Journal of Software Engineering Research and Development, 2024, 12:16, doi: 10.5753/jserd.2024.3745
This work is licensed under a Creative Commons Attribution 4.0 International License.

From Textual to Formal Requirements: A Case Study Using Spectra in Safety-Critical Systems Domain

Luiz Eduardo Galvão Martins [Universidade Federal de São Paulo | legmartins@unifesp.br]

Abstract

The requirements specification of any system is crucial for the correct development of the systems and software. It becomes even more relevant in the development of safety-critical systems (SCS). This paper aims to investigate the process of transforming requirements specification written in natural language (textual requirements) to requirements specification written in Spectra language (formal requirements). Spectra is a formal language built to specify reactive systems. The case study carried out in this research focuses on the requirements specification of a low-cost insulin infusion pump. The requirements were initially specified in natural language and later transformed into Spectra language. During the transformation process, we investigated the potential use of the Spectra language in the phase of requirements specification, identifying the difficulties in the transformation process and its advantages, taking into account the software engineer's point of view.

Keywords: *Textual Requirements, Formal Requirements, Spectra specification, Spectra Language*

1 Introduction

Safety-Critical Systems (SCS) are increasingly present in the daily lives of modern societies, which are becoming heavily dependent on these systems. SCS are Man-made systems, which are based on computational technology, in which eventual defects or failures may cause accidents that put human life at risk or cause damage to the environment or property (Hatcliff *et al*. 2014)(Leveson 2011)(Heimdahl 2007). SCS are present in aviation systems, automotive systems, control of industrial plants (chemical, oil, nuclear), medical devices, railroad control, defense, and aerospace systems, among others (Leveson 2011)(Nair *et al.* 2015).

The requirements specification of any system is crucial for its correct development, and it becomes even more relevant for the development of SCS (Sommerville 2015)(Miller *et al.* 2006). Requirements Engineering is the discipline that focuses on the development of techniques, methods, processes, and tools that assist in the design of software and systems, covering the activities of elicitation, analysis, modeling and specification, verification and validation, and management of requirements (Sommerville 2015). The complete specification of software requirements establishes the basis for its architectural design and offers a description of the functional and quality aspects that should guide the implementation and software evolution.

The adoption of formal methods for specifying the requirements for SCS has been advocated by many researchers in recent decades (Miller *et al.* 2006)(Martins and Gorschek 2016). One of the main acclaimed benefits is that the specified requirements would become complete, consistent, and unambiguous, thanks to the rigor of formal methods. However, formal methods are still little used by SCS developers, the reports of using such methods in literature usually apply to illustrative exercises or pilot projects, not becoming an extensive practice within the SCS industry (Martins and Gorschek 2017). The reasons usually indicated for non-adherence to formal methods are (Bozzano and Villafiorita 2006)(Hu *et al.* 2007): formal methods are difficult to use, requiring prior knowledge of the specific mathematical language; few people have the knowledge to properly understand and use formal methods; formal languages are not suitable for raising requirements with system stakeholders, as they are complex and have a long learning curve. The requirements specification documents produced by professionals who develop critical systems and software worldwide are still essentially based on natural language (textual requirements) (Martins and Gorschek 2016)(Martins and Gorschek 2017). On the one hand, this practice facilitates communication between stakeholders, on the other hand, it makes requirements specifications subject to inconsistencies and ambiguities (Liu *et al.* 1995)(Chen 2009)(Miller *et al.* 2006) .

A possibility to increase the interest of SCS developers in adopting formal methods for specifying their systems and software would require using a process capable of helping practitioners transform the requirements specified in natural language into requirements specified in formal language. The study presented in this paper aims to investigate the development of such a process in order to easily, quickly, and safely transform SCS requirements written in natural language to specifications written in Spectra language. Spectra is a formal specification language for reactive systems, a category in which SCS normally falls. Spectra supports temporal constructs and others that allow system engineers and software engineers to make concepts such as monitoring and counting explicit in their specifications (Maoz and Ringert 2021a). We chose Spectra as the formal language for this study, motivated by the novelty of this language and its potential to increase the productivity of software engineers throughout the software development lifecycle.

In addition to the formal language, Spectra proponents provide a tool environment for synthesizing Spectra specifications for Java code, enabling software development to be truly driven by specifications. Along with this paper, we present and discuss the results obtained from a case study performed in a SCS domain, particularly in the domain of medical devices. The case study focused on the requirements specification of a low-cost insulin infusion pump under

development with a Brazilian company. The requirements were initially specified in natural language and later transformed into Spectra language. We tried to answer the following research questions throughout the case study: What steps are necessary to convert textual requirements specification into Spectra specification? What is the learning curve of Spectra? What are the benefits and difficulties in the transformation process? Does Spectra language properly capture the semantics of the textual requirements?

The remainder of this paper is organized as follows: In Section 2, we present background and related work; in Section 3, we present the methodology adopted to conduct the study; in Section 4, we present the case study carried out; in Section 5 we discuss the results obtained from the case study, and in Section 6 we present the conclusion and final remarks.

2 Background and Related Work

2.1 Definitions

In order to set the scope and make clear the adopted terms used in this research, and to ensure consistency throughout this paper, we present the following definitions, organized in alphabetical order:

Formal Language. A language used by software engineers to specify constraints and operations of the system accurately and unambiguously. A formal language is based on mathematical constructs.

Formal requirements. A set of system or software requirements specified using a formal language.

Natural Language. It is an informal language used by software engineers to write software requirements documents.

***Spectra* language.** A formal specification language for reactive systems, the category that critical systems typically fall into.

***Spectra* specification.** A software specification produced using Spectra language.

Textual requirements. A set of system or software requirements specified using a natural language.

2.2 Spectra

The Spectra language is a formally verified software/system specification language developed for modelling reactive systems (Maoz and Ringert 2019)(Maoz and Ringert 2021a). Reactive systems are systems that continuously interact with their environment and respond to external events in real time, such as air traffic control systems, industrial automation, and software embedded in vehicles. Important features of the Spectra language are the following: (i) Formal Specification: It allows to specify system and software in a formal and precise manner, facilitating the automatic verification of desired properties; (ii) Model Checking: Using Spectra, we can check whether a system model meets specified requirements, this is done through formal verification techniques, such as model checking, to ensure that the system satisfies all specified properties; (iii) Automation: The language was designed to be compatible with automation tools, allowing the automatic generation of tests and the synthesis of controllers that guarantee compliance with requirements; and (iv) Support for Reactive Systems: It was specifically designed to deal with the complexity of reactive systems, offering constructs that facilitate the modelling of dynamic behaviours and continuous interaction with the environment.

Spectra supports temporal constructs, as well as other constructs that allow systems engineers and software engineers to make concepts like tracking and counting explicit in their specifications (Maoz and Ringert 2019) (Maoz and Ringert 2021a) (Maoz and Ringert 2021b). The following is a list of the main features available in the Spectra language:

Module declaration. Every Spectra specification document is treated as a module. Each module is defined as a separate file.

Variable declarations. The variables defined in the specifications can be classified in two ways: as environment variables or as variables controlled by the system. Both can be Boolean, Int, or Enumeration types.

Assumptions and guarantees. The behavior of the environment, observable by the variables controlled by the environment, is described through assumptions. The required behavior of the system is described by means of guarantees.

PastLTL operators. PastLTL operators evaluate formulas over past interactions between the system and the environment. The available operators are PREV, ONCE, HISTORICALLY, SINCE, TRIGGERED.

Predicate definitions. Predicates allow encapsulation and reuse of parameterized Boolean expressions.

Monitor definitions. Monitors are used to track events over time.

Counter definitions. The following operations can be applied to counters: *inc, dec, reset, overflow, underflow.*

Pattern definitions. Standard definitions can be used to reuse specification units.

Weight definitions. The language allows the definition of integer weights on states and transitions of a specification.

The specifications produced with the Spectra language can be analyzed by a set of software tools (Spectra Tools), which include the synthesis of controllers that satisfy the produced specifications, since these specifications are realizable (Bozzano and Villafiorita 2006)(Ma'ayan 2022) (Ma'ayan and Maoz 2023). Spectra Tools will automatically attempt to find and produce an implementation that satisfies the GR(1) specification (Bloem 2012)(Amram *et al.* 2022)(Gorenstein *et al.* 2024). A GR(1) specification consists of assumptions, which must be satisfied by the environment, and guarantees, which must be satisfied by the system, i.e., by the synthesized implementation. For more details see https://smlab.cs.tau.ac.il/syntech/spectra/ .

2.2 Related Work

Sayar and Souquières (2019) proposed development patterns to formalize requirements describing constraints and sequences. The formal method used by them is the Event-B method. However, they believe that any formal method can be used with the development patterns proposed by them. Throughout the paper, they present and discuss two patterns:

(1) a conditional pattern Dev-if that describes a constraint on the system functionality; and (2) a sequential pattern Dev-seq that "helps the developer to automatically introduce the order between existing operations of a given system" [18]. They show two examples of how to apply the proposed patterns in a hemodialysis case study.

Cabral and Sampaio (2008) propose a strategy that automatically translates use cases written in a controlled natural language (CNL) into the specification in CSP process algebra. The system requirements are organized as use cases written in CNL using imperative and declarative sentences. Imperative sentences allow writing the actions performed by the system actors, while declarative sentences describe characteristics, constraints, and states of the system. The grammatical rules adopted to write use cases are defined based on the knowledge bases that map verbs to CSP channels and verb complements to values of CSP datatypes.

Jin *et al.* (2010) propose a concern-based approach to generating formal requirements specification from textual requirements document, which applies "separation of concerns during requirements analysis and utilizes concerns and their relationships to bridge the gap between textual requirements statements and formal requirements documentation". The formal specification generated is mainly represented by tabular expressions. Throughout the paper, a light control system is used to show the application of the approach.

Walter *et al.* (2017) propose a solution to detect redundant specifications and test statements described in structured natural language. They present and discuss formalization process for requirements specification and test statements, allowing them to detect redundant statements and reduce the efforts for specification and validation. The formalization process is based on the Specification Pattern Systems and Linear Temporal Logic. They evaluated the process in the context of Mercedes-Benz Passenger Car Development.

Ma'ayan and Maoz (2023) conducted an exploratory case study in which they followed students in a semester-long university workshop class on their end-to-end use of a reactive synthesizer, from writing the specifications to executing the synthesized controllers. Along this case study they collected more than 500 versions of more than 80 specifications, as well as more than 2500 Slack messages written by the class participants. Based on the collected data they propose guidelines in the directions of language and specification quality, tools for analysis and execution, process and methodology, all towards making reactive synthesis more applicable for software engineers.

Gorenstein, Maoz, and Ringert (2024) present two contributions to deal with Non-Well-Separation (NWS). In the first contribution they show how to synthesize systems that avoid taking advantage of NWS, i.e., do not prevent the satisfaction of any environmental assumption, even if possible. In the second contribution the authors propose a set of heuristics for the fast detection of NWS. They carried out evaluations over benchmarks from the literature showing the effectiveness and significance of their contributions.

3 Methodology

In order to investigate the transformation of textual to formal requirements, we chose to apply a qualitative research approach adopting a case study (Wohlin *et al.* 2012)(Robson 2002) as the strategy to reach the research goals. This study aims to obtain an in-depth understanding of the difficulties and benefits of using the Spectra language to specify the software requirements of a SCS. **Table 1** shows the research questions that drove our investigation.

Table 1. Research Questions.

Research Questions	Aims
RQ1: What steps are necessary to convert textual requirements specification to Spectra specification?	To understand the main learning elements, concepts, and tasks to be executed to convert a textual requirements specification to a Spectra specification.
RQ1.1: How long does it take?	To measure the learning curve of Spectra and compare the time necessary to produce a Spectra specification in relation to the textual specification.
RQ2: What are the benefits and difficulties?	To identify the main benefits and difficulties in adopting Spectra language to write formal requirements specifications.
RQ3: Are the semantics of the textual requirements well captured by Spectra?	To analyze the power of Spectra in capturing the semantics of textual requirements.

3.1 Study Design

This investigation was divided into three parts: planning, execution, and analysis.

Planning. As part of the planning of the study, we decided to adopt a case study as a research approach. The case study was conducted in a medical device domain. Two people participated in the case study. The profiles of the participants are presented as follows:

- Participant *P1*:
  - Background in software engineering with more than 20 years of experience in requirements specification and software development.
  - High knowledge about the requirements of the chosen system (insulin infusion pump). Five years of experience in insulin infusion pump software development.
  - No previous skills in Spectra language.
- Participant *P2*:
  - Background in computer science and software engineering with more than 15 years of experience in formal methods.
  - No previous knowledge about the requirements of the chosen system (insulin infusion pump).
  - More than eight years of experience with Spectra language.

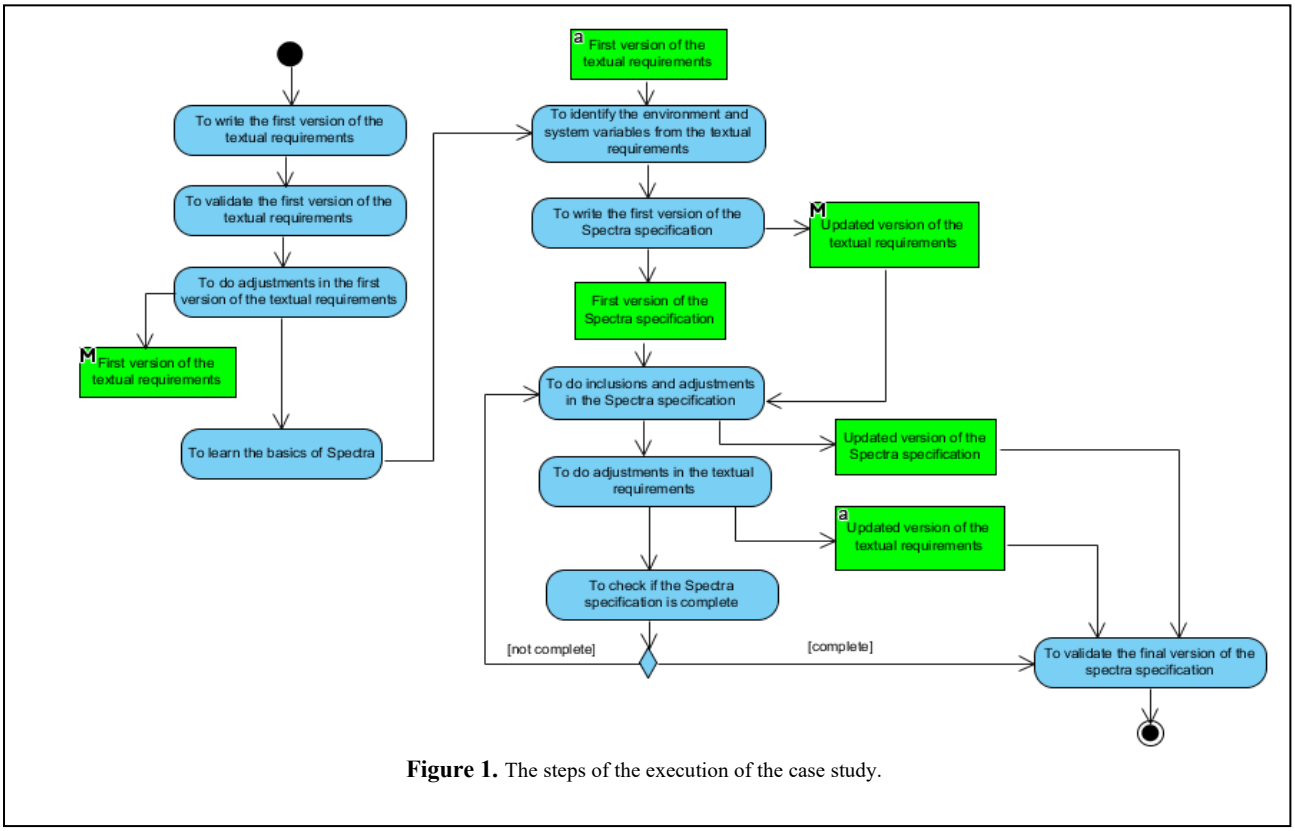

Figure 1. The steps of the execution of the case study.

The variables managed and observed along the case study were the following:

- System requirements knowledge (independent);
- Spectra skill (independent);
- Effort to write the textual requirements (dependent);
- Effort to write the Spectra specification (dependent);
- Captured semantics (dependent).

Execution. The execution of the case study was divided into four steps: (i) To write the textual requirements of the chosen system; (ii) To get a background of Spectra language; (iii) To transform the textual requirements into a Spectra specification, and (iv) To validate the Spectra specification. The details of the execution of the case study are presented in **Figure 1**.

Analysis. A quantitative and qualitative analysis of the results was performed. A quantitative analysis was performed considering the time spent writing the textual requirements, learning the basics of Spectra, and writing Spectra specifications. A qualitative analysis was performed focusing on the difficulties of using Spectra and the benefits of using it as a tool for a formal requirements specification.

3.2 Threats to Validity

The main threat to the validity of this study is related to the biases of the case study participants. The results obtained are obviously dependent on the experience and capabilities of the two participants who carried out the case study. However, since we do not intend to generalize the results to other cases, we believe that the results are helpful for an analysis of the use of Spectra language in the formal requirements specification in safety-critical systems context. Moreover, the extensive experience of the case study participants, both in software requirements specification and in software specification with the Spectra language, are factors that strengthen the obtained results.

4 Case Study

In this section, we describe the case study performed.

4.1 Context

The case study was carried out in a safety-critical system domain related to developing a medical device. The chosen system was developing a low-cost insulin infusion pump (LCIIP), which is under development in a Brazilian company in cooperation with participant P1. The first step of the case study was to write the requirements of the LCIIP in a format of textual requirements. The textual requirements were written by P1 and reviewed by P2. The second step was to get started with Spectra Language by P1, which had no previous contact. The third step was to transform the textual requirements specification into Spectra specification. This step was performed in cooperation between P1 and P2. The fourth step was the validation of the Spectra specification, which was performed for both participants. Steps 3 and 4 were performed in an iterative loop. The flow of the steps is shown in **Figure 1**. The entire case study was carried out in three months, with full time participation of P1 and part time participation of P2.

4.2 Textual Requirements of the LCIIP

The textual requirements specification is organized into two parts: a glossary of terms used in the specification and the functional requirements of the LCIIP. The functional requirements are identified by numbers. For each requirement, the time spent (in minutes) was recorded during the specification.

Glossary and Acronyms:

- Basal infusion: It is a continuous insulin infusion, which should run 24 hours/day until the insulin reservoir is empty (TIUR <=10.0). The basal infusion follows the basal profile previously configured by the user.
- Basal profile: It is divided into 24-time slots (from #0 to #23) corresponding to the 24 hours of the day. The user should configure the insulin units to be infused along each hour of the day. The system will infuse the insulin for 24 hours according to the basal profile defined by the user, changing the amount of insulin each hour of the day according to the user definition.
- Bolus infusion: It is a fast insulin infusion, which occurs according to the user's needs. Bolus infusion is actioned by the user.
- FMI: Frequency of micro infusion along one hour. FMI = (IU/6) x 60 *MI/hour*. It should be calculated for each time slot.
- Insulin reservoir: It was adopted a 3ml syringe as an insulin reservoir. The syringe is filled by the user. 300 IU is the syringe's full capacity. It must be filled with the full capacity.
- IU: Insulin units. Allowed range: 0 <= IU <= 6.0, which means that 6 IU is the max limit for one hour of infusion (0.1 IU/min), considering only the basal infusion.
- IUB: Insulin units for bolus infusion.
- IUR: Insulin units remaining (for each time slot).
- LCD: Liquid crystal display.
- LSS: Last *status* of the system. Values: “first use”; “normal”; “changing battery”.
- MI: Micro infusion. A MI occurs according to the FMI (the step motor runs N steps every M seconds). Every micro infusion corresponds to 0.1 IU. One IU corresponds to 0.01ml of insulin.
- MIC: Micro infusion counter. Every MI implies MIC = MIC +1.
- T: Saved current time (hour + minutes).
- TBMI: Time between micro infusions. TBMI = 3600/FMI *seconds*.
- TIUR: Total insulin units remaining (for the whole reservoir).
- TLB: Time when occurred the last bolus infusion.
- TR: Time when the system is restarted.
- TRS: Time remained stopped (TRS = TR – T)
- TSLB: Time since last bolus infusion (TSLB = current time – TLB).

Functional Requirements:

1. *Turn on* (7 min.)

   Description: The user pushes the “power on/off” button to turn on the system. If the battery is correctly placed and is charged, then the system is initialized.

   1.1 *Initialize system* (1h25min.)

   Description: The system should check LSS:

   - Case LSS = “first use” then the system should:
     - To get the confirmation from the user that the plunger is at the start position (this is manually done by the user) and the reservoir is connected.
     - To set TIUR = 300.
     - To show the TIUR on the LCD.
     - To wait for the user command.
   - Case LSS = “changing battery” then the system should:
     - To recovery TIUR.
     - To show the TIUR on the LCD.
     - If continuous infusion was running before to change the battery, then to perform FR4, else to wait for the user command.
   - Case LSS = “normal” then the system should:
     - To recovery TIUR.
     - To show the TIUR on the LCD.
     - To wait for the user command.

2. *Start continuous infusion* (2 min.)

   Description: The user pushes “Start button”. The system runs continuous infusion.

   2.1 *Run continuous infusion* (2h44min.)

   Description: Continuous infusion is performed driven by the basal profile previously defined by the user. The system gets corresponding IU from the basal profile and calculates FMI and TBMI. Every TBMI, a MI must be performed. After each MI, the IUR and the TIUR should be calculated and showed on the LCD. After each MI: IUR = IUR – 0.1 and TIUR = TIUR – 0.1.

   - At every full hour change, the system must override the IU and the IUR and recalculate FMI and TBMI.
   - Every minute to perform FR8.
   - Every minute to perform FR10.
   - Every MI implies MIC = MIC +1.

3. *Stop continuous infusion* (31 min.)

   Description: Continuous infusion can be stopped for three reasons: (1) "Stop" button is pushed by the user; (2) "Change" button is pushed by the user; and (3) Bolus infusion is confirmed by the user. When the continuous infusion is stopped, the system should save current time T (hour + minutes) for further recovery.

4. *Restart continuous infusion* (1h07 min.)

   Description: The system should restart when the user pushes “Restart button”; or when bolus infusion is finished (in the case of the continuous infusion has stopped because of the bolus infusion). When continuous infusion restarts, time T (hour + minutes when the system

was stopped) should be recovered. Then the system should:

- To get the current time (hour + minutes) when the system is restarted (TR).
- To calculate how long the system remained stopped (TRS = TR - T) and show TRS on the LCD.
- To perform FR2.1.

5. *Turn off* (5 min.)
   Description: The user pushes "power on/off" button for more than 5 seconds. The user shall confirm if he/she really wants to shut down the system. Set LSS = "normal" and store TIUR whenever shut down the system.

6. *Configure basal profile* (25 min.)
   Description: The user pushes "basal profile" button. If the continuous infusion is running, it must be stopped. For each hour of the day (from #0 to #23), the user should set IU to be infused.
   - Allowed range: 0 <= IU <= 6.0.
   - After IU is set, set IUR = IU (for each time slot).
   - The user shall set all 24-time slots to complete a profile. The user is forced to inform IU of each time slot.
   - After time slot #23 is filled, the system should:
     - To show the TIUR (main screen).
     - To wait for the user command.

7. *Set bolus infusion* (10 min.)
   Description: The user pushes the "Bolus" button to turn on the bolus infusion mode. The user should set IUB to be infused. The system asks the user for confirmation.
   - 0 < IUB <= 30.0.
   - TSLB >= 1h.

   7.1 *Confirm bolus infusion* (11 min.)
   Description: The user pushes the "Enter" button to confirm the bolus infusion. After confirmation, the system should:
   - To perform FR3.
   - To proceed with the bolus infusion.
   - To update TIUR (TIUR = TIUR – IUB)
   - To save TLB.
   - If the system was running continuous infusion before bolus infusion, then perform FR4, else:
     - To show the TIUR on the LCD.
     - To wait for the user command.

   7.2 *Cancel bolus infusion* (4 min.)
   Description: The user pushes the "Cancel" button to cancel the bolus infusion. If the bolus infusion has started, it cannot be canceled anymore.

8. *Check battery level* (5 min.)
   Description: Every minute, the system should check the battery level, and the battery icon should be updated.

   8.1 *Turn on battery alarm* (14 min.)
   Description: If the battery level is less or equal to 10% charged, then the system should:
   - To ring the alarm sound.
   - To show a message on the LCD.

   8.2 *Turn off battery alarm* (7 min.)
   Description: If the battery level is greater than 10%, then the system should turn off the battery alarm.

9. *Change component* (19 min.)
   Description: The user pushes "Change" button. The user may choose "battery", "reservoir", or "cancel" option.

   9.1 *Change battery* (13 min.)
   Description: The system should:
   - If the continuous infusion is running, then perform FR3.
   - To save TIUR.
   - To set LSS = "changing battery".
   - To Shut down the system.

   9.2 *Change reservoir* (14 min.)
   Description: The system should:
   - If the continuous infusion is running, then perform FR3.
   - To collect the plunger at the starting position.
   - To get confirmation from the user that the reservoir was changed.
   - To set TIUR = 300.
   - If continuous infusion was running, then to perform FR4.

   9.3 *Cancel* (1 min.)
   Description: The system should show the TIUR (main screen).

10. *Check reservoir level* (9 min.)
    Description: Every minute, the system should check the reservoir level, and the reservoir icon should be updated.

10.1 Turn on reservoir alarm (10 min.)
Description: If the TIUR <=10.0, then the system should:

- To ring the alarm sound.
- To show a message on the LCD.

10.2 Turn off reservoir alarm (4 min.)
Description: If TIUR >10.0, then the system should turn off the reservoir alarm.

4.3 Textual Requirements Validation

The textual requirements were written by *P1*, based on five years of his experience in the development of a low-cost insulin infusion pump prototype in cooperation with a Brazilian company. The requirements validation was based on an inspection process. The textual requirements presented in section 4.2 were specifically written for this case study. *P1* produced the first version of the textual requirements, which was reviewed by *P2*. The intention at this stage of the requirements validation was to assure that *P2* could completely understand the requirements of the LCIIP.

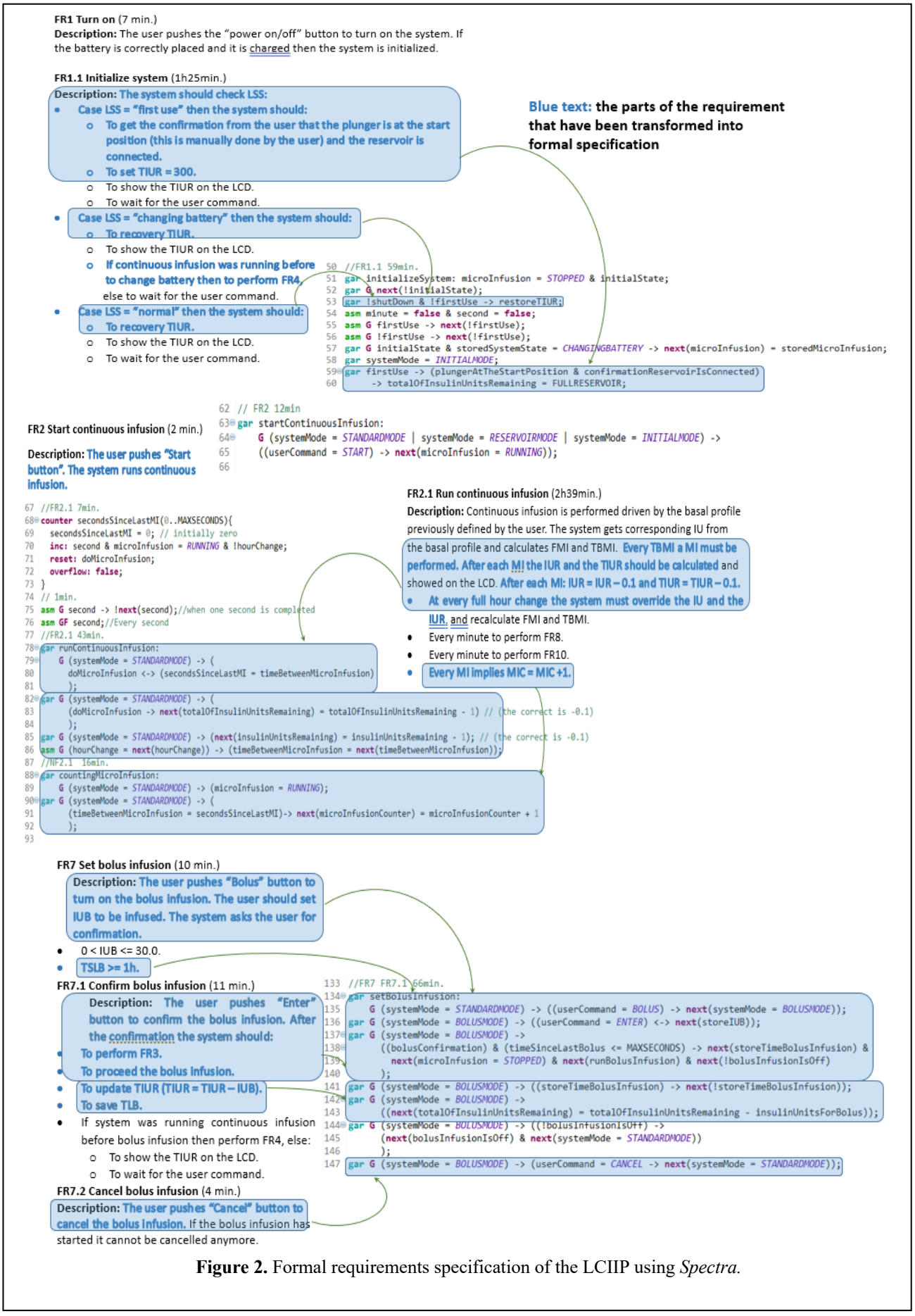

Figure 2. Formal requirements specification of the LCIIP using *Spectra*.

As stated in section 3.1, *P2* had no previous knowledge about requirements of the LCIIP. After a few interactions of the inspection process, the textual requirements received few adjustments becoming clearer to *P2*.

During the inspection process, *P2* read the requirements document, highlighting the points that raised doubts or difficulty in understanding. The main doubts raised by *P2* were not about the requirements themselves, but rather about some concepts specific to the problem domain, such as basal infusion, bolus infusion, basal profile and micro infusion.

A glossary of terms (presented in section 4.2) was created and included in the requirements document with the intention of making these concepts explicit.

4.4 Getting started with Spectra

As explained in Section 3.1, P1 had no previous experience with Spectra language. In order to get started with Spectra, P1 invested seven hours studying the document "Spectra Language & Spectra Tools User Guide" [4]. This introduction to Spectra allowed to P1 starting with the first Spectra specifications. A total of 37 hours were invested in learning during the case study: 23 hours (P1 alone) + 14 hours (supervised). The supervision was provided by P2.

4.5 Formal Requirements using Spectra

The textual requirements specification drove the whole Spectra specification of the LCIIP. Spectra specification is a formalization of the requirements of the LCIIP. Figure 2 shows the formal specification using Spectra for the functional requirements FR1, FR1.1, FR2, FR2.2, FR7, FR7.1, and FR7.2 of the LCIIP.

5 Results and Analysis

In this section, we present and analyze the results from the case study. The analysis is organized according to the research questions presented in section 3.

5.1 Steps to Convert Textual Requirements Specification into Spectra Specification (RQ1)

Spectra specification for the LCIIP was built along with the case study. This formal specification was based on the textual requirements specification created previously. Taken into consideration the structure of the Spectra language, the steps presented in Figure 3 were carried out to build a Spectra specification for the LCIIP. The whole Spectra specification for the LCIIP was written along with several iterations of the loop presented in **Figure 3**.

The loop presented in **Figure 3** is a simple and well-defined process that we proposed to transform/convert textual requirements specification into Spectra specification. As explained in Section 2, an important concept in Spectra is the separation between system and environment. The variables managed in Spectra should be defined as system variables or environment variables. System variables are ruled by guarantees, and environment variables are ruled by assumptions. Each step of the process presented in **Figure 3** is explained as follows.

1. Identify variables of interest: The identification of variables of interest is the starting point for the process of transforming textual requirements into Spectra specification. These variables must be identified from the textual requirements. Such variables will be handled as system variables or environment variables.

2. Create system variables: The creation of system variables is already part of the transformation of textual requirements into Spectra specification. These variables are the control variables of the system or software that is being formally specified. They will be used throughout the specifications of the guarantee.

3. Create environment variables: The creation of environment variables defines the variables that, although not controlled by the system or software, interact with it and will be used in specifying the assumptions.

4. Create guarantees: Guarantees will be the rules that will control the system or software being specified. These rules must be extracted from an interpretation of the functional and non-functional requirements present in the textual requirements.

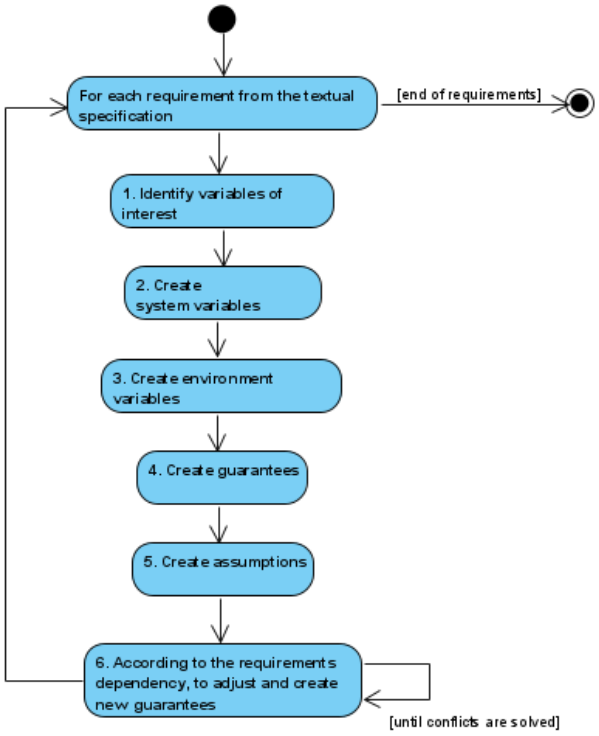

Figure 3. Main steps to convert textual requirements specification into *Spectra* specification.

5. Create assumptions: Assumptions must be created based on the analysis of non-functional requirements present in the textual requirements specification. Assumptions are rules that describe the environment's behavior.

6. Adjust and create new guarantees: After carrying out the previous steps, a minimum Spectra specification has already been produced. However, the correct interaction between the created rules (assumptions and guarantees) will require a process of refining these rules, based on a correct interpretation of the textual requirements. Both specifications

Table 2. Changes in Textual Requirements During the Formal Specification.

Rq	Change (after the validation of the textual requirements)	Trigger
FR3	Reasons for stopping the system: "alarm battery is on" and "alarm reservoir is on" were removed.	Formal specification of the guarantee "stopContinuousInfusion."
FR3	Reasons for stopping the system: "bolus infusion is confirmed by the user" was included.	Formal specification of the guarantee "stopContinuousInfusion."
FR4	Reasons for restart the system: "bolus infusion is finished" was included.	Formal specification of the guarantee "restartContinuousInfusion." The need to create the system variable "bolusInfusionIsOff."
FR2.1	More details for run continuous infusion: IUR = IUR – 0.1 and TIUR = TIUR – 0.1.	Need to create system variables to specify the guarantee "runContinuousInfusion."
FR8.2	We changed "If battery alarm is on and the battery level is greater than 10% charged, then the system should turn off the battery alarm." to "If the battery level is greater than 10%, then the system should turn off the battery alarm."	Improvement and simplification of the guarantee "checkBatteryLevel."
NFR2	Creation of a new NFR : "Battery level never goes up."	Improvement and simplification of the guarantee "checkBatteryLevel."
NFR3	Creation of a new NFR: "Micro infusion and collecting plunger should never happen together."	Formal specification of the guarantee "changeReservoir."
FR5	"Store TIUR whenever shut down the system." was included.	Formal specification of the guarantees "changeBattery" and "storeTIURWheneverShutDown."
FR1.1	"To collect the plunger for start position." was removed.	Formal specification of the guarantees related to "Initialize System."
FR1.1	We changed "To get the confirmation from the user that reservoir is connected" to "To get the confirmation from the user that the plunger is at the start position (this is manually done by the user) and the reservoir is connected."	Formal specification of the guarantees related to "Initialize System."
FR2.1	We included the definition of TBMI (time between micro infusion) and added it into the textual requirement.	Formal specification of the guarantee "runContinuousInfusion."
FR2.1	We included "Every TBMI a MI must be performed. "	Formal specification of the guarantee "runContinuousInfusion."
FR1.1	We removed "to save current time T (hour + minutes) for further recovery" when LSS = "normal". We checked that the current time is not further used.	Formal specification of the guarantees related to "Initialize System."
FR5	"Set LSS = 'normal' whenever shut down the system." was included.	Formal specification of the guarantee "turnOffSystem."
FR6	We included "If the microinfusion is running, it must be stopped."	Formal specification of the guarantee "configureBasalProfile."
FR1.1	We removed " To set LSS = 'normal' " when LSS = "first use".	Formal specification of the guarantees related to "Initialize System."
FR1.1	We removed " To set LSS = 'normal' " when LSS = "changing battery".	Formal specification of the guarantees related to "Initialize System."

(textual and Spectra) feed themselves throughout this refinement process.

In **Figure 2**, every rule initiated with *gar* means it is a guarantee. One difficulty we found during this process was to create rules to maintain consistency among the guarantees. Step 6 of the loop was executed several times until all conflicts among the guarantees were solved.

5.2 Time Spent for Specification (RQ1.1)

One dependent variable we measured along with the case study was the time spent on specification. We measured the time spent on textual requirements specification and the time spent for Spectra specification. **Figures 4 and 5** show the results in terms of the time for specifications. Figure 4 shows the comparison between the time spent for specification when used both approaches: the traditional approach based on textual specification and the innovative approach based on Spectra specification. As we can see in **Figure 5**, the total time spent for textual requirements specification was 512 minutes and 743 minutes for Spectra specification. Considering all the requirements, the time for specification using Spectra was 45% greater than using textual specification. The average rate of time spent between the two approaches was 1.45. Considering that the use of Spectra demanded a learning curve, we already expected that the time spent on specification using Spectra would be greater than using natural language (textual specification). However, it was a surprise that the average ratio between both approaches was just 1.45.

The sequence of requirements presented in **Figure 4** follows the chronology they were specified using Spectra. We started the Spectra specification with FR1 because of two reasons: (1) it seemed natural starting with the first requirements of the list presented in Section 4.2; and (2) it was considered a functional requirement of average complexity (appropriate for the first incursion with Spectra language).

The second requirement specified using Spectra was FR10. It is a simpler requirement than FR1, and there is a low relationship between them, which seemed to us a good candidate to be a second requirement to be specified using Spectra. We can see that the specification time ratio of the FR10 (0.43) is significantly less than the FR1 (1.29), even better for Spectra specification than textual specification (that was a real surprise). The same situation we observed in requirements FR2, FR6, and FR9. We believe that the reason for this finding is twofold: (1) the advance in the learning curve with Spectra; and (2) for simpler requirements, which mean requirements well-structured with cohesive functionality, it is faster to write Spectra specification than textual specification.

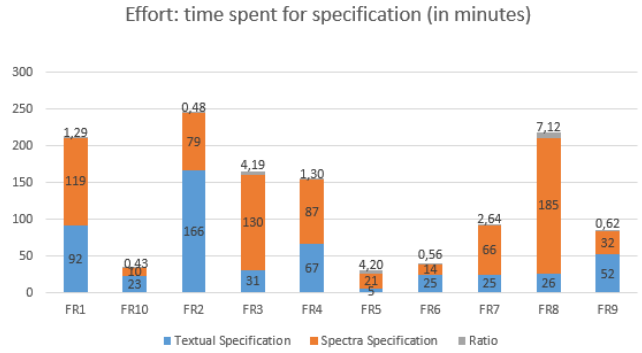

Figure 4. Comparison between the time spent for specification (for each requirement).

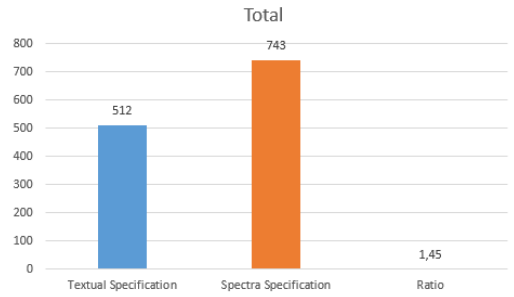

Figure 5. Total time spent for specification of all requirements.

5.3 Benefits and Difficulties along with the Formal Specification (RQ2)

In this section, we highlight the benefits and difficulties found along the formal specification using Spectra. The pros and cons commented in this section are only related to the requirements engineering process using Spectra. However, it is important to say that Spectra has a potential impact on the whole software development process. Other steps of the software development process besides the requirement process are out of scope of this study.

The main benefits are commented on below:

Separation of concerns between system and environment. This separation of concerns is quite beneficial for the conception of the system and the system requirements being specified. When specifying the requirements using Spectra, it is necessary to identify what variables belong to the system from those that belong to the environment. The separation of concerns helped to make the LCIIP requirements clearer. The initial version of the LCIIP requirements (textual requirements presented in section 4.2) was produced without the separation of concerns required by Spectra. Steps 1, 2, and 3 presented in **Figure 3** were performed in order to address the separation of concerns demanded by Spectra. After the system and environment variables were identified, it became easier to find the guarantees and the assumptions arising from the textual requirements. The requirements became more consistent and organized after the separation of concerns.

Requirements Formalization as a Requirements Validation Process. The formalization process of the textual requirements specification into Spectra specification inevitably forced the realization of a validation process of the textual requirements. During the formalization of the requirements into the Spectra specification, several inconsistencies and incompleteness were detected in the textual requirements. **Table 2** shows the changes and the triggers that motivated the changes in the textual requirements during the formalization of the requirements specification. There were 16 changes in total. **Figure 6** shows the number of changes regarding requirements. The requirement problems presented in **Table 2** only were detected during the formalization of the requirements, which we can see as a requirements validation process as well. This is a plus of using Spectra as a specification approach, taking into account that the formalization process will impact the rest of the software development process, not only the requirements engineering activity.

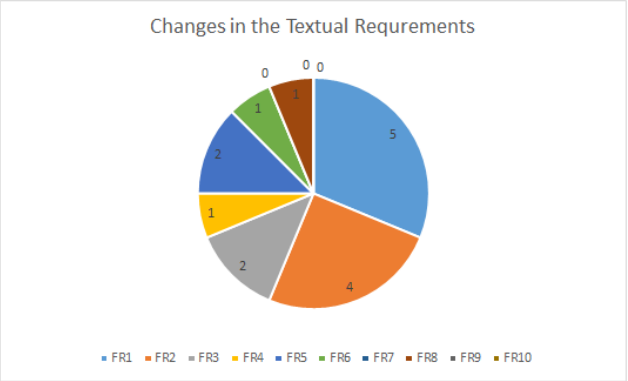

Figure 6. Number of changes by requirements after the requirements formalization (transformation to Spectra Specification).

The main difficulties are commented on below:

Learning curve (RQ1.1). It is always present when any new technology is going to be adopted. Spectra is a specification (formal) language for reactive systems, which is an automated procedure to obtain a correct-by-construction reactive system from a given specification [23]. We write Spectra specifications with the intention to find an implementation that satisfies the GR(1). A GR(1) specification consists of assumptions, which have to be satisfied by the environment, and guarantees, which have to be satisfied by the system [14]. So, the learning curve of Spectra was related mainly to understanding and correctly use the temporal operators of GR(1). As commented in section 4.4, participant P1 had no previous contact with Spectra or GR(1) concepts; however, with just seven hours of study, it was possible to

initiate the first specifications. The experience got along with this case study showed that the learning curve of Spectra can be considered as low for practitioners and students with a background in software engineering.

The semantics of the textual requirements captured by Spectra (RQ3). When we transform one specification written in one language to another there is always the problem of assuring if the original meaning (semantics of the original specification) was preserved after the transformation. Along with this case study, we tried to transform textual requirements into Spectra specifications; however, there was not a systematic process to guaranty that the original meaning of the textual requirements is completely preserved in the Spectra specification. The verification process to check if the original meaning was preserved was performed in an ad hoc way, based on the perceptions and experience of the participants P1 and P2, which is, of course, subjective. The main focus of Spectra is on the specification of the rules that cover temporal aspects of the dynamic control of the system. The textual specifications transformed throughout the case study did not take into account user interface aspects; just rules related to the internal control of the system were considered. In order to verify if the original semantics of the textual requirements was preserved in the Spectra specifications, it is necessary the definition of proper metrics. Again, this is a problem faced when one uses any formal methods; it is not a particular problem of Spectra language.

6 Conclusion

In this paper, we presented a case study that investigates the benefits and difficulties in transforming SCS requirements written in natural language into specifications written in Spectra language. The most relevant findings from this study and their implications for further research are as follows.

Transformation process. The basis for the development of any software comes from the correct specification of its requirements. Typically, requirements are written in natural language. The first bottleneck in the use of formal languages appears when we have to transform textual requirements into formal specifications. Throughout this case study, we proposed a process that helps the software engineer transform requirements into formal specifications. In this process, we present a first approach indicating some helpful steps for transforming requirements into Spectra specifications. This process was used as a framework to organize the Spectra specifications for the low-cost insulin infusion pump system. The process needs to be refined and tested in other case studies.

Validation process. The transformation process of textual requirements into Spectra specifications showed several inconsistencies and inaccuracies in the original textual requirements. The formal specification requires a detailing of terms and rules that is unparalleled in the natural language specification. Thus, during the formal specification, we noticed that 7 out of 10 requirements had some imprecision that required change and correction. In total, 16 inaccuracies were detected and corrected. Therefore, an interesting finding was that the transformation process automatically built into a requirements validation process. This requirements validation process is a valuable subproduct from the transformation process, which inevitably forces a broad assessment of the originally specified requirements. As future research, we suggest carrying out experiments to compare traditional requirements validation approaches with the validation provided by the transformation process in the context of formal languages.

Learning Curve. Surprisingly, the Spectra language learning curve proved to be short in this case study. As described throughout the article, participant P1 had no prior knowledge of the Spectra language. Despite the promising results, other studies and experiments need to be carried out to show how developers from different backgrounds and experiences perceive the effective learning and use of the Spectra language. Moreover, it seems to us that it is worthwhile to carry out comparative studies on the learning curve of Spectra with other formal languages, particularly in the context of using these languages for the development of safety-critical software.

Requirement semantics. Maintaining the semantics of the original requirements in the formal specifications produced after the transformation is still a challenge. This is an open problem that deserves research effort by the software engineering community, which is not particular when using Spectra.

As future work, we intend to extend the use of the Spectra language to specify new requirements for the insulin infusion pump that we are developing in cooperation with a Brazilian company. Furthermore, we intend to carry out other case studies involving software requirements for safety-critical systems, in the healthcare, aviation and automotive domains, in order to confirm the potential of the Spectra language as a valuable tool in the requirements specification and validation process.

Acknowledgements

This work was funded by São Paulo Research Foundation (FAPESP) under the grant agreement 2018/17592-1. Special thanks to prof. Shahar Maoz for the support during the realization of the case study at Tel Aviv University.

Data Availability Statement

All data produced and used in the case study are presented throughout this paper.

Conflict of Interest Statement (COI)

The author declares that there is no conflict of interest in the subject matter or materials discussed in this manuscript.

References

Amram, G., Maoz, S., Segall, I., and Yossef, M. (2022). Dynamic Update for Synthesized GR(1) Controllers. Proc. of ICSE 2022, pp. 786-797, ACM.

Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., and Sa'ar, Y. (2012). Synthesis of Reactive(1) Designs. J. Comput. Syst. Sci. 78(3), pp. 911–938. http://dx.doi.org/10.1016/j.jcss.2011.08.007.

Bozzano, M. and Villafiorita, A. (2006). The FSAP/NuSMV-SA Safety Analysis Platform. International Journal on Software Tools for Technology Transfer, 9(1), 5–24. http://doi.org/10.1007/s10009-006-0001-2.

Robson, C. (2002). Real World Research. 2nd Edition. USA: Blackwell Publishers.

Cabral, G. and Sampaio, A. (2008). Formal Specification Generation from Requirement Documents. In: Electronic Notes in Theoretical Computer Science, Vol. 195, 171-188, ISSN 1571-0661, https://doi.org/10.1016/j.entcs.2007.08.032.

Chen, Z. (2009). Formalizing Safety Requirements Using Controlling Automata. In Proceedings of the Second International Conference on Dependability (pp. 81–86). doi:10.1109/DEPEND.2009.18

Gorenstein, A., Maoz, S. and Ringert, J. O. (2024). Kind Controllers and Fast Heuristics for Non-Well-Separated GR(1) Specifications. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24). ACM, New York, NY, USA, Article 28, 1–12. https://doi.org/10.1145/3597503.3608131

Hatcliff, J., Wassyng, A., Kelly, T., Comar, C., and Jones, P. (2014). Certifiably safe software-dependent systems: challenges and directions. In Proceedings of the on Future of Software Engineering - FOSE, (pp. 182–200).

Heimdahl, M. P. E. (2007). Safety and Software Intensive Systems: Challenges Old and New. In FoSE 2007: Future of Software Engineering (pp. 137–152).

Hu, Y., Podder, T., Buzurovic, I., Yan, K., Ng, W. S., and Yu, I. (2007). Hazard analysis of EUCLIDIAN: An image-guided robotic brachytherapy system. In Proceedings of the 29th Annual International Conference of the IEEE EMBS (Vol. 1, pp. 1249–1252).

I. Sayar and J. Souquières. (2019). Bridging the Gap Between Requirements Document and Formal Specifications using Development Patterns. In: IEEE 27th International Requirements Engineering Conference Workshops (REW), 2019, pp. 116-122, doi: 10.1109/REW.2019.00026.

Ivarsson, M. and Gorschek, T. (2009). Technology Transfer Decision Support in Requirements Engineering Research: A Systematic Review of REj. Requirements Engineering Journal, vol. 14, no. 3, (pp. 155-175).

Jin, Y., Zhang, J., Hao, W. et al. (2010). A concern-based approach to generating formal requirements specifications. Front. Comput. Sci. China 4, 162–172. https://doi-org.ez69.periodicos.capes.gov.br/10.1007/s11704-010-0151-y.

Leveson, N. G. (2011). Engineering a Safer World: Systems Thinking Applied to Safety. The MIT Press.

Liu, S., Stavridou, V., and Dutertre, B. (1995). The Practice of Formal Methods in Safety-Critical Systems. Journal of Systems and Software, 1212(94), (pp. 77–87).

Ma'ayan, D. and Maoz, S. (2023). Using Reactive Synthesis: An End-to-End Exploratory Case Study," 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, pp. 742-754, doi: 10.1109/ICSE48619.2023.00071.

Ma'ayan, D., Shahar, M. and Rozi, R. (2022). Validating the Correctness of Reactive Systems Specifications Through Systematic Exploration. Proc. of MODELS 2022, pp. 132-142, ACM.

Maoz, S. and Ringert, J. O. (2019). Spectra Language & Spectra Tools User Guide. http://smlab.cs.tau.ac.il/syntech/spectra/userguide.pdf

Maoz, S., and Ringert, J. O. (2021a). Reactive Synthesis with Spectra: A Tutorial, IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Madrid, ES, pp. 320-321, doi: 10.1109/ICSE-Companion52605.2021.00136.

Maoz, S., Ringert, J.O. (2021b). Spectra: a specification language for reactive systems. Softw Syst Model 20, pp. 1553–1586. https://doi.org/10.1007/s10270-021-00868-z

Martins, L. E. G. and Gorschek, T. (2016). Requirements Engineering for Safety-Critical Systems: A Systematic Literature Review, Information and Software Technology, Vol. 75, July 2016, (pp.71–89).

Martins, L. E. G. and Gorschek, T. (2017). Requirements Engineering for Safety-Critical Systems: Overview and Challenges. IEEE Software, v. 34, (pp. 49-57).

Miller, S. P., Tribble, A. C., Whalen, M. W., and Heimdahl, M. P. E. (2006). Proving the shalls. International Journal on Software Tools for Technology Transfer, 8(4-5), (pp. 303–319). doi:10.1007/s10009-004-0173-6.

Nair, S., de la Vara, J. L., Sabetzadeh, M., and Falessi, D. (2015). Evidence management for compliance of critical systems with safety standards: A survey on the state of practice. Information and Software Technology, 60, (pp. 1–15).

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering, 14, 131–164. http://doi.org/10.1007/s10664-008-9102-8.

Sommerville, I. (2015) Software Engineering. Addison-Wesley, 10th edition.

Walter, B., Hammes, J., Piechotta, M. and S. Rudolph. (2017). A Formalization Method to Process Structured Natural Language to Logic Expressions to Detect Redundant Specification and Test Statements. In: IEEE 25th International Requirements Engineering Conference (RE), pp. 263-272, doi: 10.1109/RE.2017.38.

Wohlin, C., Runeson, P., Host, M., Ohlson, C., Regnell, B. and A. Wesslén. (2012). Experimentation in Software Engineering: An Introduction. Germany: Springer-Verlag.