Proceedings Paper DOI: 10.58190/icat.2023.35

PROCEEDINGS OF

INTERNATIONAL CONFERENCE ON ADVANCED TECHNOLOGIES

https://proceedings.icatsconf.org/

11th International Conference on Advanced Technologies (ICAT'23), Istanbul-Turkiye, August 17-19, 2023.

Formal Methods for an Agile Scrum Software Development Methodology

Fisokuhle Hopewell Nyembe [1], John Andrew van der Poll [2], Hugo Hendrik Lotriet [3]

[1] *School of Computing, College of Science, Engineering and Technology (CSET), Science Campus, University of South Africa (Unisa), South Africa*

37233858@mylife.unisa.ac.za, ORCID: 0000-0001-8205-5088

[2] *Digital Transformation and Innovation, Graduate School of Business Leadership (SBL), Midrand Campus, University of South Africa (Unisa), South Africa*

vdpolja@unisa.ac.za, ORCID: 0000-0001-6557-7749

[3] *School of Computing, College of Science, Engineering and Technology (CSET), Science Campus, University of South Africa (Unisa), South Africa*

lotrihh@unisa.ac.za, ORCID: 0000-0002-0353-5073

***Abstract*— Efficient, high-quality software systems embodying dependable methods are in high demand, which has led to a wide range of competitive market solutions. One effective technique that arguably has excelled above others is the Agile Software Development Methodology (ASDM). Agile approaches' capacity to produce software in a way that is flexible to changes is the main factor that makes them preferable. Scrum, a recommended Agile methodology, prioritises feature coverage and project structure. Because iterative methodologies encourage engagement from cross-functional teams, including consumers, Agile provides flexibility in responding to change. However, achieving methodological efficiency is insufficient while developing software; high-quality software should be achieved with equal consideration. Formal Methods (FMs), which are mathematically based techniques, can offer highly dependable software but suffer from a steep learning curve in mastering the underlying discrete mathematics and logic. This research investigates the extent to which FMs may be embedded in traditional Agile as embodied by Scrum. Future work in this area would be the development of a framework for embedding FMs in Scrum, followed by a survey among software practitioners to establish the feasibility of our technique.**

***Keywords*— Agile Software Development Methodology, Formal Methods (FMs), Formal Specification, Proof Obligation (PO), Scrum, Z.**

I. INTRODUCTION

Numerous software development methodologies aim to address the challenge of producing quality software on time, within budget, and preserving a company's market dominance [1], [2]. Conventional approaches often assumed that scope could be defined up front, a plan could be put in place, and the plan could be executed with little or no change. Over time many embraced the Agile Software Development methodology [3], characterised by being throughput-oriented and focusing on delivering value to customers as rapidly as possible [4]. It can also be used as a task management framework to apply familiar implementation approaches to the task's completion in line with re-usability.

Scrum, deemed to be a development, delivery, and maintenance strategy for complex products, is a well-known Agile software development methodology for approaching difficult adaptive problems and delivering high-value products in an innovative way. Scrum is described as easily understood, lightweight, yet somewhat hard to master [2]. Short project cycles, known as Sprints, are used to plan, design, build, test, review, and deploy a usable deliverable [2]. The Scrum framework comprises Scrum Teams and their associated duties, tasks, activities, objects, and guidelines. Each component of the framework serves a particular purpose and is crucial to the adoption and success of Scrum.

Scrum processes embody a small group of people who are very adaptable and flexible. These teams iterate and incrementally deliver products, facilitating possibilities for feedback. A scrum team is made up of a Product Owner, a Development Team, and a Scrum Master. Scrum Teams are also distinguished by their capacity to self-organise and collaborate across departments [2].

Despite the advantages of short development cycles and regular feedback from stakeholders, rapid software development using Agile may lead to challenges regarding lack of planning, scope creep, and overbudgeting, especially with respect to mission-critical software development where human lives may be at stake [5]. Developers may, therefore, consider using Formal Methods (FMs) as part of the development.

The use of FMs for software development involves using mathematical techniques to construct highly dependable software to meet end-user requirements. Advocates of FMs point to the benefits to be gained in producing high-quality software that may be provably correct, while critics point to the steep learning curves in mastering the underlying mathematics and logic.

With Agile at one end of the spectrum and FMs at the other end, it may be worth the effort to investigate the extent to which FMs may be embedded in Agile embodied by (e.g.) the Scrum methodology. This then leads to the objective of our paper:

Objective: Investigate the extent to which FMs may be embedded in the Agile Scrum methodology.

II. Literature Review

Our literature review centres primarily on Agile, defined by Scrum and FMs for software development.

A. *Agile Software Development*

Agile, as defined by the Agile Manifesto [6], rapidly responds to change and is attributed to having the ability to balance flexibility and structure [7]. Owing to its widespread use, our research will focus on the popular Agile variant called Scrum, characterised by piece-meal project cycles, known as Sprints, used for delivering planned, designed, built, and tested reviewed software systems [2]. Contrary to, for example, the traditional Waterfall Software Development Life Cycle (SDLC) [4], where full system requirements are available upfront and sufficient time is allocated to planning processes prior to the development, the Agile Software Development (ASD) practice known as IKIWISI (I'll know it When I See It) implies that full system requirements may not always be available upfront. Rather requirements are discovered as the system is developed. These happen once or twice a day through short sprints [2], during which the user may have regular insight into the development process. This suggests that users can better describe their full requirements after the initial idea has been translated into a functioning prototype [8].

Development interspersed by scrum sessions occurs iteratively, embodying the development of the concept, design, build, and test. In a way, therefore, an Agile iteration is a Waterfall development in the small except for the maintenance phase.

1) *User Stories*:

User stories describe scenarios to interpret how the system should function [9]. They assist in advancing the end-user perspective on the system and are the beginning and end points of the requirements coverage. System features are interpreted as user stories, recorded on storyboards, and tracked daily. Three main Agile practices are used to record system features, and these are requirements involving user story reviews, unit testing, and evolutionary prototyping., using, amongst others, JAD (refer below).

User stories are specified in a non-technical notation, e.g., natural language, and are continuously enhanced and refined throughout the development as more becomes known about the system. This is in line with Agile's principle of minimal documentation, implying that no formal requirements are to be produced [10]. The system features are piecemeal and aimed at leading developers into a fully functioning product required by the client. Therefore, one of the main differences between traditional methods and Agile is that the latter has less appetite for thorough requirements analysis [10]. Consequently, ASD has become less structured in recent years and has mostly been characterized by the three (3) faces of simplicity, namely, minimalism, quality design, and generative rules [9].

Agile defines three main requirements analysis techniques. These are JAD (Joint Application Development), prioritising users over tools and processes; modelling, adhering to the 11th principle of agility; and prioritisation, involving storyboards which organise the project according to priorities. We note that the emphasis on end users aligns with the Fifth Industrial Revolution (5IR), in which the emphasis is moved back to the human, e.g., a harmonious collaboration between humans and machines [11].

2) *Agile Challenges*

Despite its lucrative features, Agile incurs a number of disadvantages. Reference [10] notes challenges in the management of requirements in ASD – these difficulties are a result of the pressure that comes with expectations of fast deployments. Scrum teams typically do not know what the result (or just a few cycles down the line) will look like from any given point of development. It is, therefore, hard to estimate what it will cost, how long it might take, and which resources will be needed (especially when the project grows larger and more complex) [12]. It may be easy for scrum teams to get side-tracked by delivering unexpected features since it requires minimal planning at the beginning Since scrum teams often work on each component in separate cycles, the finished product usually seems fragmented instead of unified [2].

The documentation in Agile projects happens continuously and often just in time (JiT) for the output rather than from the beginning. In this manner, it becomes less detailed and is often put on the back burner [2].

Project management maturity appears to be an added challenge for Agile. Reference [13] argues that it is hard for Agile to achieve a maturity level beyond level 2 and that a Project Management Information System (PMIS) should be embedded in, for example, the PMBOK.

B. *Formal Methods (FMs)*

The use of FMs involves using (discrete) mathematical notation to detail the precision of the properties or behaviour of a software system. Formal Methods at the starting point usually focus on formal specifications, which is a way to describe system requirements formally. A popular formal specification language is Z [14], based on a strongly typed fragment of Zermelo-Fraenkel set theory [15] and first-order logic. A formal specification describes what the system must do and not how it should be achieved. A formal specification can reliably be used to verify the information system functions as determined by the customer. The systems' properties ought not to unduly constrain the specification of how the information systems' correctness is achieved [14].

An example of the state space of a simple banking system specified in Z specification of the case used in this paper is given below:

The basic types are [ACCOUNT, BALANCE].

ATM_Banking

$accounts : \mathbb{P}\ ACCOUNT$

$atm : ACCOUNT \nrightarrow BALANCE$

$accounts = \mathrm{dom}\ atm$

The state comprises two components, *accounts* and *atm*, with types as indicated, as well as an invariant that constrains accounts to be those held by the bank.

A state definition is usually followed by specifying an initial state from which the system may start off. The specification of an initial state of the system gives rise to a proof obligation (PO) that such an initial state may be realised. Operations may be defined on the state, resulting in further POs.

Discharging POs is seen as one of the strengths of FMs as embodied by a formal specification. By discharging POs, a specifier can show that the resultant system will behave as expected and that undesirable properties are absent. Examples of these are given in Section III. A formal specification also assists the development team to discover aspects of the system that may otherwise be hidden, thereby allowing developers to identify challenges early on.

1) FMs Challenges

As indicated above, one of the main challenges of FMs is the steep learning curve involved in getting to grips with the underlying discrete mathematics and formal logic. Further examples of the Z specification language appear in Section III, and it should be evident that a fair amount of mathematical maturity is needed in using Z.

Owing to their inherent complexities, formal specifications may contain errors [16]. Amongst the complexities underlying Z is that the schema calculus whereby schemas are combined may create inconsistencies [17].

Some of the above challenges may be addressed through adequate tool support for FMs, but such tools may turn out to be as hard to use as the FM itself [18].

Despite the FMs challenges indicated, there are numerous success stories of using FMs, most notably the famous CICS (Customer Information Control System) specified in Z [19].

Given the respective advantages and disadvantages of each of Agile and formal methods, we propose a combination of the two techniques. Given the more technical nature of FMs compared to Agile, we suggest the embedding of FMs in Agile instead of the other way around.

C. Are Formal Methods Ready for Agile?

Reference [20] assesses the benefits of combining FMs and ASD. They also assess the readiness of ASD to support FMs techniques to have synergy in the processes. Reference [21] describe FMs as a response to complexity by describing a software system as a mathematical entity, allowing competent stakeholders to verify and refute aspects of a requirements specification, and they dispel the widely held view of FMs as a software development methodology on its own (cf. Bowen & Hinchey’s myths of FMs [22]), similar to the concerns expressed by Corrigan et al. [13] for the project management maturity level of Agile.

A misreading that [21] deals with is that FMs are only effective as a post-factor verification. They also advise against viewing Agile Software Development as a methodology that can be implemented in all software development environments. Each software development enterprise should adopt only the ASD characteristics that are suitable for their environment and their resources [23]. A fine blend of Agile and FMs may, therefore, be the way forward.

Next, we investigate how FMs may be embedded as part of the operations of Agile by analysing a hypothetical scrum case study. The idea of using a case is part of a research strategy, as indicated by the Saunders et al. research onion [24]. Case studies may be categorized into three categories: explanatory, exploratory, and descriptive [25]. Since we are investigating an Agile-FMs interplay, our case study is exploratory.

III. CASE STUDY

Consider a scrum-based Agile development of a banking application where customers can, amongst other operations, deposit money into their bank accounts and make withdrawals, together with receiving feedback from the system.

Having analysed the requirements as high-level use cases, the scrum product owner (SPO) enters these into the Scrum Product Backlog and consults with the system architects and some senior software engineers to estimate and prioritize the items. The high-level requirements are subsequently divided into smaller-grained user stories. The SPO then schedules the first Sprint Planning meeting with the Scrum team.

Tasks start at day 0, which represents the sprint planning day, until day 28, which is typically the end of a Sprint, never taking longer than one month [26].

Sprint 1 – Day 0 (S1.0)

The Scrum Master calls a planning meeting, with the development team. Such meeting is indicated by Sprint *1*, Day *0 – S1.0*. This notation is reminiscent of Scheurer’s feature notation [27]. Agile may not explicitly provide for an *Sm.n* – Sprint *m* of Day *n* notation, but if not, then the introduction of such notation may be a pseudo advantage of embedding FMs in Agile.

Next, the team defines a number of user stories. For the purposes of the example, we consider five user stories during the *S1.0* meeting (we further ignore details of inserting a bank card, providing a pin, etc.) indicated in Table I:

TABLE I
SPRINT BACKLOG USER STORIES

No	User-story
1	Select the account to deposit into or withdraw from.
2	Select the deposit or withdrawal option.
3	Enter the deposit amount.

No	User-story
4	Enter the withdrawal amount.
5	Confirm transaction success.

Next, the team commissions a product backlog board consisting of three columns – To do, Doing, and Done. Initially, the backlog board is empty, which may be specified in Z (assuming basic types [*To_Do, Doing, Done*]) as:

```
_Sprint_Backlog_________________________
| to_do: To_Do
| doing : Doing
| done : Done
|__________________________
| partition <to_do, doing, done>
|________________________________________
```

where,

partition <*to_do*, *doing*, *done*> ≙
 disjoint (*to_do*, *doing*) ∧
 disjoint (*to_do*, *done*) ∧
 disjoint (*doing*, *done*)

Since an item in the backlog board may appear in at most one column, the components are pairwise disjoint. Observing that the columns should be pairwise disjoint presents an advantage of embedding FMs in Agile. With (pure) Agile, the scrum members may not notice that column contents overlap, i.e., the state invariant in schema *Sprint_Backlog* may be violated. With a small number of items, this may not pose any problems, but as the size of the board grows, this may become harder to notice.

Following Z's Established Strategy (ES), the next step is to define an initial state of the board and subsequently show that such a state can be realised [28].

```
_InitSprint_Backlog_____________________
| Sprint_Backlog′
|__________________________
| to do = ∅ ∧ doing = ∅ ∧ done = ∅
|________________________________________
```

A proof obligation (PO) arises to show that the said initial state may be realised [29]. This is an important point even though the PO may be trivially discharged:

Proof:

⊢ *Sprint_Backlog′* • *InitSprint_Backlog*

Hence, we need to show:

$$\vdash \exists\ to\_do' : To\_Do;\ doing' : Doing;\ done' : Done \mid to\_do' = \emptyset \wedge doing' = \emptyset \wedge done' = \emptyset \qquad (1)$$

The proof of (1) follows trivially since the empty set values are specified in schema *InitSprint_Backlog*. The proof indicates there is indeed an initial state from which the system may start. Again, Agile may not necessarily pay attention to this important aspect. We indicated above that Scrum user stories continuously enhance the system as more becomes known about the system through daily meetings. Discharging FMs proof obligations, therefore, serves the same purpose.

Sprint 1 – Day 28 (S1.28)

The SPO assesses a prototype of the system to determine whether the created user stories fulfil the requirements and whether the features are comprehensively documented. Suppose the SPO's conclusions are:

The state space of Table 1 is given by (assuming basic types [*USER_STORIES*]):

```
_User_Stories___________________________
| stories: ℕ₁ ⇸ USER_STORIES
|________________________________________
```

Schema *User_Stories* indicates user-stories are numbered using positive integers (i.e., starting from 1). This may be an important observation since Sprint days are numbered from 0. Developers and specifiers ought to be aware of these considerations as part of boundary considerations (refer also to the considerations on the deposit and withdrawal of amounts elsewhere in this paper) and would be an advantage of using FMs in Agile.

Next, we specify the five user stories as per schema *AddUserStories*.

```
_AddUserStories_________________________
| Δ User_Stories
|__________________________
| stories′ =
| {1 ↦ "Select the account to deposit into or
|        withdraw from",
|   2 ↦ "Select the deposit or withdrawal option",
|   3 ↦ "Enter the deposit amount",
|   4 ↦ "Enter the withdrawal amount",
|   5 ↦ "Confirm transaction success" }
|________________________________________
```

The formal specification of Table 1 could have followed either of two routes: Initialise component *stories′* as an empty function (cf. schema *InitSprint_Backlog*, followed by a proof that such an initial state can be realised, followed by an operation like *AddUserStories*, or specify *AddUserStories* directly as above. The considerations around these two options are reminiscent of assigning a value to a variable in computer memory or first checking whether the variable already contains the value and, if so, do nothing. As before, the formal specification makes us aware of these options earlier than the standard completion of an Agile Sprint.

Next, suppose having evaluated the user stories, the team indicates they do not have the capacity to complete user story 4, and because of that, part of user story 2 (the option to withdraw an amount). Consequently, the SPO moves these to the 2nd sprint.

Table II illustrates how sprints and user stories can be tracked and managed as initially presented by the Product Owner and changed on the backlog board, having moved user stories 2 and 4 to the 2nd sprint.

TABLE II
TRACKING AND PRIORITISATION OF USER STORIES

User Stories	Sprint Ready	Priority	Status	Sprint
1	*Yes*	*M*	*To do*	*1*
2	*No*	*H*	*To do*	*2*
3	*Yes*	*H*	*To do*	*1*
4	*No*	*M*	*To do*	*2*
5	*No*	*L*	*To do*	*1*

Legend: *M = Medium; H = High; L = Low*.

Next, we formally specify Table 2, starting with an appropriate state space.

```
_Prioritisation_User_Stories ____________
| trackings: ℕ₁ ⇸
|     Sprint_Ready × Priority × Status × Sprint
|
|________________________________________
```

We note that the Cartesian product in the above schema fixes an ordering among the columns of the table, but in practice, such ordering is probably immaterial. That said, implementations of database systems usually impose an ordering among table columns at the implementation phase. So, the schema makes such an implementation decision explicit at the specification phase already, while the Agile processes may not draw attention to this aspect. This illustrates a further advantage of embedding FMs in Agile.

The next step is to populate the schema as per the information in Table 2:

```
_Populate_Prioritisation_User_Stories________
| Δ Prioritisation_User_Stories
|____________________________
| (∀i: ℕ₁ •
|   trackings(i).Status = "To do"
|   ∧
|   (if i ∈ {1, 3} then
|     trackings(i).Sprint_Ready = "Yes"
|    else trackings(i).Sprint_Ready = "No")
|   ∧
|   (if i ∈ {1, 4} then trackings (i).Priority = "M"
|    elseif  i ∈ {2, 3} then trackings (i).Priority = "H"
|    else trackings (i).Priority = "L")
|   ∧
|   (if i ∈ {2, 4} then trackings (i).Sprint = "2"
|    else trackings (i).Sprint = "1")
|________________________________________
```

The first version of Z [14] did not incorporate an *if … then … else …* construct, but it was added in the 2nd edition of Spivey's Z user manual [30] to facilitate the user experience (readability, usability) of Z. In the above, we further extended the syntax to include an *elseif* as indicated. This may be a pseudo advantage of using FMs in Agile and elsewhere in Computing.

Returning to the functionality of the banking system (state space given by *ATM_Banking* above), we investigate what might be gained by formalising some of the user stories, starting with a customer depositing money into their account (user stories 1, 2, 3, and 5), ignoring the complexities of user story 2 having been moved to the 2nd sprint.

User-story Objective

As a bank customer:
I want to deposit cash into my bank account at an ATM;
So that I do not have to wait for the bank's branch working hours.

Acceptance criteria

1. Customer needs to enter a valid account to deposit cash.
2. System needs to validate the existence of the account number.
3. System needs to give the customer an option to enter the amount to be deposited.

The following schema formalises the user story.

```
_Cash_Deposit__________________________
| Δ ATM_Banking
| account? : ACCOUNT
| deposit? : BALANCE
| receipt! : RECEIPT
|____________________________
| deposit? > 0 ⇒
|  (∃ balance′ : BALANCE •
|    balance′ = atm(account?) + deposit? ∧
|    atm′ = atm ⊕ {account? ↦ balance′} ∧
|    receipt! = deposit?)
|________________________________________
```

Assuming basic types [*ACCOUNT*, *BALANCE*, *RECEIPT*], the account to deposit into, and the amount deposited serve as input to the system. The system generates a receipt (user story 5) for the customer.

The existing balance is overridden (⊕) with the existing balance incremented (*deposit*? > 0) by the amount. As may be observed, the formal specification makes a number of underlying assumptions clear that the scrum team might have glossed over during this user story.

The overriding operator could be interpreted as whatever amount is in the user's account beforehand is simply replaced by the amount beforehand plus the deposit made. This case is like updating the value of a variable in memory discussed earlier. Again, this consideration may not receive due consideration by the scrum team using Agile only. Therefore, possible ambiguities that existed between the system requirements described merely in natural language are clarified by the schema.

A further clarification is that a zero (0) amount may not be deposited. The user story does not adequately specify this, running the risk that an exception might be generated (thrown) by the system.

The above observations should indeed emerge once the design or programming phases are entered, but other design decisions may have to be changed. The requirements elicitation phase is regarded as being the most crucial and most challenging. The consequences of getting this critical phase wrong are far-reaching and can persist throughout the life of the software system [31]. Formal specifications assist in eliciting errors earlier during the system life cycle, thereby improving system functionality.

Next, consider the user stories (1, 2, 4, and 5) involving a withdrawal from an ATM.

User-story objective

As a bank customer:

I want to withdraw cash from my bank account through an ATM;

So that I can have physical access to my funds after hours.

Acceptance criteria

1. Customer needs to have inserted a bank card and account on the ATM.
2. System checks to see if the requested amount exceeds the balance.
3. If so, the system displays the balance and asks the user to enter a new amount.
4. If the amount entered is less than the account balance, cash is dispensed, and the new balance is displayed.

Schema *Cash_Withdrawal* formalises the essence of the user story.

```
_Cash_Withdrawal_______________________
│ Δ ATM_Banking
│ account? : ACCOUNT
│ withdrawal? : BALANCE
│ receipt! : RECEIPT
├──────────────────────────
│ withdrawal? ≤ atm (account?) ⇒
│ (∃ balance′ : BALANCE •
│   balance′ = atm(account?) - withdrawal? ∧
│   atm′ = atm ⊕ {account? ↦ balance′} ∧
│   receipt! = balance′ )
└──────────────────────────────────────
```

Amongst others, the acceptance criteria state, "System checks to see if the requested amount exceeds the balance." as well as "If the amount entered is less than the account balance, cash is dispensed, and the new balance is displayed.". Between these two criteria, it is not clear whether the case of $withdrawal? = 0$ is allowed, similar to the case of depositing a zero amount.

The formal specification clarifies this by stating that a customer may withdraw all the money in an account. Naturally, bank policies may have to regulate these aspects. The formal specification, therefore, elicits aspects that the scrum team could have missed.

Next, we consider the following day of Sprint 1, labelled Day 1.

Sprint 1 – Day 1 (S1.1)

The team gathers the next day, and the backlog board is updated on the strength of the previous day's work. Suppose user stories 4 and 5 have been completed (*Done*), user story 3 is in process (*Doing*), and user stories 1 and 2 are still to start (*To Do*).

TABLE III
SPRINT_BACKLOG_USER_STORIES STATUSES

Sprint Column	**User Stories**	
To Do	1: Select the account to deposit into or withdraw from.	2: Select the deposit or withdrawal option.
Doing	3: Enter the deposit amount.	
Done	4: Enter the withdrawal amount.	5: Confirm transaction success.

Formally Table III may be specified by:

```
_Sprint_Backlog_User_Stories_______________
│ Δ Sprint_Backlog
├──────────────────────────
│ (∀i : [1 .. 5] •
│   if i ∈ {1, 2} then trackings(i).Status = "To Do"
│   elseif i ∈ {4, 5} then trackings(i).Status = "Done"
│   else trackings(i). Status = "Doing")
└──────────────────────────────────────
```

As before, a proof obligation arises from schema *Sprint_Backlog_User_Stories*, and it is to show that the backlog board remains a partition, i.e., none of the entries appears in more than one cell in Table III, thereby supporting an FMs advantage mentioned earlier.

Naturally, the 28-day sprints continue until the project is completed or until the 28 days come to an end. If, after day 28, the project is not completed, the process continues.

Suppose next the scrum team arrives at the last day of the first sprint – day 28.

Sprint 1 – Day 28 (S1.28)

The SPO assesses a prototype of the system to determine whether the created user stories fulfil the requirements and whether the features are comprehensively documented. Suppose the SPO's conclusions are:

- User stories 1, 2, 4, and 5 are completed to expectation.
- User-story 3 remains open owing to technological challenges (essentially a defect), as indicated in Table IV. It is, therefore, placed on hold for Sprint 2.

Table IV shows the open defects, resulting in user story 3 not being completed on time.

TABLE IV
OPEN DEFECTS LIST

Defect ID	**Description**	**Status**	**User-story**
#3	The user can enter the deposit amount manually, but ATM cannot verify the amount in real-time.	Open	3

TABLE V
ADVANTAGES OF EMBEDDING FORMAL METHODS IN SCRUM

Concept	Advantages
1. Notation for a specific day within a Sprint was developed.	The need to identify specific days from 0 to 28 (4 weeks) within specific sprints led to a pseudo advantage of FMs. A notation S*m.n* for sprint *m*, day *n* was developed. For example, S*1.1* denotes Sprint *1*, day *1*.
2. State space as captured by the Z schema for a Sprint Backlog	The backlog board is defined by three columns, namely, *To_Do*, *Doing*, and *Done*. Formalizing the board revealed that the three components of the Sprint Backlog are pairwise disjoint.
3. Proof of initial Sprint Backlog	The proof shows how an initial state of the system may be realised, an aspect that Scrum developers may not necessarily pay attention to.
4. Z Schema 4 State Space for User Stories	This schema shows that using FMs makes it explicit that user stories are numbered sequentially, starting from 1. This is an important consideration since days in a Sprint are numbered from 0.
5. Z schema state space user-story Prioritization	The schema presents a cartesian product that fixes an ordering among the columns of the table. Attributes of a record in relational databases are not necessarily ordered, but the columns in the prioritization of the user stories (Table 2) appear to be ordered. The Z specification makes this explicit through the Cartesian product as a type.
6. Extending notation of conditional predicates – *if*/*else*/*elseif* statements	We have extended the predicate notation of Z by adding conditional statements in the form of *if*/*else*/*elseif* statements, as these usually occur in procedural and executable software development languages.
7. Identification of boundary conditions – Z schema cash deposit and cash withdrawal	Boundary conditions not necessarily identified during a Scrum sprint may become explicit through formally specifying conditions. Two cases arose: [1] Users may not deposit a zero amount (*deposit?* > 0). [2] With respect to a cash withdrawal, the schema specifies that a user may empty an account, e.g., *withdrawal?* $\leq$ *atm*(*account?*). The amount requested may indeed equal the amount available. These conditions may be missed in the brevity of natural language's user stories and result in defects.

Typically, user story 3 could remain open since the ATM does not embed the technology to count and verify the physical amount of money deposited by the user in real-time (banks usually have two officials who together open an ATM the next morning and manually verify each deposit made). This may be registered as a defect and allocated a defect number, e.g., #3. Generally, however, defects and user stories would not resemble a one-to-one mapping.

Table IV may be formalised by the following two schemas (basic types indicated may be new, or inferred from earlier schemas):

Open_Defects
defect_id: ID
description: DESCRIPTION
status: STATUS
s: STORY_ID

Correspondingly, Table IV could be specified as:

Add_Open_Defects
Δ *Open_Defects*
defect_id = 3
description =
"ATM unable to verify deposit amount in real-time"
status = "Open"
s = 3

Schema *Add_Open_Defects* is just for one specific case (defect) and could be enhanced to cater for more defects than just a single case. The defect will remain open until at least the next sprint.

While *Add_Open_Defects* does not convey any information not already in Table IV, it nevertheless adheres to an important formal specification design principle, namely, "Maximise communication with the user of the specification." [28].

In the following section, we summarise the value proposition of embedding FMs in Agile.

IV. ADVANTAGES OF EMBEDDING FMs IN SCRUM

Having observed the embedding of FMs as part of the above Agile case, we summarise as indicated in Table V.

The foregoing information, together with the summary in Table V, meets our objective stated at the end of Section I.

V. Conclusions and Future Work

In this paper, we introduced Agile, as captured by Scrum, as a lucrative software development methodology. Agile hastens the software development process yet it may lead to challenges, especially with respect to mission-critical software development. The lack of upfront and agreed-upon requirements may incur ambiguities with respect to accurately capturing user requirements. The high level of interaction among Scrum members is, on the one hand, desirable, while on the other hand, it may result in many interruptions in the working day of such members.

Formal methods have been introduced as a way of producing reliable or at least highly dependable software using discrete mathematics and logic. FMs, however, incur challenges of their own, amongst others, a perceived steep learning curve in mastering the underlying mathematical aspects.

The above observations led to the work reported above, namely embedding FMs in Agile Scrum Sprints. We embarked on a case study approach and formalised Scrum artefacts and processes, and in doing so, identified some of the advantages of embedding FMs in Agile. These are captured in Table V.

With respect to future work, Table V could be used as a starting point for further theoretical and empirical studies on this topic, aimed at developing a framework for embedding FMs into Agile Scrum. That said, Table V could be viewed already as a framework by some [32]. Either way, the framework could be validated through an industry survey among Agile- and FMs practitioners. Different methodological survey instruments could be used to validate these findings by developing measurement scales.

Acknowledgement

This work is based on the research supported in part by the National Research Foundation of South Africa (Grant Number SRUG2204214078).

References

[1] S. Baltes and P. Ralph, Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, *27*(4), pp.1-31, 2022.

[2] C.A. Hatcher, A Conceptual Framework for Flight Test Management and Execution Utilizing Agile Development and Project Management Concepts. 812th Test Support Squadron, 812 TSS/ENTI Edwards United States, 2019.

[3] G, Kim, J. Humble, P. Debois, J. Willis, and N. Forsgren, *The DevOps handbook*: How to create world-class agility, reliability, & security in technology organizations. IT Revolution, 2021.

[4] L. Traini, Exploring Performance Assurance Practices and Challenges in Agile Software Development: An Ethnographic Study. *Empirical Software Engineering*, *27*(3), pp.1-25, 2022.

[5] B. Moyo, The contingent use of systems development methodologies in South Africa, Doctoral dissertation, North-West University (NWU), South Africa, 2021.

[6] K. Beck et al., Manifesto for Agile Software Development, 2016. Available online at: http://agilemanifesto.org/. Accessed 27 April 2023.

[7] J. Highsmith, Agile Software Development-Why it is Hot. *Extreme Programming Perspectives*, M. Marchesi, et al., Editors, pp.9–16, 2003.

[8] V. Szalvay, An Introduction to Agile Software Development. Danube Technologies Inc., 2004.

[9] J.E. Tomayko, Engineering of unstable requirements using agile methods. In *International Conference on Time-Constrained Requirements Engineering*, September 2017.

[10] X. Franch, C. Gómez, A. Jedlitschka, L. López, S. Martínez-Fernández, M. Oriol and J Partanen, Data-driven elicitation, assessment, and documentation of quality requirements in agile software development. In *International Conference on Advanced Information Systems Engineering*, pp. 587–602. Springer, Cham, June 2018.

[11] J.A. van der Poll, Problematizing the Adoption of Formal Methods in the 4IR–5IR Transition. *Applied System Innovation*, 2022; 5, 127, 2022. Available online at: https://doi.org/10.3390/asi5060127. Accessed on 10 April 2023.

[12] K. Bhavsar, V. Shah, and S. Gopalan, Scrum: An agile process reengineering in software engineering. *International Journal of Innovative Technology and Exploring Engineering*, 9(3), pp. 840–848, 2020.

[13] M.J. Corrigan, J.A. van der Poll, and E.S. Mtsweni, E.S., The Project Management Information System as Enabler for ICT4D Achievement at Capability Maturity Level 2 and Above, *Communications in Computer and Information Science* (*CCIS*), Springer, 933, pp. 295 – 312, 2019.

[14] J.M. Spivey, *The Z notation: A Reference Manual*, Prentice Hall, Englewood Cliffs, 1989.

[15] H.B. Enderton, *Elements of set theory*. Academic Press, 1977.

[16] D. Parnas, Really Rethinking 'Formal Methods'. *Computer*. 43, pp. 28–34, 2010, 10.1109/MC.2010.22.

[17] A. Bayode, J.A. van der Poll, and R.R. Ramphal, 4th Industrial Revolution: Challenges and Opportunities in the South African Context. *Conference on Science, Engineering and Waste Management* (*SETWM-19*), pp. 174 – 180, 18 – 19, November 2019.

[18] J.G. Ackermann and J.A. van der Poll, Reasoning Heuristics for the Theorem-Proving Platform Rodin/Event-B, *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA, 2020, pp. 1800-1806, DOI: 10.1109/CSCI51800.2020.00332.

[19] L. Freitas, J. Woodcock, and Y. Zhang, Verifying the CICS File Control API with Z/Eves: An experiment in the verified software repository, *Science of Computer Programming*, 74(4), pp. 197-218, 2009, ISSN 0167-6423. https://doi.org/10.1016/j.scico.2008.09.012.

[20] M. Gleirscher, S. Foster and J. Woodcock, New opportunities for integrated formal methods. *ACM Computing Surveys* (CSUR), 52(6), pp.1-36, 2019.

[21] P. G. Larsen, J.S. Fitzgerald, and S. Wolff, Are formal methods ready for agility? a reality check. In FM+AM 2010: *Second International Workshop on Formal Methods and Agile Methods*, Newcastle University.

[22] J.P. Bowen and M.G. Hinchey, Seven more Myths of Formal Methods, *IEEE Software*, pp. 34–41, 1995.

[23] G. O'Regan, *Mathematics in computing*. Springer International Publishing, 2020.

[24] M.N.K. Saunders, P. Lewis, and A. Thornhill, *Research Methods for Business Students*, 8th Edition, London, UK.; Harlow, Pearson, 2019.

[25] Y. Cui, I. Zada, S. Shahzad, S. Nazir, S.U. Khan, N. Hussain, and M. Asshad, Analysis of service-oriented architecture and scrum software development approach for IIoT. *Scientific Programming*, 2021.

[26] W. Zayat and O. Senvar, Framework study for agile software development via Scrum and Kanban. *International journal of innovation and technology management*. June 24, 2020, 17(04):2030002.

[27] T. Scheurer, *Foundations of computing: System Development with Set Theory and Logic*, Addison-Wesley, 1994, ISBN 0-201-54429-6.

[28] J.A. van der Poll and P. Kotzé, Enhancing the Established Strategy for Constructing a Z Specification. *South African Computer Journal* (*SACJ*), Number 35, pp. 118 - 131, December 2005.

[29] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*, Second edition, Prentice-Hall, 1996.

[30] J.M. Spivey, *The Z notation: A Reference Manual*, Second edition. Hemel Hempstead, Prentice Hall, 1992.

[31] S.K. Pandey and M. Batra, Formal Methods in Requirements Phase of SDLC. *International Journal of Computer Applications*, 70(13), pp.7-14, 2013.

[32] J.A. van der Poll, A Research Agenda for Embedding 4IR Technologies in the Leadership Management of Formal Methods, *The 2022 International Conference on Computational Science and Computational Intelligence* (*CSCI'22*), pp. 1845 – 1850, Las Vegas, USA, 14 – 16 December 2022.