StatusLight* is the output of the component with an Integer type. The ports in the Light component are mapped to an *Statevar* in the component's *ReactiveClass*. *StartSignal*, *startTrafficSignal*, and *changeSignal* are the inputs for the Light component with a *Boolean* type. The Light component also has four outputs. The outputs are *GoToNextA*, *GoToNextB*, and *ChangeSignal* with *Bwoolean* type, and *initValueA* and *initValueB* with Integer type (Code 8).

```
reactiveclass Light() {

  statevars {
    boolean   goToNextSignal;
    int       initValue;
    int       statusLight;
  }
}

reactiveclass Controller() {

  statevars {
    boolean   startSignal;
    boolean   startTrafficSignal;
    int       initValueA;
    int       initValueB;
    boolean   goToNextA;
    boolean   goToNextB;
  }
}
```

Code 8: Rebeca code for the Light and Controller components after applying the Rule 4.

- The connections are mapped to the Rebeca code. The goToNextA and the goToNextB controller from the Controller component have connections. The initValueA and the initValueB controllers also have connections. The connections are mapped to a Msgsrv Rebeca code in the source and destination blocks by Mapper Rule 5. When we have the Controller as a source, then the LightA and LightB are added as known objects of the Controller component. We also include the names of the instances in the Rebeca main section, as it is shown in Code 9.

```
reactiveclass Light(3) {

  knownrebecs {
    Crossing crossing;
  }

  statevars {
    boolean   goToNextSignal;
    int       initValue;
    int       statusLight;
```

```
    }

    Light() {

    }

    msgsrv goToNext(boolean tempGoToNext) {
      goToNextSignal = tempGoToNext;
    }

    msgsrv initiateValue(int value) {
      initValue = value;
    }

    msgsrv sendStatusLight() {
      crossing.sendStatusLight(id, statusLight);
    }
  }

  reactiveclass Controller(6) {

    knownrebecs {
      Light lightA;
      Light lightB;
    }

    statevars {
      boolean   startSignal;
      boolean   startTrafficSignal;
      int      initValueA;
      int      initValueB;
      boolean   goToNextA;
      boolean   goToNextB;
    }

    Controller() {
    }

    msgsrv start(boolean tempStartSignal) {
      startSignal = tempStartSignal;
    }

    msgsrv startTraffic(boolean tempStartTrafficSignal) {
      startTrafficSignal = tempStartTrafficSignal;
    }

    msgsrv goToNextA() {
      lightA.goToNext(true);
    }

    msgsrv goToNextB() {
      lightB.goToNext(true);
    }

    msgsrv initiateValueA(int value) {
      initValueA = value;
      lightA.initiateValue(value);
    }

    msgsrv initiateValueB(int value) {
      initValueB = value;
      lightB.initiateValue(value);
    }

    msgsrv changeLight() {
      self.changeLight() after(5);
    }
  }

  main {
```

```
    Light A(crossing):();
    Light B(crossing):();

    Controller controller(A,B):();

    Crossing crossing(controller):();
  }
```

Code 9: Rebeca code for the Light and Controller components when their connectors mapped

The final Rebeca code The codes generated by the mapper are combined. We apply logic to *Msgsrvs* of the blocks. Additionally, the system must be in an operating mode in order to verify the model, so we included start and will toggle start. This model represents information which may be obtained from the traffic light component (Code 10).

```
reactiveclass Light(3) {

  knownrebecs {
    Crossing crossing;
  }

  statevars {
    boolean   goToNextSignal;
    int      initValue;
    int      statusLight;
    int      id;
  }

  Light() {

  }

  msgsrv goToNext(boolean tempGoToNext){
    goToNextSignal = tempGoToNext;

    if(statusLight == 3){    // Red
      statusLight = 5;     // Green

    }else if(statusLight == 4){ //
        Yellow
      statusLight = 3;     // Red

    }else if(statusLight == 5){ // Green
      statusLight = 4;     // Yellow

    }
  }

  msgsrv initiateValue(int value) {
    initValue = value;
    id = value;
  }

  msgsrv sendStatusLight() {
    crossing.sendStatusLight(id,
                    statusLight);
  }
}

reactiveclass Controller(6) {

  knownrebecs {
    Light lightA;
    Light lightB;
  }

  statevars {
    boolean   startSignal;
    boolean   startTrafficSignal;
    int      initValueA;
    int      initValueB;
    boolean   goToNextA;
    boolean   goToNextB;
  }


  msgsrv start(boolean tempStartSignal)
      {
    startSignal = tempStartSignal;
    self.initiateValueA(3);
    self.initiateValueB(5);
  }

  msgsrv startTraffic(
          boolean tempStartTrafficSignal
      )
  {
    startTrafficSignal =
    tempStartTrafficSignal;
    self.changeLight();
  }

  msgsrv goToNextA() {
    lightA.goToNext(true);
  }

  msgsrv goToNextB() {
    lightB.goToNext(true);
  }

  msgsrv initiateValueA(int value) {
    lightA.initiateValue(value);
  }

  msgsrv initiateValueB(int value) {
    lightB.initiateValue(value);
  }

  msgsrv changeLight() {
    self.goToNextA();
    self.goToNextB();
    self.changeLight() after(5);
  }
}

reactiveclass Crossing(3){

  knownrebecs {
    Controller controller;
  }
```

```
statevars {
  boolean startSignal;
  boolean startTrafficSignal;
  int statusLightA;
  int statusLightB;
}

Crossing() {
  self.start() after(5);
  self.startTraffic() after(10);
}

msgsrv start() {
  controller.start(true);
}

msgsrv startTraffic() {
  controller.startTraffic(true);
}
```

```
  msgsrv sendStatusLight(
  int id, int value) {
    if( id == 0 ){
      statusLightA = value;
    }else{
      statusLightB = value;
    }
  }
}

main {
  Light A(crossing):();
  Light B(crossing):();
  Controller controller(A,B):();
  Crossing crossing(controller):();
}
```

Code 10: Traffic light Rebeca model

6.3 Mapping the Safety Contracts to Rebeca Properties

All variables of Rebeca corresponding to the ports in internal block diagrams are used in the Property file in Afra.

$$A : not(startSignal.commission)$$

$$A : not(startTrafficSignal.commission)$$

StartSignal and *StartTrafficSignal* are the inputs that make up the safety contrasts. Both of them have a Boolean type. It is straightforward to define the statement mapping for two: monitor their value and store it in a new variable to use in assertions.

Generated Rebeca code of Controller assumptions shown in Code 11

```
// Controller assumptions
  startSignal = controller.startSignal;
  startTrafficSignal = controller.startTrafficSignal;
//
```

Code 11: Definitions in the Property file of Afra for Controller

It is also true for the light component, which has two inputs, but one of them has the type of integer. The value of that input is compared with the component's normal behaviour to determine its expected value, then it is compared with the expected value and saved to a variable to be checked in the assertion process.

$$A : not(goToNextSignal.commission)$$

$$A : not(initValue.commission)$$

Generated Rebeca code of Light assumptions shown in Code 12. This actor has two instances we should check both of them.

```
// Light assumptions
// Green
AisGreen = A.initValue == 5;
BisGreen = B.initValue == 5;
// Yellow
AisYellow = A.initValue == 4;
BisYellow = B.initValue == 4;
// Red
AisRed = A.initValue == 3;
BisRed = B.initValue == 3;

AgoToNextSignal = A.goToNextSignal;
BgoToNextSignal = B.goToNextSignal;
//
```

Code 12: Definitions in the Property file of Afra for Light

Two guarantee of Controller Component were added to properties file in Afra.

$$G : not(initValueA, initValueB.commission)$$

$$G : not(goToNextA, goToNextB.commission)$$

```
// Controller guarantee
// Green
isGreenA = controller.initValueA == 5;
isGreenB = controller.initValueB == 5;
// Yellow
isYellowA = controller.initValueA == 4;
isYellowB = controller.initValueB == 4;
// Red
isRedA = controller.initValueA == 3;
isRedB = controller.initValueB == 3;

goToNextSignalA = controller.goToNextA;
goToNextSignalB = controller.goToNextB;
//
```

Code 13: Definitions in the Property file of Afra for Controller

Two guarantee of Light Component were added to properties file in Afra.

$$G : not(statusLight.valueSubtle)$$

$$G : not(statusLight.valueCoarse)$$

```
// Light guarantee
// Green
AisGreenFinal = A.statusLight == 5;
BisGreenFinal = B.statusLight == 5;
// Yellow
AisYellowFinal = A.statusLight == 4;
BisYellowFinal = B.statusLight == 4;
// Red
AisRedFinal = A.statusLight == 3;
BisRedFinal = B.statusLight == 3;
//
```

Code 14: Definitions in the Property file of Afra for Light

Safety properties The system cannot contain five states (Green, Green), (Yellow, Yellow), (Red, Red), (Green, Yellow), (Yellow, Green), and (Yellow, Green) (the first element displays the value of Light A, and the second element displays the value of Light B). Five safety properties from system specification documents indicate that a crossing system accident is likely to occur. We then extract the safety properties below directly from the specifications of the system. These are all safety properties extracted from safety contracts and safety properties from safety documents.

```
property {

  define {
    // Green
    AisGreen = A.initValue == 5;
    BisGreen = B.initValue == 5;
    // Yellow
    AisYellow = A.initValue == 4;
    BisYellow = B.initValue == 4;
    // Red
    AisRed = A.initValue == 3;
    BisRed = B.initValue == 3;

    AgoToNextSignal = A.goToNextSignal;
    BgoToNextSignal = B.goToNextSignal;

    AisGreenFinal = A.statusLight == 5;
```

```
    BisGreenFinal = B.statusLight == 5;
    // Yellow
    AisYellowFinal = A.statusLight == 4;
    BisYellowFinal = B.statusLight == 4;
    // Red
    AisRedFinal = A.statusLight == 3;
    BisRedFinal = B.statusLight == 3;

    startSignal = controller.startSignal;
    startTrafficSignal = controller.startTrafficSignal;

    isGreenA = controller.initValueA == 5;
    isGreenB = controller.initValueB == 5;
    // Yellow
    isYellowA = controller.initValueA == 4;
    isYellowB = controller.initValueB == 4;
    // Red
    isRedA = controller.initValueA == 3;
    isRedB = controller.initValueB == 3;

    goToNextSignalA = controller.goToNextA;
    goToNextSignalB = controller.goToNextB;

  }

  Assertion {
    Assertion1: !(AisGreen && BisGreen );
    Assertion2: !(AisYellow && BisYellow );
    Assertion3: !(AisRed && BisRed );
    Assertion4: !(AisGreen && BisYellow );
    Assertion5: !(AisYellow && BisGreen );


    Assertion6: !(!startSignal && AisGreen);
    Assertion7: !(!startTrafficSignal && AisGreen);
    Assertion8: !(!AgoToNextSignal && AisGreenFinal);
    Assertion9: !(!BgoToNextSignal && BisGreenFinal);
  }
}
```

Code 15: Safety properties to be check in Rebeca

6.4 Model Checking via Afra tool

In this step, we run a model checker. Afra needs two files to be able to verify. We created the Rebeca model in the section 6.2 and the properties file in the Afra section 6.3. After having these two files, we used Afra to run model checking. Afra looks for a counter example to check if any assertions failed. As shown in the Figure 21, all assertions are satisfied in this example.

Attribute	Value
▾ SystemInfo	
Total Spent Time	0
Number of Reached States	44
Number of Reached Transitions	61
Consumed Memory	704
▾ CheckedProperty	
Property Name	Deadlock-Freedom and No Deadlin...
Property Type	Reachability
Analysis Result	satisfied

Figure 21: Traffic light: Afra report after model checking.

Discussion This example demonstrates how we use CHESSML models to create Rebeca models, and how we use ConcertoFLA results to create safety contracts, and then extract safety properties

for model checking from safety contracts. Based on four safety contracts, we created four assertions. For example, the safety contract states that we would not have output failures if the controller's signals trigger when they are not supposed to.

We model Commission failures on the controller's input. The correct order is *startSignal* and then *startTrafficSignal*. We trigger *startSignal* after *startTrafficSignal* again. Model checking shows that this creates faulty outputs. In ConcertoFLA technique failures are characterized qualitatively. However, formal verification enables us to identify the failures based on the specific value. We found that if the *startSignal* is enabled for the second time before 25 units of time (the time needed for *startTrafficSignal* to updates the values of the light), the controller outputs become faulty. Afra generated a state space of the model, which is shown in Figure 22. According to the state space, we need 25 units of time in order to start the system and the lights. When we call *startSignal* earlier than 25 units of time twice, the incorrect state space is created, and the assertion below fails-this assertion is derived from the safety contracts.

```
//
!( startSignal &&  startTrafficSignal  && goToNextA && initValueA )
//
```

Code 16: Assertion of Controller

The assertions we added in Afra help us to formally verify any new update in the model against existing safety contracts.

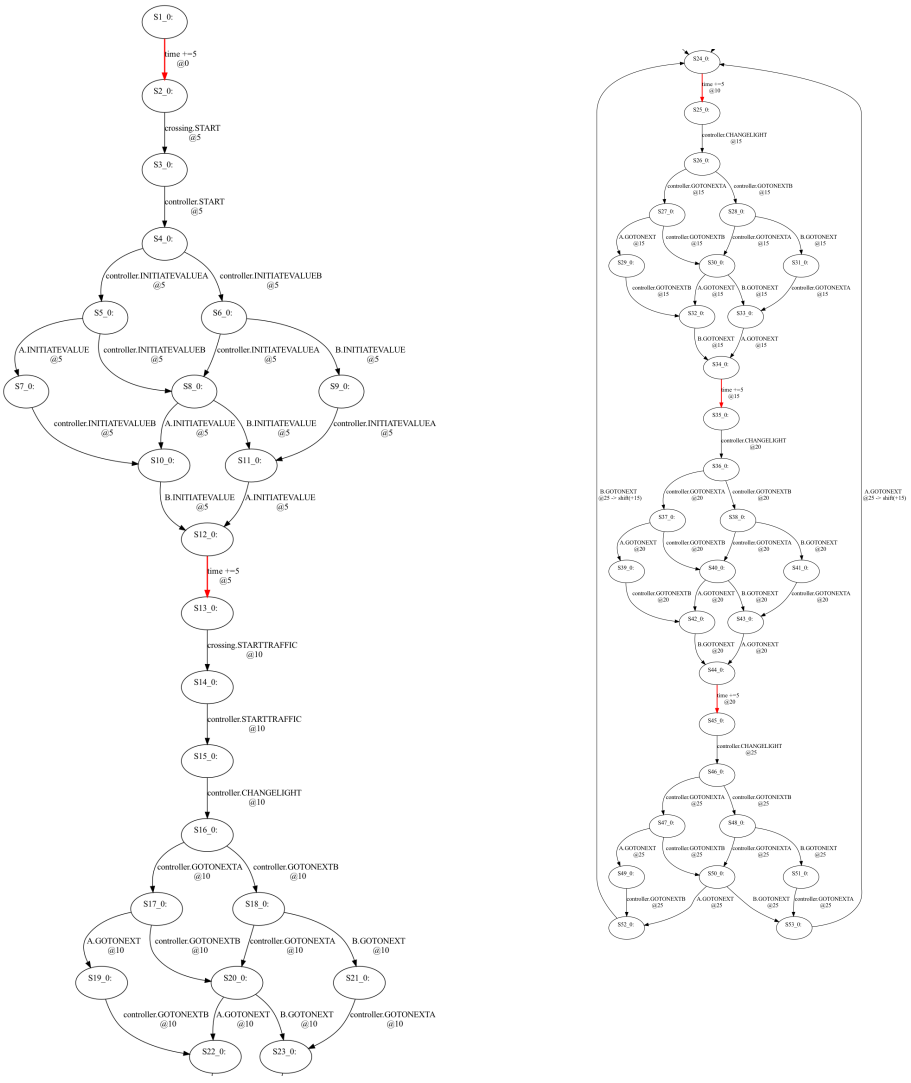


Figure 22: Traffic light: State space.

7 Train Door Controller Case Study

There are doors on all trains that allow passengers to enter and exit. A train's door must be locked before it can move. The system which controls the train's door is capable of locking the door and then instructing it to begin moving. It is not possible for the system to guarantee that the door will never become unlocked while the train is in move. This is because we do not know when the door gets locked and the train starts moving as a result of software commands [36].

System specification The primary architectural components are Input-Output (IO) units, the central Train Control Unit (TCU), and the Door Control Unit (DCU). IO units serve as system interfaces and are designed to receive/transmit input/output signals. The IO unit on the passenger side is responsible for reading the door push buttons in order to accept the passenger's open request. When a passenger presses the "open" button, the IO unit receives and transmits the open request to the DCU. The driver's orders for open, close, lock, and unlock are sent via the TCU to the DCU. The DCU is responsible for executing the correct orders to alter the door's status. TCU is responsible for central control management. TCU may be dispersed and executed on distinct physical devices. For instance, one physical control device for non-safety-related operations and another device for safety-critical tasks. DCU may be a programmable device that receives the command signal from TCU and transmits it to the matching door actuator converters. Typically, data transmission between physical devices is facilitated by a system-wide bus and a secure communication protocol. Later in our behavioural models, we represent both DCU and the corresponding IO on the passenger side as "Door" actors, as well as TCU and the driver as "Controller" components. "Train" represents a series of IO units receiving status from sensors and other methods that are used to alert the TCU and the driver that the train has arrived at the station and is prepared to depart, which is crucial for our case study. These are the states in which the TCU must alter the door status.

Conduct the Safety Analysis with ConcertoFLA Technique This system consists of four components (Door, Passenger, Train and Controller components). We fallow the approach used in thesis [32].

Model the Component Architecture in Internal Block Diagram The Internal Block diagrams have been created inside the CHESS tools. This system is composed of four blocks. The system block has the same name as the system. This system has passengers, doors, controllers, and train components. Each component has different block diagrams.

Passenger Block Diagram Based on the Internal block diagram shown in Figure 23, Passenger component has one output. It is the output that is responsible for requesting the door to be opened. The output name is *openSignal*, and it has the type of *Boolean*.

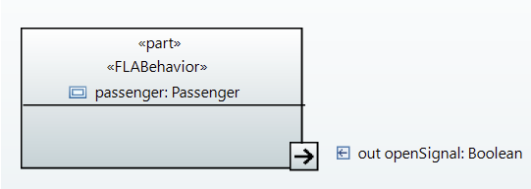


Figure 23: Passenger block diagram

Door Block Diagram The door component gets an open request from the passenger and transfer it to the controller. The door component responds to requests from the controller by locking or unlocking, opening or closing itself. A "Lock door" means no one can send an open request until someone unlocks it (Door IBD, shown in Figure 24).

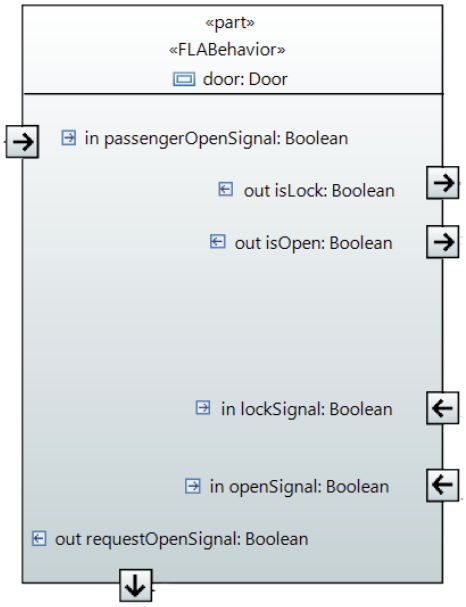


Figure 24: Door block diagram

Controller Block Diagram As shown in Figure 25, the controller block diagram has five inputs and two outputs. Aside from the three inputs from the door controller, two additional inputs are received from train components. This component has been assigned a *Boolean* type for all ports. There are two outputs from this system, both of which are directed to the door component with the same types of inputs.

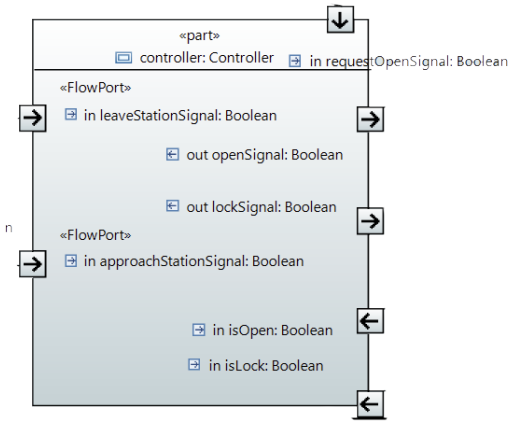


Figure 25: Controller block diagram

Train Block Diagram The train component consists of two outputs, each of which has a Boolean type output. Additionally, this component has a port which has a type of InOut, and with a Boolean type (Train IBD shown in Figure 26).

Model the Component Architecture in Sequence Diagram In the previous steps, we created all actors and their message servers. Unfortunately, the internal block diagram does not indicate how these message servers will be called. We have therefore created a sequence diagram to indicate the order in which the message servers will be called. In Figure 27 we created the Sequence diagram of the system to model the behaviour of the Train.

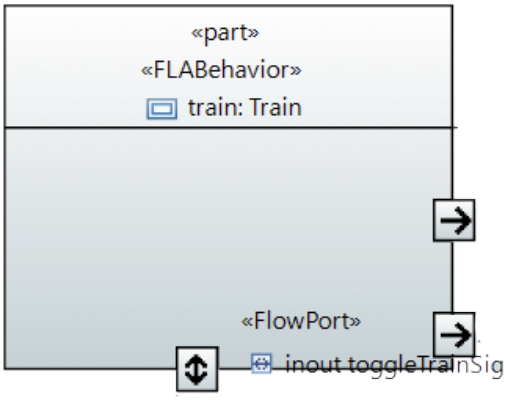


Figure 26: Train block diagram

FPTC Rules for Each Component In order to compose the FPTC rules, we first need to understand the rationale behind them.

- Assume that the train approaches or leaves the station at an incorrect time, resulting in omissions that cause incorrect output behaviour. The Code 17 contains FPTC rules for the Controller component.

```
—
Rule1: requestOpenSignal.noFailure, isOpen.noFailure,isLock.noFailure,
    leaveStationSignal.omission, approachStationSignal.noFailure ->
    openSignal.noFailure,lockSignal.omission;

Rule2: requestOpenSignal.noFailure, isOpen.noFailure,isLock.noFailure,
    leaveStationSignal.noFailure, approachStationSignal.omission ->
    openSignal.omission,lockSignal.noFailure;
—
```

Code 17: FPTC rules of Controller

- There is a pair of faulty behaviours in the door component. In the event that either of these behaviours occurs when they should not, it will affect outputs. The Code18 contains FPTC rules of component.

```
—
Rule1: passengerOpenSignal.noFailure,lockSignal.omission,openSignal.
    noFailure,requestOpenSignal.noFailure ->isOpen.noFailure,isLock.
    omission;
Rule2: passengerOpenSignal.noFailure,lockSignal.noFailure,openSignal.
    omission,requestOpenSignal.noFailure ->isOpen.omission,isLock.
    noFailure;
—
```

Code 18: FPTC rules of Door

7.1 Translate the FPTC rules into corresponding safety contracts

It is intended that all extracted FPTC rules be transferred to safety contracts for each component. The first FPTC rule applies to Controller, Code 17.Rule 1, and the input has Omission failure. According to Section 2.1.3, the first safety contract created base on this input. It is possible to guarantee that our output omission will not be excessive if we ensure that we do not have failure on input. The safety contract for this component shown in Code 19. Code 19 represents the next safety contract created from the FPTC rule Code 17.Rule 2.

```
—
1:
```

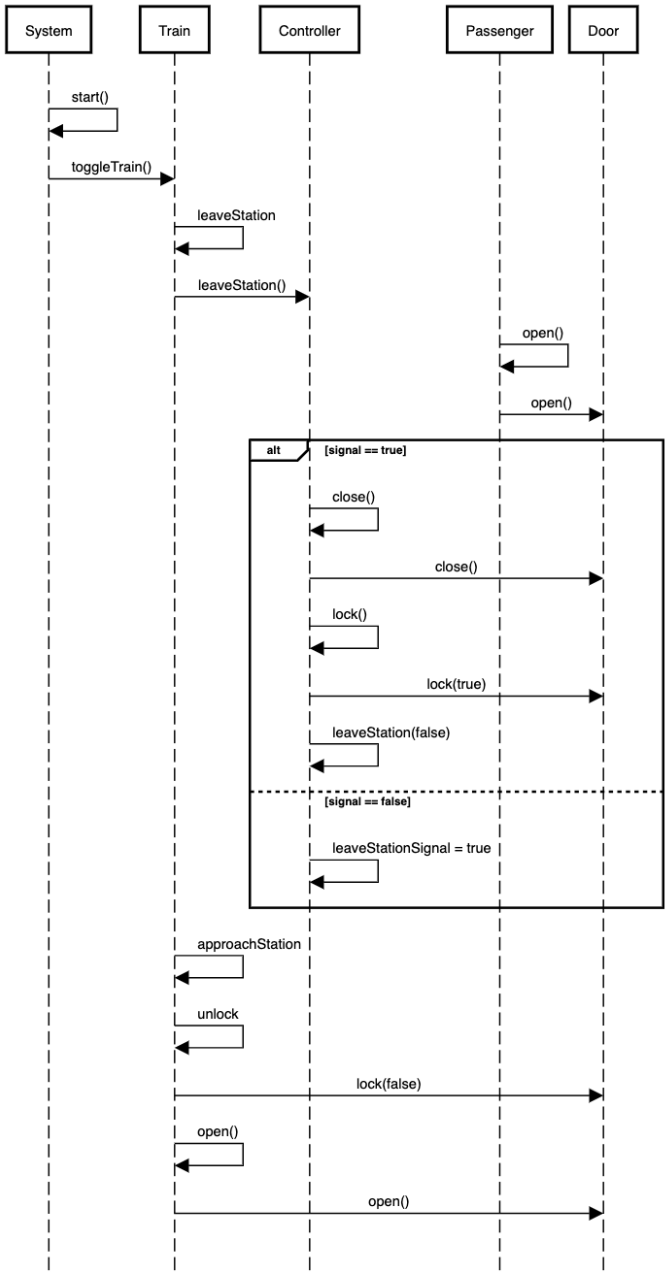


Figure 27: Sequence diagram of Traffic light system

```
1.A: not (leaveStationSignal.omission)
1.G: not (lockSignal.omission)
2:
2.A: not (approachStationSignal.omission)
2.G: not (openSignal.omission)
-
```

Code 19: Generated safety contracts for Controller component

The same approach from Section 2.1.3 will apply to creating safety contracts from the FPTC rule of Door component and the safety contract written Code 20.

```
-
1:
1.A: not (lockSignal.omission)
1.G: not (isLock.omission)
2:
2.A: not (openSignal.omission )
2.G: not (isOpen.omission)
-
```

Code 20: Door component safety contracts

7.2 Mapping Internal Block Diagram to Rebeca

All the information was added to CHESS-FLA using the proposed mapper and the Rebeca code was generated.

- In the first stage, we implemented Mapper Rule 1. Code 21.

```
main {
}
```

Code 21: Start point of mapper first step

- Using Mapper Rule 2, all blocks in this system are migrated to Rebeca. This system includes four components. Three new reactive classes and one controller instance were added to the code. Also, an instance of them added to *main* part.

```
reactiveclass Passenger() {
}

reactiveclass Train() {
}

reactiveclass Door() {
}

reactiveclass Controller() {
}

main {
  Passenger passenger():();

  Train train():();

  Door door():();

  Controller controller():();
}
```

Code 22: Rebeca code of all blocks rule 2

- There is only one system of IBD in this case study, which is System. However, when we map it to *reactiveclass*, system blocks will have a constructor and a *msgsrv* will be executed within them. As it shows in Code 23.

```
reactiveclass System(){
    System() {
        self.start();
    }

    msgsrv start() {

    }
}
main {
    System system():();
}
```

Code 23: System System block Rebeca code

- This step involves mapping all ports to the Rebeca Code. These are ports for each component.
  - The Passenger component has an output signal of *openSignal* with the *Boolean* type; the mapped Rebeca code is an *statevar*.
  - The train component also has two output ports, which are both *boolean* types. *leaveStationSignal* and *approchStationSignal* are the outputs that go to Controller. Also, this component has a *InOut* port which has the type of *boolean* and named *toggledTrainSignal*.
  - The Controller component contains five inputs and two outputs. Aside from the three inputs from the door controller, two additional inputs are received from train components. All ports in this component have been assigned a *boolean* type. There are two outputs from this system, both of which are directed to the door component with the same types of inputs; all these ports are converted to *statevar* in Rebeca, The input ports name are *requestOpenSignal*, *leaveStationSignal*, *approchStationSignal*, *isOpen* and *isLock* and the outputs are *openSignal* and *lockSignal*.
  - The door component has three inputs and three outputs which all have the same type of *Boolean*. The inputs are *passengerOpenSignal*, *lockSignal*, *openSignal*, and the outputs are *requestOpenSignal*, *isLock*, and *isOpen* with the type of *Boolean*.

```
reactiveclass Passenger() {
  statevars {
    boolean openSignal;
  }
}

reactiveclass Train() {
  statevars {
    boolean leaveStationSignal;
    boolean approachStationSignal;
  }
}

reactiveclass Door() {
  statevars {
    boolean passengerOpenSignal;
    boolean lockSignal;
    boolean openSignal;
    boolean requestOpenSignal;

    boolean isOpen;
    boolean isLock;
  }

}

reactiveclass Controller() {
  statevars {
    boolean lockSignal;
    boolean openSignal;
```

```
    boolean isOpen;
    boolean isLock;
    boolean leaveStationSignal;
    boolean approachStationSignal;
  }
}
```

Code 24: Ports mapped to Rebeca model

- This time, we will map each connection to Rebeca. In this system, there are multiple connections. We used Rule 4 of mapper and mapped all of them to Rebeca, and you can find your generated Msgsrv in Code.

```
reactiveclass Passenger(2) {
  knownrebecs {
    Door door;
  }

  statevars {
    boolean openSignal;
  }

  Passenger() {
    self.open();
  }

  msgsrv open() {
    door.open();
  }
}

reactiveclass Train(4) {
  knownrebecs {
    Controller controller;
  }

  statevars {
    boolean leaveStationSignal;
    boolean approachStationSignal;
  }

  Train() {
    self.toggleTrain();
  }

  msgsrv leaveStation() {
    leaveStationSignal = true;
    controller.leaveStation(true);
  }

  msgsrv approachStation() {
    approachStationSignal = true;
    controller.approachStation(true);
  }

  msgsrv toggleTrain() {
    self.leaveStation();
    self.approachStation() after (DELAY);
    self.toggleTrain() after(15);
  }
}

reactiveclass Door(8) {
  knownrebecs {
    Controller controller;
    System system;
  }

  statevars {
```

```
    boolean passengerOpenSignal;
    boolean lockSignal;
    boolean openSignal;
    boolean requestOpenSignal;

    boolean isOpen;
    boolean isLock;
  }

  msgsrv open() {
    openSignal = true;
    isOpen = true;
    requestOpenSignal = true;
    controller.requestOpen() after (DELAY);
  }


  msgsrv lock(boolean signal) {
    lockSignal = signal;
    isLock = signal;
  }

  msgsrv feedbackOpen() {
    system.feedbackOpen(isOpen);
    controller.feedbackOpen(isOpen);
  }

  msgsrv feedbackLock() {
    system.feedbackLock(isLock);
    controller.feedbackLock(isLock);
  }
}

reactiveclass Controller(9) {
  knownrebecs {
    Door door;
  }

  statevars {
    boolean lockSignal;
    boolean openSignal;
    boolean isOpen;
    boolean isLock;
    boolean leaveStationSignal;
    boolean approachStationSignal;
  }

  Controller() {
    isLock = false;
    isOpen = false;
    //leaveStationSignal = true;
  }

  msgsrv leaveStation(boolean signal) {
    if(signal){
      self.close();
      self.lock();
      self.leaveStation(false);
    }else {
      leaveStationSignal = true;
    }
  }

  msgsrv approachStation(boolean signal) {
    approachStationSignal = signal;
    self.unlock();
    self.open();
  }

  msgsrv open() {
```

```
        door.open(true);
        isOpen = true;
      }

      msgsrv close() {
        door.open(false);
        isOpen = false;
      }

      msgsrv lock() {
        door.lock(true);
        isLock = true;
      }

      msgsrv unlock() {
        door.lock(false);
        isLock = false;
      }

      msgsrv feedbackOpen(boolean value){
        isOpen = value;
      }

      msgsrv feedbackLock(boolean value){
        isLock = value;
      }

      msgsrv requestOpen() {

      }
    }
```

Code 25: Map connections to Rebeca model

Final Rebeca code By combining all the codes generated by the mapper with additional ConcertoFLA data, we will also provide additional functional logic to certain blocks (/(Msgsrvs/) as internal block diagrams are lacking information. .

```
env byte DELAY = 10;

reactiveclass Passenger(2) {
  knownrebecs {
    Door door;
  }

  statevars {
    boolean openSignal;
  }

  Passenger() {
    self.open();
  }

  msgsrv open() {
    door.open();
  }
}

reactiveclass Train(4) {
  knownrebecs {
    Controller controller;
  }

  statevars {
    boolean leaveStationSignal;
    boolean approachStationSignal;
  }

  Train() {
    self.toggleTrain();
  }

  msgsrv leaveStation() {
    leaveStationSignal = true;
    controller.leaveStation(true);
  }

  msgsrv approachStation() {
    approachStationSignal = true;
    controller.approachStation(true);
  }

  msgsrv toggleTrain() {
    self.leaveStation();
    self.approachStation() after (DELAY)
        ;
    self.toggleTrain() after(15);
  }
}

reactiveclass Door(8) {
  knownrebecs {
    Controller controller;
    System system;
  }

  statevars {
    boolean passengerOpenSignal;
```

```
    boolean lockSignal;
    boolean openSignal;
    boolean requestOpenSignal;

    boolean isOpen;
    boolean isLock;
  }

  msgsrv open() {
    openSignal = true;
    isOpen = true;
    requestOpenSignal = true;
  }

  msgsrv close() {
    openSignal = false;
    isOpen = false;
    requestOpenSignal = false;
  }

  msgsrv lock(boolean signal) {
    lockSignal = signal;
    isLock = signal;
  }

  msgsrv feedbackOpen() {
    system.feedbackOpen(isOpen);
    controller.feedbackOpen(isOpen);
  }

  msgsrv feedbackLock() {
    system.feedbackLock(isLock);
    controller.feedbackLock(isLock);
  }
}

reactiveclass Controller(9) {
  knownrebecs {
    Door door;
  }

  statevars {
    boolean lockSignal;
    boolean openSignal;
    boolean isOpen;
    boolean isLock;
    boolean leaveStationSignal;
    boolean approachStationSignal;
  }

  Controller() {
    isLock = false;
    isOpen = false;
  }

  msgsrv leaveStation(boolean signal) {
    if(signal){
      self.close();
      self.lock();
      self.leaveStation(false);
    }else {
      leaveStationSignal = true;
    }
  }

  msgsrv approachStation(boolean signal)
      {
    approachStationSignal = signal;
    self.unlock();
    self.open();
  }

  msgsrv open() {
    door.open();
    isOpen = true;
  }

  msgsrv close() {
    door.close();
    isOpen = false;
  }

  msgsrv lock() {
    door.lock(true);
    isLock = true;
  }

  msgsrv unlock() {
    door.lock(false);
    isLock = false;
  }

  msgsrv feedbackOpen(boolean value){
    isOpen = value;
  }

  msgsrv feedbackLock(boolean value){
    isLock = value;
  }

  msgsrv requestOpen() {

  }
}

reactiveclass System(2) {
  knownrebecs {
    Train train;
  }

  statevars {
    boolean isOpen;
    boolean isLock;
  }

    System() {
        self.start();
    }

    msgsrv start() {
      train.toggleTrain();
    }

  msgsrv feedbackOpen(boolean value){
    isOpen = value;
  }

  msgsrv feedbackLock(boolean value){
    isLock = value;
  }
}

main {
  System system(train):();

  Passenger passenger(door):();

  Train train(controller):();

  Door door(controller, system):();
```

```
Controller controller(door):();
}
```

Code 26: Train door Rebeca code

7.3 Mapping the Safety Contracts to Rebeca Properties

There are some safety properties derived from safety requirements, such as the train door should not be opened when it is locked, and the train should not leave the station if the door is open and not locked. However, we used our method to derive more safety properties from the safety contracts of each component. In the first step, safety contract assumptions have been added to Afra's properties. Two assumptions of Controller Component were added to properties file in Afra.

In order to debug the model, the safety analysis results are now being used. The controller and door components have four safety contracts. Now we examine them and if the verification fails, we should update the design and repeat our approach, which is beyond the scope of this thesis.

$$A : not(leaveStationSignal.omission)$$

$$A : not(approachStationSignal.omission)$$

leaveStationSignal and approachStationSignal are the inputs that make up the safety contrasts. Both of them have a Boolean type. It is straightforward to define the statement mapping for two: monitor their value and store it in a new variable for assertions.

Generated Rebeca code of Controller assumptions shown in Code 27

```
// Controller assumptions
  leaveStationSignal = controller.leaveStationSignal;
  approachStationSignal = controller.approachStationSignal;
//
```

Code 27: Definitions in the Property file of Afra for Controller

It is also true for the door component, which has two inputs with type of Boolean. The value of that input is compared with the component's normal behaviour to determine its expected value, then it is compared with the expected value and saved to a variable to be checked in the assertion process.

$$A : not(lockSignal.omission)$$

$$A : not(openSignal.omission)$$

Generated Rebeca code of Door's assumptions, shown in Code 28. This actor has two instances, we should check both of them.

```
// Door assumptions
isLock = door.lockSignal;
isOpen = door.openSignal;
//
```

Code 28: Definitions in the Property file of Afra for Door

The same approach used for inputs is also applied to outputs. Two guarantee of Controller Component were added to properties file in Afra.

$$G : not(lockSignal.omission)$$

$$G : not(openSignal.omission)$$

```
// Controller guarantee
    lockSignal = controller.lockSignal;
    openSignal = controller.openSignal;
//
```

Code 29: Definitions in the Property file of Afra for Controller

$$G : not(isLock.omission)$$
$$G : not(isOpen.omission)$$

Generated Rebeca code of Door's guarantee shown in Code 30. This actor has two instances we should check both of them.

```
    // Door guarantee
    isLockDoor = door.isLock;
    isOpenDoor = door.isOpen;
    //
```

Code 30: Definitions in the Property file of Afra for Door

Final Safety properties These are all safety properties extracted from safety contracts and safety properties from safety documents.

```
property {

  define {
        leaveStationSignal = controller.leaveStationSignal;
      approachStationSignal = controller.approachStationSignal;
    isLock = door.lockSignal;
    isOpen = door.openSignal;
    lockSignal = controller.lockSignal;
        openSignal = controller.openSignal;
        isLockDoor = door.isLock;
    isOpenDoor = door.isOpen;
  }

  Assertion {
      Assertion1: !(!leaveStationSignal && lockSignal);
      Assertion2: !(!approachStationSignal && openSignal);
      Assertion3: !(!isLock && isLockDoor);
      Assertion4: !(!isOpen && isOpenDoor);
  }
}
```

Code 31: Safety properties to be check in Rebeca

Model Checking via Afra tool In this step, we run a model checker. Afra needs two files to be able to verify. We created the Rebeca model in the section 7.2 and the properties file in the Afra section 7.3. After having these two files, we used Afra to run model checking. Afra looks for a counter example to check if any assertions failed. As shown in Figure 28, all assertions are satisfied in this example.

▾ SystemInfo	
Total Spent Time	0
Number of Reached States	186
Number of Reached Transitions	486
Consumed Memory	3720
▾ CheckedProperty	
Property Name	Deadlock-Freedom and No Deadlin...
Property Type	Reachability
Analysis Result	satisfied

Figure 28: Afra's report after model checking.

Discussion This example demonstrates how we use CHESSML models to create Rebeca models, and how we use ConcertoFLA results to create safety contracts, and then extract safety properties for model checking from safety contracts. Based on four safety contracts, we created four assertions. The assertions we added in Afra help us to formally verify any new update in the model against existing safety contracts.

8 Discussion and Future Work

In general, writing the properties of a model is one of the most difficult tasks in model checking. Moreover, safety properties for model checking are formulated in temporal logic and it is not easy for safety experts to specify properties in temporal logic. Extraction of properties for model checking from safety contracts, provide us with a systematic way for formulating properties. Safety contracts give us component-based properties. We derive the safety properties in a (semi-)automatic way and they are formulated in a structured manner. As a result of using safety contracts to derive safety properties, we know what type of failure to model check. To the best of our knowledge, this is the first work on defining safety properties for Rebeca models in a systematic, structured, and semi-automatic way.

As part of the method we present in this thesis, an expert uses ConcertoFLA to analyse the modeled system and uses the results to create safety contracts. When the industry uses CHESS to run safety analyses, CHESSML diagrams are created.

In CHESSML, Internal Block Diagrams (IBDs) model the system's structures. ConcertoFLA is applied to IBDs for failure transformation and propagation analysis. This thesis introduces a method to map Internal Block Diagrams of CHESSML to Rebeca code. IBDs do not capture the behavior of the system. The behavioral models include sequence diagrams. We apply a method to map sequence diagrams to Rebeca codes. Model checking is then conducted using the generated Rebeca codes. We formally verify the model against the safety properties derived from safety contracts to identify potential hazards.

After modeling the system in Rebeca modeling language and adding safety properties, we demonstrate how to debug the model. We can track how assertions fail using the counter example provided by the model checking tool, Afra. The assertions we created enabled us to verify new model updates formally. We can use assertions to verify them against the new model.

One of the challenges in using formal verification for industry is the gap between their safety requirements on one hand and the models and properties on the other hand. Our thesis helps in decrease this gap.

8.1 Responses to the Research Question

Using the steps described in Section 5, we are able to extract safety properties from safety contracts and use them in Afra to model check Rebeca models. We also proposed a method for mapping CHESSML behavioral models (Internal block diagrams) to Rebeca model in order to improve the process. The mapper transforms Internal block diagrams to Rebeca models.

RQ: How to derive safety properties from qualitative failure logic analysis (FLA) results and use them for formal verification? More specifically, how to derive safety properties from safety contracts and use them as properties for model checking Rebeca models?

Answer: Our research question focuses on determining the safety properties of safety contracts—the safety contracts generated from ConcertoFLA analysis results. We use safety contracts to derive safety properties in order to address this research question. There are assumptions and guarantees in all safety contracts. Afra's properties file contains statements that represent assumptions and guarantees. We created an assertion for each safety contract. The component-based ConcertoFLA technique helps in deriving the safety properties for model checking. Our thesis presents approaches that enable us to extract safety properties from individual components.

Additionally, the results can be used as evidence in the industry's safety assessment, allowing the industry to improve the design and create a strong design at an early stage.

8.2 Future Work

We can foresee the following ways to enhance the thesis. Future work can be undertaken in order to extend this thesis as listed below.

- In this thesis, the main objective is to derive safety properties from safety contracts. In the future, however, it will be important to extend this thesis on how to derive security properties from safety-security analysis, including our mappers, and how to extend mapper approaches to extract security properties from safety-security analysis.
- This work introduces two mappings: safety contracts to safety properties and internal block diagrams to Rebeca modeling languages. In order to run these approaches in CHESS and Afra tools, they will need to be code-based. Through this implementation, we will be able to automate our approach and enable industries to bridge safety analysis and formal verification.
- We tried to extract FPTC rules and create diagrams in CHESS; however, we can use them to add other diagrams and define security requirements in CHESS. Although we attempted to extract FPTC rules and create diagrams in CHESS, we can use them to add other diagrams and define security requirements. Furthermore, CHESS is capable of running dependability analyses and constructing fault trees. In future work, we can try to extract more information to use in Afra, like fault trees.

This thesis has paved the way for further research in connecting safety analysis approaches to Rebeca modeling language and formal verification techniques. This thesis provides the basis for new research and can contribute to academia and industry through the use of the results and process.

References

[1] J. F. Nunamaker Jr, M. Chen, and T. D. Purdin, "Systems development in information systems research," *Journal of management information systems*, vol. 7, no. 3, pp. 89–106, 1990.

[2] "Iec 61508-1:2010, functional safety of electrical/electronic/programmable electronic safety-related systems," http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm, accessed: 2010-09-30.

[3] C. Paulsen and R. Byers, "Glossary of key information security terms," Tech. Rep., Jul. 2019. [Online]. Available: https://doi.org/10.6028/nist.ir.7298r3

[4] O. Hasan and S. Tahar, "Formal verification methods," in *Encyclopedia of Information Science and Technology, Third Edition*. IGI Global, 2015, pp. 7162–7170.

[5] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 1977, pp. 46–57.

[6] [Online]. Available: https://rebeca-lang.org/alltools/Afra

[7] [Online]. Available: http://rebeca-lang.org/Rebeca

[8] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and verification of reactive systems using rebeca," *Fundam. Informaticae*, vol. 63, no. 4, pp. 385–410, 2004. [Online]. Available: http://content.iospress.com/articles/fundamenta-informaticae/fi63-4-05

[9] M. Sirjani, "Rebeca: Theory, applications, and tools," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2006, pp. 102–126.

[10] P. M. Alday, M. Schlesewsky, and I. Bornkessel-Schlesewsky, "Towards a computational model of actor-based language comprehension," *Neuroinformatics*, vol. 12, no. 1, pp. 143–179, 2014.

[11] B. Gallina, Z. Haider, and A. Carlsson, "Towards generating ecss-compliant fault tree analysis results via concertofla," in *IOP Conference Series: Materials Science and Engineering*, vol. 351, no. 1. IOP Publishing, 2018, p. 012001.

[12] B. Gallina and Z. Haider, "Making safeconcert security-informed to enable multi-concern modelling," in *30th European Safety and Reliability Conference ESREL-2020, 1 Nov 2020, Venice, Italy*. Research Publishing Services, 2020.

[13] L. Montecchi and B. Gallina, "Safeconcert: A metamodel for a concerted safety modeling of socio-technical systems," in *International Symposium on Model-Based Safety and Assessment*. Springer, 2017, pp. 129–144.

[14] I. Šljivo, B. Gallina, J. Carlson, H. Hansson, and S. Puri, "A method to generate reusable safety case argument-fragments from compositional safety analysis," *Journal of Systems and Software*, vol. 131, pp. 570–590, 2017.

[15] V. Djukanovic, "Mapping uml diagrams to the reactive object language (rebeca)," 2019.

[16] D. C. Schmidt, "Model-driven engineering," *Computer-IEEE Computer Society-*, vol. 39, no. 2, p. 25, 2006.

[17] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 633–642.

[18] M. Chaudron, S. Larsson, and I. Crnkovic, "Component-based development process and component lifecycle," *Journal of Computing and Information Technology*, vol. 13, no. 4, pp. 321–327, 2005.

[19] T. Stolte, G. Bagschik, A. Reschka, and M. Maurer, "Hazard analysis and risk assessment for an automated unmanned protective vehicle," in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 1848–1855.

[20] M. R. Chaudron, W. Heijstek, and A. Nugroho, "How effective is uml modeling?" *Software & Systems Modeling*, vol. 11, no. 4, pp. 571–580, 2012.

[21] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[22] M. Hause *et al.*, "The sysml modelling language," in *Fifteenth European Systems Engineering Conference*, vol. 9, 2006, pp. 1–12.

[23] L. Bressan, A. L. de Oliveira, L. Montecchi, and B. Gallina, "A systematic process for applying the chess methodology in the creation of certifiable evidence," in *2018 14th European Dependable Computing Conference (EDCC)*. IEEE, 2018, pp. 49–56.

[24] S. Mazzini, J. M. Favaro, S. Puri, and L. Baracchi, "Chess: an open source methodology and toolset for the development of critical systems." in *EduSymp/OSS4MDE@ MoDELS*, 2016, pp. 59–66.

[25] B. Gallina, M. A. Javed, F. U. Muram, and S. Punnekkat, "A model-driven dependability analysis method for component-based architectures," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2012, pp. 233–240.

[26] R. F. Paige, L. M. Rose, X. Ge, D. S. Kolovos, and P. J. Brooke, "Fptc: automated safety analysis for domain-specific languages," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 229–242.

[27] I. Sljivo, B. Gallina, J. Carlson, and H. Hansson, "Strong and weak contract formalism for third-party component reuse," in *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2013, pp. 359–364.

[28] J. A. McDermid, M. Nicholson, D. J. Pumfrey, and P. Fenelon, "Experience with the application of hazop to computer-based systems," in *COMPASS'95 Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security'*. IEEE, 1995, pp. 37–48.

[29] E. Khamespanah, P. Mrvaljevic, A. Fattouh, and M. Sirjani, "Using afra in different domains by tool orchestration," in *Composing Model-Based Analysis Tools*. Springer, 2021, pp. 283–299.

[30] M. Sirjani, L. Provenzano, S. A. Asadollah, M. H. Moghadam, and M. Saadatmand, "Towards a verification-driven iterative development of software for safety-critical cyber-physical systems," *Journal of Internet Services and Applications*, vol. 12, no. 1, pp. 1–29, 2021.

[31] B. Gallina, L. Montecchi, A. L. de Oliveira, and L. Bressan, "Multiconcern, dependability-centered assurance via a qualitative and quantitative coanalysis," *IEEE Software*, vol. 39, no. 4, pp. 39–47, 2022.

[32] I. Sljivo, "Assurance aware contract-based design for safety-critical systems," Ph.D. dissertation, Mälardalen University, 2018.

[33] A. Cimatti, M. Dorigatti, and S. Tonetta, "Ocra: A tool for checking the refinement of temporal contracts," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 702–705.

[34] S. F. Alavizaedh, A. H. Nekoo *et al.*, "Reuml: A uml profile for modeling and verification of reactive systems," in *International Conference on Software Engineering Advances (ICSEA 2007)*. IEEE, 2007, pp. 50–50.

[35] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega, "Chess: a model-driven engineering tool environment for aiding the development of complex industrial systems," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 362–365.

[36] M. Sirjani, E. A. Lee, and E. Khamespanah, "Model checking software in cyberphysical systems," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 1017–1026.