Towards Safety Standard Compliance of IoT Software Systems Using Modelling and Verification with DCR Graphs

Anastasia Orishchenko

Thesis submitted for the degree of
Master in Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Towards Safety Standard Compliance of IoT Software Systems Using Modelling and Verification with DCR Graphs

Anastasia Orishchenko

Towards Safety Standard Compliance of IoT Software Systems Using Modelling and Verification with DCR Graphs

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

During the past years, software has become a large part of our personal and professional lives. We rely increasingly more on different kinds of applications to support our daily routines. Although failures in most of the applications we use do not have severe consequences, errors in some of the systems may cause damage to the environment, people, or economics and, in the worst case, human loss. For such safety-critical systems, it is crucial to ensure that they function correctly to reduce the risk of error occurrence. Various safety standards specify requirements for the software development process that should lead to more reliable software. However, meeting these requirements often increases the costs and development time. At the same time, many businesses require the software development and delivery process to be as fast and cheap as possible in order to survive in competitive environments.

This thesis addresses the trade-off between faster software development and the quality of the final product and discusses how modelling and verification can contribute to satisfying safety standard requirements. We use Tellu Diabetes App, a mobile application that forms a part of a safety-critical Internet of Things system, as a use case. The modelling and verification process is carried out with the help of DCR Graphs, a form of mathematical notation that has previously been used for modelling and reasoning about business processes. In the thesis, we construct and verify models of the process of making changes to the application code and the Tellu Diabetes App. For each of the models, we discuss which parts of the safety standard requirements it contributes to cover and to which extent. In addition, we reflect on the suitability of DCR Graphs as a tool for modelling an Internet of Things system since they have not been applied for this purpose previously. We also address the limitations of the DCR Tool, the graphical web-based interface used for constructing DCR Graphs encountered during the thesis work.

Acknowledgements

First, I would like to thank my supervisors, Christian Johansen and Olaf Owe, for their support during thesis work. My special appreciation goes to the main supervisor, Christian Johansen, for suggesting such an interesting use case for this thesis and providing invaluable guidance and feedback during the whole process. His advice and recommendations always helped, and his questions challenged me to learn more and gain a better understanding of the subject.

I would also like to thank Tellu for letting me use their Diabetes App as a use case in this thesis. In particular, I am grateful to Lars Thomas Boye for taking his time to describe the application and the development process.

Another gratitude goes to the DCR Community and, in particular, Håkon Normann for being open for any questions regarding DCR Graphs.

Finally, I would like to thank my parents and friends for their moral support, encouragement and providing some needed distractions to rest my mind throughout the master studies. This period would have been much harder without their help.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

During the past years, software has become a large part of our personal and professional lives. We rely increasingly more on different kinds of applications to support our daily routines. Therefore, it is essential that errors in the software occur as rarely as possible. Although failures in most of the applications we use do not have severe consequences, errors in some of the systems may cause damage to the environment, people, or economics and, in the worst case, human loss. Such systems are called safety-critical.

History shows examples of software bugs resulting in car accidents, plane crashes and cancer patients receiving wrong doses of radiation [30, 45, 50]. To avoid history repeating, it is crucial to gain trust in the systems developed by allocating time and resources on verification during the development process. Various safety standards specifying requirements for the software development process have also been introduced to reduce the risk of error occurrence. Meeting these requirements often implies higher costs and longer development time.

At the same time, many businesses that involve software as part of their processes operate in competitive and rapidly changing environments. In order to beat competitors, stay relevant for the current market and make a profit, the software development and delivery process must be as fast and cheap as possible.

The trade-off between faster software development and the quality of the final product has served as the main motivation for this thesis. By studying a safety-critical application Tellu Diabetes App as a use case, the thesis addresses whether modelling and verification as part of the development process can contribute to achieving both.

Tellu Diabetes App is a mobile application developed by Tellu that forms a part of an Internet of Things (IoT) system for monitoring diabetes patients. The application is intended to be used by elderly people with diabetes type 2 to report measurements like blood glucose levels and blood pressure. The measurements can then be accessed by medical personal who can react to abnormal values and provide help if needed. The system is considered safety-critical because any application or data transmission failure could cause serious health problems for the patients.

1.2 Thesis objectives

One of the main challenges faced by Tellu in satisfying safety standard requirements is related to code modification control. For each change made to the application code, it must be proved that the application still behaves correctly. In other words, it must be shown that modifications in the code are introduced in a way that maintains or improves the quality of the application. Therefore, the first objective of this thesis is to study whether it can be done by modelling and verifying the process of introducing new changes in the code. The tool chosen for this purpose is Dynamic Condition Response Graphs (DCR Graphs), a form of mathematical notation that has previously been used for modelling and reasoning about business processes.

Furthermore, to achieve a higher safety level, safety standards pose requirements for the use of formal methods as a part of the software development process, i.e. specifying and verifying the system developed in notation with formally described semantics. Dynamic Condition Response Graphs (DCR Graphs) represent such a notation but have not been used for modelling computer systems previously. Therefore, it would be interesting to investigate whether Dynamic Condition Response Graphs (DCR Graphs) are suitable for modelling an Internet of Things system as a part of the software development process. It is the second objective of the thesis.

Based on the two previous paragraphs, the goals of this work can be summarised as follows:

1. Study how modelling and verification of the process of introducing new changes to the code can contribute to satisfying safety standard requirements.

2. Investigate how well Dynamic Condition Response Graphs (DCR Graphs) can be applied to modelling a real Internet of Things (IoT) system as a part of the software development process.

1.3 Thesis outline

Chapter 2 provides theoretical background on topics discussed in the thesis. First, the concept and challenges related to the Internet of Things systems are presented in Section 2.1, proving the need for additional methods to reduce the risk of using them. Section 2.2 explains how safety standards address the risks of failures that can occur in a system. Then, Section 2.3 describes activities in the software development process and the requirements posed by safety standards and presents the trade-off between developing software fast and developing secure software. Sections 2.4 and 2.5 look deeper into software modelling activity and narrow it down to formal modelling. Finally, Section 2.6. gives an overview of some approaches that have been used to model Internet of Things systems previously.

Chapter 3 provides the theory required to construct and reason about DCR Graphs. First, a brief history of DCR Graphs and an overview of the DCR method is given in Sections 3.1 and 3.2. Then, Sections 3.3 and 3.4 describe the basic structure and graphical notation of DCR Graphs as well as their formal semantics. Finally, Section 3.5 explains the methodology used in this thesis to construct DCR Graphs and verify the desired properties.

Chapters 4, 5 and 6 constitute the central part of the thesis. The use case is described in detail in Chapter 4, while Chapters 5 and 6 present the work done on modelling and verification of the process of introducing new changes to the code and mobile application, respectively.

Chapter 7 summarises and discusses the findings made during the thesis work and presents some thoughts on future work.

2 Background

2.1 Internet of Things (IoT)

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it" [53] was one of the central statements in Mark Weiser's paper published in Scientific American magazine in 1991. He proposed a notion of ubiquitous computing, describing a world where computers exist everywhere, becoming a significant yet invisible part of people's lives.

Indeed, there has been considerable growth in the number and diversity of physical devices over the past years, including not only personal computers and mobile phones but also various types of sensors. Connecting such physical objects into a network and enabling them to interact and cooperate with each other in the same manner as people do over the Internet would provide an enormous amount of opportunities for improving the quality of our lives. This idea was in 1999 coined as the Internet of Things by Kevin Ashton [34].

Since then, the term Internet of Things (IoT) has been widely used in industry. Applications of IoT span a wide range of domains like homes, cities, retails, logistics and healthcare, helping people perform their daily tasks. IoT devices can be found everywhere from our homes and cities to inside our bodies.

Along with the opportunities, the goal of IoT "to enable things to be connected anytime, anyplace, with anything and anyone ideally using any path/network and any service" [20] also brings many research and development challenges and requirements that IoT systems have to satisfy.

2.1.1 Challenges of designing and developing IoT systems

Distribution

Distributed systems consist of separate components that may be executed either sequentially or in parallel on different, interconnected nodes. Building distributed systems and especially creating middleware solutions that enable easy implementation of services on top of them has a set of challenges [49]. Since IoT applications are by nature distributed, they inherit many issues of distributed systems like implementation of routing protocols, robustness and fault tolerance in case of communication links, nodes software and hardware failures and synchronisation.

Adaptability

Many IoT systems consist of nodes with resource constraints (e.g. memory, processor and power) and are mobile or wirelessly connected to the network. Due to these factors, the nodes may be arbitrarily connected and disconnected from the system, making it highly dynamic. To manage the communication between the nodes and services which use them, handle the nodes leaving and joining the network spontaneously and detect, diagnose and attempt to fix the problems as they occur, IoT applications have to be self-adaptive, self-organised, self-optimising, and self-healing [20].

Real time

Some IoT applications perform operations where delay in communication and data delivery may have dangerous consequences. Such time-critical systems have to provide on-time delivery of data and services.

Intelligence (data management)

IoT applications are often associated with generation of vast amounts of data. These data typically have high volume and various forms and are made available at different speeds. Applications have to be able to interpret and reason about the data to make correct decisions and execute tasks without human interoperability. In other words, IoT applications need to obtain intelligence. Achieving this goal requires that the systems incorporate decision-making techniques such as context-aware computing service, predictive analytics, and complex event processing [51].

Furthermore, the data can sometimes be corrupted for reasons like a sensor's failure, malicious introduction of invalid data, delays in data delivery or wrong data format. IoT applications have to be designed and developed so they can establish the presence of invalid data.

Security [32]

Security is a critical requirement of IoT applications. Due to a large number of devices, services and people sharing the same network, it is essential to ensure that the data cannot be eavesdropped on or interfered with by non-authorised users (confidentiality) and that it can only be accessed and processed by the corresponding user (privacy). Trust is another important security principle since it ensures that security and privacy objectives are achieved during the interaction among different objects, IoT layers and applications. In addition, IoT applications must check data integrity to avoid erroneous operation of applications.

Regulatory compliance [51]

Many IoT applications collect information about people's daily activities that people may consider confidential. The leakage of this sensitive

information could affect the privacy of the individual. In order to avoid privacy violation, IoT applications must be compliant with law-enforced requirements like EU data protection rules [19].

Scalability

According to IoT Analytics, there are around 12 billion IoT devices worldwide by the end of 2020, and this number is expected to grow to 30 billion by 2025 [33]. Many questions across several fields have to be answered to deal with such a large scale of objects. For example, existing communication protocols may no longer suffice, and new ones may have to emerge. New solutions for collecting, using and storing massive amounts of data produced by the devices have to be found. Moreover, new architectural models supporting the variety of devices and applications must be developed [47].

Humans in the loop

Many IoT applications involve humans as a part of the system, i.e. humans and objects work in synergy. Although including humans in the loop can have some advantages (e.g. human assistance in self-driving cars can improve safety), modelling human behaviour is a challenge since complex physiological, psychological and behavioural aspects of human beings must be considered [47].

Limitations of IoT devices

Many IoT devices, such as sensors, have resource limitations like low memory, processing power, and power shortage. As communication is the most energy-consuming task on devices, the need for low power communication technologies arises [20].

Heterogeinety

IoT applications involve interaction between heterogeneous devices, i.e. devices with different operating conditions, capabilities and functionalities. Furthermore, the network used for the interaction can be heterogeneous (e.g. fixed, wireless or mobile). Therefore, enabling seamless integration of IoT devices and achieving portability of IoT applications is seen as one of the significant challenges [42].

Designing and implementation challenges

IoT systems consist of several hardware and software components that communicate with each other. Designing IoT systems can be a complex and challenging task since these systems involve interactions between various components such as IoT devices and network resources, web services, analytic components, application and database servers. The wide range of choices available for each of these components makes it difficult to evaluate

the alternatives and design systems generically enough to be able to change a particular choice of product or service [3]. If that is not the case, it may become complex to update the system design to add new features or replace previously chosen components.

Application maintenance challenges

The code running on IoT devices will have to be debugged and updated regularly. Since IoT applications may be distributed over a wide geographical area, maintenance operations present several challenges. Support for remote debugging and application updates on IoT devices poses problems related to privacy, security and limited bandwidth of these devices.

2.2 Functional safety

As shown in the previous section, IoT systems have many potential points of failure, making their development and operation a challenging and error-prone process. For many of these systems (like transport or medical systems), failures could have serious consequences such as causing harm to people or the environment or, in the worst case, lead to a fatal outcome. In other words, these systems depend on the correct function of each individual hardware and software component, i.e. functional safety of the system. Unfortunately, it is impossible to completely exclude the risk of failure from the system. However, it is possible to define and accept a tolerable risk target and try to reduce the risk to this level.

Two types of failures can occur in a system, random failures and systematic failures. Random failures are usually related to specific hardware components and happen at random time intervals. Using historical data for similar components, it is possible to predict how often such failures can occur in the future and set a quantified safety target based on this failure rate. Systematic failures, on the other side, are not related to specific hardware components but rather to mistakes done during the development or operation of the system. Therefore, it is challenging to predict the frequency of systematic failures and set quantified targets in the same way as for random failures. However, safety targets for systematic failures can be addressed qualitatively by establishing a set of controls and methods with the purpose of minimising their occurrence. The higher level of rigour in the development and operation process, the less likely is the occurrence of systematic failures.

Since both quantitative and qualitative factors need to be considered during risk assessment, the need for a measure of a system's safety performance arises. For that purpose, the concept of safety integrity levels (SIL) was introduced. There are four levels, each corresponding to the severity of the identified risk and defining a set of requirements. A higher safety integrity level means higher requirements but a lower probability of failure occurrence.

Safety integrity levels are defined differently in different standards.

Here, the IEC 61508 Standard is used for providing examples of requirements since it is a general standard concerned with the functional safety of electrical/electronic/programmable electronic devices, which serves as a base for many other industry-specific standards. The IEC 61508 Standard defines safety integrity levels 1 to 4 (SIL1 - SIL4) where SIL1 is the lowest and SIL4 is the highest [44]. Requirements defined for the different levels are related to a system's hardware and software components. Since this thesis uses a mobile application (i.e. a software component of the system) as a use case, only the requirements for the software development are explained in detail.

2.3 Software engineering process

A software process is a set of activities that leads to the production of a software system [46]. Since two software systems may differ with regard to their type, goals, functionality, complexity and criticality, the paths towards the final products will probably not be identical either. In addition, software processes will vary depending on other factors like the company developing the software or size, experience and skills of the development team.

This section presents two distinguished process methodologies, a more rigorous process typically proposed for safety-critical systems (such as described in the previous section) and an agile process.

2.3.1 Rigid software engineering process

Part 3 of IEC 61508 Standard is called "Software requirements" and addresses, as the title suggests, techniques and activities used during the software development process [44].

The Standard suggests an approach to the software development process called V-model. V-model is a top-down approach containing several phases from the overall software specification at the top to the actual software code at the bottom. Each phase has a corresponding activity whose purpose is to verify the output produced during the phase. The verification process starts from the bottom with testing of a specific module, followed by testing of integrations of modules and finishing at the top with testing of the entire system. A variant of the V-model demonstrating its principle is illustrated in Figure 2.1. However, the number of phases and techniques used during different phases may vary based on the complexity of the system and target safety integrity level.

The Standard requires that any modifications and deviations from the suggested development process are documented and explained.

Software specification

The first activity proposed by V-model is overall software specification. Software specification is the process of defining and understanding system

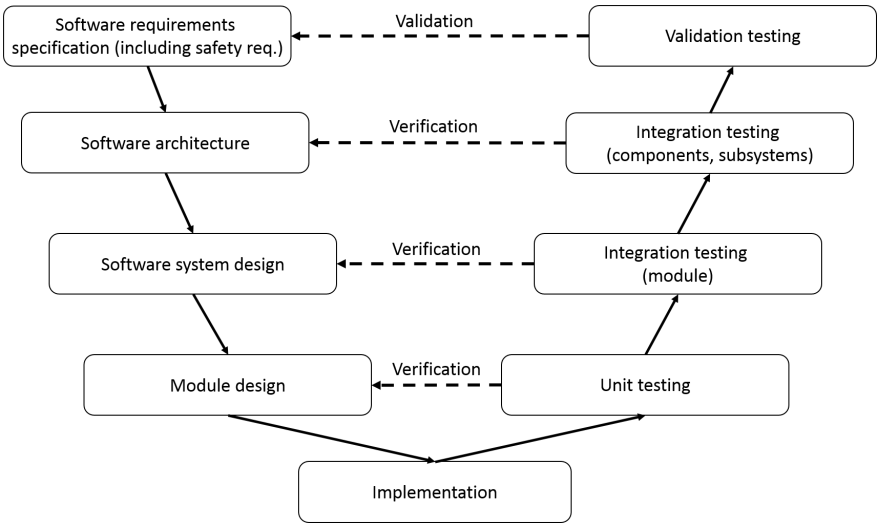

Figure 2.1: V-model

requirements carried out by customers and the development team [46]. The output of this process is a document containing the description of a system that satisfies stakeholders requirements. This phase of the development process is critical since mistakes and misunderstanding occurring at this stage may lead to problems at later stages, longer and more resource consuming development process or, in the worst case, unusable software.

The requirements collected during software specification can be divided into two major categories: functional and non-functional requirements. Functional requirements specify what the software should do, i.e. which functionality it must provide and how it should behave under certain conditions. Non-functional requirements, on the other hand, specify how the software should work and can be seen as quality attributes. Typical non-functional requirements include, but are not limited to, performance (for instance, response time and throughput), scalability, availability and usability. Safety, which takes a central part in the IEC 61508 Standard, is also an example of a non-functional requirement.

The Standard specifies some guidelines for how the software specification should be written. First, it states which items should be covered by the overall specification (among others "all the modes of operation, the capacity and response time performance requirements, maintenance and operator requirements, self-monitoring of the software and hardware as appropriate, and details of all internal/external interfaces") and by the software safety requirements in particular (like "capacities and response times, equipment and operator interfaces including misuse, functions which force a safe state, and out of range values") [44].

Furthermore, the specification must be clear, precise and free from ambiguity and should easily be traced back to other relevant documents. In addition to specification in natural language, the Standard suggests using semi-formal or formal methods for some or all requirements depending on the target safety integrity level. For example, semi-formal methods should

be used to describe critical parts of the specification for SIL1 and SIL2 and for the entire specification for SIL3 and SIL4. Some of the requirements for the specification process posed by the IEC 61508 Standard can be summarised as presented in Table 2.1.

Technique/measure	SIL1	SIL2	SIL3	SIL4
Computer-aided specification tools	Recommended	Recommended	Highly recommended	Highly recommended
Semi-formal methods	Recommended	Recommended	Highly recommended	Highly recommended
Formal methods	—	Recommended	Recommended	Highly recommended

Table 2.1: Requirements involving software specification

Software design and implementation

When the system requirements are collected and analysed, the process of developing an executable system can start. The software development activity typically consists of software design and implementation.

Software design is concerned with describing the structure of the system, data models used by the system and the interfaces between the components [46]. As can be seen from V-model in Figure 2.1, the design process is usually carried out in several stages, starting from overall system architectural design to a more detailed design of system components (modules). However, these stages are usually not executed linearly but are interdependent and interleaved. For instance, some design decisions made at the component level may affect overall system design, which means that previous steps have to be reworked and adjusted.

Output from the system design process serves as a base for software implementation, where programmers write and debug software code. Software implementation is often supported by various development tools. Most known are probably integrated development environments (IDEs), software applications that enable programmers to work more effectively by providing helpful features like syntax highlighting, code completion, refactoring and debuggers. Some software development tools may also be used to generate parts of code based on the design as explained and exemplified later in Section 2.4.

The IEC 61508 Standard specifies some requirements related to software design and implementation processes based on the target safety integrity level. These requirements are summarised in Table 2.2.

As we can see from the table, structure and modularity are central for all safety integrity levels. The reason for that is that smaller software modules are easier to implement, test and maintain, i.e. they are more manageable. Reuse of software modules is also encouraged as long as they are trusted and verified in similar situations. Furthermore, semi-formal and formal methods are required for software design purposes from SIL2

and above. Finally, coding standards and defensive programming should be used during the implementation of the code to increase the quality of the software.

Technique/measure	SIL1	SIL2	SIL3	SIL4
Computer-aided design tools	Recommended	Recommended	Highly recommended	Highly recommended
Semi-formal methods	Recommended	Highly recommended	Highly recommended	Highly recommended
Formal methods	—	Recommended	Recommended	Highly recommended
Modular approach	Highly recommended	Highly recommended	Highly recommended	Highly recommended
Structured programming	Highly recommended	Highly recommended	Highly recommended	Highly recommended
Use of trusted/verified software modules (if available)	Recommended	Highly recommended	Highly recommended	Highly recommended
Design and coding standards	Recommended	Highly recommended	Highly recommended	Highly recommended
Defensive programming	—	Recommended	Highly recommended	Highly recommended

Table 2.2: Requirements for software design and implementation

Software verification and validation

Although developers usually check their code for errors and defects during the implementation process described above, additional steps must be performed to ensure that both individual components and the system as a whole work as expected. This process is referred to as software verification. Verification is defined as the "process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase" [28]. Today, software verification is usually associated with two techniques, peer reviewing and testing.

A peer review is a process of analysing code statically, i.e. without executing it, performed by one or several engineers who have not taken part in the development of a particular piece of code. Peer reviews are an effective technique for discovering errors in code and improving overall code quality [46]. However, since the code is not executed, it can be hard to catch runtime errors or performance problems during a peer review.

Unlike peer reviews, testing is a dynamic technique, meaning that the software gets executed. The purpose of testing is "to show that a program does what it is intended to do and to discover program defects before it is put into use" [46]. Errors in software are discovered by running a program in many possible situations (e.g. for different input values). As

V-model in Figure 2.1 demonstrates, the testing process consists of several stages. The first stage is unit testing, where program components like individual functions or object classes are tested. Unit testing is followed by integration testing, where individual components are integrated into composite components and further to complete system so that component interfaces and interactions between components are tested.

The IEC 61508 Standard specifies a number of requirements involving software verification which are summarised in Table 2.3. According to the Standard, both peer reviews and testing must be carried out at the level of individual modules to verify that they perform the intended function and do not perform unintended functions. Integration testing with predefined test cases and test data is also required. All results of testing and corrective actions, if any, must be documented. These requirements apply to all safety integrity levels. For SIL2 and higher, formal proof is suggested as a verification technique in addition to peer reviews and testing.

Technique/measure	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Formal proof	—	Recommended	Recommended	Highly recommended
Static analysis	Recommended	Highly recommended	Highly recommended	Highly recommended
Dynamic analysis and testing	Recommended	Highly recommended	Highly recommended	Highly recommended
Software complexity metrics	Recommended	Recommended	Recommended	Recommended

Table 2.3: Requirements for software verification

The last phase included in V-model (figure 2.1) is software validation. Validation is about confirming that the software as a whole satisfies the requirements and meets the expectations of the customer collected during the software specification process [46]. The IEC 61508 Standard requires that a validation plan is written. It should cover all life-cycle activities, specify pass/fail criteria and show how all the requirements (including safety requirements) have been addressed. In addition, various metrics like test coverage and conformance to coding standards should be collected and presented.

Software evolution

Even after the software is developed, tested and put into use, the software development process continues since the software may be further modified as a result of feedback from the customers, new requirements or discovery of errors. This stage is usually referred to as software evolution (or maintenance) and is historically seen as a separated process [46]. However, the distinction between development and maintenance is increasingly blurred, and maintenance is seen as a continuation of the development process.

Software evolution poses a challenge on standard compliance since it must be proven that the development process and the software still satisfy all the requirements for each modification. The IEC 61508 Standard specifies a number of steps that must be performed if a change is made. The modification itself and its reasons must be well documented, and the impact of the modification must be analysed. Methods and procedures should be similar to those used during the original process. Furthermore, re-testing of either individual modules or the whole system is required based on the target safety integrity level. Only changed modules or all affected modules needs to be re-verified for SIL1 and SIL2, respectively, while the whole system must be re-validated for SIL3 and SIL4.

2.3.2 Agile software development methods

Nowadays, businesses operate in a competitive and rapidly changing environment. They have to adapt to new opportunities and markets, changing economic conditions and competing products and services that emerge. As software has become a part of almost all business operations, it is crucial that it can be developed and changed quickly to reflect new requirements. In many cases, time to market, i.e. time it takes to develop and deliver the software, is one of the most critical requirements. If the software is not developed quickly enough, it may be irrelevant for the current market, meaning that the business waists resources (such as money and people) without getting any benefits. Such hard deadlines and resource limitations imply that the quality of software design, architecture and code is often compromised in favour of delivering a functioning system or application in time.

Requirement for rapid development and delivery of software presents a set of challenges with regard to the development process. Since software requirements change rapidly (e.g. new requirements may be added or existing ones adjusted), it is impossible to define a complete specification of the software before it is developed. It may also be challenging to design software architecture in detail because of uncertainty about which other systems the software may interact with, how the relationship between different parts of the system may change or which modules have to be added in the future. As we see, traditional plan-driven development processes (like the V-model presented in the previous section), where specification, design, implementation and testing steps are carried out sequentially without the possibility to go back to the previous step, are unsuitable under these circumstances.

The need for development processes that can handle changing requirements has been recognised for many years [31]. Already in the 1970s and 1980s, there were proposed modifications and adjustments to traditional sequential approaches. However, the real growth of rapid software development occurred in the late 1990s with the introduction of methodologies such as Extreme Programming (XP) [5], Scrum [43], and Dynamic systems development method (DSDM) [48]. These and other methods for rapid software development became known as agile development or agile meth-

ods. The processes and practices of the different agile approaches vary from one approach to another. However, all of them have common philosophy and characteristics described in the Agile Manifesto by the leading developers of these methods [6].

First, the agile methods propose a development process where specification, design, implementation and testing activities are interleaved in contrast to plan-driven approaches. The system specification and implementation evolve simultaneously rather than sequentially. As shown in Figure 2.2, the activities are performed in cycles, with new requirements added to the specification and implemented iteratively.

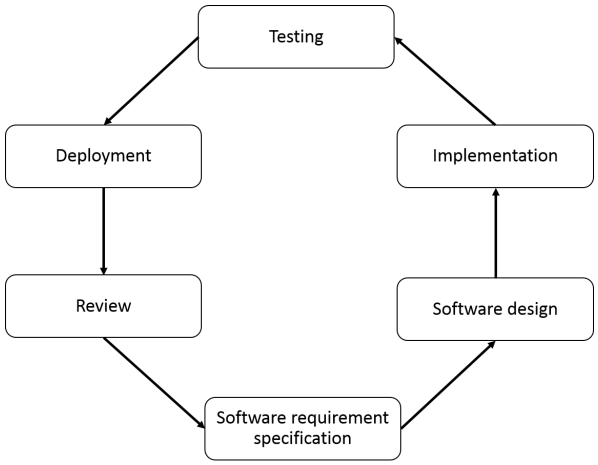

Figure 2.2: Agile software development life cycle

Second, detailed system specification and documentation is minimised or generated automatically by development tools. It allows developers to focus mainly on software implementation and testing. This concept is summed up in Agile Manifesto as "Working software over comprehensive documentation" [6].

Furthermore, all agile methods suggest that software should be developed in increments (e.g. pieces). Each increment focuses on developing a particular feature or functionality during a short period of time. By the end of the increment, a new version of the software is delivered to the customers. They evaluate the work that has been done and propose changes or provide and prioritise new requirements for the next increment.

This implies that the customers are closely involved in the development process. Customer engagement and collaboration are essential in agile methodologies since they ensure that the product developed satisfies current business requirements. Rapid releases allow the customers to track the progress made by the development team and provide feedback frequently. Continuous communication between the customers and developers helps to clarify existing requirements or changes to them, uncover misunderstanding and recognise missing functionality. This way, the development of outdated or unnecessary functionality can be avoided, leading to less wastage of financial and time resources.

In addition to changing business requirements, there are several other reasons why the software is constantly changing during the development process. The development team or the customers may find bugs and error that have to be fixed. New versions of software components may be released, so the system must be adapted. Moreover, the quality of the codebase should continuously be improved in order for it to be readable and maintainable. Managing, remembering and communicating information about all changes to the development team members and customers is not an easy task. Therefore, the development process should be supported by tools that help keep track of the changes.

For example, requests for changes or bug and issues reports can be kept track of using change management tools. Such tools also help determine the costs and impact of making these changes and decide if and when the changes should be implemented.

Additionally, version control systems track and provide control over changes made to the source code, documentation or configuration files. They also ensure that changes made by different developers do not interfere with each other, promoting collaboration between team members and across the teams.

Tools are also used to make the development process more effective by automating one or several steps. For instance, automated testing tools are used to test the system after the changes are made in order to check that the changes did not introduce any errors in the system. System building, e.g. a process of creating an executable system by assembling program components, data and libraries, compiling and linking them, is also often automated by using continuous integration and continuous deployment tools (CI/CD tools).

The automation described in the previous paragraph is a fundamental idea of DevOps, a concept that can be seen as an extension of agile methodologies. DevOps is a set of practices focused on bridging the gap between the development and operation of software systems. By automating processes of building, testing and deployment of the system, DevOps aims to provide feedback from operation to development faster than when different teams perform these processes. This way, the software development cycle can become even shorter, and the quality of the software increased.

2.4 System modelling

The previous section explained the software development activities and how they are typically organised in different process methodologies. This section will look more in-depth at one of the activities, namely system modelling.

System modelling is the process concerned with developing models of a system that present different views or perspectives of that system. A model is not a complete description of the system. Rather, it is an abstraction of the system that simplifies a system design and picks out one

or several main characteristics, leaving out the details. For example, one can construct different models showing interactions between the system and its environment, interactions between the components of a system, structure of the data processed by a system or behaviour of the system as a reaction to internal or external events.

System models are useful for several reasons during different stages of the software development process.

First, models can be used during collection, definition, refinement and prioritisation of the requirements as a base for discussion between the development team, customers or other stakeholders. As modern software engineering processes often require a collaboration of experts from different disciplines, models can help decide on the system scope (i.e. what is and is not a part of the system being developed) and achieve a similar understanding of the system requirements [22].

Second, models can serve as a base for the implementation of the system by software engineers. For instance, models specifying information about and the relationship between real-world objects may be helpful for deriving internal data representation of these real-world object in the system or deciding upon technologies for data persistence (e.g. SQL versus NoSQL database). Models can also stimulate discussions among the development team regarding architectural patterns or programming paradigms to use when developing the system. Furthermore, models that are complete and correct can even be used to generate source code of the system by automated tools, like Umple [36], Xpand [55], or Acceleo [21].

Another application of system models is as a way to document a system or some parts of it. Especially in the context of agile methodologies, where time spent on writing documentation should be minimised, using models for that purpose may seem attractive.

There are many existing notations for system modelling, ranging from graphical, easier to understand notations to various mathematical notations. Nowadays, system modelling is usually associated with representing a system using graphical notation, like Unified Modelling Language (UML) [8]. However, it is also possible to construct system specifications using formal (mathematical) models.

2.5 Formal modelling and verification

The requirements for the use of formal methods as a part of the process to achieve a higher safety integrity level were mentioned several times in Section 2.3.1. This section will explain what it means to model and verify a system formally, how this process can be carried out, and the advantages and disadvantages formal methods have in a software development process.

A formal model is a model represented in a notation that has formally described semantics. Formal models are developed by translating the system requirements expressed in natural language, diagrams or tables into a mathematical language. The resulting formal model is an unambiguous

description of the system's behaviour which serves as a formal specification of the system.

2.5.1 Advantages and disadvantages

Developing formal models has many advantages with regard to the rest of the development process. First, the process of formal modelling forces a detailed analysis of the requirements, which may uncover potential problems and make their correction less resource consuming than at the later stages of the development. Requirements analysis also reduces the scope of misunderstanding introduced by the imprecisions in the natural language specifications. Thus, formal models are the most precise way to specify the system.

Most importantly, formal models can be applied to check that a program behaves correctly and consistently with the specification. In other words, formal models can serve as a verification method, along with other more widespread techniques as peer reviews and testing.

Only a small proportion of all possible situations is executed during testing since their number is usually huge. Therefore, we can not be sure that the software is error-free [52]. Compared to testing, verification using formal models has the advantage of providing a formal (mathematical) proof of program correctness [28]. Using automated tools, called model checkers, it is possible to analyse all possible runs of a model exhaustively. Model checkers take a formal model and a formal description of the desired property as an input, analyse the specification and report that the desired property is satisfied by the model or give an example that shows that it is not satisfied. If the former is the case, then the program is guaranteed to meet its specification, while the latter case shows inconsistencies and incompleteness that should be fixed. This way, costs associated with the testing of the program can be reduced since the program model has been verified against its specification.

All in all, formal modelling and verification can help increase the quality and reliability of software. Academic researchers have been advocating for their use in software development for decades [12, 54]. Also, industrial experience in certain areas shows the benefits of applying formal methods [40, 41]. Even technology giants such as Amazon and Facebook have seen the need to apply formal methods [10, 39]. Engineers working on Amazon Web Services have been using formal specification and model checking for several years and believe that they "improve both time-to-market and quality of [their] systems" [39]. In more safety- and business-critical domains, like the aviation industry, the use of formal methods has even been integrated as a part of development standards, and mandatory certification procedure [29]. However, for the majority of software-intensive companies, formal methods are still not systematically integrated into the daily development process. There are several reasons for that.

First, formal models expressed with mathematical notations may be challenging to understand for customers or domain experts, so they can not

check whether the model precisely reflects the requirements. On the other side, software engineers, who understand the model, may lack domain knowledge, so they can not be sure if the model was constructed correctly with regards to the application domain.

Second, formal modelling is an expensive process because much time is needed to translate the requirements into a formal language and check the specification. Since there is little experience in using formal methods during software development processes, it is difficult to see and estimate possible cost savings that its use may bring. Therefore, managers rarely want to take the risk of adopting this approach.

Third, creating formal specification of large and complex systems is hard. It may be feasible to model the most critical parts of a system formally, but scaling it to the complete systems is difficult.

One of the most important disadvantages of formal modelling is that it may be hard to combine with agile software development approaches, which are most popular in the industrial context today. In order to develop the formal specification, it is required that the system requirements and design are defined, analysed and checked in detail. It is usually inconvenient in an environment where system requirements change rapidly. Therefore, formal models are primarily developed as a part of a rigid software process.

2.5.2 Formal verification process

Verification using formal models was mentioned in the previous section as one of the main advantages of using formal methods in the software engineering process. This section provides a more detailed explanation of how the formal verification process is usually carried out and which techniques can be used during each of the steps.

The process of formal verification can be divided into three phases: modelling, running and analysis.

Modelling

The modelling phase consists of constructing a model of the system under consideration using chosen modelling language and defining the properties that the system should satisfy.

Formal models are often expressed as transition systems. Transition systems are directed graphs with nodes corresponding to states and edges corresponding to transitions. A state is some information about a system at a particular point of its behaviour, while a transition is a specification of how the system can progress from one state to another. For example, for a computer program, the state is the current values of all program variables and the current value of the program counter indicating which program statement to execute next. The result of executing this statement is that the program counter will increase, and values of other program variables may be changed (i.e. change in the state). Thus, execution of a program statement can be seen as a transition [4].

Just like models, properties to be checked have to be expressed in an accurate and unambiguous way using property specification language. An example of property specification languages commonly used for verifying system properties is temporal logic. Temporal logic has operators for representing concepts of time (for example, "always", "eventually", and "never"), making it convenient to express properties describing desired behaviour of a system over time. Two main types of system properties that can be specified in temporal logic are safety and liveness properties.

Safety properties

A safety property states that an undesirable event will never occur under certain conditions. The most known examples of safety properties in the context of distributed systems are deadlock freedom (i.e. that a state where all processes in the system are waiting for action from another process in order to continue never is reached) and mutual exclusion (multiple processes never access a shared resource).

The above safety properties are called invariance properties (or invariants). Invariants are requirements that should hold in all reachable states of program executions. In terms of transition systems, it means that the property has to be satisfied by all initial states, and for all transitions, it is true that if a state s satisfies the property, then the state s' reached by the transition also satisfies the property. Verification of invariance properties can be done by exploring all reachable states and trying to find a state where the desired property is violated. If such a state exists, then it is proved by counterexample that the model does not satisfy the property.

However, not all safety properties are invariants. Some properties impose requirements on fragments of execution paths instead of separate states like invariance properties do. For such properties, a counterexample is a path fragment (called "bad prefix") that shows a violation of the desired property.

Safety properties alone are not sufficient for verifying a system since a system can easily fulfil safety properties by doing nothing. Thus, safety properties are usually complemented by liveness properties.

Liveness properties

A liveness property expresses that some event will eventually occur under certain conditions. Unlike safety properties, liveness properties do not represent requirements that have to hold continuously, but those that must ultimately take place [7]. An example of liveness property is eventual consistency (e.g. several nodes in a distributed database will eventually reach consistency in data).

Running and analysing results

The model and properties specified during the modelling phase are used as input for model checkers, i.e. tools that can exhaustively analyse all

possible runs of a program to check the validity of the properties in the system model. Model checker reports whether the property is satisfied or violated. In the latter case, a counterexample is returned and has to be further analysed in order to discover the cause of the error. If the error lies in the model or property, then they have to be revisited and verification restarted with the corrected model. Otherwise, the verification served its purpose by showing a problem in the system. The system can then be improved.

2.6 Modelling IoT applications

In literature, several approaches for modelling IoT applications have been proposed. Some of the methods and tools presented were developed with the primary purpose of supporting the development process, while others aim to facilitate formal analysis and verification of certain aspects or properties of the developed system. This section summarises some of the distinctive approaches that deal with the specification of IoT applications.

2.6.1 UML-based approaches

Unified Modelling Language (UML) is a modelling language for visualising, specifying, constructing and documenting the artefacts of software-intensive systems that was approved as a standard approach [8].

UML has served as a base for development of other modelling languages that are extended into the realm of IoT applications, like SysML4IoT [13], PervML [11] or ThingML [23]. These modelling languages aim to solve some challenges specific to the design process of IoT systems by using conceptual primitives suitable for this domain.

PervML and ThingML also provide code generation frameworks that transform model specifications into code in one or multiple programming languages (Java for PervML and C, Java or JavaScript for ThingML).

2.6.2 Bigraph-based approaches

Bigraphs is a mathematical model proposed by Robin Milner as a method for modelling ubiquitous systems [37]. Bigraphs focus on two aspects of such systems, namely locality and connectivity. This focus is reflected in the bigraph structure, consisting of two graphs: the place graph, which expresses the physical or logical location of system components (locality) and the link graph, which describes relationships and interactions between them (connectivity). In addition, dynamics of a specified system (i.e. changes in placing or linking of a part of the system) are defined using reaction rules.

Set of bigraphs representing the states of the system obtained from an initial bigraph, and the reaction rules applied successively on it is called a Bigraphical Reactive System (BRS) [37].

The BRS has been adopted to specify the physical part of an IoT application dealing with the spatial distribution, the mobility and the heterogeneity of its components in a model called Bigraphical Communicating Agent Model for the Internet of Things (BCAM4IoT) [35].

3 DCR Graphs

This chapter provides theoretical background on Dynamic Condition Response Graphs (DCR Graphs), the modelling language used for modelling and verification purposes in this thesis. First, the motivation for the development of the DCR Graphs and their applications are introduced in Section 3.1. Then, Section 3.2 presents the graphical DCR Tool used to construct the graphs. Further, the basic structure and graphical notation of DCR Graphs is explained in Section 3.3 before Section 3.4 provides formal definitions necessary for reasoning about the graphs. Finally, Section 3.5 demonstrates the methodology that was followed for modelling and verification using DCR Graphs in this thesis.

3.1 Introduction

For the last years, businesses have tried to model their business processes in order to increase productivity and quality. There are many existing techniques and notations, with the Business Process Management Notation (BPMN) being the industrial standard today. BPMN or other notations such as swim lane diagrams, flow charts or sequences of state-changing commands are examples of imperative process models, meaning that all possible control flows are described explicitly.

Imperative process models serve their purpose in providing a common way of describing business processes. However, they have several limitations. First, the number of all possible paths may become relatively high for complex processes. Describing all of them is almost impossible, which may lead to the incomplete specification of the process. Second, a collection of all possible routes does not capture the rules and constraints that the routes are based on. That means that models must be supplemented with textual descriptions with no formal relationship. Both of these limitations make updating and maintaining imperative models to reflect changes in business processes a tedious, error-prone and time-consuming operation. Furthermore, such notations are often difficult to understand, making it hard to engage end-users and leading to implementations that do not support real business needs.

In order to counter these issues, Dynamic Condition Response Graphs (DCR Graphs) has been proposed as a new model. In opposite to BPMN, DCR Graphs is a declarative, event-based model for describing process flow. It means that control flow is defined implicitly as a set of constraints

or rules. Therefore, there is no need to explicitly describe all possible flows since all flows satisfying given constraints are allowed. In this way, DCR Graphs capture the logic behind the process rather than a few possible routes.

DCR Graphs is a method that has been developed in collaboration between the Process and System Models research group, lead by Thomas Hildebrandt at the IT University of Copenhagen and Exformatics A/S. DCR Graphs technology has evolved through a series of research projects over the past years with substantial amount of academic publications on both formal aspects [16, 24, 26, 38] and industrial case studies [15, 17, 18, 25, 27].

The development of DCR Graphs continues as interest in the technology increases in the industry. The DCR method is now supported by a full technology stack based on formal notation and semantics of DCR Graphs. For instance, an interactive tool, called DCR Tool, allows teams or individuals to design, simulate, analyse, document and execute declarative processes. DCR Tool is continuously improved and extended with new features and plugins.

3.2 DCR Tool

As mentioned in the previous section, the DCR Tool is a graphical and interactive web-based tool that allows businesses and academia to create and simulate DCR Graphs.

The main features of the DCR Tool include creating new graphs, saving or exporting graphs, editing existing graphs and simulating graphs. Simulations enable the user to test the DCR Graphs and verify that the process behaves as expected. The tool has an easy to use interface and provides a guided tour containing key aspects of creating DCR Graphs with the tool.

The DCR tool also has support for version control, providing a way to see the history of graph revisions and changes made from one version to another and making it easy to revert to a particular version.

In addition, the DCR Tool adds a social aspect to the DCR technology by providing a way to share created graphs and collaborate with friends or co-workers. It is possible to do collaborative simulations, enabling users to play the processes like a game. This contributes to keeping employees and end-users engaged in the modelling process.

In addition to core functionality, the DCR Tool has an App Store containing several applications that can be added to the tool. Applications that were used in this thesis are presented later in Section 3.5.

3.3 Structure and graphical notation

A Dynamic Condition Response Graph (DCR Graph) is a directed graph with nodes representing the events that can happen and edges representing relations between the events.

3.3.1 Activities

An activity or an event is a process or task within a workflow. It can represent invocation of an operation, a step in a business process or an entire business process. Activities can have a name, description and role(s) assigned to them. They can also have relations to other activities. In addition, an activity has three state values expressing whether it is included, pending and executed.

Events/activities may be standard, data and computations. Standard activity is an activity that can just be executed. A data activity is assigned a value when executed. A computation activity holds an expression, which is evaluated and assigned as the value of the activity.

3.3.2 Relations

A relation or a connection describes how two activities relate to each other. Connections are used to express business rules that a process must follow. There are five main types of relations between events:

1. Condition, Milestone and Pre-condition
2. Response and No-Response
3. Include
4. Exclude
5. Spawn

Condition

The condition relation, graphically represented as shown in Figure 3.1, between an activity A and an activity B means that activity B can only occur if activity A has occurred previously.

Figure 3.1: Graphical representation of condition relation in DCR Tool

An example of condition relation could be activities A "Implement task" and B "Create pull request". The "Implement task" activity must precede the "Create pull request" activity in a process, so "Implement task" is a condition for "Create pull request". This example is graphically demonstrated in Figure 3.2.

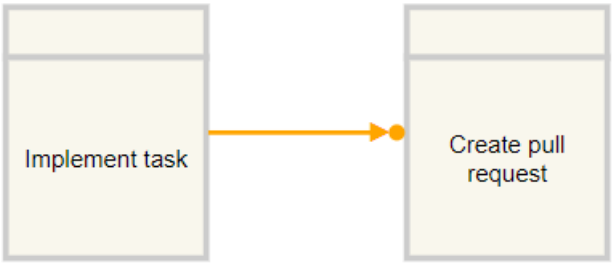

Figure 3.2: Example of condition relation

Milestone

The milestone relation creates a dependency between activity A and activity B such that B can occur initially. But if A becomes pending, then B cannot occur until A has occurred. The milestone relation is graphically represented as shown in Figure 3.3.

Figure 3.3: Graphical representation of milestone relation in DCR Tool

An example of milestone relation could be activities "Merge changes to the main branch", "Report merge conflict", and "Resolve merge conflict". "Merge changes to the main branch" can occur initially. But if "Report merge conflict" occurs, then "Resolve merge conflict" becomes pending. Then "Merge changes to main branch" can not occur until "Resolve merge conflict" is executed. This example is demonstrated in Figure 3.4.

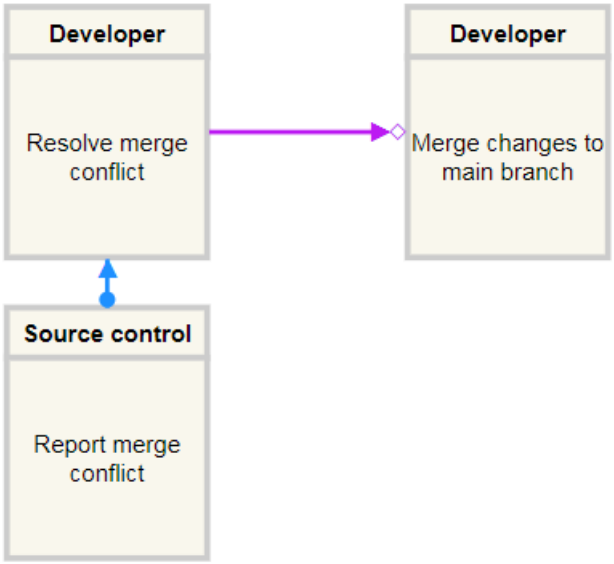

Figure 3.4: Example of milestone relation

Pre-condition

The pre-condition relation is a combination of condition and milestone relations, meaning that if an event A is pre-condition for an activity B, then A must be executed and not pending for B to be enabled. These relations are frequently used together for expressing business rules, so pre-condition relation was recently introduced to the graphical tool for convenience and is demonstrated in Figure 3.5.

Figure 3.5: Graphical representation of pre-condition relation in DCR Tool

An example of pre-condition relation could be activities "Fetch measurements from sensor", "Pair devices" and "Switch off Bluetooth on the smartphone", where "Pair devices" is a pre-condition for "Fetch measurements from sensor". It means that in order for the "Fetch measurements from sensor" activity to be enabled for execution, the "Pair devices" activity must be executed and can not be pending. If the "Switch off Bluetooth on the smartphone" activity occurs, it makes "Pair devices" pending, preventing "Fetch measurements from sensor" activity from execution until the devices are paired again. This example is demonstrated in Figure 3.6.

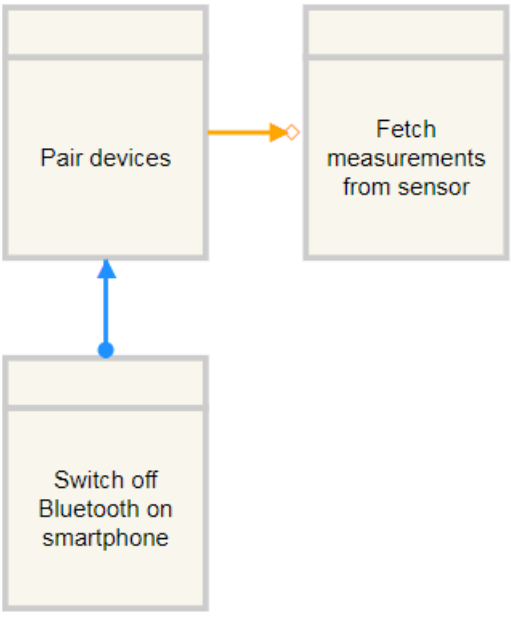

Figure 3.6: Example of pre-condition relation

Response

The response connection is a relation between activity A and activity B such that if A occurs, then B has to occur at least once at some point after. B can also occur even if A never occurs. The response connection is graphically represented as shown in Figure 3.7.

Figure 3.7: Graphical representation of response relation in DCR Tool

For example, if activity A is "Measurement device sends measures to the application", then activity B, "Application receives measurements", must take place at least once. This example is graphically represented as shown in Figure 3.8.

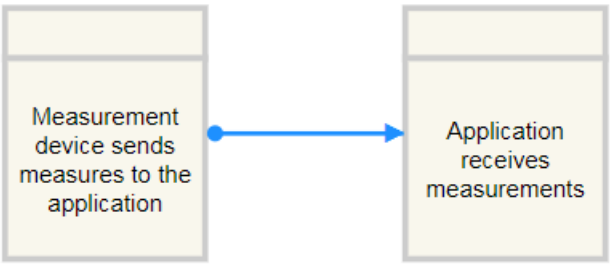

Figure 3.8: Example of response relation

No-Response

The no-response connection creates a relation between activity A and activity B such that B does not have to occur if A occurs. In other words, if activity A occurs, it removes the pending state of activity B. The no-response connection is graphically represented as shown in Figure 3.9.

Figure 3.9: Graphical representation of no-response relation in DCR Tool

An example of no-response relation is presented in Figure 3.10. It shows the activities "Hold meeting" and "Cancel meeting". Initially, the "Hold meeting" activity is pending (expressed with an exclamation mark). However, if the "Cancel meeting" activity occurs, it removes the pending state of the "Hold meeting" activity, so it does not need to be executed.

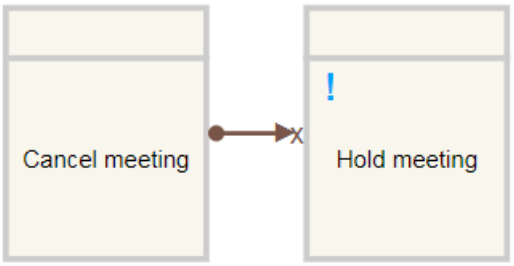

Figure 3.10: Example of no-response relation

Exclude

The exclude connection is a relation between activity A and activity B such that B cannot occur if first A has occurred. When A occurs, B becomes excluded from the set of included events. Activity B can become included again afterwards if activity with an include connection to B occurs. The exclude relation is graphically represented as shown in Figure 3.11.

Figure 3.11: Graphical representation of exclude relation in DCR Tool

Activity A could, for instance, be "Log in rejected" and B "Access personal information". This example is graphically represented as shown in Figure 3.12.

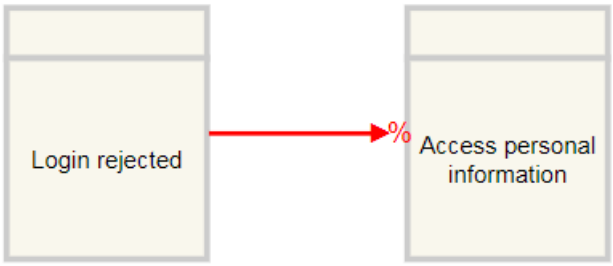

Figure 3.12: Example of exclude relation

Include

The include connection is a relation between activity A and activity B such that if activity A occurs, then activity B can occur if it was not previously included in the process. The include relation is graphically represented as shown in Figure 3.13.

Figure 3.13: Graphical representation of include relation in DCR Tool

For example, activity B could be "Access personal information", which has been previously excluded because of wrong login information. Activity A could be a new login trial, now with correct credentials. This example is graphically represented as shown in Figure 3.14.

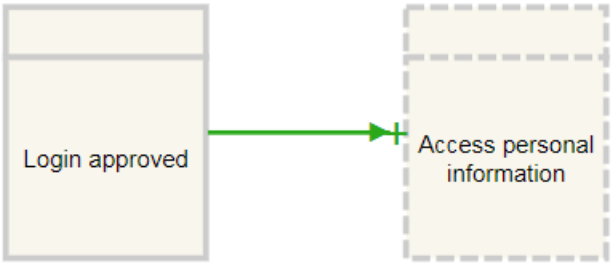

Figure 3.14: Example of include relation

Spawn

The spawn connection creates a relation between an activity A and a subactivity B such that, when A occurs, a new instance of B is created. The spawn relation is graphically represented as shown in Figure 3.15.

Figure 3.15: Graphical representation of spawn relation in DCR Tool

Activity B could, for instance, be "Document handling" and A be "Create new document". Execution of the "Create new document" activity will lead to the creation of a new instance of "Document handling activity" that in turn can contain other subactivities. This example is demonstrated in Figure 3.16.

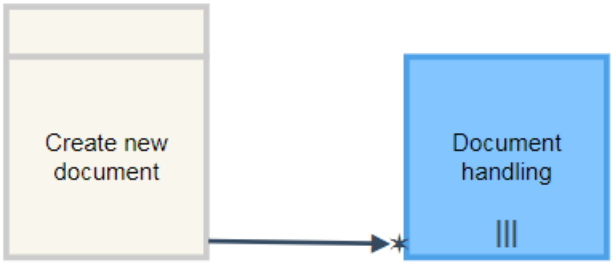

Figure 3.16: Example of spawn relation

3.3.3 Grouping of events

A useful extension of DCR Graphs is the possibility to logically group multiple activities under another activity. There are two supported ways to group activities: nesting and subprocesses.

Nesting is a way to syntactically avoid many connections in a graph since connections in and out of a parent activity go to and from each child activity nested inside it. It means that for a connection going out from a parent activity, all included child activities must be executed before the process can proceed.

Subprocesses, on the other hand, keep their own state. In order to proceed, it is required that the parent activity is in an accepting state. It means that if there is a connection going out from the parent activity, then this activity can not be pending for the process to proceed. However, it is not required that all included activities get executed, which is the case for nesting.

In the graphical tool, grouping is demonstrated by putting child activities "inside" parent activity, as shown in Figure 3.17. The difference between nesting and subprocess is indicated with an "n" or an "s" at the bottom corner of a parent activity.

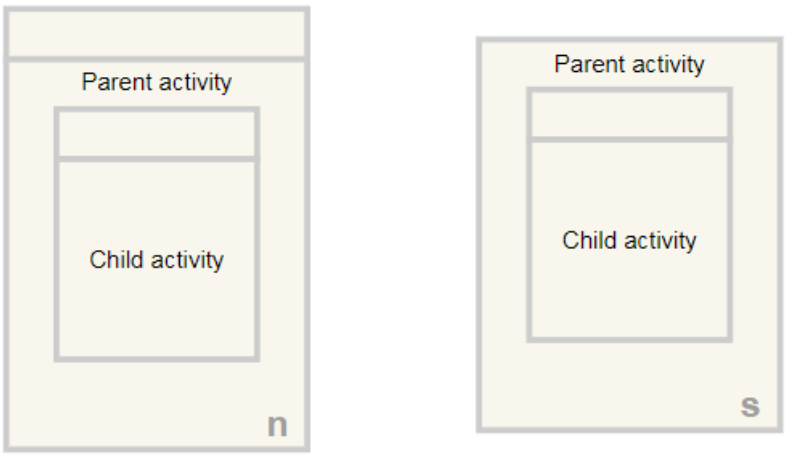

Figure 3.17: Graphical representation of grouping of events in DCR Tool

3.4 Formal semantics

The previous section provided an intuitive understanding and some simple examples of the structural components of DCR Graphs. However, formal modelling means that models are represented in a notation with formal semantics. Therefore, this section will describe the semantics of DCR Graphs in a formal way, giving the basis for constructing and reasoning about models of a real system.

First, Section 3.4.1 defines a basic DCR Graph. Then these definitions are extended to definitions of Distributed DCR Graph by adding roles and principals in Section 3.4.2. Finally, formal semantics of grouping of events are introduced in Section 3.4.3 by giving definitions of Nested DCR Graphs.

The definitions in this section are based on [38], but are extended to include the pre-condition and no-response relations, which were

introduced to the DCR Tool at a later point in time.

3.4.1 DCR Graphs

Definition 3.4.1 *A Dynamic Condition Response Graph is a tuple* $G = \langle E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, l\rangle$ *where*

- E *is the set of events*
- $M \subseteq E \times E \times E$ *is the marking and* $\mathcal{M}(G)$ *is the set of all markings*
- Act *is the set of actions*
- $\rightarrow\bullet \subseteq E \times E$ *is the condition relation.*
- $\bullet\rightarrow \subseteq E \times E$ *is the response relation.*
- $\rightarrow+, \rightarrow\% \subseteq E \times E$ *are the dynamic include relation and exclude relation, satisfying that* $\forall e \in E. e\rightarrow + \cap \rightarrow\% = \emptyset$
- $\rightarrow\diamond \subseteq E \times E$ *is the milestone relation*
- $\rightarrow\times \subseteq E \times E$ *is the no-response relation*
- $\rightarrow\bullet\diamond \subseteq E \times E$ *is the pre-condition relation such that* $\rightarrow\bullet\diamond = \rightarrow\bullet \cap \rightarrow\diamond$
- $l : E \rightarrow$ Act *is labelling function mapping every event to an action*

The condition relation $e\rightarrow\bullet e'$ means that event e is prerequisite for the event e'. In other words, it imposes precedence among the events.

The response relation $\bullet\rightarrow$ is a relation, such that $\{e' | e\bullet\rightarrow e'\}$ is the set of events that must happen after the event e has happened for the flow to be accepting.

The marking $M = \langle Ex, Re, In\rangle \in \mathcal{M}(G)$ consists of three sets of events: events that has previously been executed (Ex), events that are pending responses required to be executed or excluded (Re) and events that are currently included (In).

The dynamic inclusion and exclusion relations $\rightarrow+$ and $\rightarrow\%$ allow events to be dynamically included and excluded in the graph. Only the currently included events are considered in evaluating the constraints. It means that if an event e' has event e as a condition, but the event e is excluded from the graph, then it is no longer required to execute e for e' to happen. Also, if event e has event e' as response and if the event e' is excluded, then it is no longer required to happen for the flow to be accepting. Formally, $e\rightarrow + e'$ means that when event e occurs, it will include e' in the graph. $e\rightarrow\%e'$ expresses that when e occurs it will exclude e' from the graph.

The milestone relation $\rightarrow\diamond$ is, like condition relation, blocking. However, the milestone relation blocks based on events in the pending response set. For example, if an event e' has the event e as milestone (written as $e\rightarrow\diamond e'$), then event e' is not allowed to execute, if the event e is in the set of

pending responses (Re). As for the condition relation, the milestones are blocking only if they are included in the graph.

The no-response relation $\rightarrow\times$ is a relation opposite to $\bullet\rightarrow$. For example if an event e has no-response relation to an event e' (written as $e\rightarrow\times e'$), the event e will cancel the pending state of the event e'. In other words, if event e occurs, then event e' will be removed from the set of pending events (Re).

The pre-condition relation $\rightarrow\bullet\diamond$ is a relation that combines condition relation $\rightarrow\bullet$ and milestone relation $\rightarrow\diamond$ in a sense that $\rightarrow\bullet\diamond$ will always be an intersection of $\rightarrow\bullet$ and $\rightarrow\diamond$. For example, if there are two events e and e' such that $e\rightarrow\bullet e'$ and $e\rightarrow\diamond e'$, then $e\rightarrow\bullet\diamond e'$.

We will now formally define what it means that an event is enabled for execution and which conditions must be satisfied.

Definition 3.4.2 *For a Dynamic Condition Response Graph* $\mathrm{G} = \langle \mathrm{E}, \mathrm{M}, \mathrm{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, l\rangle$ *with marking* $\mathrm{M} = \langle \mathrm{Ex}, \mathrm{Re}, \mathrm{In}\rangle$*, an event* $e \in \mathrm{E}$ *is enabled (written* $\mathrm{M} \vdash_{\mathrm{G}} e$*) if*

- $e \in \mathrm{In}$
- $(\rightarrow\bullet e \cap \mathrm{In}) \in \mathrm{Ex}$
- $(\rightarrow\diamond e \cap \mathrm{In}) \in \mathrm{E} \setminus \mathrm{Re}$

The definition tells that for an event e to be enabled, it has to be included in the graph, all the included events which are condition to the event e must be in the set of executed events, and none of the included events that are milestones for it are in the set of pending responses.

When an event is enabled, it means that this event is ready to be executed. Execution of an event results in changes in the marking, which are formally defined as follows.

Definition 3.4.3 *For a Dynamic Condition Response Graph* $\mathrm{G} = \langle \mathrm{E}, \mathrm{M}, \mathrm{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, l\rangle$ *with marking* $\mathrm{M} = \langle \mathrm{Ex}, \mathrm{Re}, \mathrm{In}\rangle$ *and with en enabled event* $\mathrm{M} \vdash_{\mathrm{G}} e$*, the result of executing the event* e *will be a dynamic condition response graph* $\mathrm{G} = \langle \mathrm{E}, \mathrm{M}', \mathrm{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, l\rangle$*, where* $\mathrm{M}' = \mathrm{M} \oplus_{\mathrm{G}} e = \{\mathrm{Ex}', \mathrm{Re}', \mathrm{In}'\}$ *such that*

- $\mathrm{Ex}' = \mathrm{Ex} \cup \{e\}$
- $\mathrm{Re}' = (\mathrm{Re} \setminus (\{e\} \cup e\rightarrow\times)) \cup e\bullet\rightarrow$
- $\mathrm{In}' = (\mathrm{In} \cup e\rightarrow+) \setminus e\rightarrow\%$

The definition says that the event which gets executed is added to the set of executed events (Ex) and removed from the set of pending responses (Re). Then, all events that are a response to the event are added to the set of pending responses. If the event is a response to itself, it will remain in the set of pending responses after execution. All the events that event e has no-response relation to are removed from the set of pending events (Re). Set of included events (In) is also updated by including and excluding events

that have include and exclude relation from the executed event. Further, an event e' can not be both included and excluded by the same event e. Finally, an event e may have a relation to itself.

Series of several events being executed form runs or executions which can either be finite or infinite. Executions are called accepting if any required, included response in any intermediate marking is eventually executed or excluded. Concepts of runs/executions and accepting runs/executions are formally defined in the following definition.

Definition 3.4.4 *An execution of a dynamic condition response graph* $\mathrm{G} = \langle \mathrm{E}, \mathrm{M}, \mathrm{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, l \rangle$ *is a (finite or infinite) sequence of tuples* $\{\langle \mathrm{M}_i, e_i, a_i, \mathrm{M}'_i \rangle\}_{i \in [k]}$ *each consisting of a marking, an event, a label and another marking (the result of executing an event) such that*

- $\mathrm{M} = \mathrm{M}_0$
- $\forall_i \in [k].a_i \in l(e_i)$
- $\forall_i \in [k].\mathrm{M}_i \vdash_{\mathrm{G}} e_i$
- $\forall_i \in [k].\mathrm{M}'_i = \mathrm{M}_i \oplus_{\mathrm{G}} e_i$
- $\forall_i \in [k-1].\mathrm{M}'_i = \mathrm{M}_{i+1}$

The execution (or a run) is accepting if $\forall_i \in [k].(\forall e \in \mathrm{In}_i \cap \mathrm{Re}_i.\exists j \geq i.e_j = e \vee e \notin In'_j)$*, where* $\mathrm{M}_i = \langle \mathrm{Ex}_i, \mathrm{In}_i, \mathrm{Re}_i \rangle$ *and* $\mathrm{M}'_j = \left\langle \mathrm{Ex}'_j, \mathrm{In}'_j, \mathrm{Re}'_j \right\rangle$*. In words, a run is accepting if no required response event is continuously included and pending without it happens or become excluded. A marking* M' *is reachable in* G *(from the marking* M*) if there exists a finite execution ending in* M'*.* $\mathcal{M}_{\mathrm{M}\rightarrow *}(\mathrm{G})$ *denotes the set of all reachable markings from* M*.*

3.4.2 Distributed DCR Graphs

When modelling workflows, it may sometimes be interesting to include information about who executes a particular action. For that purpose, DCR Graphs can be extended to Distributed DCR Graphs by adding roles and principals as the following definition describes.

Definition 3.4.5 *A Distributed Dynamic Condition Response Graph is a tuple* $\mathrm{DG} = \langle \mathrm{G}, \mathrm{Roles}, \mathrm{P}, \mathrm{as} \rangle$ *where*

- $\mathrm{G} = \langle \mathrm{E}, \mathrm{M}, \mathrm{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, l \rangle$ *is a Dynamic Condition Response Graph,*
- Roles *is a set of roles,*
- P *is a set of principals (e.g. persons or processors)*
- $\mathrm{as} \subseteq (\mathrm{P} \cup \mathrm{Act}) \times \mathrm{Roles}$ *is the role assignment relation to principals and actions.*

The role assignment relation indicates the access rights (roles) assigned to principals and which roles give the right to execute which actions.

Some concepts defined for DCR Graphs in the previous section must now be extended so that roles and principles are taken into account. For example, for an event to be enabled, it is no longer enough for it to satisfy conditions in Definition 3.4.2.

Definition 3.4.6 *For a Distributed Dynamic Condition Response Graph* $\mathrm{DG} = \langle \mathrm{G}, \mathrm{Roles}, \mathrm{P}, \mathrm{as} \rangle$ *with* $\mathrm{G} = \langle \mathrm{E}, \mathrm{M}, \mathrm{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, l \rangle$, *an event e is enabled (written* $\mathrm{M} \vdash_{\mathrm{DG}} e$*), if* $\mathrm{M} \vdash_{\mathrm{G}} e$*, (p as r) and (a as r).*

When it comes to executing an enabled event, the result of execution will be the same in a Distributed DCR Graph as in DCR Graph since event execution only involves changes to the marking, which are not affected by the roles and principals. It is formally defined in the following definition.

Definition 3.4.7 *For a Distributed Dynamic Condition Response Graph* $\mathrm{DG} = \langle \mathrm{G}, \mathrm{Roles}, \mathrm{P}, \mathrm{as} \rangle$ *with* $\mathrm{G} = \langle \mathrm{E}, \mathrm{M}, \mathrm{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, l \rangle$ *with* $\mathrm{M} \vdash_{\mathrm{DG}} e$*, executing event e in* DG *will have the same effect as that of executing the event e in the underlying DCR Graph* G*. The resulting marking will be the same in both cases.*

A run or an execution in Distributed DCR Graphs and when it is accepting is defined as follows.

Definition 3.4.8 *For a Distributed Dynamic Condition Response Graph* $\mathrm{DG} = \langle \mathrm{G}, \mathrm{Roles}, \mathrm{P}, \mathrm{as} \rangle$ *with* $\mathrm{G} = \langle \mathrm{E}, \mathrm{M}, \mathrm{Act}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, l \rangle$ *with marking* $\mathrm{M} = \langle \mathrm{Ex}, \mathrm{Re}, \mathrm{In} \rangle$*, a run (finite or infinite) is a sequence of labels* $(e_0, (p_0, a_0, r_0))(e_1, (p_1, a_1, r_1))\ldots$ *of a sequence of transition* $\mathrm{M}_i \xrightarrow{(e_i,(p_i,a_i,r_i))} \mathrm{M}_{i+1}$ *for* $i \geq 0$ *starting from initial marking such that* $\mathrm{M}_i \vdash_{\mathrm{DG}} e_i$ *and* $\mathrm{M}_{i+1} = \mathrm{M}_i \oplus_{\mathrm{DG}} e_i$*. A run is accepting if its underlying DCR Graph* G *is accepting i.e.* $\forall_i \in [k].(\forall e \in \mathrm{In}_i \cap \mathrm{Re}_i.\exists j \geq i.e_j = e \vee e \notin \mathrm{In}'_j)$.

3.4.3 Nested DCR Graphs

When a DCR Graph contains many events and relations between them, it may be desirable to logically group the events. This section provides formal semantics of Nested DCR Graphs that allow doing that. Before defining Nested DCR Graphs, definitions of DCR Graph (Definition 3.4.1) and Distributed DCR Graphs (Definition 3.4.5) are generalised to a definition of DCR Graphs that abstracts away from roles and principals.

Definition 3.4.9 *A Dynamic Condition Response Graph (DCR Graph)* G *is a tuple* $\langle \mathrm{E}, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, \mathrm{L}, l \rangle$ *where*

- E *is the set of events (or activities)*
- $\mathrm{M} = \langle \mathrm{Ex}, \mathrm{Re}, \mathrm{In} \rangle \in \mathcal{M}(\mathrm{G})$ *is the marking*
- $\rightarrow\bullet \subseteq \mathrm{E} \times \mathrm{E}$ *is the condition relation.*

- $\bullet\rightarrow \subseteq \mathrm{E} \times \mathrm{E}$ *is the response relation.*
- $\rightarrow+, \rightarrow\% \subseteq \mathrm{E} \times \mathrm{E}$ *is the dynamic include relation and exclude relation, satisfying that* $\forall e \in \mathrm{E}. e\rightarrow + \cap \rightarrow\% = \emptyset$
- $\rightarrow\diamond \subseteq \mathrm{E} \times \mathrm{E}$ *is the milestone relation*
- $\rightarrow\times \subseteq \mathrm{E} \times \mathrm{E}$ *is the no-response relation*
- $\rightarrow\bullet\diamond \subseteq \mathrm{E} \times \mathrm{E}$ *is the pre-condition relation such that* $\rightarrow\bullet\diamond = \rightarrow\bullet \cap \rightarrow\diamond$
- L *is the set of labels*
- $l : \mathrm{E} \rightarrow \mathcal{P}(\mathrm{L})$ *is labelling function mapping events to sets of labels*

In words, each event is mapped to the set of labels, which can consist of a name of the event and a role which defines who can execute that event.

This definition will now be extended to the definition of Nested DCR Graphs by allowing events to be grouped under a super-event.

Definition 3.4.10 *A Nested Dynamic Condition Response Graph (Nested DCR Graph)* G *is a tuple* $\langle \mathrm{E}, \triangleright, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, \mathrm{L}, l\rangle$ *where*

- $\langle \mathrm{E}, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, \mathrm{L}, l\rangle$ *is a DCR Graph*
- $\triangleright : \mathrm{E} \rightharpoonup \mathrm{E}$ *is a partial function mapping an event to its super-event (if defined)*
- $\mathrm{M} \in \mathcal{P}(\mathrm{atoms}(\mathrm{E})) \times \mathcal{P}(\mathrm{atoms}(\mathrm{E})) \times \mathcal{P}(\mathrm{atoms}(\mathrm{E}))$, *where* $\mathrm{atoms}(\mathrm{E}) = \mathrm{E} \setminus \{e \in \mathrm{E} | \exists e' \in \mathrm{E}. \triangleright (e') = e\}$ *is the set of atomic events*

We write $e \triangleright e'$ *if* $e' = \triangleright^k(e)$ *for* $0 < k$, $e \trianglerighteq e'$ *if* $e \triangleright e'$ *or* $e = e'$, *and* $e \trianglelefteq e'$ *if* $e' \triangleright e$ *or* $e = e'$. *The resulting relation,* $\trianglerighteq \subset E \times E$, *referred to as the nesting relation, must be a well founded partial order. The nesting relation must also be consistent with respect to dynamic inclusion/exclusion in the following sense: if* $e \triangleright e'$ *or* $e' \triangleright e$, *then* $e\rightarrow + \cap e'\rightarrow\% = \emptyset$

To define the execution semantics for Nested DCR Graph, we first need to define how to flatten a nested graph to a simpler DCR Graph.

Definition 3.4.11 *For a Nested Dynamic Condition Response Graph* $\mathrm{G} = \langle \mathrm{E}, \triangleright, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, \mathrm{L}, l\rangle$ *define the underlying flat Dynamic Condition Response Graph as*

$$\mathrm{G}^b = (\mathrm{atoms}(\mathrm{E}), \mathrm{M}, \rightarrow\bullet^b, \bullet\rightarrow^b, \rightarrow\diamond^b, \rightarrow+^b, \rightarrow\%^b, \rightarrow\times^b, \rightarrow\bullet\diamond^b, \mathrm{L}, l)$$

where $rel^b = \trianglerighteq rel \trianglelefteq$ *for some relation* $rel \in \{\rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \rightarrow\times, \rightarrow\bullet\diamond\}$

Using the corresponding flat graph, we can now define when an event is enabled and the result of executing an event for Nested DCR Graphs.

Definition 3.4.12 *For a Nested Dynamic Condition Response Graph* $\mathrm{G} = \langle \mathrm{E}, \triangleright, \mathrm{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, \mathrm{L}, l \rangle$ *where* $\mathrm{M} = \langle \mathrm{Ex}, \mathrm{Re}, \mathrm{In} \rangle$*, an event* $e \in \mathrm{atoms}(\mathrm{E})$ *is enabled, written* $\mathrm{M} \vdash_{\mathrm{G}} e$*, if* $\mathrm{M} \vdash_{\mathrm{G}^b} e$*. Similarly, the result of executing* $\mathrm{M} \oplus_{\mathrm{G}} e$ *is the same as the result executing the event in flattened graph and is defined as:* $\mathrm{M} \oplus_{\mathrm{G}^b} e = \langle \mathrm{Ex}, \mathrm{Re}, \mathrm{In} \rangle \oplus_{\mathrm{G}^b} e$

A run/execution in a Nested DCR Graph and the conditions for the run/execution to be accepting are the same as for underlying flat DCR Graph.

3.5 DCR process methodology

This section addresses the methodology for the modelling of a process as a DCR Graph and different ways of verifying correction of the model available in the DCR Tool.

In order to construct a model of a process as a DCR Graph, one must first identify the actions that can be performed (activities) and who can perform them (roles). Next, the rules describing order, precedence, dependencies and other forms of restrictions among activities can be defined.

When the information about roles, activities and rules is collected, a graphical representation of the DCR Graph can be drawn using the DCR Tool. First, the activities are added. For each activity, the DCR Tool allows to add a title, assign one or several roles and choose whether the activity is initially included, pending or executed, as well as to write an additional description of the activity if needed for documentation purpose. Then, the rules that activities must conform to are expressed as relations described in Section 3.3.2. For the relations, it is also possible to add a description if desired.

The goal and purpose of creating a model are often to verify some properties that should be valid. For DCR Graphs, it means checking that the process expressed by the graph behaves as expected. Verifying the final and complete graph is most important. However, for larger and more complicated processes, it might also be a good idea to do some checking during the graph creation process. One way to do so is by creating scenarios.

A scenario is a sequence of activities representing a path through the process. Scenarios can either describe desirable or undesirable (invalid) behaviours of the process.

The DCR Tool provides two different ways of creating scenarios. The first one is by using DCR Swimlane Editor. DCR Swimlane Editor allows to drag and drop activities one by one, so they form a path from beginning to the end of the process. For a scenario, it is possible to add a title and description, as well as to choose whether the scenario is required (desired behaviour, called "happy path" in the DCR Tool) or forbidden (undesired behaviour). It is then possible to validate the scenario against created DCR Graph in order to check that the graph conforms to defined rules.

Another way of creating scenarios in the DCR Tool is by using DCR Simulator. Compared to DCR Swimlane Editor, DCR Simulator allows for collaborative simulation of the graph with co-workers or so-called machine users. When executing activities in DCR Simulator, the state of the graph is updated after each step. It shows which activities are included, pending or executed and if the execution is accepting. User can save a simulation as a scenario.

DCR Swimlane Editor and DCR Simulator are useful tools to test some particular executions. However, verification using scenarios has a disadvantage similar to the disadvantage of writing tests for computer programs: only a limited number of situations gets executed. Especially for more complex processes that involve many activities, it would not be feasible to consider all possible orders in which the activities can be executed. Therefore, using DCR Swimlane Editor and DCR Simulator alone is insufficient to verify the DCR Graph.

The DCR Tool itself does not provide other inbuilt functionalities for this purpose. However, the DCR Community provides an App Store containing multiple applications that can be added to the DCR Tool. When it comes to verification goal, there are primarily two applications that are worth the attention.

One of them is the Scenario Search application. Figure 3.18 demonstrates the application's functionality, which is twofold. One option is to search for a specific path (called "happy path") through the process starting from one activity and ending in another. It also allows specifying activities to be used or avoided during the execution. If one or several paths conforming to the specified parameters exist, the application returns a swimlane diagram showing an example of such a path. Otherwise, a message notifying about the absence of such paths is returned. Another option, which can be accessed by choosing "Full" instead of "Happy path", returns a diagram showing all possible transitions in the DCR Graph.

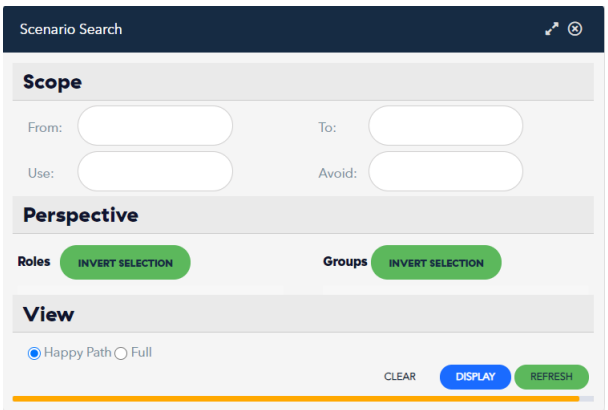

Figure 3.18: Scenario Search application window

The second application is called Dead-end Analyzer. As the name suggests, this application can be used to check whether the graph contains paths leading to a state where the goal of the process can never be reached.

If such a path exists, the Dead-end Analyzer returns a swimlane diagram as an example. Otherwise, it returns a message stating that the graph can not reach a dead-end.

The Scenario Search and Dead-end Analyzer applications have been useful tools for verification purpose in this thesis. More details on their use, as well as their limitations, are discussed later in the sections dedicated to modelling and verification of this thesis' use case.

4 Tellu Diabetes App as the use case

As mentioned in the introduction, the use case chosen for this thesis is Tellu Diabetes App. This chapter provides all the information about the use case needed for modelling and verification in the following chapters. First, Sections 4.1-4.4 explain the overall architecture, technology choices and functionality of the application. The information presented in these sections is collected from the deliverable from the Secure Connected Trustable Things (SCOTT) project [9]. The description will serve as a foundation for modelling of Tellu Diabetes App in Chapter 6. Then, Section 4.5 outlines the process of introducing changes in the application code supported by the tools used by the development team in their daily work. The described process presents a usual implementation process flow based on studying the possibilities of these tools. This process will be modelled and verified in Chapter 5.

Tellu Diabetes App is a mobile application for monitoring diabetes patients. The main user group of the application is elderly people with diabetes type 2. Patients with more severe cases use insulin or other drugs, which introduce the danger of running too low on blood glucose. In addition, elderly with diabetes have an increased risk of heart conditions, and they often have high blood pressure. Diabetes App should therefore help users monitor weight, blood pressure and blood glucose levels using measurements from Bluetooth Low Energy (BLE) devices.

4.1 Architecture

The overall system architecture consists of three main parts: medical BLE devices, an application running on a phone or tablet and Tellu's backend service TelluCloud which is used for user authentication and storage of measurements from devices. The medical BLE devices communicate with the application over Bluetooth, while the communication between the application and TelluCloud happens over the Internet. The architecture is graphically demonstrated in Figure 4.1.

4.2 Application functionality

Tellu Diabetes App has a number of functional requirements, i.e. functions that the application must provide. The main functionalities of the Tellu Diabetes App include adding new measurements (either automatically by

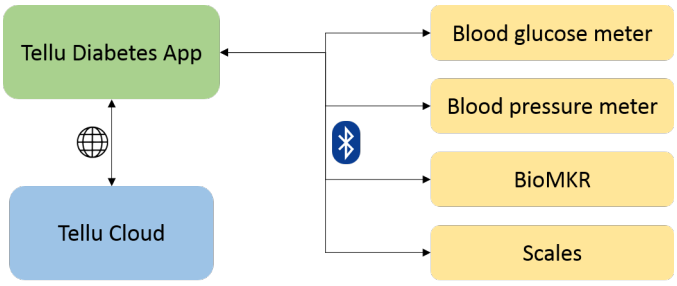

Figure 4.1: Overall architecture of Tellu Diabetes App

transmitting them from the BLE device or manually by filling out the form) and viewing user's previous measurements. Furthermore, to ensure that the data can only be accessed by the user it belongs to, the application must provide login and logout functionality. These features can be expressed as following use cases:

- As a user of the application, I want to get measurements from measurement devices, so I do not need to enter them manually.
- As a user of the application, I want to be able to enter measurements manually in case I have trouble connecting measurement devices.
- As a user of the application, I want to see my previous measurements in order to compare them.
- As a user of the application, I want to get guidance on how to pair a medical device to my mobile phone/tablet in order to get started with using the application easily.
- As a user of the application, I want to be reminded of when I need to make new measurements, so I do not forget it.
- As a user of the application, I want to be sure that my measurements are only visible to me and the medical personal.

Considering the main user group of the application, it is essential that it also satisfies a non-functional requirement related to usability. The application must be intuitive to use and provide clear guidance and feedback to the user.

A typical workflow of the application can be described in the following steps:

1. User authenticates himself in the application.
2. BLE device is paired with the phone/tablet. This step is only required the first time the measurement device is used with a new phone/tablet. After that, the devices will be paired automatically as long as the phone/tablet has Bluetooth switched on. Some guidance on pairing the devices should be provided in application since this process can be a little tricky, especially for elderly users.

3. User takes measurements using BLE devices.

4. Measurements are transmitted to the mobile application. In addition to getting data over Bluetooth connection, the application must allow manual input in case of connection failures or for a device without Bluetooth. This step can be seen as the main step in the process since it will be repeated many times. It is, therefore, important to ensure that the process of measuring and receiving data is as intuitive as possible and that status of the process is communicated to the user.

5. Measurements are posted to TelluCloud where they are persisted and can be accessed, for example, by medical personal.

6. User of the application can also choose to look at the history of previous measurements. In this case, the application fetches data stored at TelluCloud and displays it to the user.

4.3 Devices

As previously mentioned, the application should support several measurement types such as weight, blood pressure and glucose levels. Tellu has worked with several devices to cover all required measurements.

- Blood glucose meter Contour next ONE, which measures blood glucose in blood applied to a test strip inserted in the device.
- Blood pressure meter AD Medical UA-651BLE
- Weight scale AD Medical UC-352BLE

All devices are Bluetooth enabled so that measurements can be transmitted to mobile application over Bluetooth.

4.4 Technology

One of the requirements for the Diabetes App is support for both Android and iOS platforms since both have a significant market share. To avoid implementing two different application for those platforms, Tellu chose to use Xamarin Forms as the framework for application development. This way the most of the code can be shared between platforms, except for the platform-specific parts dealing with Bluetooth communication.

4.5 Process of task implementation

The team developing Tellu Diabetes App uses a variety of tools to support the development process. Examples of the tools relevant for the modelling process are Jira [1], Bitbucket [2] and CI/CD tool for executing automated build and testing.

Jira is software that is used to keep track of all tasks that must be done. It also makes project management more accessible by providing intuitive and visual tools for project planning, prioritising tasks and following the progress. Bitbucket, in its turn, is a code management tool, which allows team members to share and collaborate on making changes in the code. Finally, CI/CD tool (or build system) helps to automate the process of building application and running unit and integration tests found in the codebase. All the tools can be integrated with each other in order to increase traceability. For example, the tasks in Jira can be directly connected to a development branch in Bitbucket, which makes it convenient to track the changes related to precisely that task. Furthermore, the build tool can be set up to listen to changes in the Bitbucket repository, so each build corresponds to specific changes in the code.

The usual flow of a task implementation process is graphically demonstrated in Figure 4.2 and can be described as follows. A task (which can, for example, be a bug to be fixed or a new feature to be implemented) is put in the project backlog, i.e. a list containing all tasks that require to be done. Each task in the list is typically called a ticket. This list should be prioritised by the severity and urgency of its tasks.

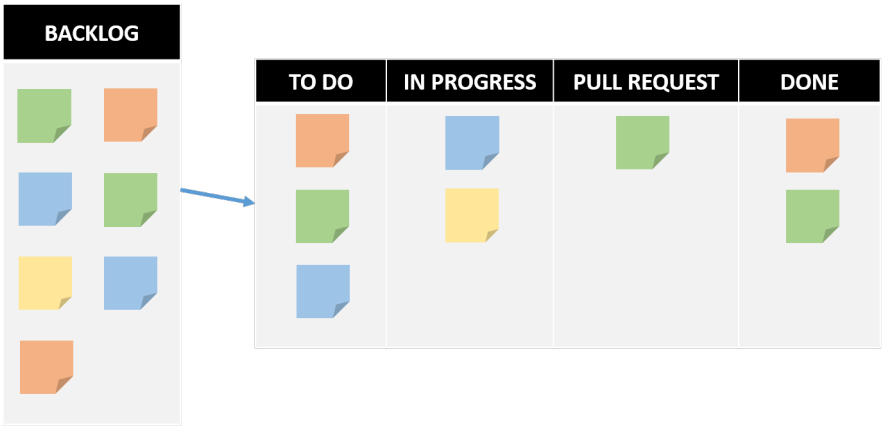

Figure 4.2: Flow of a Jira ticket through the task implementation process

While backlog contains all the tasks to be done, another project management tool is needed to keep track of workflow in the current period of time (like a single sprint in Scrum). This tool is called the board and consists of several columns with tickets being moved between columns as their status changes. A typical board includes columns like TO DO, IN PROGRESS, PULL REQUEST and DONE.

When a task from backlog is chosen for development, it is initially put to the TO DO column. When a developer picks a task to solve or is assigned to it by the project manager, the ticket is moved to the IN PROGRESS column. The developer creates a separate branch where he can add new code or modify existing code. Each logical unit of work forms a commit. Changes are committed to the local repository and are pushed to the remote repository at Bitbucket, where other team members can access them.

When the changes are pushed to the remote repository, the automated build process that builds the application and runs unit and integration tests is triggered. The build process is essential to ensure that new changes in the code did not break any existing features by checking that the application still builds and the tests still pass. When the build process is finished, the status is reported. The status can either be a failed or a successful build. The former indicates some errors during compilation, assembling or testing of the code and means that additional measures should be taken to fix it. It usually involves making changes in the code locally, committing and pushing them to the remote repository again.

While the build process helps to verify the changes dynamically by running tests, the code should be verified statically as well. Thus, when the developer considers his task as ready to be merged to the main branch, he must create a pull request and ask a team member to approve the changes. This process is called peer review (or code review) and is performed by the reviewer going through the modifications and searching for errors and potential improvements in the code. The peer can write some comments explaining which parts of the code should be changed and why. At the end of the process, the peer can approve the pull request if the code looks good or request changes if there is something to improve.

Only when the changes are statically and dynamically verified (i.e. approved by a peer and pass the automated tests), they can be merged into the main branch.

Sometimes, two or more developers may need to modify the same part of the code during the implementation of their tasks. When one of the developers merges in his changes, the second developer's changes may conflict with newly added modifications. In this case, the code management system will inform the second developer about the arisen merge conflict and disallow proceeding with merging until the conflict is resolved.

5 Modelling and verification of task implementation process

This chapter is dedicated to the work done on modelling and verification of the task implementation process with the purpose of satisfying safety standard requirements concerned with code modification control. The model and properties verified are based on the process and tools as described in Section 4.5.

5.1 Properties

When it comes to the task implementation process, two important properties should be satisfied by the process.

1. Any code changes merged into the main branch must be quality checked (peer-reviewed and tested)
2. A task that is chosen for development should eventually be completed

5.2 Model

Before the specified properties can be verified, we must construct the model first. The final model describes the process of implementing a single task from the point it gets chosen for development until the solution is merged into the main branch.

The modelling process was carried out in several iterations. The first versions of the model were reworked and adjusted to fix errors in the relations between activities and achieve an appropriate level of abstraction and complexity. The model in this section presents the final version. Please refer to Section 5.2.5 for some of the considerations taken into account during the early phases of the modelling process and background for the final decisions.

Following the DCR process methodology described in Section 3.5, the first step in the modelling process is defining the activities executed during the task implementation process and who executes them.

5.2.1 Activities

The task implementation process described in Section 4.5 can be summarised by the following list of main activities and their subactivities.

- Project manager moves a Jira ticket from BACKLOG to TO DO in the current sprint
- Developer picks (or is assigned to) the ticket from Jira board and moves it from TO DO to IN PROGRESS
- Developer creates a new branch
- Developer solves the task
  - Developer makes changes in the code
  - Developer writes tests and runs them
  - Developer commits the changes
  - Developer pushes changes (the newly created branch) to the remote repository
- Build process performed by the build system
  - Build system recognises that there are new changes, builds the application and runs automated tests
  - Build system reports build status (successful or unsuccessful)
- Peer review process
  - Developer creates a pull request
    - Developer asks colleague/peer for review
    - Developer moves the Jira ticket from IN PROGRESS to PULL REQUEST
  - Peer can approve the pull request
    OR
  - Peer can request changes in the code
- Task completion
  - Developer merges the branch into the main branch
  - Developer moves the Jira ticket from PULL REQUEST to DONE

5.2.2 Roles

Following is a list of all roles that appeared in the activities defined above.

- Project manager
- Developer
- Peer
- Build system

5.2.3 Rules

Next, the rules that the activities must conform to are defined. These rules describe order, precedence, dependencies and other forms of restrictions among activities. The rules will later be expressed as relations in the DCR Graph.

For the task implementation process, the following rules were identified in addition to the properties which were already defined in Section 5.1.

- The task must be included in the current sprint before the developer can pick it
- New changes must be done on a new branch
- The code must contain changes since the last commit before a new commit can be executed
- There must be commits on a branch before the developer can push the changes to the remote repository
- Build system (on feature branch) is only triggered when new commits are pushed to remote branch
- Result of the build can either be success or failure, not both
- A pull request can only be created if there are new changes that were pushed to the remote branch
- Once a pull request is created, the developer can not create new pull requests – only the same pull request is updated
- A peer can not do code review before he is asked to do so
- A peer can either approve the pull request or request changes

5.2.4 Model as DCR Graph

From defined roles, activities and rules, we construct the DCR Graph modelling task implementation process as graphically demonstrated using the DCR Tool as shown in Figure 5.1.

We will now explain the typical relation patterns used in the graph to demonstrate how some of the above rules are converted to relations in a DCR Graph.

As we see in the final model, multiple activities have an exclude relation (red arrow) to themselves. One of them is the "Move task from BACKLOG to TO DO" activity shown in Figure 5.2. It means that when the "Move task from BACKLOG to TO DO" activity is executed, it is added to the set of executed events and removed from the set of included events (by Definition 3.4.3). Since this activity is not included, it can not be enabled for execution unless it is included again (according to Definition 3.4.12). However, the "Move task from BACKLOG to TO DO" activity does not have any include relations going to it, meaning that it can not be executed for the rest of the

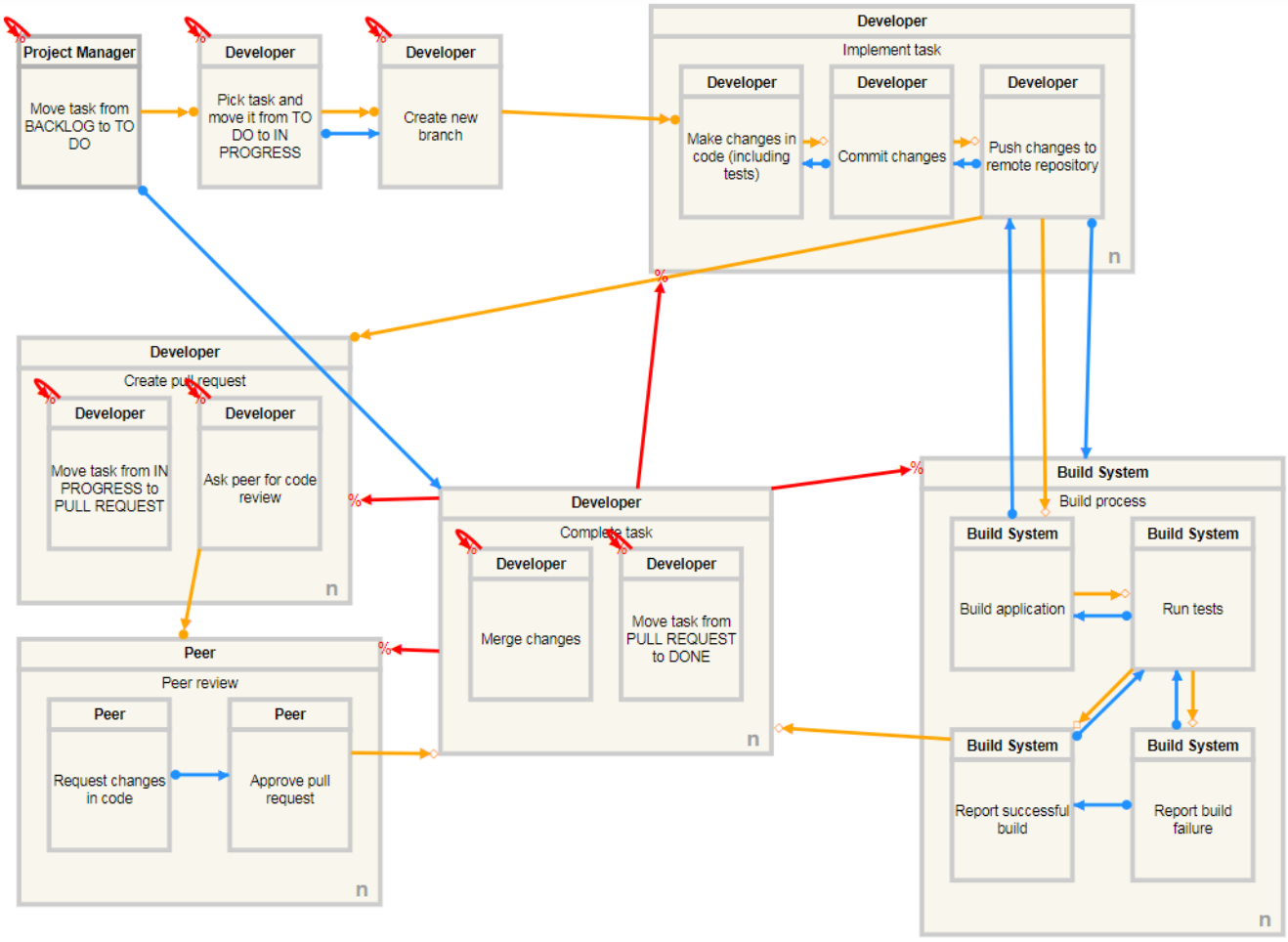

Figure 5.1: Task implementation process modelled as DCR Graph

process. In other words, an activity with an exclude relation to itself (and without include relation from other activity) represents that the activity can only be executed once in the entire process.

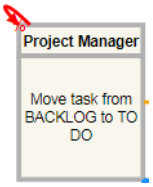

Figure 5.2: Activity with exclude relation to itself

Another relation pattern used in the model is the combination of pre-condition and response relations. Two variants of this combination are presented in the graph in Figure 5.1. The first variant includes two activities with pre-condition relation (yellow arrow) going from the first activity to the second and response relation (blue arrow) going back from the second

activity to the first one. An example shown in Figure 5.3 includes the activities "Make changes in code" and "Commit changes". As we can recall from Chapter 3, pre-condition relation is a combination of condition and milestone relations. "Make changes in code" being a condition for "Commit changes" means that the "Commit changes" activity will not be enabled for execution until the "Make changes in code" activity is in the set of executed events. This relation expresses that one activity must precede the second activity. After the "Make changes in code" activity is executed once it will stay in the set of executed events for the rest of the execution. Thus, the "Commit changes" activity can be executed at any point in time. However, it would lead to undesired behaviour since there must be new changes in the code before the developer can make a commit. Therefore, we use a combination of response and milestone relation to ensure that the activities' precedence is preserved for the rest of execution after the "Make changes in the code" activity has occurred once. The response relation from the "Commit changes" activity means that it will put the "Make changes in code" activity in the set of pending events. The milestone relation will then prevent "Commit changes" from being executed. When new changes are introduced (i.e. "Make changes in code" activity is executed again), it is removed from the set of pending events, so the "Commit changes" activity can occur.

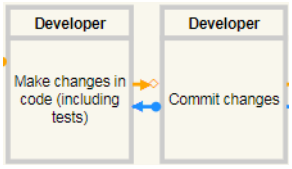

Figure 5.3: Combination of pre-condition and response relation between two activities

The second variant of the combination of pre-condition and response relations consists of three activities. It is demonstrated in Figure 5.4 with activities "Request changes in code", "Approve pull request" and "Complete task". The condition part of the pre-condition relation between "Approve pull request" and "Complete task" expresses that a task can not be completed unless the peer approves the pull request. However, if the peer changes his mind and requests changes in the code, the "Approve pull request" activity will become pending (due to response relation between these activities). The milestone part of the pre-condition relation between "Approve pull request" and "Complete task" activities will then ensure that "Complete task" can not happen until "Approve pull request" has happened again.

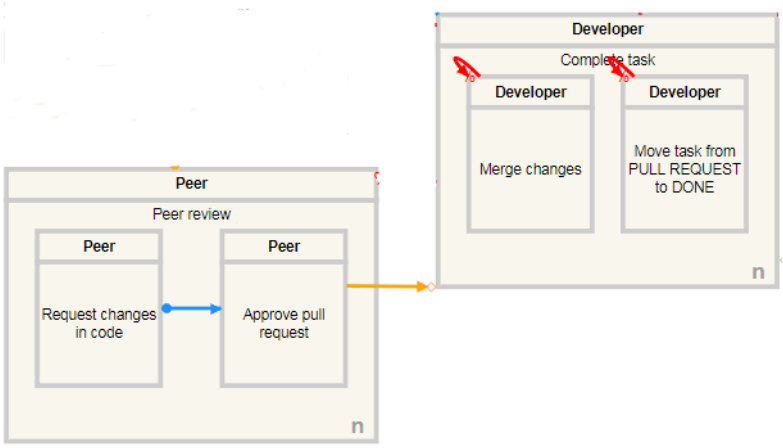

Figure 5.4: Combination of pre-condition and response relation between three activities

According to Definition 3.4.10 of a Nested DCR Graph, the same graph is formally defined as shown in listing 5.1. Note that we introduce the following abbreviations for roles and activity names for convenience: Project manager (PM), Developer (D), Build system (BS), Peer (P), "Move task from backlog to TODO" (moveTaskBacklogTodo), "Pick task and move it from TODO to IN PROGRESS" (pickTask), "Create new branch" (createBranch), "Implement task" (implTask), "Make canges in code, including tests" (changeCode), "Commit changes" (commit), "Push changes to remote repository" (push), "Create pull request" (createPR), "Move task from IN PROGRESS to PR" (moveTaskProgressPR), "Ask peer for code review" (addReviewer), "Peer review" (peerReview), "Request changes in code" (reqChange), "Approve pull request" (approvePR), "Build process" (buildProcess), "Build application" (buildApp), "Run tests" (runTest), "Report successful build" (buildSuccess), "Report build failure" (buildFailure), "Complete task" (completeTask), "Merge changes" (merge), "Move task from PR to DONE" (moveTaskPRDone).

Listing 5.1: Task implementation process as Nested DCR Graph

Define a Nested DCR Graph
$G = \langle E, \triangleright, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\times, \rightarrow\bullet\diamond, L, l \rangle$ such as

$E = \{$moveTaskBacklogTodo, pickTask, createBranch, implTask, changeCode, commit, push, createPR, moveTaskProgressPR, addReviewer, peerReview, reqChange, approvePR, buildProcess, buildApp, runTest, buildSuccess, buildFailure, completeTask, merge, moveTaskPRDone$\}$

$\triangleright = \{\langle$changeCode, implTask$\rangle$, $\langle$commit, implTask$\rangle$, $\langle$push, implTask$\rangle$, $\langle$moveTaskProgressPR, createPR$\rangle$, $\langle$addReviewer, createPR$\rangle$, $\langle$reqChange, peerReview$\rangle$, $\langle$approvePR, peerReview$\rangle$, $\langle$merge, completeTask$\rangle$, $\langle$moveTaskPRDone, completeTask$\rangle\}$

$\mathtt{atoms}(E) = \{\text{moveTaskBacklogTodo, pickTask, createBranch, changeCode, commit, push, moveTaskProgressPR, addReviewer, reqChange, approvePR, buildApp, runTest, buildSuccess, buildFailure, merge, moveTaskPRDone}\}$

$M = \langle \emptyset, \emptyset, E \rangle$

$\rightarrow\bullet = \{\langle \text{moveTaskBacklogTodo, pickTask} \rangle, \langle \text{pickTask, createBranch} \rangle, \langle \text{createBranch, changeCode} \rangle, \langle \text{push, createPR} \rangle, \langle \text{addReviewer, peerReview} \rangle, \langle \text{changeCode, commit} \rangle, \langle \text{commit, push} \rangle, \langle \text{push, buildApp} \rangle, \langle \text{buildApp, runTests} \rangle, \langle \text{runTests, buildSuccess} \rangle, \langle \text{runTests, buildFailure} \rangle, \langle \text{buildSuccess, completeTask} \rangle, \langle \text{approvePR, completeTask} \rangle\}$

$\bullet\rightarrow = \{\langle \text{moveTaskBacklogTodo, completeTask} \rangle, \langle \text{pickTask, createBranch} \rangle, \langle \text{commit, changeCode} \rangle, \langle \text{push, commit} \rangle, \langle \text{push, buildProcess} \rangle, \langle \text{runTests, buildApp} \rangle, \langle \text{buildFailure, buildSuccess} \rangle, \langle \text{buildFailure, runTests} \rangle, \langle \text{buildSuccess, runTests} \rangle, \langle \text{reqChange, approvePR} \rangle\}$

$\rightarrow\% = \{\langle \text{moveTaskBacklogTodo, moveTaskBacklogTodo} \rangle, \langle \text{pickTask, pickTask} \rangle, \langle \text{createBranch, createBranch} \rangle, \langle \text{moveTaskProgressPR, moveTaskProgressPR} \rangle, \langle \text{merge, merge} \rangle, \langle \text{moveTaskPRDone, moveTaskPRDone} \rangle, \langle \text{completeTask, peerReview} \rangle, \langle \text{completeTask, createPR} \rangle, \langle \text{completeTask, implTask} \rangle, \langle \text{completeTask, buildProcess} \rangle\}$

$\rightarrow+ = \emptyset$

$\rightarrow\diamond = \{\langle \text{changeCode, commit} \rangle, \langle \text{commit, push} \rangle, \langle \text{push, buildApp} \rangle, \langle \text{buildApp, runTests} \rangle, \langle \text{runTests, buildSuccess} \rangle, \langle \text{runTests, buildFailure} \rangle, \langle \text{buildSuccess, completeTask} \rangle, \langle \text{approvePR, completeTask} \rangle\}$

$\rightarrow\times = \emptyset$

$\rightarrow\bullet\diamond = \{\langle \text{changeCode, commit} \rangle, \langle \text{commit, push} \rangle, \langle \text{push, buildApp} \rangle, \langle \text{buildApp, runTests} \rangle, \langle \text{runTests, buildSuccess} \rangle, \langle \text{runTests, buildFailure} \rangle, \langle \text{buildSuccess, completeTask} \rangle, \langle \text{approvePR, completeTask} \rangle\}$

$L = \{\langle \text{moveTaskBacklogTodo, PM} \rangle, \langle \text{pickTask, D} \rangle, \langle \text{createBranch, D} \rangle, \langle \text{implTask, D} \rangle, \langle \text{changeCode, D} \rangle, \langle \text{commit, D} \rangle, \langle \text{push, D} \rangle, \langle \text{createPR, D} \rangle, \langle \text{moveTaskProgressPR, D} \rangle, \langle \text{addReviewer, D} \rangle, \langle \text{peerReview, P} \rangle, \langle \text{reqChange, P} \rangle, \langle \text{approvePR, P} \rangle, \langle \text{buildProcess, BS} \rangle, \langle \text{buildApp, BS} \rangle, \langle \text{runTest, BS} \rangle, \langle \text{buildSuccess, BS} \rangle, \langle \text{buildFailure, BS} \rangle,$

$\langle \text{completeTask}, D \rangle$, $\langle \text{merge}, D \rangle$, $\langle \text{moveTaskPRDone}, D \rangle \}$

$l = \{ \langle \text{moveTaskBacklogTodo}, \langle \text{moveTaskBacklogTodo}, PM \rangle \rangle$,
$\langle \text{pickTask}, \langle \text{pickTask}, D \rangle \rangle$, $\langle \text{createBranch}, \langle \text{createBranch}, D \rangle \rangle$,
$\langle \text{implTask}, \langle \text{implTask}, D \rangle \rangle$, $\langle \text{changeCode}, \langle \text{changeCode}, D \rangle \rangle$,
$\langle \text{commit}, \langle \text{commit}, D \rangle \rangle$, $\langle \text{push}, \langle \text{push}, D \rangle \rangle$, $\langle \text{createPR}, \langle \text{createPR}, D \rangle \rangle$,
$\langle \text{moveTaskProgressPR}, \langle \text{moveTaskProgressPR}, D \rangle \rangle$,
$\langle \text{addReviewer}, \langle \text{addReviewer}, D \rangle \rangle$, $\langle \text{peerReview}, \langle \text{peerReview}, P \rangle \rangle$,
$\langle \text{reqChange}, \langle \text{reqChange}, P \rangle \rangle$, $\langle \text{approvePR}, \langle \text{approvePR}, P \rangle \rangle$,
$\langle \text{buildProcess}, \langle \text{buildProcess}, BS \rangle \rangle$, $\langle \text{buildApp}, \langle \text{buildApp}, BS \rangle \rangle$,
$\langle \text{runTest}, \langle \text{runTest}, BS \rangle \rangle$, $\langle \text{buildSuccess}, \langle \text{buildSuccess}, BS \rangle \rangle$,
$\langle \text{buildFailure}, \langle \text{buildFailure}, BS \rangle \rangle$, $\langle \text{completeTask}, \langle \text{completeTask}, D \rangle \rangle$,
$\langle \text{merge}, \langle \text{merge}, D \rangle \rangle$, $\langle \text{moveTaskPRDone}, \langle \text{moveTaskPRDone}, D \rangle \rangle \}$

5.2.5 Additional considerations

One of the main challenges in the modelling process is to choose a suitable level of abstraction for a particular situation. The key point in the process is to consider the purpose of creating the model. It will help to decide which information should be included in the model, how detailed it should be and which parts should be abstracted away.

Deciding on the scope of the model

The textual description of the task implementation process described in Section 4.5 covers a notion of a merge conflict. In this situation, several developers make changes in the code that conflict with each other. An early version of the model reflected this part of the process and can be seen in Figure 5.5. However, the "Merge conflict" activity did not become a part of the final model in Figure 5.1.

The decision to abstract away this activity was made based on the aim of the modelling process, namely verifying the properties defined in Section 5.1. The properties focus mainly on the requirement of quality checking of the code, like running automated tests and statical code review. The technical details, such as reported merge conflict, do not seem to be of the same importance to the process verification. Thus, they were omitted in the model as well. Otherwise, it could result in an overcomplicated model without providing any benefits in terms of verification.

Detailed modelling of "Implement task" activity

When developers implement a task, they usually write unit tests to check that the new code behaves as expected. They can also run tests that already exist in the codebase to check that changes in the code did not break any previously developed functionality. These steps are essential to ensure that the software behaves correctly after the code is modified. Therefore, it seemed reasonable to model these steps as separate activities.

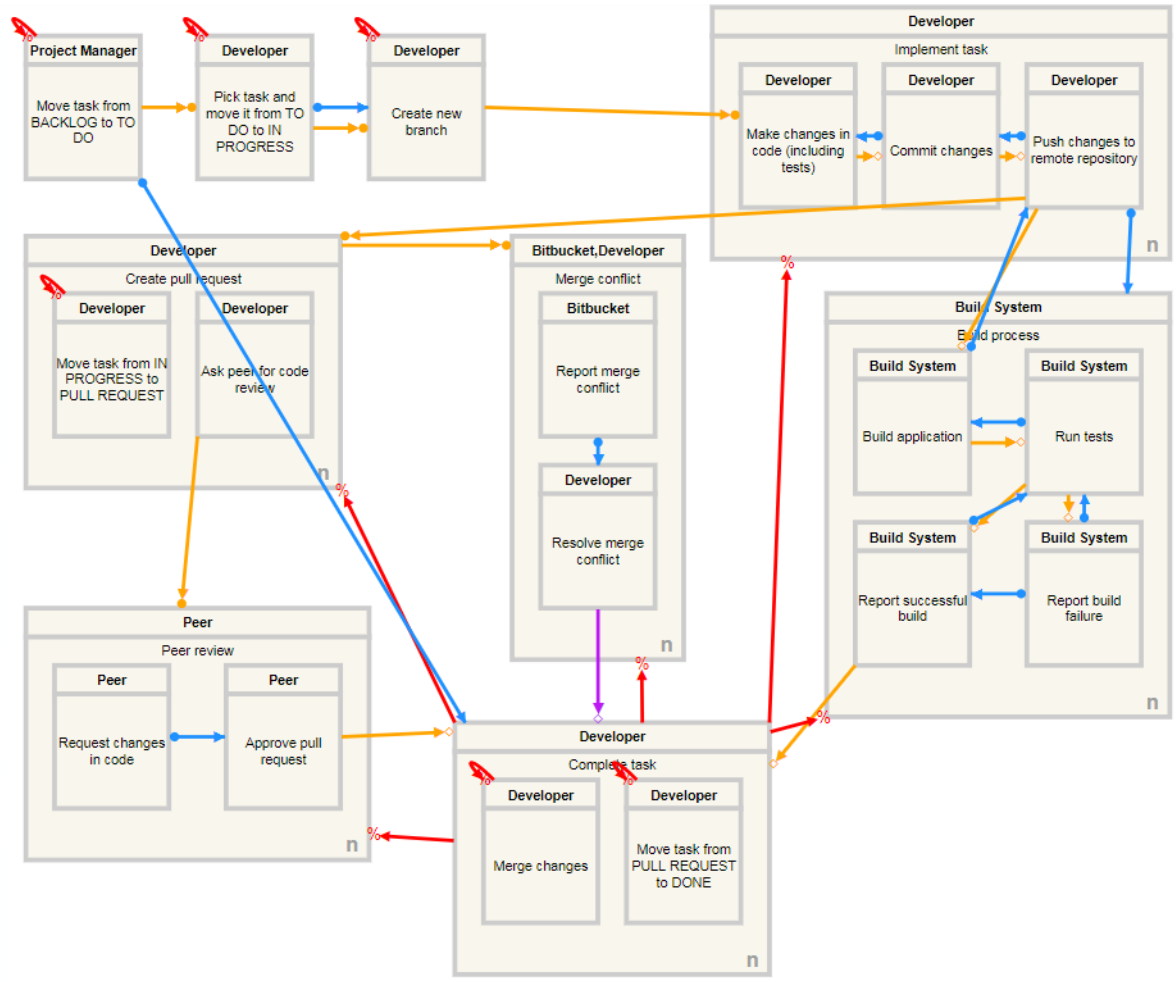

Figure 5.5: Task implementation process extended with "merge conflict" activity

Figure 5.6 shows the excerpt from an early version of the model where the "Implement task" activity included more child activities to describe it in more detail. Instead of a single "Make changes to the code (included tests)" activity as child activity, it included a nested activity "Code changes", which, in turn, contained child activities "Write code", "Write tests" and "Run tests locally".

However, this way of modelling caused problems related to the number of possible executions during the verification process. The only constraint on the "Code changes" subprocess is the condition relation from the "Create branch" activity. By Definition 3.4.2 all three child activities of the "Code changes" activity are enabled for execution at any point of time after the "Create branch" activity is executed. It means that for every step of graph execution (until the end state is reached), the number of activities that can be executed as the next step is at least three. For the final model (Figure 5.1), where these three activities are combined into a single "Make changes to the code (included tests)" activity, this number is at least one. The former

leads to an increased number of possible executions compared to the latter.

The difference is illustrated in Figures 5.7 and 5.8 which show possible executions for simplified and detailed versions of "Implement task" subprocesses respectively. It is clear that the number of executions is significantly bigger for the detailed version. Considering that the complete model also contains other activities, the difference becomes even more visible, making verification with the DCR Tool infeasible for the reasons explained later in Section 5.3.2.

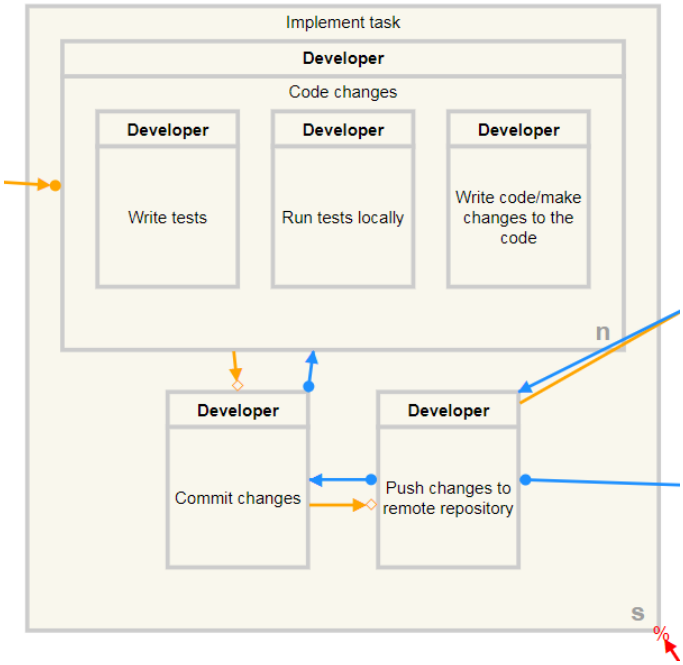

Figure 5.6: Detailed model of "code changes" process (excerpt from task implementation process)

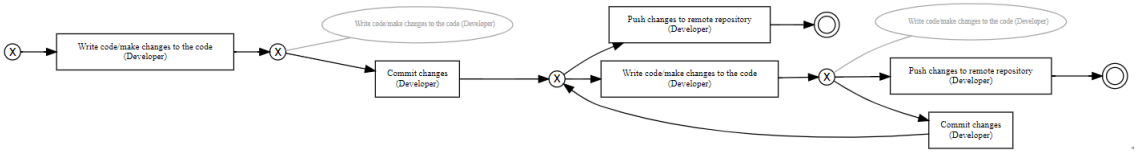

Figure 5.7: All possible executions of simplified "Implement task" subprocess

5.3 Verification

Using the constructed model, the properties can now be verified. This section describes how the verification process was carried out, the tools used and the achieved results.

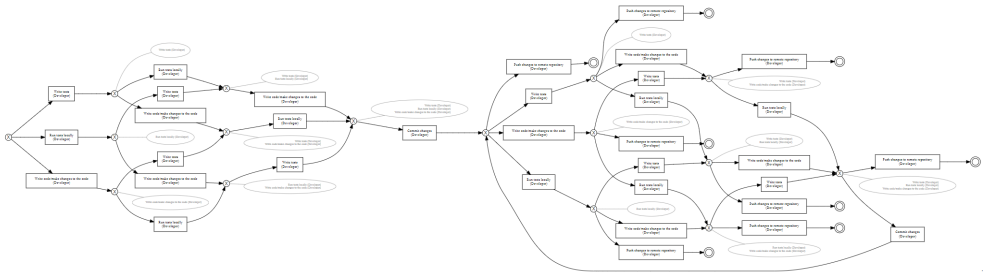

Figure 5.8: All possible executions of extended "Implement task" subprocess

5.3.1 Testing process behaviour with scenarios

One method used to test the model's behaviour during the development process and on its completion was checking that some desired and undesired paths were respectively accepted and rejected by the model.

As can be recalled from Section 3.5, the DCR Tool allows to specify and validate the paths (scenarios) using DCR Swimlane Editor.

Figure 5.9 shows the list of defined scenarios, including one execution that is required to be accepting and two executions that are forbidden in the model. The checkmarks on the left indicate that scenario verification was successful.

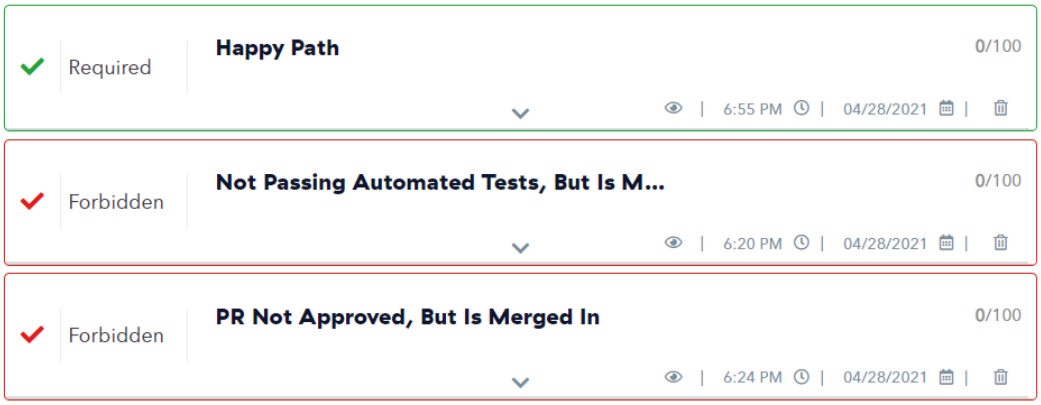

Figure 5.9: Examples of scenarios for task implementation process

The required scenario is the path where all activities are executed as expected, with both automated tests passing and pull request approved before the changes are merged into the main branch. The swim lane diagram in Figure 5.10 shows that all the boxes (activities) are green, meaning that the run is accepting.

The two forbidden scenarios represent paths where either the automated tests fail or the pull request gets rejected, but the changes are still merged in. Swim lane for one of these paths is demonstrated in Figure 5.11. One of the boxes ("Merge changes" activity) is marked as yellow because it violates the rules expressed by the DCR Graph, meaning that the

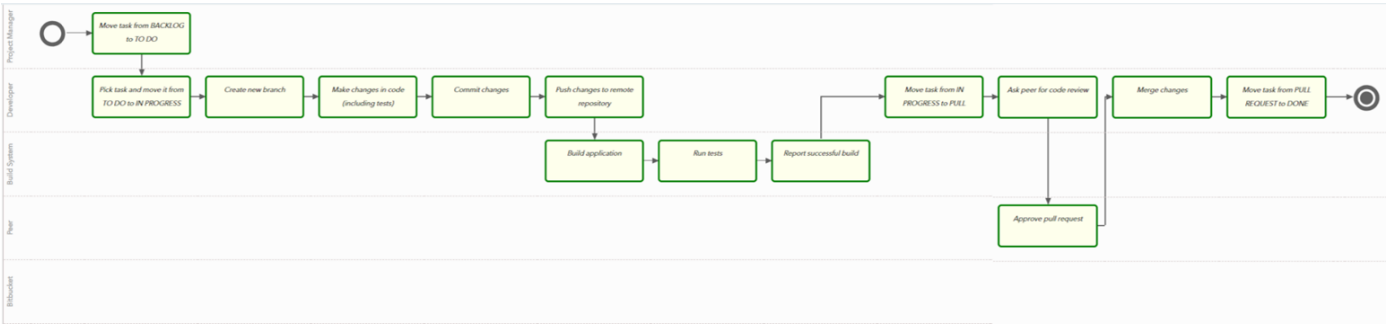

Figure 5.10: Accepting execution

execution is not accepting.

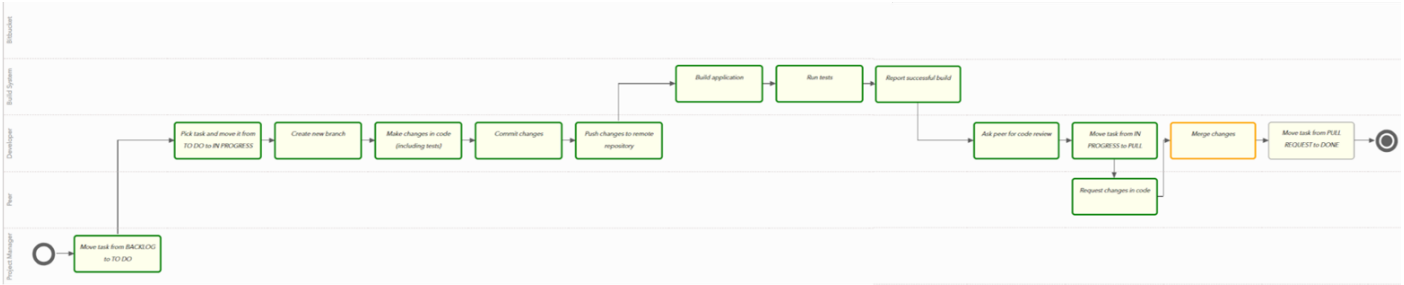

Figure 5.11: Not accepting execution

The same can be proved using the formal definitions from Section 3.4. In order to do that, the Nested DCR Graph defined in 5.1 must first be transformed to a flat DCR Graph by using Definition 3.4.11.

Listing 5.2: Flattening Nested DCR Graph for task implementation process

Define an underlying flat DCR Graph for Nested DCR Graph in listing 5.1 as
$G^b = \langle \text{atoms}(E), M, \rightarrow\bullet^b, \bullet\rightarrow^b, \rightarrow\%^b, \rightarrow+^b, \rightarrow\diamond^b, \rightarrow\times^b, \rightarrow\bullet\diamond^b, L, l \rangle$ such as

atoms(E), L and l are the same as in listing 5.1

$M = \langle \emptyset, \emptyset, \text{atoms}(E) \rangle$

$\rightarrow\bullet^b = \{\langle \text{moveTaskBacklogTodo}, \text{pickTask} \rangle,$
$\langle \text{pickTask}, \text{createBranch} \rangle, \langle \text{createBranch}, \text{changeCode} \rangle,$
$\langle \text{push}, \text{createPR} \rangle, \langle \text{addReviewer}, \text{reqChanges} \rangle,$
$\langle \text{addReviewer}, \text{approvePR} \rangle, \langle \text{changeCode}, \text{commit} \rangle,$
$\langle \text{commit}, \text{push} \rangle, \langle \text{push}, \text{buildApp} \rangle, \langle \text{buildApp}, \text{runTests} \rangle,$
$\langle \text{runTests}, \text{buildSuccess} \rangle, \langle \text{runTests}, \text{buildFailure} \rangle,$
$\langle \text{buildSuccess}, \text{merge} \rangle, \langle \text{buildSuccess}, \text{moveTaskPRDone} \rangle,$
$\langle \text{approvePR}, \text{merge} \rangle, \langle \text{approvePR}, \text{moveTaskPRDone} \rangle \}$

$\bullet\rightarrow^{b}$ = {⟨moveTaskBacklogTodo, merge⟩ ,
⟨moveTaskBacklogTodo, moveTaskPRDone⟩ ,
⟨pickTask, createBranch⟩ , ⟨commit, changeCode⟩ ,
⟨push, commit⟩ , ⟨push, buildApp⟩ , ⟨push, runTest⟩ ,
⟨push, buildSuccess⟩ , ⟨push, buildFailure⟩ ,
⟨runTests, buildApp⟩ , ⟨buildFailure, buildSuccess⟩ ,
⟨buildFailure, runTests⟩ , ⟨buildSuccess, runTests⟩ ,
⟨reqChange, approvePR⟩ }

$\rightarrow\%^{b}$ = {⟨moveTaskBacklogTodo, moveTaskBacklogTodo⟩ ,
⟨pickTask, pickTask⟩ , ⟨createBranch, createBranch⟩ ,
⟨moveTaskProgressPR, moveTaskProgressPR⟩ ,
⟨merge, merge⟩ , ⟨moveTaskPRDone, moveTaskPRDone⟩ ,
⟨merge, moveTaskProgressPR⟩ , ⟨merge, addReviewer⟩ ,
⟨merge, reqChange⟩ , ⟨merge, approvePR⟩ , ⟨merge, changeCode⟩ ,
⟨merge, commit⟩ , ⟨merge, push⟩ , ⟨merge, buildApp⟩ ,
⟨merge, runTests⟩ , ⟨merge, buildSuccess⟩ ,
⟨merge, buildFailure⟩ , ⟨moveTaskPRDone, moveTaskProgressPR⟩ ,
⟨moveTaskPRDone, addReviewer⟩ , ⟨moveTaskPRDone, reqChange⟩ ,
⟨moveTaskPRDone, approvePR⟩ , ⟨moveTaskPRDone, changeCode⟩ ,
⟨moveTaskPRDone, commit⟩ , ⟨moveTaskPRDone, push⟩ ,
⟨moveTaskPRDone, buildApp⟩ , ⟨moveTaskPRDone, runTests⟩ ,
⟨moveTaskPRDone, buildSuccess⟩ , ⟨moveTaskPRDone, buildFailure⟩
}

$\rightarrow+^{b}$ = ∅

$\rightarrow\diamond^{b}$ = {⟨changeCode, commit⟩ , ⟨commit, push⟩ ,
⟨push, buildApp⟩ , ⟨buildApp, runTests⟩ , ⟨runTests, buildSuccess⟩ ,
⟨runTests, buildFailure⟩ , ⟨buildSuccess, merge⟩ ,
⟨buildSuccess, moveTaskPRDone⟩ , ⟨approvePR, merge⟩ ,
⟨approvePR, moveTaskPRDone⟩ }

$\rightarrow\times^{b}$ = ∅

$\rightarrow\bullet\diamond^{b}$ = {⟨changeCode, commit⟩ , ⟨commit, push⟩ ,
⟨push, buildApp⟩ , ⟨buildApp, runTests⟩ , ⟨runTests, buildSuccess⟩ ,
⟨runTests, buildFailure⟩ , ⟨buildSuccess, merge⟩ ,
⟨buildSuccess, moveTaskPRDone⟩ , ⟨approvePR, merge⟩ ,
⟨approvePR, moveTaskPRDone⟩ }

Using the flat DCR Graph, we can reason about the states of the DCR Graph during execution. The following reasoning shows the execution of activities from the project manager chooses the task for development until a pull request is created and approved, but the build of the application fails. It then explains why the execution can not proceed to **merge** activity.

The state of the graph is described by the marking M = ⟨Ex, Re, In⟩, i.e. the sets of executed, pending and included events. For the graph in

listing 5.2, the initial state is M = $\langle \emptyset, \emptyset, \text{atoms}(E) \rangle$. It means that the sets of executed and pending events are empty and all atomic events in the graph are included.

According to Definition 3.4.12, an event is enabled for execution if it is included, all events that have condition relation to it are executed and none of the events that are milestones for it are pending. Initially, the only event that satisfies these requirements is **moveTaskBacklogTodo** because it is included and does not have any conditions or milestones.

Executing **moveTaskBacklogTodo** events causes change in state. By Definition 3.4.3, the executed event is added to Ex set, is removed from Re set (if it was pending) as well as Re and In sets are updated according to response, include and exclude relations going from the executed event. Event **moveTaskBacklogTodo** does not have include relation to any event. However, it has exclude relation to itself, meaning that it can only be executed only once in the process, unless it is included by any other event again. It has also response relation to the events **merge** and **moveTaskPRDone**, meaning that one a task is chosen for development, the activities related to completing the task must be executed in the future. The new marking becomes

$$\text{M} = \langle \{\text{moveTaskBacklogTodo}\}, \{\text{merge}, \text{moveTaskPRDone}\}, \text{atoms}(E) \setminus \{\text{moveTaskBacklogTodo}\} \rangle.$$

Now, the only event that is enabled for execution is the **pickTask** event. Executing it results in the marking

$$\text{M} = \langle \{\text{moveTaskBacklogTodo}, \text{pickTask}\}, \{\text{merge}, \text{moveTaskPRDone}, \text{createBranch}\}, \text{atoms}(E) \setminus \{\text{moveTaskBacklogTodo}, \text{pickTask}\} \rangle.$$

Note that **createBranch** event is in the set of pending requests, because it is a response to **pickTask** event. It means that the developer must create a new branch to introduce changes related to the task he has picked.

Let us then assume that **createBranch**, **changeCode**, **commit**, **push** events get executed. The marking is then

$$\text{M} = \langle \{\text{moveTaskBacklogTodo}, \text{pickTask}, \text{createBranch}, \text{changeCode}, \text{commit}, \text{push}\}, \{\text{merge}, \text{moveTaskPRDone}, \text{changeCode}, \text{commit}\}, \text{atoms}(E) \setminus \{\text{moveTaskBacklogTodo}, \text{pickTask}, \text{createBranch}\} \rangle.$$

The events that are enabled for execution now are **changeCode**, **moveTaskInProgressPR**, **addReviewer** and **buildApp**.

Assume that the execution continues with performing activities related to creation and approving a pull request. The events **moveTaskInProgressPR** and **addReviewer** get executed and the marking becomes

$$\text{M} = \langle \{\text{moveTaskBacklogTodo}, \text{pickTask}, \text{createBranch}, \text{changeCode}, \text{commit}, \text{push}\} \cup \{\text{moveTaskInProgressPR}, \text{addReviewer}, \text{approvePR}\}, \{\text{merge}, \text{moveTaskPRDone}\} \cup \{\text{changeCode}, \text{commit}\}, \text{atoms}(E) \setminus \{\text{moveTaskBacklogTodo}, \text{pickTask}, \text{createBranch}, \text{moveTaskInProgressPR}, \text{addReviewer}\} \rangle.$$

Further, let us look at the case where the build process results in build failure. Events **buildApp**, **runTests** and **buildFailure** get executed. The marking is updated to

$$M = \langle \{\text{moveTaskBacklogTodo}, \text{pickTask}, \text{createBranch}, \text{changeCode}, \text{commit}, \text{push}\} \cup \{\text{moveTaskInProgressPR}, \text{addReviewer}, \text{approvePR}, \text{buildApp}, \text{runTests}, \text{buildFailure}\}, \{\text{merge}, \text{moveTaskPRDone}, \text{changeCode}, \text{commit}, \text{buildSuccess}\}, \text{atoms}(E) \setminus \{\text{moveTaskBacklogTodo}, \text{pickTask}, \text{createBranch}, \text{moveTaskInProgressPR}, \text{addReviewer}\} \rangle.$$

By the definition of an enabled event, the execution can not proceed with **merge** or **moveTaskPRDone** events. The reason is that both of these activities have **buildSuccess** as a milestone. It means that **buildSuccess** can not be in the set of pending events to execute them. Looking at the marking above, we see that it is not the case. By choosing **merge** activity for execution, we would therefore violate rules of DCR Graphs. Thus, the path where an application build fails is not valid in the model.

DCR Swimlane Editor is a helpful and intuitive way of testing the behaviour of the process for some scenarios that are expected to be accepting or forbidden. However, it is impossible to verify all possible executions of a graph using scenarios. The number of executions is so large that it is not feasible to create a swim lane for each of them. Furthermore, maintaining all the swim lanes in case of changes in the graph would be a tedious task. Therefore, the model needs to be verified in another way that allows checking all possible executions.

5.3.2 Verification through execution of all possible paths

As mentioned in Section 3.5, the DCR Tool gives access to two applications that provide the functionality to find a path through the graph based on parameters and analysing the graph for deadlocks, namely Scenario Search and Dead-end Analyzer applications. This section explains the attempts and results of verifying the properties specified in Section 5.1 using these applications.

Property 1

The first property that has to be satisfied is "Any code changes merged into the main branch must be quality checked (peer review approved and build successfully)". We can also formulate this property as two following sentences: "No code changes that a peer does not approve can be merged" and "No code changes that do not pass the build can be merged". These sentences express requirements about some undesired behaviour that should not occur. As can be recalled from Section 2.5.2, these are therefore safety properties.

In order to prove a safety property, it must be proven that there are no executions where the undesired behaviour occurs among all possible executions. If at least one such execution is found, the property is not valid in the model. In the context of DCR Graphs, it means that if there is at

least one path such that an undesired sequence of activities occurs, then the property is not valid. This path will, in this case, be a counterexample.

Following this reasoning, the Scenario Search application was found suitable for proving safety properties "No code changes that a peer does not approve can be merged" and "No code changes that do not pass the build can be merged". By filling out the parameters shown in Figure 3.18 in a way that expresses the undesired behaviour and running Scenario Search, it will either return a counterexample or a message saying that no such paths exist. If the latter is the case, then the property is valid in the model.

Take "No code changes that a peer does not approve can be merged"-property as an example first. A counterexample showing that the property does not hold in the model would be a path where activity "Merge changes" can be executed even if the "Approve PR" activity was not executed before it. The parameters filled out as shown in Figure 5.12 mean that we search for a path to "Merge changes" activity where "Approve PR" activity was avoided (not executed). The result of this search is seen at the top of Figure 5.12. It says that no paths that satisfy given parameters were found, meaning that "No code changes that a peer does not approve can be merged"-property is valid in the model.

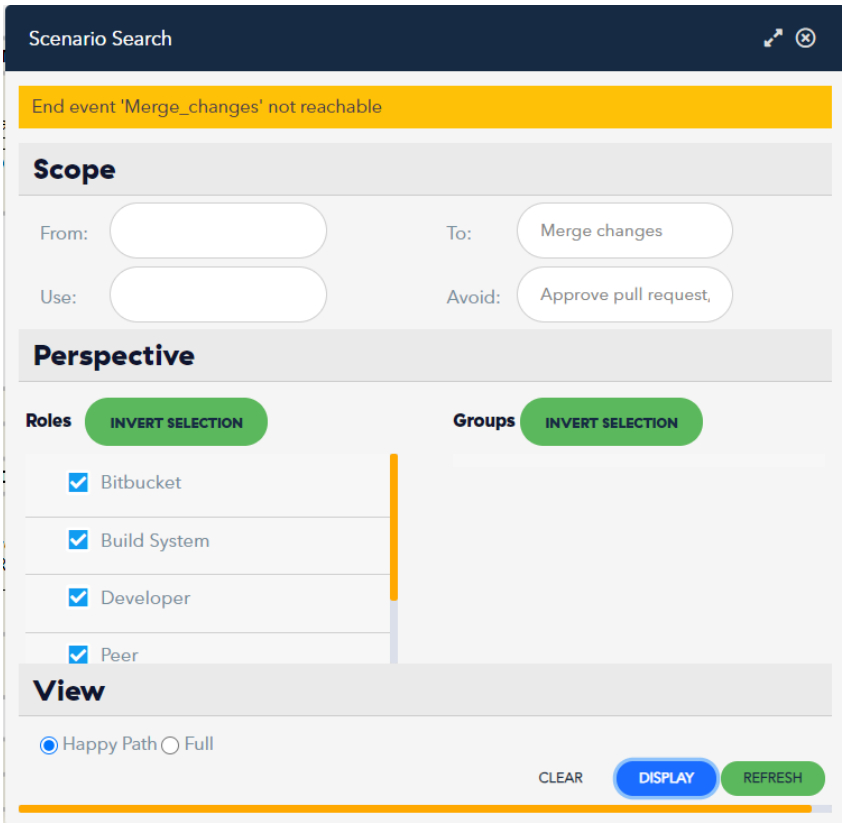

Figure 5.12: Verifying "No code changes that are not approved by a peer can be merged"-property with Scenario Search application

"No code changes that do not pass the build can be merged"-property

can be verified in a similar way as shown in Figure 5.13.

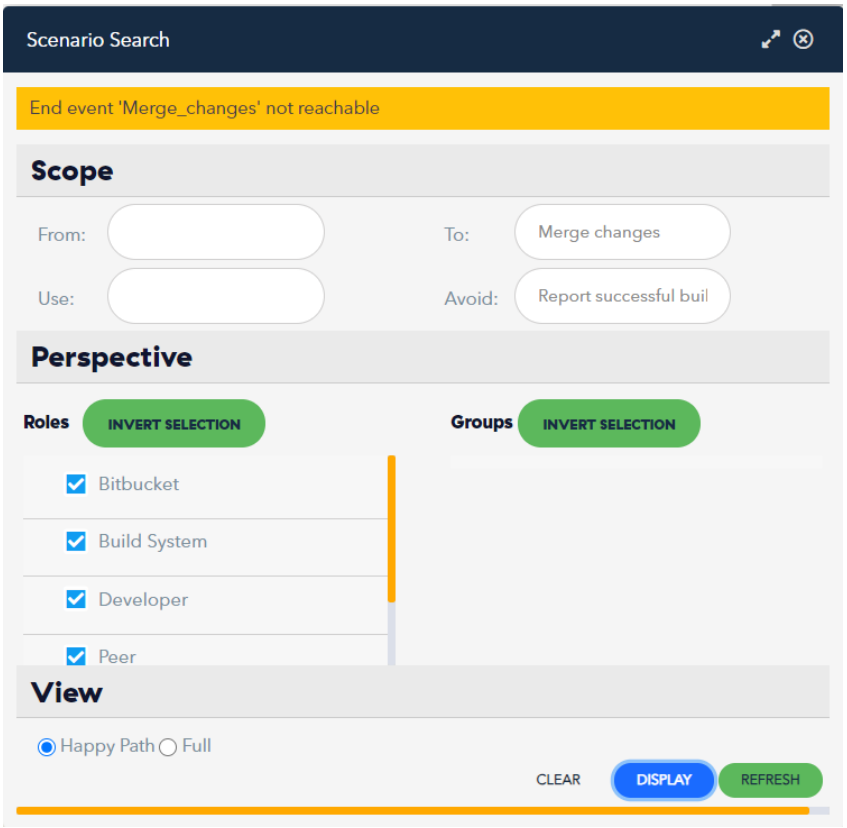

Figure 5.13: Verifying "No code changes that do not pass the build can be merged"-property with Scenario Search application

Since there are no counterexamples for each of the properties, we can conclude that the desired property "Any code changes merged into the main branch must be quality checked (peer review approved and build is successful)" holds in the model.

Property 2

The second property that has to be verified for the software development process is that "A task that is chosen for development must eventually be completed". It means that each time the process of implementing a task starts, it can not finish until the task is completed. In other words, the "Complete task" activity must occur in each valid path. Based on Section 2.5.2, this is therefore a liveness property.

One way to verify this property could be to search for a path where the process has reached its end (is completed), but the subactivities "Merge changes" and "Move task from PR to Done" of "Complete task" activity has not been executed. However, the Scenario Search application does not provide a way to express the end state (the completion) of the process in parameters. Thus, the verification of this property can not be done in the

same way as for the first property.

Another possibility is to use the "full" option of the Scenario Search application that returns a figure showing all possible executions of the DCR Graph. This figure includes the end states, i.e. states where the process is completed (execution is accepting). Then, it is possible to find whether all paths leading to end states include "Merge changes" and "Move task from PR to Done" activities. The result of the full path search for the model of the task implementation process is shown in Figure 5.14. From the figure, it is clear that going through it manually is an infeasible task.

A third option is making use of the Dead-end Analyzer application provided in the App Store of DCR Tool. As mentioned in Section 3.5, the Dead-end Analyzer application searches for paths where the goal of the process can not be reached. In the context of DCR Graphs, it means finding a path where an accepting state can not be reached.

For the process of task implementation, the accepting state can only be reached in two cases. The first case is the initial state since there are no pending events in this state. However, once the "Move task from backlog to TODO" activity is executed, the "Merge changes" and "Move task from PR to Done" activities become pending. It means that the only way an accepting state can be reached again is if "Merge changes" and "Move task from PR to Done" activities get executed since they become pending. Thus, if it can be proved that all executions can be completed (i.e. there is no deadlock in the process), then the desired property is satisfied. The absence of a situation that prevents the process from completing can be proven with the Dead-end Analyzer application. Figure 5.15 shows the result of running dead-end analysis on the DCR Graph modelling the task implementation process. It shows that the dead-end is not reachable in the graph, so it can be concluded that the property "A task that is chosen for development must eventually be completed" is satisfied.

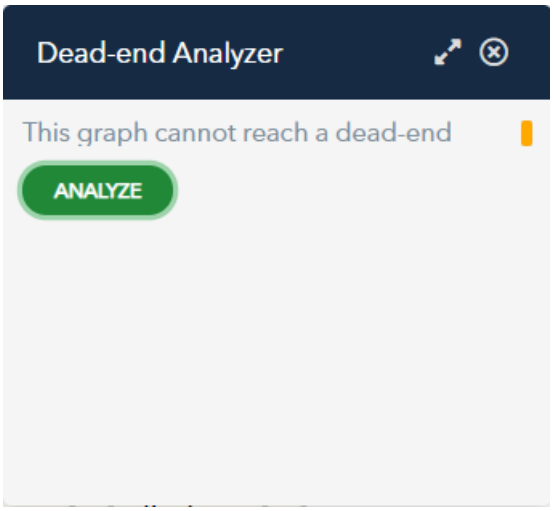

Figure 5.15: Dead-end Analyzer result

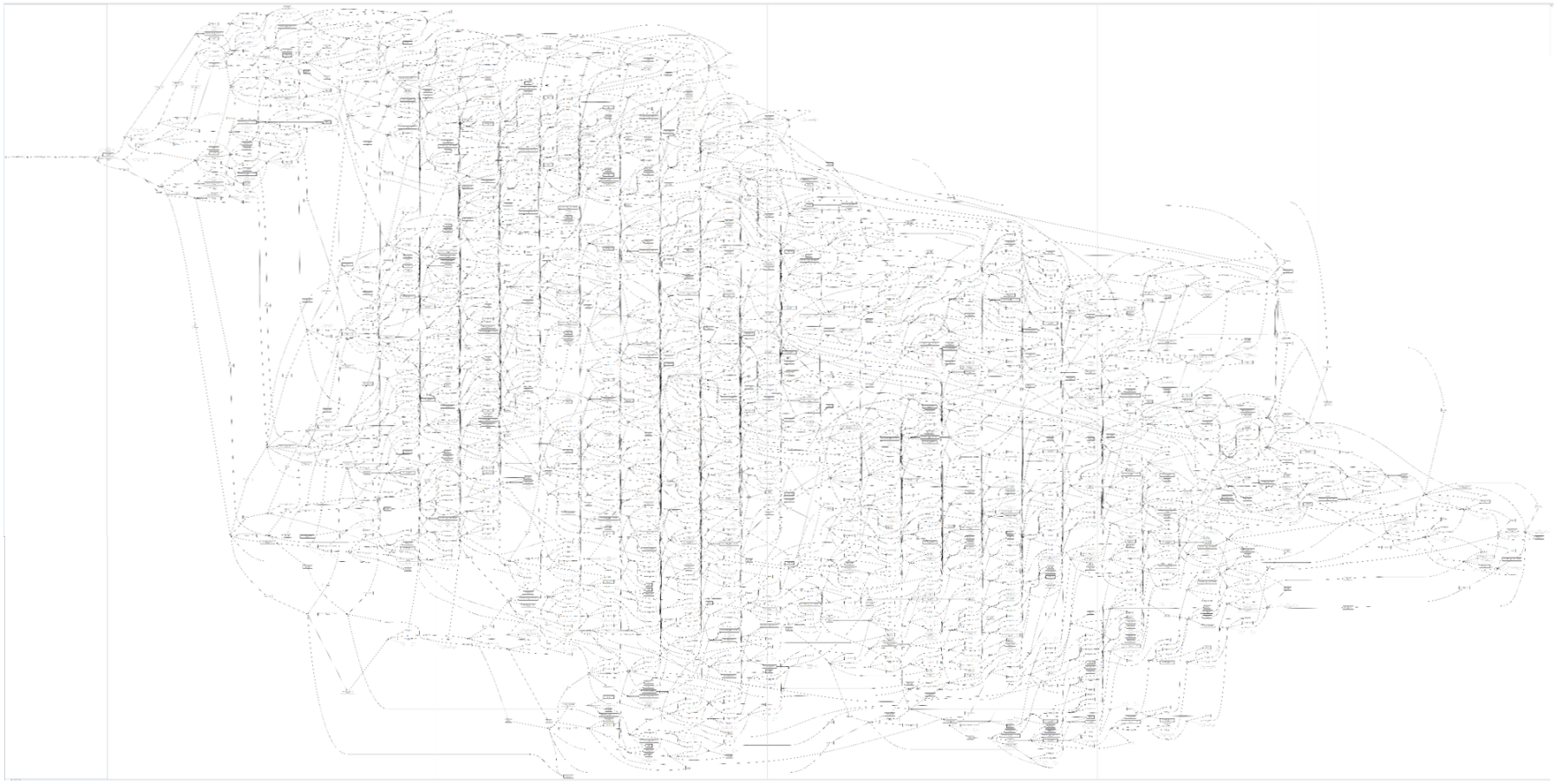

Figure 5.14: All possible executions of task implementation process DCR Graph

Limitation for verification using the DCR Tool

The DCR Tool, which has been used for modelling and verification in this thesis, is a web-based tool. When the user initiates a computation process, for example, searching for a scenario with the Scenario Search application or finding dead-ends with the Dead-end Analyzer application, the tool sends a request to a service that takes care of executing this process. Then, the DCR Tool waits until the service returns a response containing the execution result. However, the DCR Tool can not wait for the response indefinitely. If a request takes long time to answer, it could indicate that the service is down and will never return a response. Therefore, The DCR Tool is set to wait for a limited amount of time (1,7 minutes) before it assumes that the service did not receive a request and shows an error message to the user. However, the reason for the service not responding in time could also be that the computation takes longer time to complete than expected. In the context of DCR Graphs executions, this would often be the case when the number of all possible executions becomes large.

Section 5.2.5 mentioned that extending the model with more activities to present a more detailed picture of task implementation activity led to complications for verification. It also explained that the main reason is the rapidly growing number of all possible executions when more activities are enabled for execution. A larger number of possible executions also means longer time needed for executing them.

During verification of the extended model, the problem related to the time-out limit was faced. When running the Dead-end Analyzer application on the extended graph, a generic error message saying that something went wrong was returned. However, no counterexample showing how the graph can reach a dead-end was presented. This response does not provide a valuable result of verification. No counterexample can only indicate that the dead-end was not reached before the request timed out after 1,7 minutes. However, a dead-end could be reached after the time-out. Therefore, it is impossible to interpret the result and state whether the verification succeeded or not.

6 Modelling and verification of Tellu Diabetes App

This section addresses the work done on modelling and verification of the system developed, focusing on the mobile application, Tellu Diabetes App. The model is based on the application functionality described in Section 4.2.

6.1 Properties

The system as a whole and the application must satisfy several properties. The data handled by the application (mainly health-related measurements) is sensitive, so it must be ensured that it is transmitted and stored securely. Some of the properties are summarised as follows:

1. Only logged in users can access the application functionality related to the viewing of personal information
2. The data transmitted between medical BLE devices and the application can not be received by any mobile device (smartphone, tablet) other than the user's
3. The data transmitted over the network (for example, between application and TelluCloud) can not be available for attackers to read
4. TelluCloud must eventually receive measurements sent from the mobile application so that the medical personal can be notified in time in case any additional actions are needed

As we can see, many of these properties are related to parts of the system that lie outside of the mobile application. For example, properties 2 and 3 are related to communication links (network and Bluetooth connections), while property 4 relies on the correct function and uptime of TelluCloud services. Since this thesis focuses mainly on modelling and verification of the mobile application, some assumptions must be made.

6.2 Assumptions

Analysing the properties specified in the previous section, we can list the following assumptions:

- The services provided by TelluCloud behave correctly. TelluCloud may still respond with error codes or be temporarily down, but if it responds with success codes, it is expected that TelluCloud handles the data correctly.
- Once the Bluetooth connection is established between two devices (in this case, medical BLE device and smartphone/tablet running the application), the connection is secure.
- When the application makes a request to TelluCloud, the library used by the application for making network calls, handles the data transmission in a secure way (for example, by encrypting the data sent)

6.3 Model

We will construct a model of application functionality based on the functional requirements described in Section 4.2.

Following the DCR process methodology from Section 3.5, we will first define activities that can be executed in the application and who executes them.

6.3.1 Activities

- Login process
  - User loges into the application by filling out the login form
  - Mobile application validates the login form
  - In case login form validation fails, the mobile application displays an error message
  - In case login form validation succeeds, the mobile application sends a request to TelluCloud to authenticate the user
  - TelluCloud returns a response (either success or error)
  - In case of authentication failure, the mobile application displays an error message
- Pairing of BLE device and mobile
  - User turns off Bluetooth on smartphone
  - User turns on Bluetooth on smartphone
  - User pairs the devices
- Process of adding new measurement automatically
  - Mobile application fills out the measurement form automatically with data received from medical BLE device
  - User submits the form

- Process of adding new measurement manually
  - User fills out the measurement form
  - User submits the form
- Process of validating measurement form
  - Mobile application validates the measurement form
  - If form validation fails, the mobile application displays an error message
- Process of sending "add measurement" request to TelluCloud (same for submitting either manually or automatically filled out form)
  - Mobile application makes a request to TelluCloud to add a new measurement
  - TelluCloud returns a response (either success or error)
  - In case of error response code, the mobile application displays an error message
- Viewing previous measurements
  - User navigates to the screen with previous measurements in the application
  - Mobile application sends "get all my measurements" request to TelluCloud
  - TelluCloud returns a response (either success or error)
  - If TelluCloud responds with OK, the mobile application displays the list of previous measurements
  - If TelluCloud responds with error code, mobile application displays an error message
- Logout process
  - User logs out of the application by clicking a button
  - Mobile application sends logout request to TelluCloud

6.3.2 Roles

Following is a list of all roles that appeared in the activities defined above.

- User
- Mobile application
- TelluCloud

6.3.3 Model as DCR Graph

The DCR process methodology described in Section 3.5 suggests defining all rules representing dependencies between the activities before constructing the model. However, this process contains many activities that, in turn, contain other subactivities. Thus, it was considered more appropriate to start with defining rules and construct model including top-level activities only before narrowing down to the rules between subactivities.

Top-level model

In order to explain dependencies between different features the application provides, a model that only includes top-level events is presented first.

Rule 1 The main rule that the top-level activities need to conform to is that no features are accessible before the user logs in. Put in a more formal way, the events representing the execution of any application feature (like accessing previous measurements or submitting new measurements) can not be executed unless the "Log in" event has occurred before. Furthermore, if the user logs out of the application, the "Log in" event must occur again before executing any other events. We assume that the user is initially logged out.

Rule 2 The second rule is that in order to get measurements from a BLE device, it must be paired with the user's smartphone/tablet. We assume that the devices are initially not paired.

These rules can be expressed as DCR Graph as demonstrated in Figure 6.1. Initially, only "Log in" and "Pair BLE device to smartphone" activities are included in the process. The other activities are not included (graphically represented with dotted lines), meaning that they are not enabled for execution (by Definition 3.4.2).

The "Log in" activity has include relations (green arrows) to events representing other functionalities of the application. It also has an exclude relation (red arrow) to itself. These relations express that when "Log in" activity gets executed, it becomes excluded from the process, while "Log out", "Fill out measurement form manually", "Fill out measurement form automatically" and "Access previous measurements" activities become included (by Definition 3.4.3). It means that after the user logs in, he or she can access the other features of the application. However, it is not possible to log in when the user is already logged in.

Further, the "Log out" activity has exclude relations to all events representing application functionality (also itself) and an include relation to "Log in" activity. These relations express that after the user logs out, the only thing that the user can do in the application is to log in.

Two previous paragraphs explain how Rule 1 is modelled. This is the rule that is mainly concerned with access to the mobile application's functionality. Rule 2, on the other hand, specifies a requirement related to an event outside of the main application flow. "Pair BLE device to

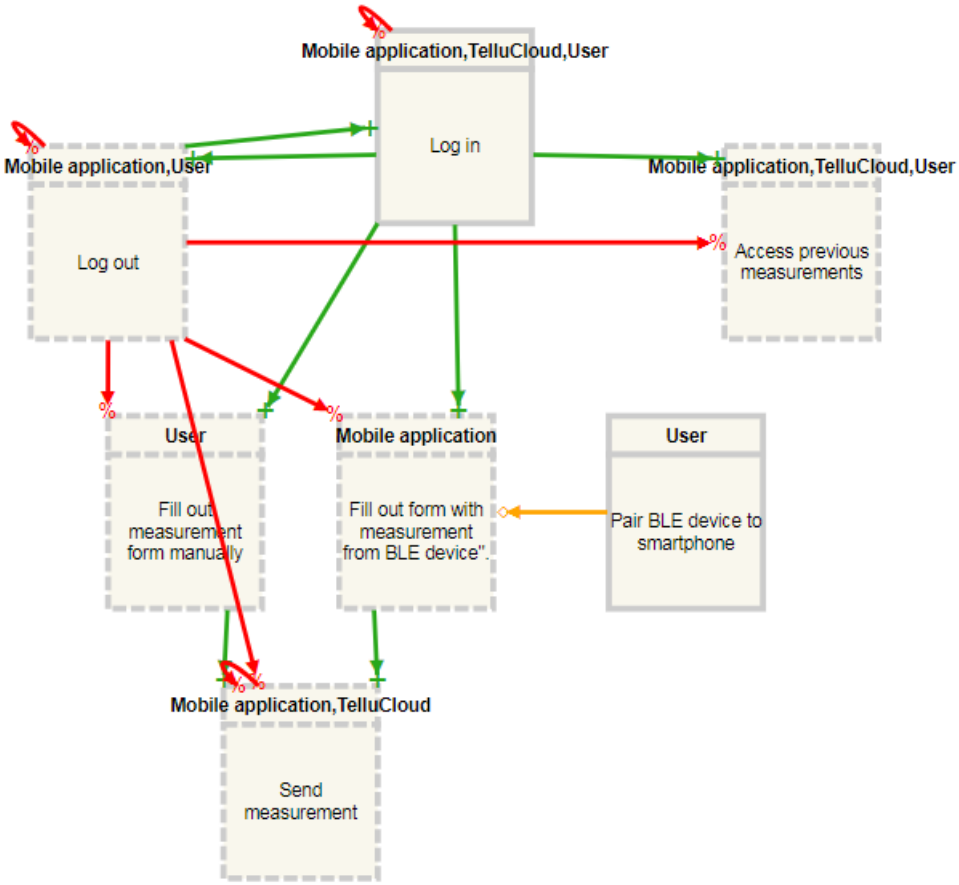

Figure 6.1: Top-level model of Tellu Diabetes App as DCR Graph

smartphone" is an activity that can occur independently from the other activities. However, one of the application functionalities depends on its execution. This dependency is expressed with pre-condition relation (yellow arrow), going from "Pair BLE device to smartphone" to "Fill out measurement form automatically" activity. As can be recalled from 3.3, the pre-condition relation is a combination of condition and milestone relation. The condition part means that "Pair BLE device to smartphone" must occur before "Fill out measurement form automatically" can occur. The milestone part means that if "Pair BLE device to smartphone" becomes pending, then "Fill out measurement form automatically" can not occur until pairing has occurred again. The milestone relation between these activities is redundant in the top-level model, but it is essential for the further extension of the device pairing process. Therefore, the need for milestone relation is explained later in Section 6.3.3.

We will now continue the modelling process by extending the top-level activities with their subactivities and specifying the rules that apply between them.

Extending "Log in" activity

First, we will consider the "Log in" activity in more detail and explain some of the relation patterns used for expressing certain rules. Subactivities of "Log in" activity have to conform to the following rules:

1. User has to fill out the login form before it can be submitted
2. When the user submits the form, the mobile application must verify it. If form validation fails (for example, if no password was provided), an error message is displayed. Otherwise, a request to a service responsible for authentication at TelluCloud is sent.
3. When a request is sent to TelluCloud, it must eventually respond either with OK or an error response (in case the request has timed out, it is interpreted as an error response). On successful authentication, the user is logged in and gets access to other application features. Otherwise, an error message is displayed.
4. After the user has submitted the login form, he or she can not fill out the form again until the processes of form verification and authentication request to TelluCloud have completed. If either of them is completed with an error, the user can fill out the form again.

Figure 6.2 demonstrates the relations between subactivities of "Log in" activities. "Log out" activity and three include relations are included in the model, so the relations described in the previous subsection are visible.

Initially, only the "Fill out login form" activity is included in the process. Other subactivities are not included (graphically represented with dotted lines). It means that only the "Fill out login form" activity is enabled for execution by Definition 3.4.2. For example, it would not be reasonable to be able to execute the "Send authentication request" activity before the form is filled out and submitted. The include relations (green arrows) going from one activity to another can be interpreted as the order in which the activities can happen. For instance, "Fill out login form" includes "Submit login form" in the process, so it becomes enabled for execution (by Definitions 3.4.3 and 3.4.2). This include relation expresses Rule 1 specified in the list above.

Further, "Submit login form" includes "Login form validation failure" and "Login form validation OK" since they only can happen after the form is submitted. We also see response relations (blue arrows) between "Submit login form" and "Login form validation failure" and "Login form validation OK" relations. Additionally, the "Login form validation failure" and "Login form validation OK" activities have exclude relations (red arrows) to each other and themselves. This pattern is used to express that when a validation process starts, it must complete either with success or error. When one of the activities gets executed, it excludes the other, preventing the same verification from completing with both results simultaneously. The exclude relation to itself means that the activity can not be repeated until it is included again. For example, suppose the "Login form validation failure" activity occurs. In that case, it is removed from the set of pending

responses and the set of included events and "Login form validation OK" is excluded, so it can not happen until the next time the user submits the form and it is validated again. The same pattern is used multiple times for expressing rules similar to Rules 2 and 3 from the list above.

We can also see that the "Submit login form" activity excludes the "Fill out login form" activity. It can then be included again by "Login form verification failure" and "Send auth failed response" activities. It means that the user can not fill out the login form until the form verification process and the following request are completed. The combination of exclude relation between "Submit login form" and "Fill out login form" activities and include relations going from "Verification failure" and "Send auth failed response" activities to the "Fill out login form" activity express Rule 4 from the list above.

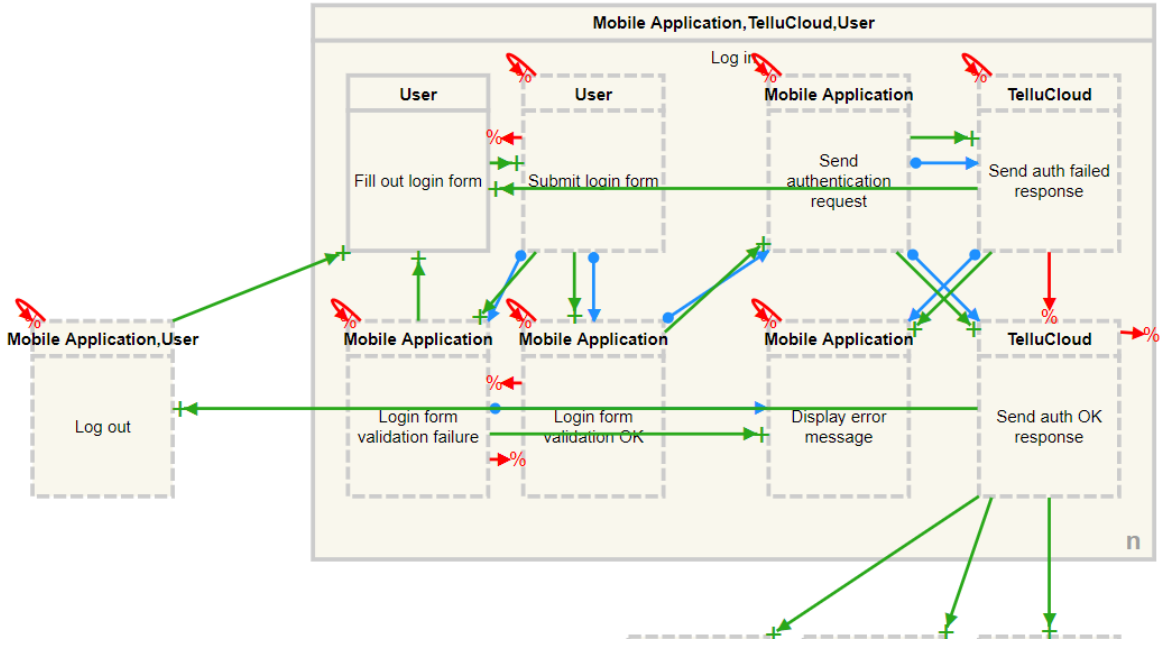

Figure 6.2: Extended model of "Log In" activity (excerpt from complete DCR Graph)

Extending "Device pairing" activity

Section 6.3.3 mentioned the pre-condition (combination of condition and milestone) relation going from "Pair BLE device to smartphone" activity to "Fill out measurements automatically" activity. The condition relation is necessary to express that the devices have to be paired before measurements can be transmitted from BLE device to smartphone. Now, we will explain how the extension of "Pair BLE device to smartphone" introduces the need for milestone relation and why condition relation alone is not sufficient.

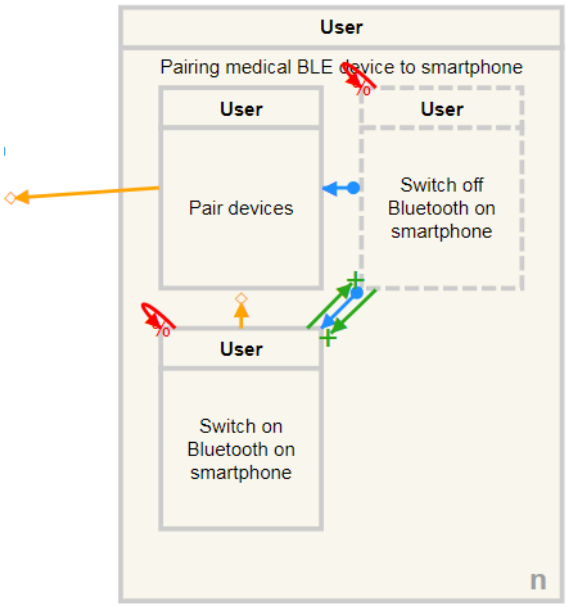

Figure 6.3: Extended model of "Device pairing" activity (excerpt from complete DCR Graph)

The definition of condition relation says that if an event A is a condition for an event B, A must be executed before B. In this case, it would mean that the "Pair BLE device to smartphone" activity must occur before "Fill out form with measurement from BLE device" can occur. After "Pair BLE device to smartphone" has been executed once, the "Fill out measurements automatically" activity can be executed (provided that it is included in the process and no other conditions going to it prevent it from being executed). Let us now consider the following situation: the user pairs the devices and then switches off Bluetooth on the smartphone. In this case, the devices are no longer paired, so measurement transmission should not be allowed to be executed. However, this execution would be accepting in the model containing only condition relation. When the events "Pair BLE device to smartphone" and "Switch off Bluetooth on smartphone" are executed, they are added to the set of executed events. Since "Pair BLE device to smartphone" is executed, it does not prevent "Fill out form with measurement from BLE device" from being executed. This behaviour is undesired, so the model must be adjusted.

It can be done by introducing activities and relations as shown in Figure 6.3. The response relations (blue arrows) going from "Switch off Bluetooth on smartphone" to "Switch on Bluetooth on smartphone" and "Pair devices" mean that they become pending when the "Switch off Bluetooth on smartphone" activity is executed. By definition of milestone relation, "Fill out form with measurement from BLE device" can then not be executed. "Pair devices" can, in turn, not be executed until "Switch on Bluetooth on smartphone" is executed since these activities have a pre-condition relation. Therefore, the combination of pre-condition and

response relations ensures that after Bluetooth is switched off, it must be switched on again for the devices to get paired.

Extending activities related to submission of new measurements

We will now consider the activities related to submitting new measurements to TelluCloud, either by filling out the form manually or automatically. This process has several rules that should be satisfied.

1. When the form filled out manually by the user or automatically by the application (based on measurements received from the BLE device) is submitted, it gets validated. Validation completes either with success or failure. If the former is the case, an "add measurement" request is sent to TelluCloud. Otherwise, an error message is displayed.

2. When the "add measurement" request is sent to TelluCloud, it must eventually respond with either OK or error code. In both cases, a suitable message is displayed to the user.

3. The form can not be filled out again until the processes of form validation and request to TelluCloud complete.

These rules are expressed with activities and relations as demonstrated in Figure 6.4. The combination of include, response and exclude relations between "Submit form", "Form validation failure", "Form validation success" and "Send add measurement request to TelluCloud", "Send OK response", "Send error response" express Rules 1 and 2 from the list above. The same pattern was explained earlier for validation of login form in Section 6.3.3.

Rule 3 is expressed by the milestone relations (violet arrows) from "Send measurement to TelluCloud" and "Measurement form validation" to "Fill out measurement form manually" and "Fill out form with measurement from BLE device". When the form is submitted, the "Form validation failure" and "Form validation success" activities become pending. By definition of milestone relation, the activities representing the process of filling out the form are then no longer enabled for execution. Further, when the form is successfully validated and the request is sent to TelluCloud, the "Send OK response" and "Send error response" activities become pending, preventing form filling activities from being executed. "Fill out measurement form manually" and "Fill out form with measurement from BLE device" will remain unenabled for execution until all child activities of "Send measurement to TelluCloud" and "Measurement form validation" either are executed or excluded.

Extending activity related to viewing previous measurements

The process of viewing previous measurements has similar rules and representation to those of measurement submission.

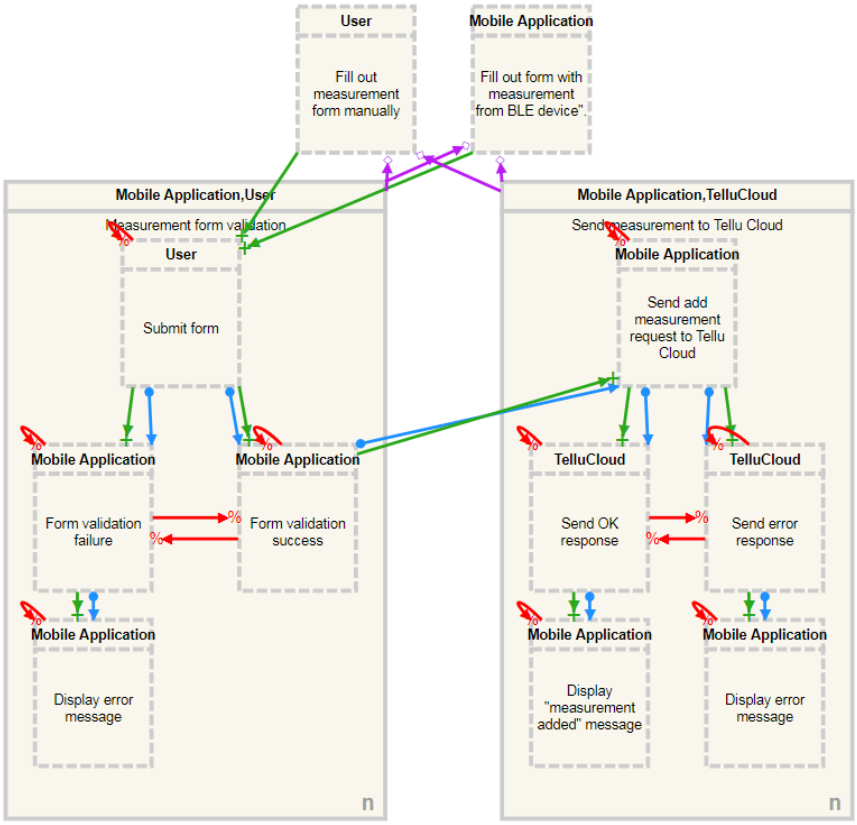

Figure 6.4: Extended model of activities related to measurement submission (excerpt from complete DCR Graph)

1. When a "get all measurements" request is sent to TelluCloud, it must eventually respond with either OK (returning the list of previous measurements) or an error code. In the former case, the list is displayed. Otherwise, an error message is displayed.

2. User can not navigate to the "my measurements" screen again until the request process is completed.

Both rules are expressed with the relation patterns that were already explained for similar cases in previous paragraphs. The process of viewing all previous measurements is illustrated in Figure 6.5.

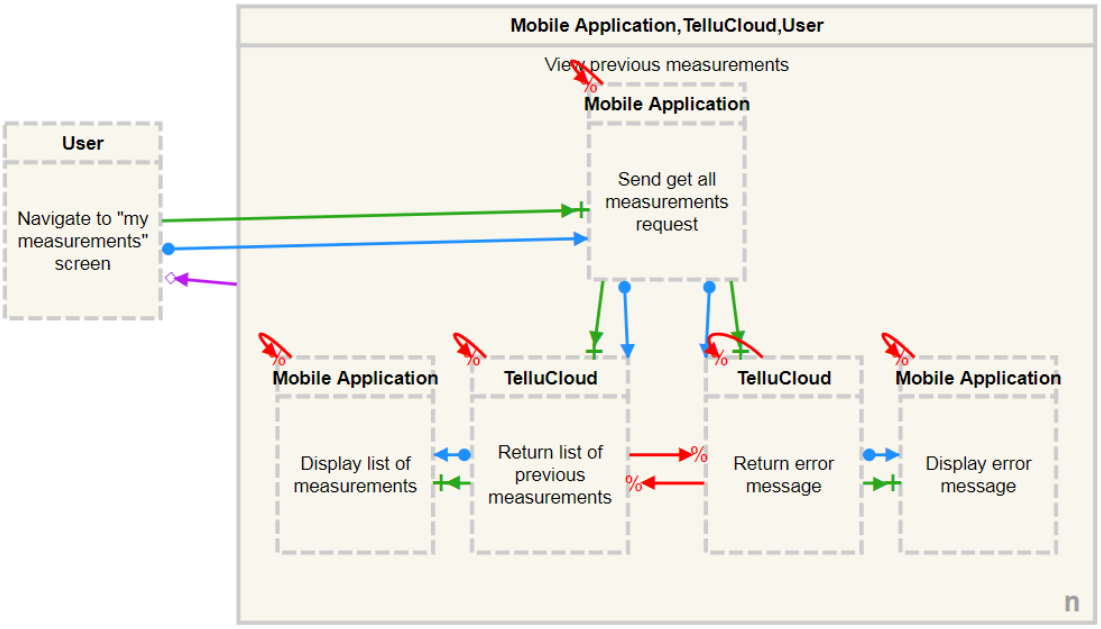

Figure 6.5: Extended model of activities related to viewing previous measurement (excerpt from complete DCR Graph)

The final model

The previous paragraphs presented the main parts of the model and explained the relations between the activities. These parts can now be unified in the final model as presented in Figure 6.6. Additional milestone relations (violet arrows) were added between activities related to processes of form validation and request sending. They express that no other activities can be executed while they are ongoing.

6.4 Verification

6.4.1 Testing process behaviour with scenarios

The first method used to verify the correctness of the model was defining some executions representing desired and undesired behaviour and checking that the actual results correspond to the expected ones. As we may recall from Section 3.5, this can be done with the DCR Swimlane Editor.

Figure 6.7 shows a selection from the list of defined scenarios, including paths expected to be accepting and forbidden in the model. Results of the verification indicating that the graph behaves as expected are represented with checkmarks on the left of each scenario.

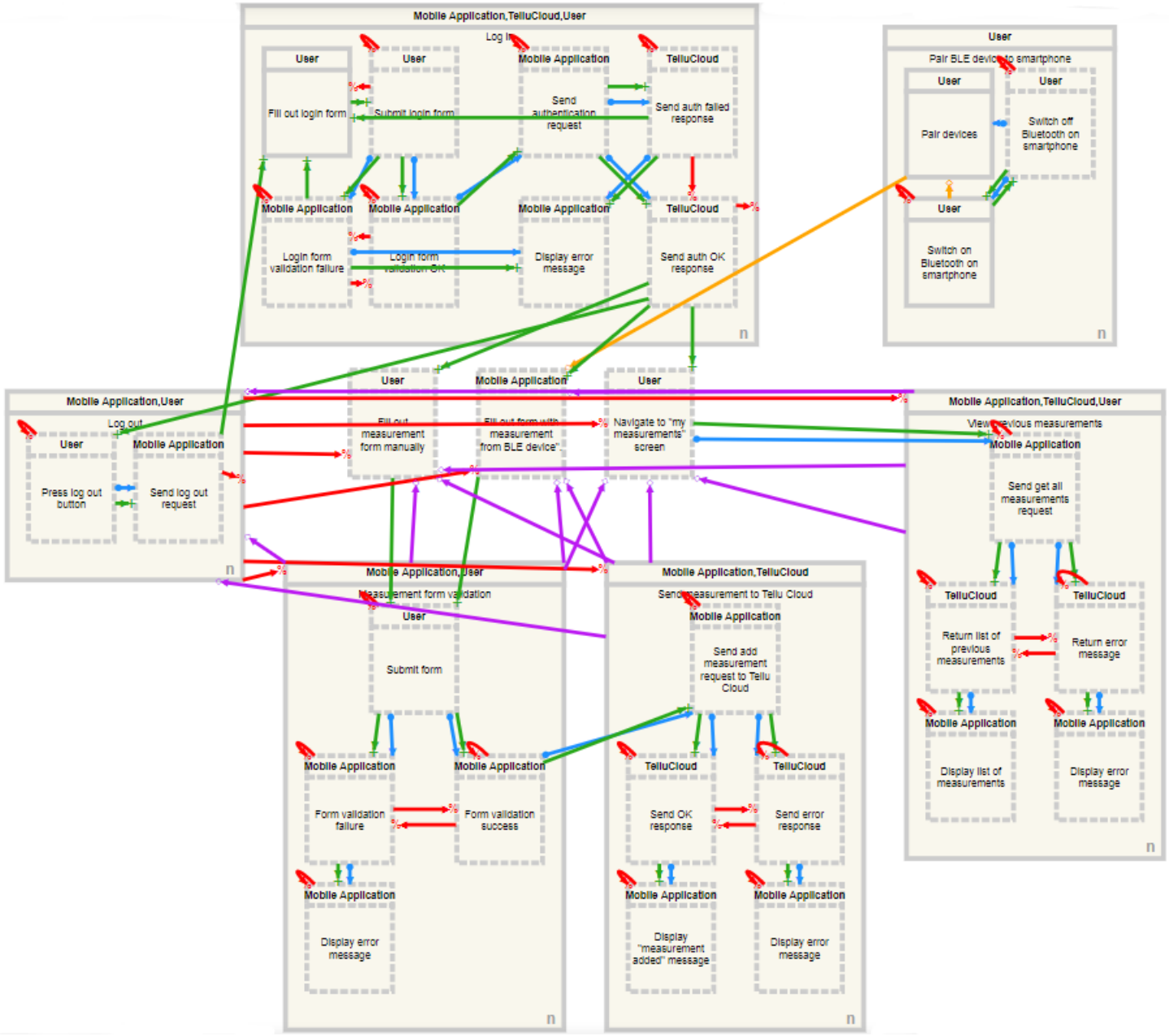

Figure 6.6: Final model of Tellu Diabetes App functionality as DCR Graph

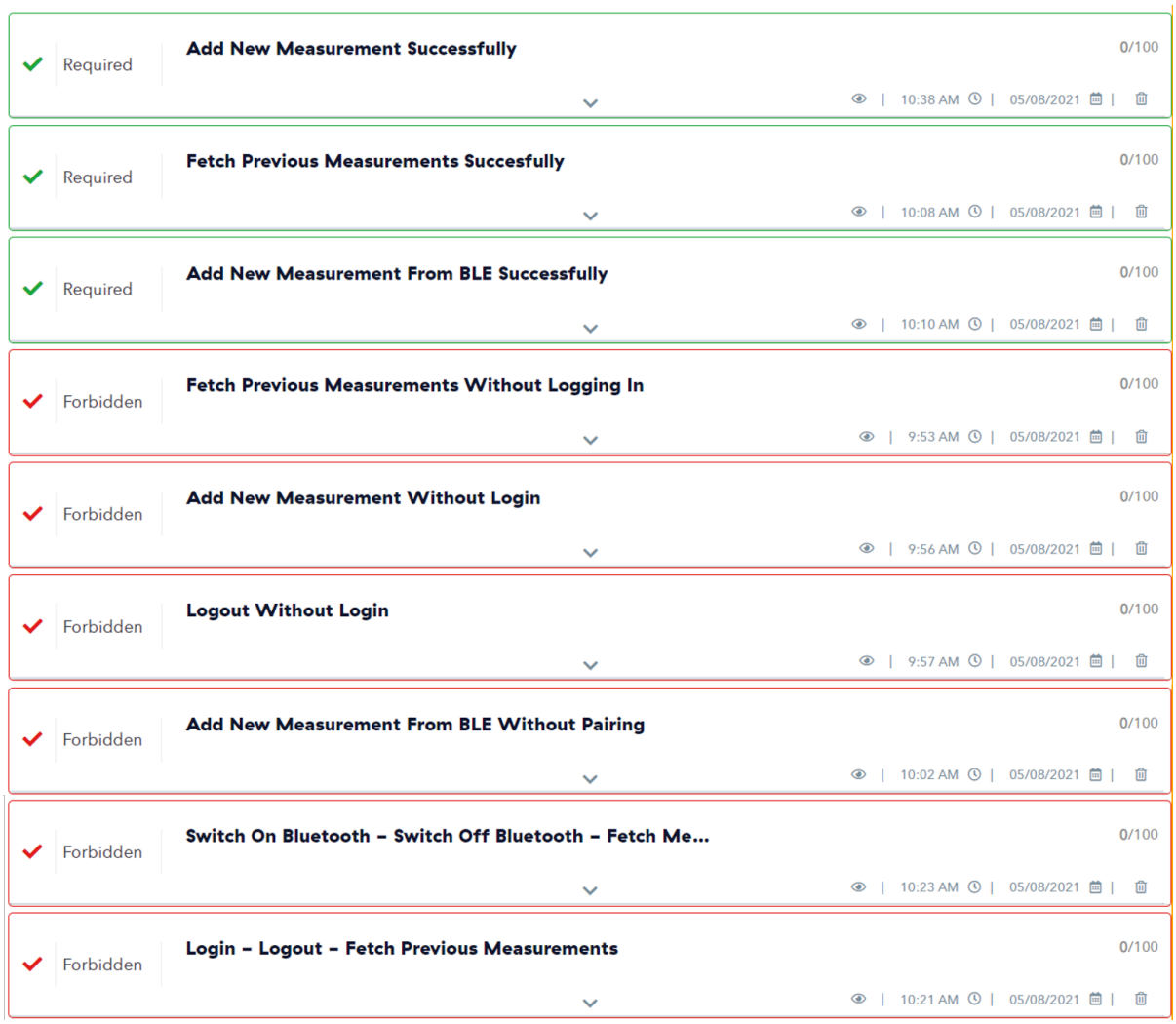

Figure 6.7: Examples of scenarios for Tellu Diabetes App model

The required scenarios are paths expressing the desired behaviour of the model. Figure 6.8 shows a swim lane diagram for one of the required scenarios, namely a successful attempt to submit a new measurement from the BLE device. In this scenario, the user logs in to the application, switches on Bluetooth on the smartphone and pairs it to the medical BLE device before adding a new measurement. As shown in the figure, all boxes (activities) are green, meaning that the scenario is accepting.

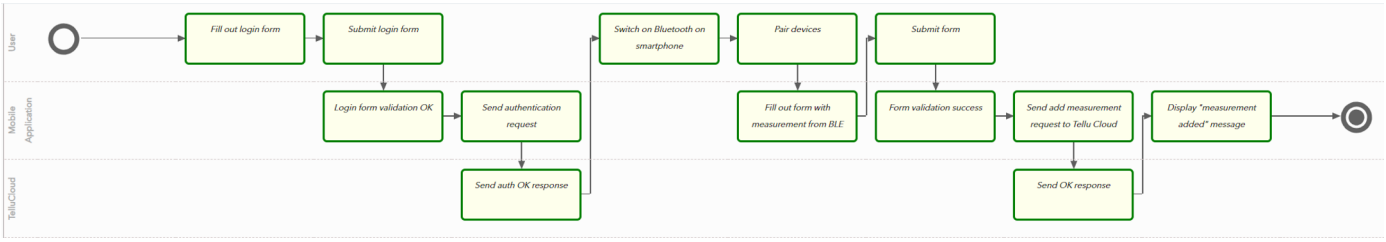

Figure 6.8: Swim lane diagram showing an accepting scenario

On the other hand, the forbidden scenarios are paths expressing the undesired behaviour of the model. Figure 6.9 demonstrates one of such scenarios. This scenario represents a path where the user logs in to the application, then logs out and attempts to navigate to the screen with previous measurements afterwards. This behaviour is undesired since only logged in users should access the application functionality, like fetching previous measurements. From the figure, we can see that this execution is not accepting in the model since one of the boxes ("Navigate to my measurements screen" activity) is marked yellow, meaning that executing this event violates the rules of the DCR Graph. It indicates that the model from Figure 6.6 contains correct relations to prevent this scenario from being accepting.

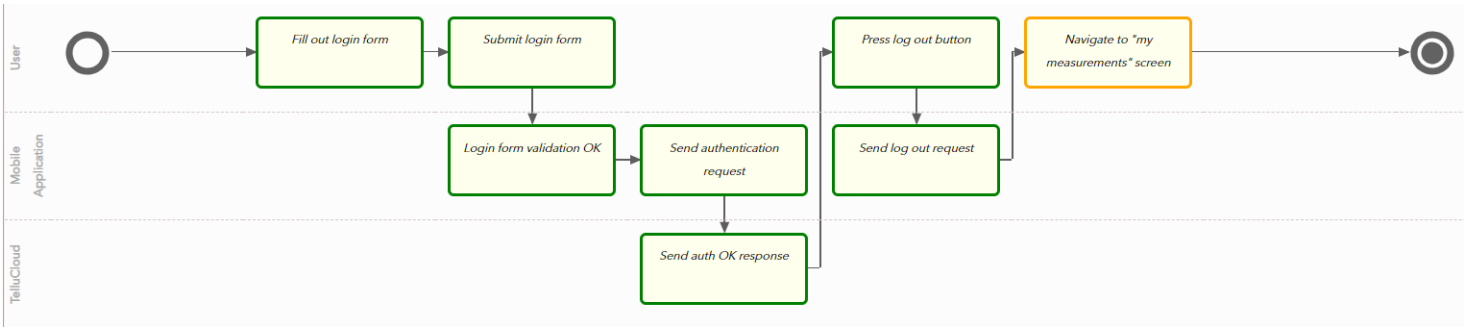

Figure 6.9: Swim lane diagram showing a forbidden scenario

Figure 6.7 lists several scenarios specified for testing purposes. However, it is only a fraction of all possible executions of the DCR Graph from Figure 6.6. Therefore, stating that the model behaves correctly in all possible situations based exclusively on the scenarios would be wrong. Thus, we need to use another method to verify the model's correctness.

6.4.2 Verification through execution of all possible paths

As Sections 6.1 and 6.2 explained, many of the desired properties depend on factors that lie outside of the mobile application. Verifying them would require lower level abstraction models of the specific parts of the system that the properties are related to. However, Property 1, "Only logged in users can access the application functionality related to viewing of personal

information", depends mainly on the application functionality and can be verified in the model from Section 6.3.3 using DCR Tool.

The property "Only logged in users can access the application functionality related to viewing of personal information" can be rephrased to "A user that has not logged into the application can not view any personal information". This sentence expresses a requirement about undesired behaviour that should not occur. As can be recalled from Section 2.5.2, this is, therefore, a safety property.

To verify a safety property, it must be proven that there are no executions among all possible executions where the undesired behaviour occurs. If at least one such execution can be found, then the property is not valid in the model. In the context of DCR Graphs, it means that if a sequence of events violating the desired behaviour exists, then this path will be a counterexample proving that the model is wrong. As we can recall from Section 3.5, the DCR Tool has an application that analyses all possible executions and helps to search for a scenario based on specified parameters, namely Scenario Search application. We will now attempt to use this application to verify "A user that has not logged into the application can not view any personal information" property by filling out the parameters in a way that expresses the undesired behaviour and running the search.

Since Tellu Diabetes App has multiple features that should not be accessed without logging in to the application, we must verify the property for each of the features. We will start with the functionality of accessing previous measurements without successful login. The successful login process is completed when TelluCloud sends an OK response ("Send auth OK response" activity in the model). The process of accessing previous measurements is initiated when the user goes to the "my measurements" screen of the application ("Navigate to "my measurements" screen" activity in the model). Thus, to find out whether the undesired situation when the user can access previous measurements without logging in, we can search for a path such that "Navigate to "my measurements" screen" is the goal activity and "Send auth OK response" is avoided in the execution. This is demonstrated in Figure 6.10. The result can be seen at the top of the figure. Scenario Search application returned a message saying that no such paths were found. It means that paths where the undesired sequence of events occurred, do not exist. Thus, we can conclude that the user can never access previous measurements without logging in first.

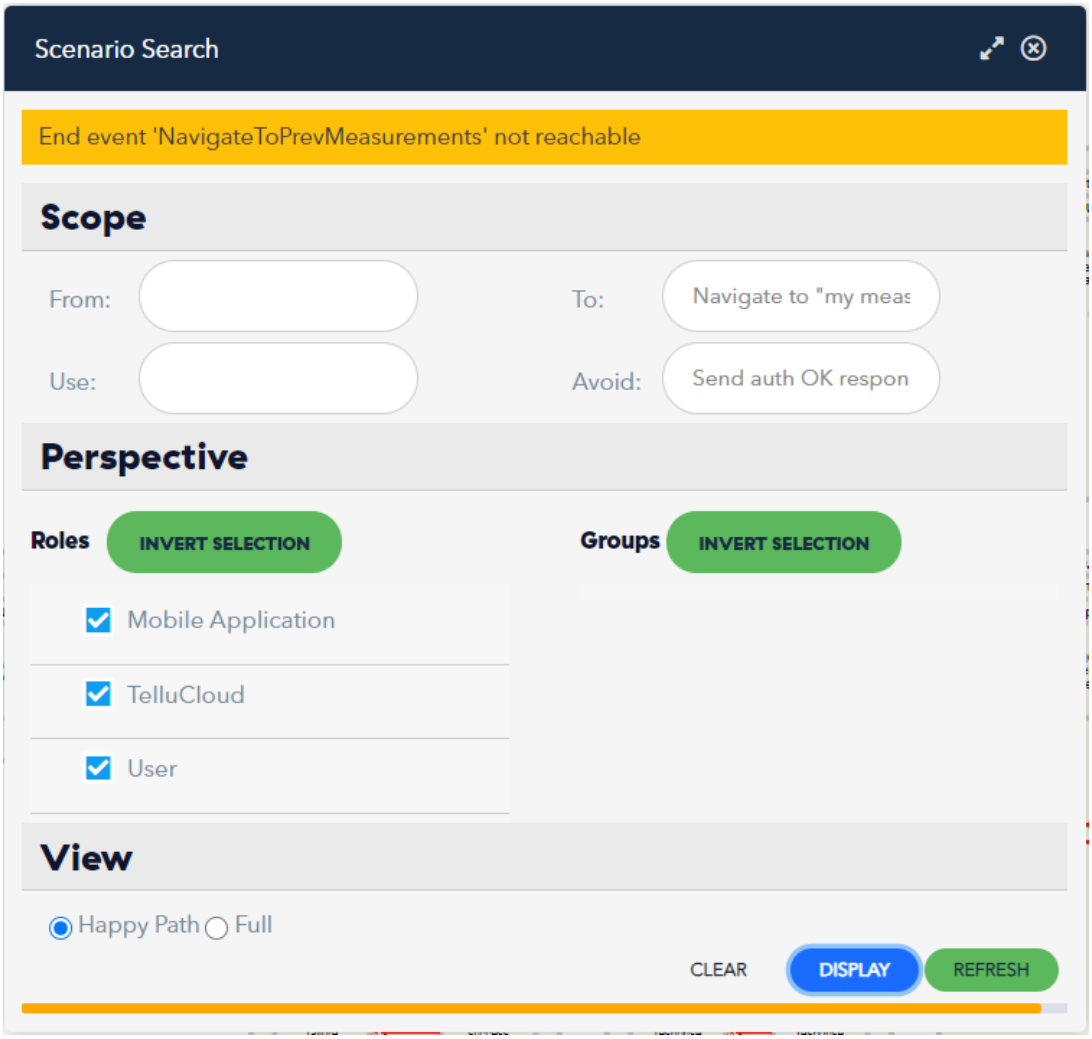

Figure 6.10: Scenario Search result for fetching previous measurements activity

The same procedure was repeated for the other features of the Tellu Diabetes App, "Fill out measurement form manually", "Fill out measurement form from BLE device" and "Press log out" as goal activities and gave similar results. They can be seen in Figure 6.11.

From the results of applying Scenario Search on the model, we can conclude that no features are available for a user who has not logged in previously. However, it is only sufficient to verify a part of "A user that has not logged into the application can not view any personal information" property. Scenario Search application helps us verify the situations where the "Log in" activity is avoided altogether. However, it does not consider executions where the user logs in first and then logs out. In this case "Send auth OK response" activity is not avoided since it is executed once. However, the user should still not be able to execute activities expressing other application functionalities (like "Navigate to "my measurements" screen"). This case demonstrates a significant limitation of

the Scenario Search application as a tool for verifying safety properties since its' parameters only allow to specify individual activities and not sequences and relations between them.

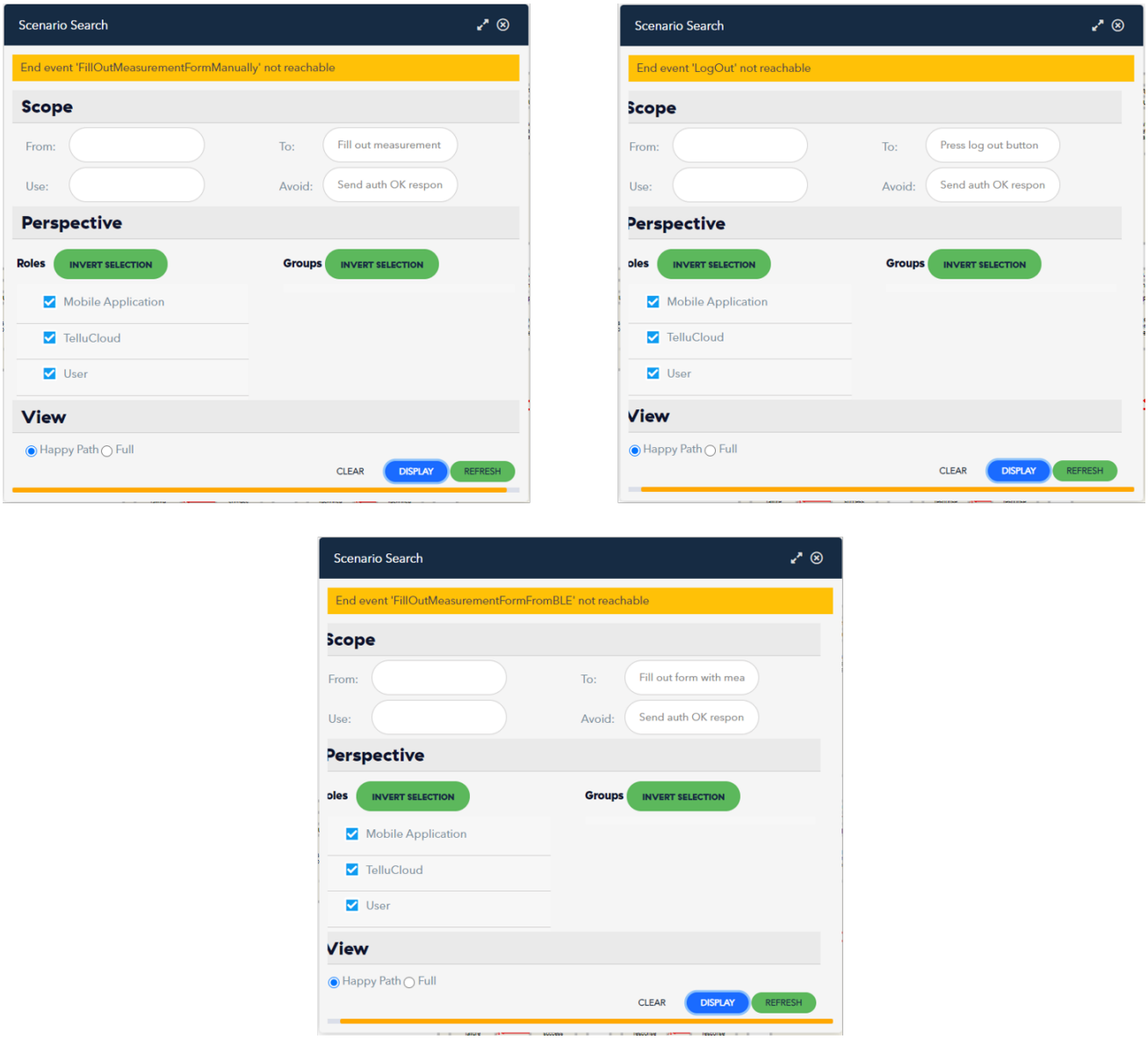

Figure 6.11: Scenario Search results for other application features

7 Discussion and future work

Chapters 5 and 6 explained the process of modelling and verification of the task implementation process and application functionality with DCR Graphs. They also presented considerations that were made and limitations that were met during the process, as well as the results of the verifications. This chapter summarises and reflects on the findings. The first two parts of the chapter are related to the two goals defined for this thesis, while the third part provides additional thoughts on using the DCR Tool. Further, the possible extensions of this work are outlined in the last part.

7.1 Towards satisfying requirements for software development process

The first goal of this thesis was to study how modelling and verification of software development process using DCR Graphs can help to satisfy safety standard requirements. As we can recall from Section 2.3.1, safety standards specify a number of requirements for each of the software development activities, specification, design, implementation, verification, validation and evolution. However, the main challenge pointed out by Tellu was that for each change in the code, it must be proved that the software still behaves as expected. Therefore, the main focus of the modelling process in Chapter 5 was on the evolution of the software, including re-verification.

First, the DCR Graph presenting the process of implementing an individual task (which could be a bug fix or new feature) was developed in Section 5.2. The final model was demonstrated in Figure 5.1. Then, the correctness of the model was verified as explained in Section 5.3 using several different methods available in the DCR Tool. It was proved that in all possible executions of the graph, the changes made during the implementation of the task go through two quality check mechanisms before they become a part of the main codebase.

The first mechanism modelled is peer review, the process where the developer working on a task asks a colleague (peer) to review the modifications and additions made in the code. As we may recall from Section 2.3.1, peer review is a static verification technique. The peer review process is initiated when the developer creates a pull request in the version control system and adds a reviewer to it. The peer can comment on

errors in the code, bad coding standards, structural flaws or other points of improvement and require changes that should be made to increase code quality. The verification process in Section 5.3 proved that only when the peer is satisfied and approves the pull request, the changes are merged into the main codebase. In other words, it was verified that all modifications in the code go through static analysis as a part of the verification process.

The second mechanism modelled is running tests automatically with the help of the build system. As we may recall from Section 2.3.1, testing is an example of dynamic verification techniques. When the changes are pushed to the remote repository, the build process gets triggered. This process consists of building the application, running all the tests that exist in the codebase and reporting the results. Re-running the tests for each modification ensures that the changes in the code did not break any existing functionality. The verification process in Section 5.3 proved that only if all the tests pass, the changes can be merged into the main codebase. Thus, it was verified that all modifications in the code undergo testing as a part of the verification process.

As presented in Table 2.3, both static analysis and testing are recommended for the lowest safety integrity level (SIL) and highly recommended for safety integrity level (SIL) higher than 2. The model developed in Section 5.2 shows that both techniques are used in the development process, so both of these requirements are satisfied by the model. Furthermore, since this process is repeated for each modification in the code, it also satisfies some of the requirements related to software evolution specified in Section 2.3.1. First, the model shows that the system is re-tested when new changes are introduced. In addition, the description of the task in the Jira ticket may serve as documentation of the modification and the reason for why it must be introduced.

All in all, the DCR Graph in Figure 5.1 and the following verification process shows that the process model satisfies several requirements for the software development process posed by the safety standards.

However, one of the weaknesses of modelling is that a model is only a reflection of reality. Verifying that safety standard requirements are satisfied by the model developed in Section 5.2 does not necessarily mean that the actual process satisfies them. For example, if version control system history contains at least one pull request that was merged to the main branch without approval, the actual process does not correspond to the model any longer.

Nevertheless, The DCR Graph and verification results can serve as documentation of the development process in formal mathematical language and be an addition to descriptions in natural language.

7.2 DCR Graphs for modelling of Internet of Things system

As Section 3.1 describes, the DCR Graphs were developed with the primary purpose of modelling and analysis of business processes in order to

increase their productivity and quality. A business process is a sequence of activities that must be performed by people or machines in order to achieve a business goal. Similarly, computer systems can be seen as a process consisting of a number of activities performed by the system components. However, DCR Graphs have not been widely used to model this kind of processes. Therefore, the second goal of this thesis was to investigate whether DCR Graphs are suitable for this purpose using a real system as the use case.

The process of modelling and verification of the Tellu Diabetes App, a mobile application that is a part of an Internet of Things system, was explained in Chapter 6. Based on the described work, this section will first address the similarities and differences between modelling a business process and computer system with DCR Graphs and which challenges the differences lead to. Then, some advantages of using DCR Graphs for the task are pointed out. Finally, the reflection on whether modelling with DCR Graphs can be a part of a software development process and contribute to safety standard compliance is presented.

As mentioned previously, components of computer systems perform a sequence of activities in a similar way as participants of a business process. When constructing a DCR Graph, each operation executed by a system can be seen as an event with the component executing the operation assigned as a role. The operations may have dependencies among each other, for example, "operation A must precede operation B" or "operation A triggers execution of operation B". These dependencies can in DCR Graphs be expressed using relations between activities. Therefore, the basic structural components of the DCR Graphs make it possible to model a computer system just like business processes to a large extent. This point is demonstrated by the DCR Graph in Figure 6.6 that models the functionality of the Tellu Diabetes App, its' communication with the services of TelluCloud and the user's interaction with it.

At the same time, a mobile application differs from a business process in an important respect. While a business process usually has a goal, representing the end of the process, a mobile application does not necessarily need to achieve any goal. Furthermore, the processes inside the mobile application should not have an end since a finished process would indicate a crash of the application. The two parts of this thesis can even exemplify this difference. While the process of task implementation presented in Chapter 5 has an end goal - completing the task and merging the changes into the main codebase - Tellu Diabetes App does not have such a goal. Of course, when a user initiates an operation in the application (for example, by submitting a new measurement), he has a goal of completing this operation. However, it is not the end goal of the process. On the contrary, once the user enters the Tellu Diabetes App, it should continue running until he chooses to leave the application. It means that the execution of the DCR Graph modelling the processes inside the Tellu Diabetes App could be infinite.

However, infinite executions do not pose a challenge for reasoning about the model. The Definition 3.4.4 provides a condition for an execution

to be accepting, namely that no required response event can continuously be included and pending without it happens or becomes excluded. This condition does not rely on a run/execution to be finite in order to determine whether it is accepting or not.

In addition, a mobile application differs from a business process in the way the current state of the process is represented. When describing the state of a business process, one is usually interested in which activities can be executed in the next step, must be executed to complete the process or which of them are blocked. On the other hand, for a mobile application, it is common to describe the state in terms of the current values of variables. For example, in Tellu Diabetes App, there could be a variable holding information about the current user of the application. If the variable's value is null, it indicates that no user is logged in, so the only functionality available to the user is to log in. When this variable is initialised as a result of a user logging in, it means that the user can access other features of the application. When the user logs out, the variable is reset to null value again. As we have seen, the current state of a DCR Graph is expressed by the marking M containing sets of executed, pending and included events. When modelling the application, it was, therefore, necessary to think of variable value state in terms of whether the activity causing the change in this state was executed or not. While applying this mindset for the first-time execution of "Log in" activity is easy, it is more challenging to consider the possibility of "Log out" activity happening afterwards. As described in Section 6.3.3, the result is an increased number of additional relations that make the model more complex.

Otherwise, the DCR Graphs contain the relations needed to express the relations between activities that were needed for our purpose.

The DCR Graphs as a tool for modelling the systems as a part of a software development process has several advantages. The first advantage is related to the agile development processes widely used in industry today. The mobile application itself and the system it is a part of can undergo many changes both during the development process and after the application is deployed for the users. Various errors and bugs may need to be fixed and new features introduced to the application. In this case, the DCR Graphs make it easy to make changes in the model as well, so that it reflects the modification of the application. In order to adjust the model, one needs to add new activities and relations to the graph or modify or delete existing ones.

However, for larger and more complex graphs, the process of modifying the model can be more difficult and error-prone. For example, the model of Tellu Diabetes App in Figure 6.6 contains a significant number of activities and relations between them. Even at this relatively high level of abstraction, it might be challenging to follow all the dependencies between activities without additional explanations, considering the number of boxes and arrows. When adding new activities, it can be easy to forget an essential relation to some of the existing activities and introduce a mistake in the model. Fortunately, the verification mechanisms, like specifying predefined scenarios that should be accepting or forbidden in the

process using DCR Swimlane Editor, can help to reassure that the graph still behaves correctly after new changes.

Another advantage of using DCR Graphs as a part of the software development process is that they have a graphical representation in addition to formal semantics. The graphical notation may make it easier for developers without logical and mathematical background or experience to create and understand the models. Therefore, usage of DCR Graphs could lower the costs of using formal methods as part of the software development process compared to other formal method tools that exclusively use hard-to-understand mathematical notations.

Furthermore, the use of DCR Graphs can contribute to satisfying some of the requirements for the use of formal methods posed by safety standards. As can be recalled from Section 2.3.1, formal methods are recommended during the software specification and design process for SIL2 and SIL3 and highly recommended for SIL4. The formal proof is also suggested as a part of the verification process. The model developed in Chapter 6 can be used to satisfy some of these requirements.

Demonstrating the functionality of the application, communication with backend services, and interactions between the user and the application, the model in Figure 6.6 serves as an overall formal model of the system. It presents the functional requirements of the Tellu Diabetes App, how the data flows between the system components and how the application reacts both to user inputs and the different responses from TelluCloud. Thus, it could satisfy the requirements for the use of formal methods during software requirement specification and the overall system architecture process. However, when it comes to designing individual components, the model should have a lower level of abstraction (for example, include more details on which operations are performed by which modules of the application). Such a model would be larger and more complex. To avoid this problem, one could create several models, each addressing a single functionality of the application. This way, the model could be more detailed but feasible to understand at the same time. However, the disadvantage of this approach is that the dependencies between different functionalities of the application would not be visible any longer.

7.3 Reflection on using the DCR Tool

Since the DCR Tool was used as the primary tool for modelling and verification in this thesis, it is in its place to comment on some findings related to the tool, including the limitations for the thesis work.

First, the DCR Graphs Community has created comprehensive documentation of DCR Graphs that has been useful during the work on this thesis. Ranging from simple and intuitive explanations on the website [14] to formal definitions in academic research papers, it provides an excellent base for understanding DCR Graphs. However, the amount and organisation of documentation on the website have occasionally made it challenging to find the desired information. In addition, since the DCR Graphs

technology is still developing, some of the formal descriptions were outdated and some features that are available in the DCR Tool have yet to appear in academic papers.

Second, the DCR Tool is easily accessible since it is web-based. Since it does not require installation of any software, it has a lower threshold for getting started for anyone who wants to explore the possibilities of DCR Graphs. In addition, the DCR Tool provides a guide on using the tool, called DCR Tour. It is an interactive guide that demonstrates the basic functionality of the graphical tool required to create and edit a DCR Graph. The DCR Tool also keeps a log of all changes the user makes to the graph, so it is easy to revert the changes in case of a mistake. The web-based tool also makes it easy to share the graphs with other people and collaborate on simulations of the graphs.

Furthermore, the App Store provides several useful applications in addition to the core functionality of the DCR Tool. Some of the applications, like Scenario Search and Dead-end Analyzer, were already described in detail throughout the thesis. Another application, called Publisher App, can be used to generate documentation of the model containing information about activities, roles, relations and scenarios. This feature allows collecting both description of the model in natural language and the model itself in one file instead of keeping it in several different places.

Despite all the benefits provided by the DCR Tool, it contains some bugs and problems with long loading times and has a somewhat unintuitive user interface for particular features. The following errors were encountered and had an impact on the thesis work:

- Somewhat long loading time when logging into the DCR Tool and opening the graphs. During the editing of the graphs, the page sometimes hangs up, so the user is forced to reload the page and can lose the recent changes.
- When using DCR Swimlane Editor in one window and making changes to the graph in another window, the user has to refresh the DCR Swimlane Editor window to get the most recent version of the graph. Before the need for refreshing was discovered, it was a source of confusion about the results of scenario validations.
- Scenario Search and Dead-end Analyzer applications open in their own dialogue boxes. When running one of the applications first and then the other, the result of running the second application appears in the first application's dialogue box. It is confusing since the user does not get any feedback in the current dialogue box.
- The DCR Swimlane Editor displays the results of scenario validation in a snack bar that appears on the screen just for a couple of seconds. If the validation fails, the content of the snack bar is an error message explaining the cause of failure. Sometimes the error message is too long to manage reading it before the snack bar disappears. Thus, making correct modifications in the scenario or the graph becomes more challenging.

However, the DCR Tool evolves continuously and the design was improved significantly while the thesis work was in progress.

Finally, the DCR Tool had a couple of limitations related to the verification process in this thesis. Chapters 5 and 6 explained how the models of the task implementation process and Tellu Diabetes App functionality were verified using different features of the DCR Tool, among others, the Scenario Search application. As we saw from these chapters, the Scenario Search application allowed us to find paths through the graph by specifying parameters like end goal activity (in the "To" field) and activities that should be used or avoided in the execution ("Use" and "Avoid" fields). This feature was used to verify some of the safety properties that had to be satisfied in the models. However, this way of specifying properties has limitations. Section 5.3.2 showed that there is no way to specify the end of the process as a "To" event. Further, Section 6.4.2 mentioned that it is impossible to express any more complex relations between activities other than that they should be used or avoided.

The DCR Tool clearly focuses on the verification of models by specifying individual scenarios with DCR Swimlane Editor and simulations. Scenario Search and Dead-end Analyzer applications provide some functionality for verification of properties for all possible executions of the graph. However, it is limited to a particular type of properties and may not be intuitive to understand. In addition, these applications have a limitation related to time-outs that were visible for verification of larger and more complex models as explained in Section 5.3.2.

7.4 Future work

Considering the reflections and limitations provided previously in this chapter, some items can be pointed out as possible extensions of the work performed in the thesis. These items are presented in the following list.

1. As explained in the previous section, the DCR Tool does not provide extensive enough support for analysis of all possible executions of the graph. Instead, it could be an idea to use model checkers for this purpose. Bringing this idea to life requires mapping the DCR Graphs to the modelling language used by the model checker first. Some work on the mapping and verification of some properties for simple DCR Graphs using SPIN model checker and verification modelling language Promela has already been described in [38]. However, scaling it up for larger and more complex DCR Graphs and attempting to integrate it in the DCR Tool would be an interesting task.

2. As Section 7.2 explained, the model of Tellu Diabetes App constructed in this thesis has too high abstraction level to provide significant benefits in terms of satisfying safety standard requirements for the use of formal methods during the design of individual software modules. Therefore, it would be interesting to study whether the construc-

ted model can be extended further with more code-specific activities and relations while still being feasible to understand and modify.

3. The task implementation process modelled in Chapter 5 and the functionalities of the application modelled in Chapter 6 go hand in hand since newly introduced modifications in the code can cause a change in the application functionality. Therefore, an idea could be to investigate whether these models can be joined, so the result of activities executed during the task implementation process can be reflected in the model of the application.

8 Conclusion

This thesis's main aims were to study the use of DCR Graphs to ensure that a development process satisfies safety standard requirements and investigate whether DCR Graphs are suitable for modelling an Internet of Things system. With the help of the Tellu Diabetes App as the use case, two models were constructed and verified using the DCR Tool.

The first goal was addressed by modelling and verifying the process of introducing new changes in the code of the application and looking at which parts of safety standard requirements could be satisfied with the help of the model. The discussion showed that the model satisfies several requirements for the software verification and evolution processes and could serve the purpose of documenting the process. However, the possible mismatch between the model and the actual process was pointed out as a weakness.

Achieving the second goal involved constructing a DCR Graph representing the functionalities of the Tellu Diabetes App. This process and further discussion showed that DCR Graphs are suitable for modelling an Internet of Things system in a similar way as business processes to a large extent. However, some challenges related to differences between computer systems and business processes and the size and complexity of the model were encountered. In addition, dynamicity and graphical notation of the DCR Graphs were pointed out as advantages for using them for modelling purposes as a part of the software development process. Finally, the model developed could contribute to satisfying some of the requirements for the software development process. Although, only the requirements related to top-level system design activities were covered by the model in this thesis.

To sum up, this thesis touched upon a broad spectrum of topics, like the Internet of Things, safety standards, software development process methodologies and formal methods. It also required insight into various software development tools and understanding of DCR Graphs syntax and semantics. This variety made the thesis work was an exciting process with a great learning outcome. Theoretical knowledge and practical experience gained throughout the process are invaluable and will surely be helpful for my future career in Information Technology.

Bibliography

[1] Atlassian. *Jira: Issue amp; Project Tracking Software*. URL: https://www.atlassian.com/software/jira.

[2] Atlassian. *The Git solution for professional teams*. URL: https://bitbucket.org/product/.

[3] Arshdeep Bahga and Vijay Madisetti. *Internet of Things: A Hands-On Approach*. Atlanta, GA, USA: Vijay Madisetti, 2014. ISBN: 0996025529.

[4] Christel Baier and Joost-Pieter Katoen. 'Principles of model checking'. In: 2008.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201616416.

[6] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: http://www.agilemanifesto.org/.

[7] Béatrice Bérard et al. 'Systems and Software Verification, Model-Checking Techniques and Tools'. In: 2001.

[8] Grady Booch, James Rumbaugh and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005. ISBN: 0321267974.

[9] Ewout Brandsma. 'Applicability of Use Case and Building Blocks for Assisted Living Demonstrated (Iteration 1)'. unpublished. N.D.

[10] Cristiano Calcagno et al. 'Moving Fast with Software Verification'. eng. In: *NASA Formal Methods*. Vol. 9058. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 3–11. ISBN: 3319175238.

[11] Carlos Cetina et al. 'Tool Support for Model Driven Development of Pervasive Systems'. In: *Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES'07)*. 2007, pp. 33–44. DOI: 10.1109/MOMPES.2007.17.

[12] Edmund M. Clarke and Jeannette M. Wing. 'Formal Methods: State of the Art and Future Directions'. In: *ACM Comput. Surv.* 28.4 (Dec. 1996), pp. 626–643. ISSN: 0360-0300. DOI: 10.1145/242223.242257. URL: https://doi-org.ezproxy.uio.no/10.1145/242223.242257.

[13] Bruno Costa, Paulo F. Pires and Flávia C. Delicato. 'Modeling IoT Applications with SysML4IoT'. In: *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2016, pp. 157–164. DOI: 10.1109/SEAA.2016.19.

[14] *DCR Graphs Documentation*. Feb. 2021. URL: https://documentation.dcr.design/documentation (visited on 14/05/2021).

[15] Søren Debois, Thomas Hildebrandt and Lene Sandberg. 'Experience Report: Constraint-Based Modelling and Simulation of Railway Emergency Response Plans'. eng. In: *Procedia computer science* 83 (2016), pp. 1295–1300. ISSN: 1877-0509.

[16] Søren Debois, Thomas Hildebrandt and Tijs Slaats. 'Hierarchical Declarative Modelling with Refinement and Sub-processes'. In: vol. 8659. Sept. 2014. ISBN: 978-3-319-10171-2. DOI: 10.1007/978-3-319-10172-9_2.

[17] Søren Debois et al. 'A Case for Declarative Process Modelling: Agile Development of a Grant Application System'. In: *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*. 2014, pp. 126–133. DOI: 10.1109/EDOCW.2014.27.

[18] Søren Debois et al. 'Hybrid Process Technologies in the Financial Sector: The Case of BRFkredit'. eng. In: *Business Process Management Cases*. Management for Professionals. Cham: Springer International Publishing, 2017, pp. 397–412. ISBN: 9783319583068.

[19] *EU data protection rules*. July 2020. URL: https://ec.europa.eu/info/law/law-topic/data-protection/eu-data-protection-rules_en.

[20] Nicolaie Fantana et al. 'Internet of Things - Converging Technologies for Smart Environments and Integrated Ecosystems'. In: Jan. 2013, pp. 153–204. ISBN: ISBN 978-87-92982-73-5 (print) ISBN 978-87-9282-96-4 (ebook).

[21] *Generate anything from any EMF model*. URL: https://www.eclipse.org/acceleo/.

[22] Stefan Hallerstede, Peter Gorm Larsen and John Fitzgerald. 'A Non-unified View of Modelling, Specification and Programming'. In: *Leveraging Applications of Formal Methods, Verification and Validation: Part 1*. Ed. by Tiziana Margaria · Bernhard Steffen. https://doi.org/10.1007/978-3-030-03418-4_4. ISoLA 2018. Limassol, Cyprus: Springer Nature Switzerland AG, Nov. 2018, pp. 52–68.

[23] Nicolas Harrand et al. 'ThingML: a language and code generation framework for heterogeneous targets'. In: Oct. 2016, pp. 125–135. DOI: 10.1145/2976767.2976812.

[24] Thomas Hildebrandt and Raghava Rao Mukkamala. 'Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs'. In: *PLACES* 69 (Oct. 2011). DOI: 10.4204/EPTCS.69.5.

[25] Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. 'Declarative Modelling and Safe Distribution of Healthcare Workflows'. eng. In: *Foundations of Health Informatics Engineering and Systems*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 39–56. ISBN: 9783642323546.

[26] Thomas Hildebrandt, Raghava Rao Mukkamala and Tijs Slaats. 'Nested Dynamic Condition Response Graphs'. In: vol. 7141. Apr. 2011. DOI: 10.1007/978-3-642-29320-7_23.

[27] Thomas Hildebrandt et al. 'Dynamic Condition Response Graphs for Trustworthy Adaptive Case Management'. eng. In: *On the Move to Meaningful Internet Systems: OTM 2013 Workshops*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 166–171. ISBN: 3642410324.

[28] 'ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary'. In: *ISO/IEC/IEEE 24765:2010(E)* (2010), pp. 1–418.

[29] Stephen A Jacklin. 'Certification of Safety-Critical Software Under DO-178C and DO-278A'. eng. In: *Certification of Safety-Critical Software Under DO-178C and DO-278A*. 2012.

[30] Christopher Jensen. *Nissan Recalls 3.5 Million Vehicles for Airbag Problems*. Apr. 2016. URL: https://www.nytimes.com/2016/04/30/business/nissan-recalls-3-5-million-vehicles-for-airbag-problems.html.

[31] Craig Larman and Victor Basili. 'Iterative and Incremental Development: A Brief History'. In: *Computer* 36 (July 2003), pp. 47–56. DOI: 10.1109/MC.2003.1204375.

[32] Jie Lin et al. 'A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications'. In: *IEEE Internet of Things Journal* 4.5 (2017), pp. 1125–1142. DOI: 10.1109/JIOT.2017.2683200.

[33] Knud Lasse Lueth. *State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time*. Nov. 2020. URL: https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/.

[34] Somayya Madakam, R Ramaswamy and Siddharth Tripathi. 'Internet of Things (IoT): A Literature Review'. In: *Journal of Computer and Communications* 3 (Apr. 2015), pp. 164–173. DOI: 10.4236/jcc.2015.35021.

[35] Souad Marir, Faiza Belala and Nabil Hameurlain. 'A Formal Model for Interaction Specification and Analysis in IoT Applications: 8th International Conference, MEDI 2018, Marrakesh, Morocco, October 24–26, 2018, Proceedings'. In: Sept. 2018, pp. 371–384. ISBN: 978-3-030-00855-0. DOI: 10.1007/978-3-030-00856-7_25.

[36] *Merging Modeling with Programming*. URL: https://cruise.umple.org/umple/.

[37] Robin Milner. *The Space and Motion of Communicating Agents*. Vol. 20. Jan. 2009. ISBN: 978-0-521-73833-0. DOI: 10.1017/CBO9780511626661.

[38] Raghava Rao Mukkamala. 'A formal model for declarative workflows: dynamic condition response graphs'. PhD thesis. 2012. URL: https://raghavamukkamala.github.io/files/pubs/DCRGraphs-raghava-PhD-thesis.pdf.

[39] Chris Newcombe et al. 'How Amazon web services uses formal methods'. eng. In: *Communications of the ACM* 58.4 (2015), pp. 66–73. ISSN: 0001-0782.

[40] Ammar Osaiweran et al. 'Evaluating the effect of a lightweight formal technique in industry'. eng. In: *International journal on software tools for technology transfer* 18.1 (2016), pp. 93–108. ISSN: 1433-2779.

[41] Ammar Osaiweran et al. 'Experience Report on Designing and Developing Control Components Using Formal Methods'. In: *FM 2012: Formal Methods*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 341–355. ISBN: 978-3-642-32759-9.

[42] Chayan Sarkar et al. 'DIAT: A Scalable Distributed Architecture for IoT'. In: *IEEE Internet of Things Journal* 2.3 (2015), pp. 230–239. DOI: 10.1109/JIOT.2014.2387155.

[43] Ken Schwaber. 'SCRUM Development Process'. In: *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA*. 1995, pp. 117–134.

[44] David J. Smith. *The safety critical systems handbook : a straightforward guide to functional safety: IEC 61508 (2010 edition), IEC 61511 (2016 edition) related guidance, including machinery and other industrial sectors*. eng. Amsterdam, Netherlands, 2016.

[45] Alistair Smout and David Shepardson. *New 737 MAX software flaw found during tests, Boeing sticks to return timeline*. Feb. 2020. URL: https://www.reuters.com/article/us-boeing-737max-idUSKBN20026S.

[46] Ian Sommerville. *Software Engineering*. 10th. Pearson, 2015. ISBN: 0133943038.

[47] John A. Stankovic. 'Research Directions for the Internet of Things'. In: *IEEE Internet of Things Journal* 1.1 (2014), pp. 3–9. DOI: 10.1109/JIOT.2014.2312291.

[48] DSDM Consortium Stapleton and Jennifer Stapleton. *DSDM: A Framework for Business-Centered Development*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321112245.

[49] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. third. distributed-systems.net, 2017.

[50] *Therac-25*. Apr. 2021. URL: https://en.wikipedia.org/wiki/Therac-25.

[51] Itorobong S. Udoh and Gerald Kotonya. 'Developing IoT applications: challenges and frameworks'. In: *IET Cyber-Physical Systems: Theory Applications* 3.2 (2018), pp. 65–72. DOI: 10.1049/iet-cps.2017.0068.

[52] Erik Van Veenendaal, Dorothy Graham and Rex Black. *Foundations of Software Testing: Istqb Certification*. 3rd. Delmar Learning, 2012. ISBN: 1408044056.

[53] Mark Weiser. 'The Computer for the 21st Century'. In: *Scientific American* 3 (Sept. 1991), pp. 94–104. DOI: 10.1038/scientificamerican0991-94.

[54] Jim Woodcock et al. 'Formal Methods: Practice and Experience'. In: *ACM Comput. Surv.* 41.4 (Oct. 2009). ISSN: 0360-0300. DOI: 10.1145/1592434.1592436. URL: https://doi-org.ezproxy.uio.no/10.1145/1592434.1592436.

[55] *Xpand*. URL: https://wiki.eclipse.org/Xpand.