Correct by contruction

Author:
J. Maree FOURIE

December 17, 2019

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

Leslie Lamport

Abstract

Correct by contruction

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

Correct by Construction (CbyC)

1.1 Overview

A lot of time and money is lost dealing with the consequences of software bugs. The ”Correct by Construction” software development methodology proposes to limit software bug by:

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

To achieve this we apply the following strategies [4]:

- use formal notation for deliverables;
- use tool-supported methods to validate deliverables;
- carry out small steps and validate the deliverables for each step;
- sate things only once;
- design software that is easy to validate;
- do the hard things first.

Software development can be divided in to two parts: specification and design; and implementation. To make this applicable to my work environment I chose to use TLA+ as a specification language and C# as a programming language.

1.2 Specification and Design

Regular language is inherently imprecise. Writing a specification and design in regular language leaves the documents open to conflicting interpretations.

Using a mathematical, formal, language to define the specification and high-level design of a system provides a precise description of the system's behaviour and a precise model of its characteristics. Using a mathematical language also enables the use of automated tools to verify the specification and design [4].

Formal specifications and designs are more precise and this forces us to understand issues and questions before we start coding [6].

1.2.1 TLA$^+$

TLA$^+$ has a large community and is proven useful in industry [8]. TLA$^+$ was created by Leslie Lamport in the late 1980's. TLA (Temporal Logic of Actions) is a simple variant of Pnueli's temporal logic. Most TLA specifications consists of ordinary non-temporal mathematics. Temporal logic is only used when appropriate. TLA is written in an assertional reasoning style [9].

1.3 Construction

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].

CbyC suggests the generation of evidence of correctness by using programming languages and tools, that allow for verification and analysis [4].

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].

1.3.1 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].

Chapter 2

Agile and Correct by Construction

2.1 Correct by Construction development workflow

The main properties of the CbyC development process are [3]:

- each life-cycle phase can be validated;
- the semantic gap between life-cycle phases are reduced, making it possible to show the conformance of later life-cycle phases with earlier phases.

The CbyC development process products are shown in Figure 2.1. We now describe each product and how it is created.

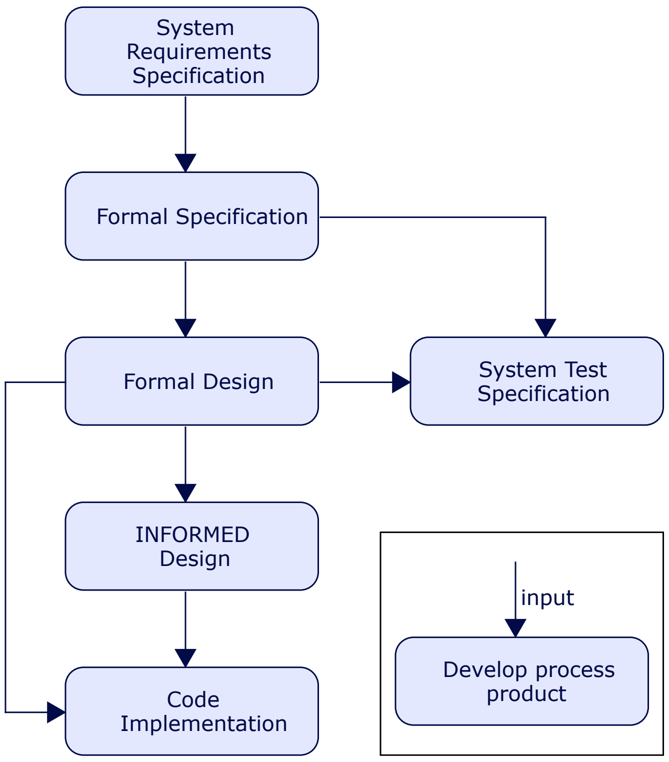

FIGURE 2.1: CbyC development process products.

2.1.1 System Requirements Specification

The "System Requirements Specification" development process product is the result of the requirements analysis process.

The aim of requirements analysis is to identify the needs of the stakeholders, the desired behaviour of the system, and any non-behavioural characteristics that are needed [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.

Every requirement should be traceable through each level of system representation (requirements, specification, design, code, test) [3].

The reasons for producing the System Requirements Specification are [3]:

- To early in the project clarify the system's boundary (what is in scope and what is out of scope, and the interfaces necessary to external systems).
- To agree on the system requirements with all of the stakeholders.
- To document the requirements in a sufficiently precise manner to allow subsequent development of the formal specification to proceed smoothly with little customer input.
- To clarify and document the assumptions about the behaviour of external systems, a common source of error.
- To identify and manage conflicting expectations between stakeholders.

2.1.2 Formal Specification

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

The abstraction level is important. The formal specification should not address how the system is implemented, internal details are deliberately left very abstract. Interactions with the external environment are specified, but may be left 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 and then combined to describe the system as a whole [3].

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

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

2.1.3 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.

2.1.4 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.

2.1.5 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.

2.1.6 Code Implementation

2.2 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.

Chapter 3

Applying the methodology

3.1 First iteration

3.2 Second iteration

3.3 The rest of the life cycle

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 [11].

There is more information about the toolbox on the TLA+ website [10]

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. *Industrial Use of TLA+*. `https://lamport.azurewebsites.net/tla/industrial-use.html`. [Online; accessed 3-December-2019]. 2018.

[9] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 032114306X.

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

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