Correct by Construction

Author:
J. Maree FOURIE

January 20, 2020

"Writing is hard; writing is hard because thinking is hard. It is easier to think you are thinking."

Leslie Lamport

Abstract

Correct by Construction

by J. Maree FOURIE

ToDo. ...

Contents

List of Figures

List of Tables

List of Abbreviations

CbyC	**C**orrect **by** **C**onstruction
TLA	**T**emporal **L**ogic **A**ctions

Chapter 1

Process and Technique

Software development can be seen as having two aspects:

- process and
- technique.

Process describes the people involved in developing the software and the interactions between them. The process can describe things like team structures, work planning, reviewing work done and delivering the software.

Technique describes technical aspects of software development. Technique can describe things like what programming languages and technologies to used, how the software is structured and tested, and also how the software is delivered.

Process and technique can overlap and inform each other as in the case of software deployment. If we wanted to do continuous delivery we would have to review the completed software on an as needed basis and not a fixed schedule (process). We might then also structure the software as micro-services and not a monolith application (technique).

Some development methodologies focus more on process and others more on technique.

Agile software development (Agile) Agile focuses on delivering useful software in a timely manner by focusing on process. Agile's only opinion on technique is that the design should be kept simple.

Correct by Construction (CbyC) CbyC uses formal methods as a technique along with a process that aligns with the formal methods. Formal methods are mathematical rigorous techniques used to validate software designs and code.

If we combine Agile process with CbyC technique and process we can have a methodology that is both rigorous and productive.

Chapter 2

Correct by Construction (CbyC)

2.1 Overview

The CbyC software development methodology proposes to limit software defects by:

1. making it difficult to introduce errors, and
2. detecting and removing errors as early as possible.

CbyC technique emphasises the use of mathematically rigorous tools. CbyC proposes using mathematical formal language to define the specification and high-level design of a system. This provides a precise description of the system's behaviour and a precise model of its characteristics. Using a mathematical language also allows the use of automated tools to verify the specification and design [4].

CbyC code is designed around information flow. The information flow is defined using a contract-based notation. The contract-based notation is used to define the abstract state of the program and the information relationships across boundaries. CbyC proposes using programming languages and tools that allow for the code and contracts to be validated using static analysis [4].

2.2 Correct by Construction Process

The CbyC process is described in terms of the artefacts it produces (Figure 2.1):

- System requirements specification;
- Formal specification;
- Formal design;
- System test specification;
- INFORMED design;
- Code;

Using the CbyC process every process artefact can be validated.The semantic gap between process artefacts are kept small. This makes it possible to show that later process artefacts conform to earlier process artefacts [3].

We now describe each artefact and how it is created.

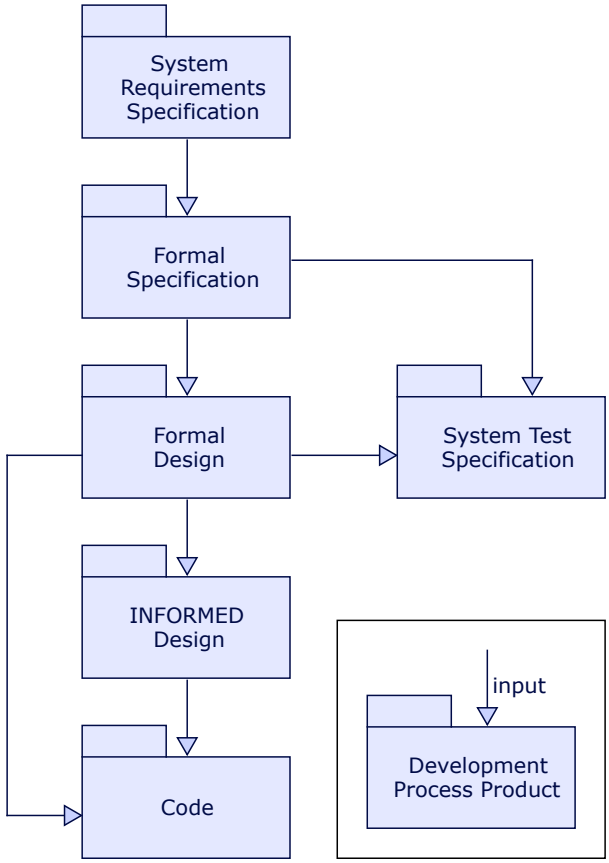

FIGURE 2.1: CbyC process artefacts.

2.2.1 System Requirements Specification

The requirements analysis identifies the needs of the stakeholders, the desired behaviour of the system, and any non-behavioural system characteristics [3].

Requirements management extends throughout the system's development, but it is most significant at the beginning, where it is used to identify [3]:

- the stakeholders, who have an interest in the development and use of the system;
- the system boundary, to clarify the scope of the project and the interfaces to external systems;
- the expected use, in terms of interactions between users and the system;
- system properties, such as security properties, performance properties, etc.

Each requirement should be traceable through every process artefact [3]. The requirements are compiled into the System Requirements Specification document.

The System Requirements Specification is created to [3]:

- early in the project clarify the system's boundary (what is in scope and what is out of scope, and the necessary interfaces to external systems);
- agree on the system requirements with all of the stakeholders;

- document the requirements with enough procession so that the Formal Specification can be developed with minimal customer input;
- clarify and document the assumptions about the behaviour the environment external to the system being developed;
- identify and manage conflicting expectations between stakeholders.

2.2.2 Formal Specification

The aim of the Formal Specification is to unambiguously describe what the system will do. This is to help the developer and the client gain a common understanding of the system [3].

The level of abstraction is important. The Formal Specification should not describe the system's implemented. Internal details are deliberately left very abstract. Interactions with the external environment are specified, but may also be abstract [3].

The Formal Specification is written in mathematical notation with an English narrative. The specification is divided into small components that can be reasoned about individually. The small components are then combined to describe the system as a whole [3].

The Formal Specification is created because it [3]:

- provides an unambiguous description of what the system does. This is important for gaining client approval of the system's behaviour.
- can be shown to be complete.
- can be formal verified, i.e. it can be proved consistent.

2.2.3 Formal Design

The aim of the formal design is to elaborate the abstract aspects of the Formal Specification to explain how the system will be implemented. The Formal Design describes the system in terms of concrete state and operations using types that are easily implemented. The Formal Design is the source of the required functional behaviour used during implementation [3].

The design is written in the same mathematical notation as the Formal Specification. This means that the design can be formally verified and errors can be uncovered before implementation starts [3].

The reasons for producing a Formal Design are [3]:

- It provides an unambiguous description of how the system does what the formal specification requires.
- It can be shown to be complete.
- It can be formal verified, i.e. it can be proved consistent.

2.2.4 System Test Specification

The aim of the System Test it to demonstrate that the system has the correct behaviour as specified in the Formal Specification. This differs from the goals of acceptance testing which is designed to demonstrate that the System meets its requirements. System Testing aims to achieve 100% coverage of the formal specification. Thus so all possible behaviours described in the formal specification should be exercised at least once [3].

The Formal Design is a refinement of the Formal Specification. Therefore the Test Specification can also be written against the Formal Design if you want to tests details of the design [3].

All System tests are specified in a System Test Specification prior to their execution. The tests are specified as scenarios that might occur in typical usage of the system. The specification includes documentation of the expected outcome of the test. System test specifications also traces each test to the components of the Formal Specification/Design that the test attempts to exercise [3].

Where code coverage metrics need to be captured, this is be done during the System test. This allows us to question the use of any code that cannot be covered by a system test. If the code is valid then focused unit tests should be added to cover the code [3].

The reasons for producing a System Test Specification are [3]:

- A system test focuses on testing the behaviour of the whole system against the expected (specified) behaviour.
- A system test is likely to find faults caused by the incorrect interaction of modules within the system.
- System testing complements static analysis, in that it confirms the dynamic behaviour.

2.2.5 INFORMED Design

CbyC design methodology is based on information flow. The information flow is defined using a contract-based notation. The contract-based notation is used to define the abstract state of the program and the information relationships across boundaries [4].

Consideration of information flows at the design stage leads to programs with the desirable properties of abstraction, encapsulation, high cohesion and loose coupling [3].

The **IN**formation **F**low **OR**iented **ME**thod of object **D**esign (INFORMED) design provided an architectural framework in which to perform the implementation. Consideration of information flows at the design stage results in programs with the desirable properties of abstraction, encapsulation, high cohesion and loose coupling [3].

The INFORMED design aids maintenance and upgrades of the software by providing a route-map from the Formal Design to the code [3].

The reasons for producing an INFORMED Design are [3]:

- It focuses on the system architecture and ensures that the architecture fits the information flow model.
- It provides the mapping from the Formal Design to the Code before writing the code.
- It complements the Formal Design without duplicating functional information.

2.2.6 Code Implementation

We start by writing the module specification. The specification is the public interface and the contracts governing the global state of the module. The contacts specify how inputs are allowed to influence outputs [3].

After writing the specification we implement the module body. The Formal Design is detailed enough to make that the mapping from the design to the code simple [3].

Modules providing infrastructure are developed early. Module are implementation in a order that allows system level functionality to be added incrementally. This means that a basic system can be built as soon as possible and functionality is added in subsequent builds. This has the advantage of addressing code integration risks as early as possible [3].

By using languages that allows static analysis we can evaluate the code contracts at build time. Thus if the code builds it is correct as specified by the contracts [3].

Chapter 3

Agile and Correct by Construction

3.1 Agile Software Development

3.1.1 Manifesto for Agile Software Development

The manifesto sets out the overarching principles of agile software development [1]:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - **Individuals and interactions** *over* processes and tools
> - **Working software** *over* comprehensive documentation
> - **Customer collaboration** *over* contract negotiation
> - **Responding to change** *over* following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

3.1.2 Agile Software Development using Scrum

Scrum is an agile project management methodology. It focuses on short feedback loops all development process. The client is involved in all iterations of the process ensuring that the completed project meets the client's needs and expectations [10].

Scrum roles

There are three roles in Scrum [10]:

Product Owner: This person represents the customer's interest, and determines the goal of each iteration. It is the product owner's responsibility to prioritise the different tasks ensuring that the most important functionality is implemented first.

Scrum Master: It is the Scrum Master's responsibility to facilitate the Scrum process and protect the agile principles. He also needs to protect the team by removing any impediments encountered by the team, ensuring that the team is not distracted from the task at hand.

Team: A team consists of five to nine people working together to create a functional system which satisfies the user needs. The team consists of people with a broad set of competences so that the team is self-organised and contained.

Activities and Artifacts

Scrum has several activities, that describe the development workflow, and artifacts, that are produced by the process [10]:

Product vision: This is the initial idea phase. The vision of the product is defined during this process.

Product backlog: This is a prioritised list of artifacts that are needed in the product. The artifacts are described as user stories, the story describes how we are adding value for the user.

Sprint: This is the development iteration. Sprints are time boxed between two to six weeks.

Sprint planning: The first day of each sprint is used to plan what is going to be done during the sprint.

Sprint backlog: These are the stories, from the product backlog the team has committed to implementing in the sprint.

Daily Scrum meeting: This is a short, 15 minutes, daily meeting used to plan the days work and checking on the state of the stories in the sprint.

Sprint review: The meeting is held at the end of the sprint to show the product owner the work that has been completed during the sprint.

Sprint retrospective: This meeting is held after the sprint review. The focus of the meeting is not on the work completed during the. The purpose of the meeting is to discuss the people, relationships, processes, and tools in order to improve the team's effectiveness.

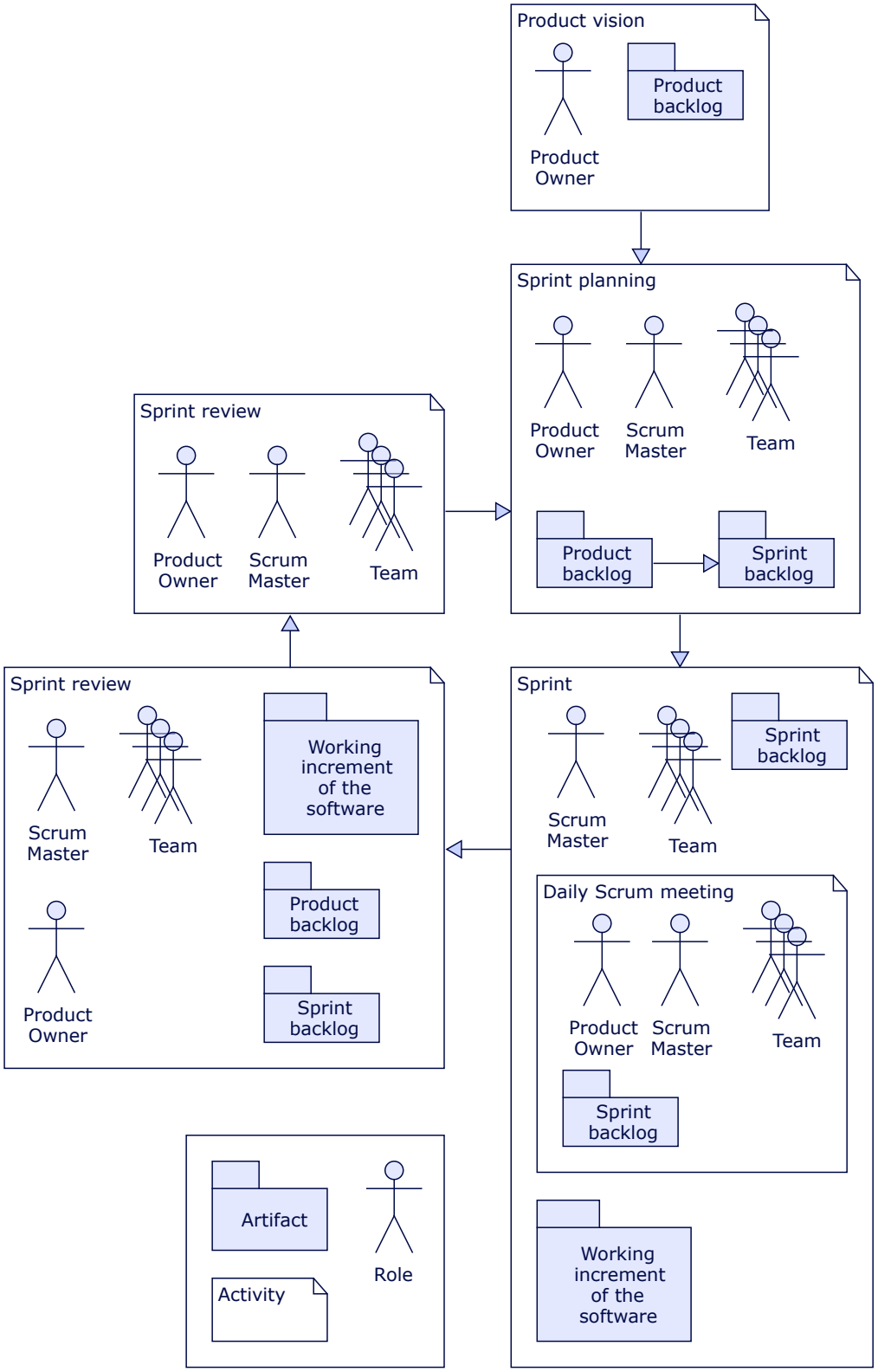

FIGURE 3.1: The scrum workflow.

3.2 Using CbyC in Agile Scrum

The strived for characteristic of agile software development is the ability to react to change. With scrum we will have to implement the change in order to test it. If we evaluate this against the Agile manifesto:

Responding to change: In order to evaluate a change: a story has to be create, and the story has to be refined and sized. Then, depending on the priority, existing work has to be moved to make resources available to implement the change.

Working software: We have to implement the requested change in order to validate it. If the change does not work we might end up with invalid or non-functional software.

Customer collaboration For a change to be actioned the team will reason through it on a whiteboard. This reasoning is based on opinion and results in arguments that cannot be resolved without implementing the change.

If we now augment our agile scrum workflow with CbyC the process changes as shown in Figure 3.2.

Roles

Team Architect: This person helps the Product Owner to create the formal specification. This role can be shared amongst the team members and does not have to belong to a specific person.

Activities and Artifacts

Product Vision: Here the Team Architect incorporates the Product Owner's requirements into the Formal Specification. The Formal Specification is then validated. If the requirements are valid stories are created on the Product Backlog.

Sprint: During the sprint we now incorporate the Formal Design, System Test Specification, INFORMED Design, and Code Implementation in our development process as described in paragraph 2.2

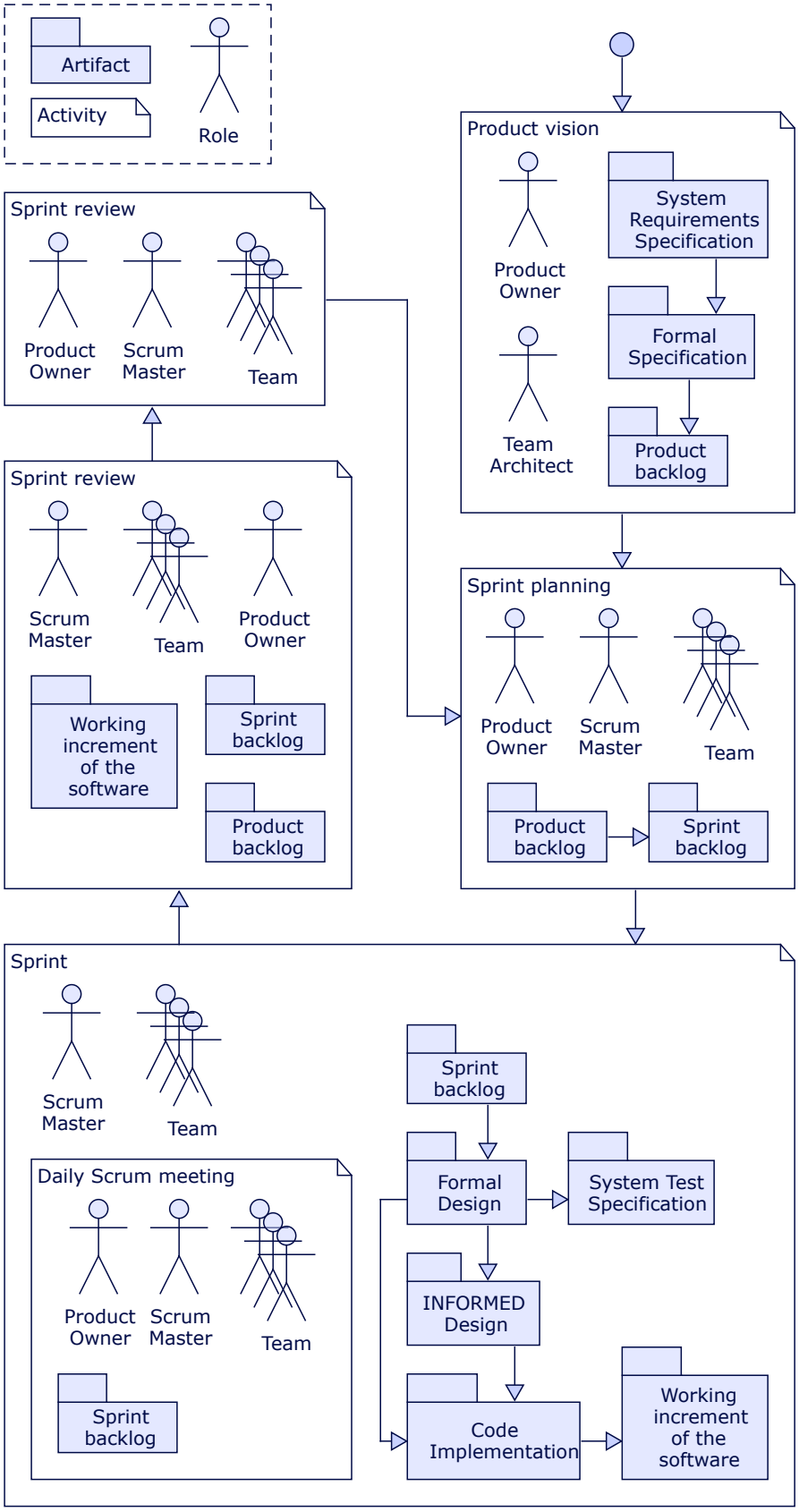

FIGURE 3.2: The scrum workflow augmented with CbyC.

Chapter 4

Applying the methodology

4.1 First iteration

4.2 Second iteration

4.3 The rest of the life cycle

Testing is usually the main method of verification and validation. The normal testing method follows these steps:

1. test individual units;
2. integrate them and test the integration;
3. then test the system as a whole.

This approach is inefficient because unit testing is ineffective and expensive. Unit testing is ineffective because most errors are interface errors, not internal errors in units. Unit testing is expensive because you have to build test harnesses to test units in isolation [6].

A more efficient and effective approach is to incrementally built the system from the top down. Each build is a real (if small) system and the system can be completely exercised in a real environment. This reduces the integration risk [6].

4.3.1 C#

Design by contract

CbyC program design is based on information flow expressed as code contracts [4]. C# does not have built in code contracts any more, but it is very simple to implement code contracts using standard C# language features.

Property based testing

C# has very little static analysis tools able to mathematically prove correctness. As an alternative we will use property based testing to exercise the code and show correctness [2] [7]. We will be using the FsCheck framework for our property based tests [5].

Appendix A

TLA+

A.1 TLA$^+$ Toolbox

Use the TLA$^+$ Toolbox to evaluate your specifications. You can download the latest toolbox from the tlaplus GitHub repository [9].

There is more information about the toolbox on the TLA+ website [8]

A.2 More information

Appendix B

FsCheck

B.1 Installing

B.2 More information

Appendix C

Hoare Logic

C.1 Program execution

$$\{P\}Q\{R\}$$

If the assertion P is true before the program Q executes, then the assertion R will be true after Q has executed.

C.1.1 Axion of Assignment

$$\vdash P_0\{x := f\}P$$

x is a variable identifier;

f is an expression;

P_0 is obtained from P by substituting x with f;

C.1.2 Rule of Consequence

$$\text{If } \vdash \{P\}Q\{R\} \text{ and } \vdash R \supset S \text{ then } \vdash \{P\}Q\{S\}$$

$$\text{If } \vdash \{P\}Q\{R\} \text{ and } \vdash S \supset P \text{ then } \vdash \{S\}Q\{R\}$$

This rule allows the strengthening of the precondition and/or the weakening of the postcondition.

C.1.3 Rule of Composition

$$\text{If } \vdash \{P\}Q_1\{R_1\} \text{ and } \vdash \{R_1\}Q_2\{R\} \text{ then } \vdash \{P\}(Q_1; Q_2)\{R\}$$

C.1.4 Rule of Iteration

$$\text{If } \vdash \{P \wedge B\}S\{P\} \text{ then } \vdash \{P\} \text{ while } B \text{ do } S\{\neg B \wedge P\}$$

Bibliography

[1] Kent M. Beck et al. *Manifesto for Agile Software Development*. http://agilemanifesto.org. [Online; accessed 7-December-2019]. 2001.

[2] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP* 46 (Jan. 2000). DOI: 10.1145/1988042.1988046.

[3] David Cooper. *Tokeneer ID Station EAL5 Demonstrator: Summary Report*. https://www.adacore.com/tokeneer. [Online; accessed 12-December-2019]. Aug. 2008.

[4] Martin Croxford and Dr. Roderick Chapman. "Correctness by Construction: A Manifesto for High-Integrity Software". In: *The Journal of Defense Software Engineering* 18.12 (Dec. 2005), pp. 5–8.

[5] *FsCheck*. https://fscheck.github.io/FsCheck/index.html. [Online; accessed 3-December-2019].

[6] Anthony Hall and Roderick Chapman. "Correctness by Construction: Developing a Commercial Secure System". In: *IEEE Software* 19.1 (Jan. 2002), pp. 18–25.

[7] Richard Hamlet. "Random Testing". In: *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[8] Leslie Lamport. *The TLA$^+$ Toolbox*. https://lamport.azurewebsites.net/tla/toolbox.html. [Online; accessed 4-December-2019]. 2019.

[9] *TLA$^+$ GitHub release*. https://github.com/tlaplus/tlaplus/releases/latest. [Online; accessed 4-December-2019].

[10] Sune Wolff. "Scrum goes formal: Agile methods for safety-critical systems". In: *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)* (2012), pp. 23–29.