CS 3EA3: Optional Assignment - Binding Theory to Practise: The Benefits of Correct-By-Construction Programming

James Zhu `001317457`

26 April 2017

1 Abstract

Correct-by-construction programming is a programming philosophy wherein the proof of the proposed solution (often in the form of mathematical specifications) is developed alongside (and ideally, prior to) to the implementation. Thus, the specification and the program forms a theorem which expresses that the program satisfies the specification and the problem statement. The use cases in industry for the design and development of large systems is presented and compared with other models of development (eg. Test-Driven Development). Finally, a case study of the use of correct-by-construction programming in the deviation of an algorithm to verify well-definedness of Guards and Bodies in the pState probablistic model checker is presented.

2 Introduction

The definition of the term "correct-by-construction" has permeated and formed the basis of the discussion and material presented in the CS 3EA3: Software Specification and Correctness course. Indeed, this concept of proof-by-construction declares more of a *philosophy* under which any product (software or otherwise) ought to be developed. Moreover, it will be demonstrated that the use of such a development model is ever more essential when large, multi-layered and multi-modal systems are under consideration, especially when the latter are utilised in production within *safety critical environments*. This latter point will be additionally illustrated through the examination of a case study in relation to the implementation of the well-definedness of Guards and Bodies for the *pState* verification toolkit.

3 Background and Mathematical Preliminaries

3.1 What is *Correct-by-construction?*

As discussed above, the idea of *Correct-by-construction* describes a development philosophy wherein, given a specific problem, the solution *program and its proof [are] developed hand in hand, [and] with the proof usually leading the way.* [1]. In practise, it may be said that this process consists of the following steps, in order to derive a *solution* from any given *problem statement*: [1]

1. (As succinctly as possible) **describe the requirements of the problem using *natural language.***
2. **Formulate these requirements into a *mathematical specification.*** This chiefly includes the following steps [2]:
   (a) Formalisation of *Givens* and *Requireds* presented by the Problem.
   (b) Derivation of a corresponding *invariant* P, and initialise the variables in order to make it *true*.
   (c) Bridge from the *invariant* to the *post-condition*. This involves solving for B within the statement $P \wedge \neg\, B \Rightarrow R$.
   (d) If $\neg\, B$ holds, then nothing more needs to be done; otherwise, a *loop* should be constructed to obtain it.
   (e) Solve for a *bound function* bf such that $P \wedge B \Rightarrow bf > 0$.
   (f) ***Very importantly***, any solution must *make progress towards termination* — or else an output may never be returned! — *See these notes by Dr. Jeffery Zucker at McMaster University on 'The Halting Problem" for further discussion on divergence.* [3].
   (g) *It must also be ensured* that the solution upholds the invariant.
3. **Convert the specifications into an *implementation (ie. program code*.**

3.2 Correct-By-Construction vs Other Methods of Development

Although not of the all the above steps in **Step 2** may be applicable to the given problem, dependent on the latter's context and domain, the application of this paradigm nonetheless brings many advantages when considering the *formal verification* of the correctness of any solution. ***Namely, the provision of formalised specifications and proof for any implementation enables for the assertion that this solution satisfies the problem in all aspects within the relevant domain***.

3.2.1 Comparison with Test-Driven Development (TDD) Within Varying Domains

In comparison with other paradigms of development, such as ***Test-Driven Development (TDD)***, which encourages rapid prototyping, and re-factoring of small "units" within the system until it satisfies some pre-defined unit test(s), Correct-By-Construction fosters careful consideration and analysis of the problem *holistically*, in both a *functional* and *mathematical* manner, prior to the start of the implementation process.

3.2.2 The Financial Case for Correct-By-Construction

Whilst certainly a *TDD*-based model may be highly beneficial (and even desirable!) for certain (smaller) domains of application which *prioritise high-turnover of features and versions over correctness*, which do not justify the financial and resource expenditure required in developing a formal specification, for larger, more complex systems, the adoption of a *proof-first* development model may even *reduce* expenditure in the longer-term.

This is chiefly due to the fact that for more complex systems containing multiple different systems operating and interacting under a common interface, a unit-based development approach will *not* enable the developer to analyse and appreciate fully how the interactions between the system affects the *behaviour* and *run-time* of the system as a whole. The latter will introduce potentially exponential risks for the developer should a component require re-factoring, as should this component be depended on by (and dependent on) many other sub-systems, a lack of formal specifications will make tracing a very time-consuming and labour-intensive process. Indeed, even if the issue is identified, within a *highly-coupled* system, the rectification process may involve the revision (and even rebuilding) of many inter-dependent modules — the latter which may have either *contributed to* or *inherited* the fault.

Thus, it is clear that for large commercial systems containing many hundreds and even *thousands* of interwoven and interdependent modules, and for which *stability* and *robustness* of implementation is a highly-important consideration in deployment, that such *rapid-revision* development paradigms *do not* provide the mathematical certainty to which these products require. Take for instance, the example of a Mars-bound rover. Should any major, unexpected system faults occur, the developers are highly limited in the scope of modifications and fixes which may be issued. Thus, the financial risk of any unaccounted for behaviours may potentially be in the millions (if not billions!) of dollars.

In contrast, a considered development approach based strictly on adherence to *proven specifications* will largely negate many of the risks outlined above, as the development of these descriptions (and justifications) of behaviour will ensure that the system(s) will *always make progress towards* their expected output,

while in addition, operating within their expected domains. While these systems built on specification will may also be susceptible to faults, (for example, due a discrepancy between the *expected* and *actual* operational domains), the latter are *generally* much less time-consuming and much more straight-forward to fix: one only has to refer to the proof, and either expand the scope of the system, or remedy some disproven assumption.

However, as with all development models, *correct-by-construction* is not without its shortcomings. Namely, one must be able to ensure that the implementation of the solution proposed has been constructed *with the utmost loyalty to the specification*, as only then will the benefits of a *proof-based* development model be fully realised. Indeed, it is often the case that if the specification *has not* been followed fully and unequvocially in implementation, that it may introduce significant challenges during debugging, in the case of an error.

To illustrate the use of the *correct-by-construction* philosophy, a case study will be presented and analysed relating to the implementation of *verification for the well-definedness of Guards and Bodies* within the *pState* model-verification system.

4 Binding Theory to Practise: A Case Study in *pState*

To demonstrate the process of correct-by-construction programming as applied to a large system, a case study will now be explored with regards to the specification and implementation of a verification system for the well-definedness of guards and Bodies within *pState*, as described by Lee and Zhu in [5].

4.1 A Brief Overview of *pState*

pState is a software toolkit which facilitates for the design, validation and formal verification of complex systems. A system is chiefly represented through *a hierarchy of statecharts*, which, together with *probabilistic transitions* adjoining them, represent the behaviour of the modelled system. [5].

- A ***state*** is defined in this context as *the values of all the variables, including input and output, of a system at a specific moment in time.* [3].
- A ***transition*** is additionally defined as *an n-tuple, consisting of an* ***Event name E****, an* ***optional guard g****, (the latter which is a boolean expression), as well as a non-empty set of probabilistic alternatives, [each of which leads] to a set of optional statements referred to as* ***bodies****.* [5]. Each transition must also *originate from a Source State*, as well as *lead to a Target State*, which are the states of the system prior to and right after taking the aforementioned Transition, respectively.

4.1.1 Invariants and Accumulated Invariants

In addition, each *State* within the System has an associated ***state invariant*** defined, which is described by Lee and Zhu [5] as ***a condition which must hold for every incoming transition into a particular state*, and such that every outgoing transition may assume the validity of this condition**. This invariant is represented within *pState* internally as an `expression`, consisting of the grammar `<term> | <operator>`.

Within the "correct-by-construction" paradigm, this is analogous to the concept of the (identically-named) *invariant*, which is defined in its domain as *a property of the data being manipulated that holds irrespective of the number of repetitions that have been executed.* [6]. An invariant in this instance is also represented by an `<Expression>`.

Thus, in both instances, an *invariant* is used to formally describe properties and conditions which must be met, in order for a program (or system) to be considered correct in relation to its defined requirements. Similarly, this condition is also expressed in the form of an *Expression.* This is an unsurprising notion, however, as the *pState* implementation under study facilitates in the modelling of systems using *correct-by-construction programming.*

However, as *pState* is a hierarchy of states (as stated above), any state within the hierarchy must both *inherit the conditions of its ancestor states*, as well as *itself be a combination of the conditions of its child states.* [5]. Therefore, Lee and Zhu introduces the concept of an ***accumulated invariant***, which is a *conjunction of:*

- Invariants from all the *ancestor states* of the current State.
- Invariants from all states which *form the closure* of the current State.
- Invariants from all the *child states* of the current State.

As part of the state hierarchy, there additionally exists a *root* state at the top, whose invariant will *always* be *true.*

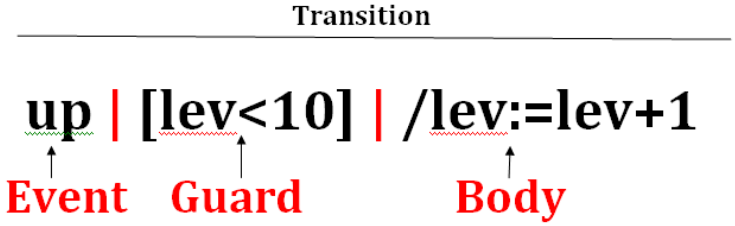
4.2 Well-Definedness of Guards and Bodies

As Lee and Zhu in [5] set out to specifically solve the problem of the *verification of Guards and Bodies within*pState, within the conditions of correctness (or *invariant*) as defined with any system modelled in *pState*, it must be established what these *invariants are*, in order to be able to derive and assert the correctness of any implementation in code.

Thus, Lee and Zhu define *well-definedness* of an expression or statement to be *true "if and only if it does not violate any of the operational conditions and constraints which may be imposed on it by the components of the system from which it is derived (or a child of")* [5]. As this clearly reflects the definition of the *invariant* as established in the previous section, the solution for the verification of a Guard or Body must therefore ensure that a Guard and/or Body does not violate the invariants established on any components of the system of which it is a part of.

In order to determine this above condition, recall that a ***Guard*** is an optional boolean expression which forms one component of an n-tuple *Transition*. This is illustrated below in the components of a Transition, as depicted in **Figure 1**.

Figure 1: The components of a Transition in pState. Image Credit: [5]



More specifically, the Guard is evaluated in *pState* on the occurrence of the Event on the Transition to which it belongs, in order to determine if the Transition may be taken.

Similarly, a ***Body*** is also a component in the n-tuple of a Transition, which are *optional statements* which are executed as its associated Transition is taken from the latter's Source and Target States (Refer to **Figure 1 above)**.

Notice that as both the *Guards* and the *Bodies* within a system are defined as a component of a Transition, then from above, it must follow that any Guard or Body must satisfy (at least some) portion of the invariant established on its hosting Transition (It will be discussed as to why it may not be required to satisfy the *entire* invariant of the Transition shortly). Thus, an examination of the notion and implementation of the well-definedness of *Transitions* is in order.

4.2.1 Well-Definedness of Transitions

Lee and Zhu in [5] defines the invariant of a Transition to be definable by the **Hoare Triple {P}** T **{Q}**, where:

1. ***P*** is the **pre-condition** — the *conjunction* of (i) the accumulated invariant in the **Source States(s)** of the Transition, along with (ii) the guard, the latter which must hold initially prior to the traversing of any Transition.

2. T is the Transition *trigger*, in this case the **Event**. Should Event occur and the Transition be taken (assuming that the Guard holds), then any actions which occur in the *Bodies* of the Transition are also performed in T.

3. ***Q*** is the **post-condition**: This is the accumulated invariant of the Target State(s). If were not the case, then the Transition would never be able to be taken without invalidating the correctness of the system as a whole.

Thus, it can be said that a *Transition* is **well-defined** if it satisfies this above Hoare Triple.

4.2.2 An Extension to the Well-Definedness of Guards

By the definition for the well-definedness of Transitions given in the previous section, it can be reasoned that in the context of the well-definedness for the associated ***Guard***, the *pre-condition* ***P*** established in this former definition should hold. This is as the system must be both: (i) in the *Source State* of the Transition to which the guard belongs (and thus, must satisfy all of the conditions placed on it); as well as (ii) satisfy the guard condition (or else the Transition will never be taken and the guard will be functionally useless).

However, it is interesting to note that the *post-condition* does *not* have to hold when considering the well-definedness of Guards. This is as the accumulated invariant of the **Target State** do not have to be satisfied, as it could be the case that the conditions for "truthifying" the Guard may not exist any longer after the traversal of the Transition.

Thus, as Lee and Zhu stipulate, the invariant (or condition) which must hold to ensure the well-definedness of *Guards* is: ***accumulated invariant of the Source State of its associated Transition*** $\wedge$ ***Guard***.

4.2.3 Another Extension to the Well-Definedness of Bodies

With consideration to the conditions for establishing the well-definedness of Guards above and the definition of the ***Bodies*** of a Transition, the latter which are carried out as its respective Transition is being taken to advance the System from its Source State to its Target State, it can be inferred that *all the conditions established for the well-definedness of the Guard must hold*. This is due to the fact that if the Guard of its' Host Transition is *not satisfiable*, then it cannot be the case that this latter Transition will ever be traversed. If this is the case, then it follows that the *Body* will never be executed as well.

Contrastingly, one major difference may be observed when in the correctness of Bodies which are not present in Guards: ***Namely, the Accumulated Invariant of the Target State of its associated Transition must also hold!***

To establish why this is the case, it can be noticed that as the Bodies are executed as the system transitions between the Source State of its affiliated Transition to Target State. Should the Target State's invariant *no longer hold*, then the Transition itself cannot *make progress towards* this Target State, without invalidating the invariant – and thus *correctness* of the entire system. It can be observed that this effect is amplified by the hierarchical nature of a *pState* system, in that modifications impacting the status of an invariant — (or the *negation of*) — will always affect *("*distribute to") the status of the invariants which are declared on either it's *ancestor* and or *child* states, as well. (And the propagation of this "invalidation" of invariants continues to spread through the state hierarchy of the entire system.)

Once again, as Lee and Zhu states, the invariant on a ***Body*** must then consist of: ***accumulated invariant of the Source State of its associated Transition*** $\wedge$ ***Bod(ies)*** $\wedge$ ***accumulated invariant of the Target State of its associated Transition***.

4.3 Ready for Implementation

Having defined:

- The **pre-condition** to be the existence of a ***Guard*** or ***Body*** on a Transition leaving a State;
- The **invariants** which must hold for ***Guards*** and ***Bodies*** respectively, in order for the former to be classified as being *well-defined*;
- The *conjunction* of many (accumulated invariants) to **make progress** towards obtaining a verification result; and
- The **post-condition** to be the result indicating the *well-definedness* of a Guard and/or Body

the specification now contains the required information to be implemented in code, which will be discussed in the following section.

4.4 Implementation Details

To benefit the understanding of the mapping of the various components of the ***formal specification*** to the `Program`, the Program will be discussed in relation to the various *components* indicated above. Zhu and Lee [5] indicated in their implementation that due to a technical limitation with a solver which is used to

verify the satisifability of first-order the predicate logic of which invariants are composited as, only the *verification of the well-definedness of Guards* is currently effective in implementation. The program for Bodies, however, is expected to be similar, in accordance with the variations as indicated in the mathematical specification in the previous sections.

4.4.1 Well-Definedness for Guards

1. **Pre-Condition:** For the selected Transition, check to see if there is an associated Guard. If not, the Guard is assumed to be `true`, as per *pState* convention.

2. Construction of the **Invariant**:

   (a) **Firstly, the Accumulated Invariant of the Source State is derived, and progress to made towards the post-condition:** The Source State of the selected Transition is retrieved. The Invariants of all of it's ancestor States, which are composited to form the Accumulated Invariant, is recursively retrieved and placed in a list. Then, iterate through the list to format it for use in the YICES Satisifiability Modulo Theories (SMT) solver (*outside of the scope of this report– see [5] for further details). This invariant is then parsed into suitable YICES syntax and sent off to YICES (through its API) to be verified for satisifiability. The *conjunction* of the satisifability of each Invariant within the Accumulated Invariant indicates whether the entire Accumulated Invariant is satisifiable.

   (b) **Then, the Guard is retrieved and conjuncted with the result from the above step to check for satisifiability**.

3. **We have reached the post-condition** As the *well-definedness of Guards* has been successfully verified and returned in accordance with the mathematical specification indicated in the previous section, it may be said that the ***program along with its specification is a theorem which expresses that the program satisfies its specification***. [1]

4.5 Conclusion

As the exercise in algorithm derivation examined in this paper with regards to the well-definedness of Guards and Bodies within the *pState* software has strongly asserted, the design and implementation of solutions which solve problems within larger and more complex systems is a highly *concise task*.

Whilst it has been established that there are cases in which such a stringent development process as dictated by the "correct-by-construction" philosophy is not warranted (and may even prove to be a hindrance to effective solutions!)—in the case of more complex, and interconnected systems, consisting with many

modules, the effectiveness of this approach is seldom under question. Indeed, for such systems in which operation will be conducted in safety-critical environment, and wherein robustness is prized above all else, the use of a *proof-first* development model, while adding to the initial production time to a usable prototype, brings long-term benefits far outweighing the burden brought on by the increased initial investment.

Inasmuch as "correct-by-construction" allows for the quicker detection, tracing and effective remedy of errors in complex systems, when conducted to a concise enough degree, it achieves something even more remarkable. Namely, it allows scientists to send a billion-dollar rover a billion kilometers into the depths of unexplored space, turn it on, and with confidence, know precisely when and where it will "call home".

References

[1] M. Al-hassy, Course Quiz 1 Solutions, Topic: "The Main Purpose of This Class" CS 3EA3, Department of Computing and Software, McMaster University, Hamilton, Canada, 15 January 2017.

[2] M. Al-hassy, Course Theorem Sheet, Topic: "Program Construction Theorem List" CS 3EA3, Department of Computing and Software, McMaster University, Hamilton, Canada, 10 April 2017.

[3] J. Zucker, Class Lecture, Topic: "The Halting Problem; The Universal Function Theorem" CS 3TC3, Department of Computing and Software, McMaster University, Hamilton, Canada, March 2017.

[4] D. Janzen, Class Lecture, Topic: "Software Construction: What is Test-Driven Development" CSC 405, Computer Science, California Polytechnic State University, San Luis Obispo, United States, Winter Term 2010.

[5] J. Lee, J. Zhu, *Verifying Well-Definedness of Guards and Bodies in pState*, Unpublished technical report for the Department of Computing and Software, McMaster University, Hamilton, Canada, 13 April 2017.

[6] R. Backhouse, *Program Construction: Calculating Implementations from Specifications*. Chichester, United Kingdom: John Wiley & Sons, Ltd, 2003.