https://doi.org/10.1007/s00165-020-00512-5
The Author(s) 2020
Formal Aspects of Computing

Formal Aspects of Computing



# Legislation-driven development of a Gift Aid system using Event-B

David M. Williams[1], Salaheddin Darwish[2], Steve Schneider[3], and David R. Michael[4]

[1] School of Computing, University of Portsmouth, Portsmouth, PO1 3HE, UK
[2] Information Security Group, Royal Holloway, University of London, Egham, TW20 0EX, UK
[3] Department of Computer Science, University of Surrey, Guildford, GU2 7XH, UK
[4] Streeva Ltd., Surrey Technology Centre, Guildford, GU2 7YG, UK

**Abstract.** This work presents our approach to formally model the Swiftaid system design, a digital platform that enables donors to automatically add Gift Aid to donations made via card payments. Following principles of Behaviour-Driven Development, we use Gherkin to capture requirements specified in legislation, specifically the UK Charity (Gift Aid Declarations) Regulations 2016. The Gherkin scenarios provide a basis for subsequent formal modelling and analysis using Event-B, Rodin and ProB. Interactive model simulations assist communication between domain experts, software architects and other stakeholders during requirements capture and system design, enabling the emergent system behaviour to be validated. Our approach was employed within the development of the real Swiftaid product, launched by Streeva in February 2019. Our analysis helped conclude that there was not a strong enough business case for one of the features, which was shown to provide nominal user convenience at the expense of increased complexity. This work provides a case study in allying formal and agile software development to enable rapid development of robust software.

**Keywords:** Behaviour-driven development, Formal modelling, Gherkin, Event-B, Gift Aid, Swiftaid

## 1. Introduction

In the UK, Gift Aid is a system of tax relief enabling charities to reclaim the income tax paid on donations made by UK taxpayers [HMR16]. Upon the donor declaring that the donation qualifies for Gift Aid, charities can claim an additional 25p from HM Revenues and Customs (HMRC) for each £1 donated at no additional cost to the donor. UK charities miss out on £560 million a year due to unclaimed Gift Aid [HMR18b], the amount that would be paid to charities were it not for donors failing to add Gift Aid to donations when eligible to do so.

*Correspondence to*: D.M. Williams, E-mail: david.m.williams@port.ac.uk, URL: https://research.streeva.com

Published online: 25 May 2020

The inconvenience of having to complete a Gift Aid declaration form by hand upon each donation is often sufficient discouragement. Increased charity adoption of donations via contactless payment cards presents an opportunity to streamline the Gift Aid process and reclaim much of this lost income. The Swiftaid project [UKR19] was an initiative to automate Gift Aid processing. Donors sign up for a free account, to which they add one or more contactless payment cards [Str19]. By capturing relevant data from the payment, Swiftaid automatically adds an extra 25% to donations at no additional charge to the donor.

In this work, we present the formal and agile development approach followed to ensure the Swiftaid system design satisfies requirements specified in legislation, namely the Donations to Charity (Gift Aid Declarations) Regulations 2016 [HMR16]. This work provides a case study for allying agile software development (used by technology startups for increased productivity, accelerated delivery and allowing for adaption to change) with formal methods (used by safety and security critical industries for applying mathematical rigour to increase systems reliability and robustness) to enable rapid development of high assurance software.

In software engineering, the term *'formal methods'* denotes a broad set of mathematical techniques for the specification and verification of software systems. Their use is often mandated or highly recommended for safety and security critical software components that require the highest levels of assurance [CC17]. Formal methods apply mathematical rigour to assert or refute claims made about the correct behaviour of systems. Distributed concurrent software systems find particular benefit in the application of formal methods to address challenges arising from their inherently complex behaviour. The application of formal methods enables the identification of defects early in the development cycle. Formal methods are employed in safety and security critical circumstances where the reliability and robustness of software is imperative [RT13].

Conversely, agile software development methodologies are employed in environments where increased productivity, accelerated delivery and an ability to adapt to change are paramount [Man19b]. Agile techniques are characterised as productive, incremental, adaptable, efficient, sustainable, cooperative and reflective. Proponents of agile methodologies assert their ability to increase productivity by delivering value early and often, with frequent delivery of enhanced utility. In principle, agile methodologies are adaptable and responsive to changing requirements throughout development. When used effectively, the iterative process of structuring work done in short bursts enables development to maintain a consistent, steady pace and encouraging frequent communication and interaction between developers, managers and external stakeholders.

Formal methods and agile are perceived to be in conflict, for example, in their willingness to welcome changing requirements. Changes to the requirements or design after the modelling and analysis stage often lead to additional overheads due to the need to repeat the validation and verification effort. However, formal models can be used as a useful basis for understanding and appreciating inherent consequences arising from requirements change. Using a formal model to analyse the extent to which the change in requirements impacts the system may avoid costs associated with uncovering unforeseen consequences much deeper into implementation and/or testing. For formal methods to ally with agile software development it is necessary for the re-analysis to be quick, efficient and supported by automated tools [LFW10]. Where formal methods stand to benefit from agile techniques, is in the close collaboration with all stakeholders that they advocate. To effectively analyse a model of a system against its requirements, the requirements must have been accurately expressed, captured, communicated and formalised. Close collaboration between stakeholders can help avoid requirements being inaccurate, incomplete or open to misinterpretation.

Our approach united agile software development (we used Behaviour-Driven Development, expressing legislation as Gherkin scenarios [Ghe19]) with formal methods (we used Event-B whose theory of refinement enables incremental system design, expressing desired system behaviour at different levels of abstraction while proving consistency between them.) Event-B has been used within sectors including automotive, rail, space telecommunication and business information by Bosch [GJ13], Siemens [FLDL+13], Finland Space Systems [ILL+13] and SAP [WKW+13]. Event-B benefits from significant tool support; we used the Rodin Platform [ABH+10] (an Eclipse-based IDE supporting refinement and mathematical proof) and ProB [LB03] (an animator and model checker for Event-B, as used by Thales, Alstom and others [HSL16].)

Section 2 provides background on Gift Aid, Behaviour-Driven Development and Event-B. Section 3 presents Gherkin scenarios that express legislated requirements from the Donations to Charity (Gift Aid Declarations) Regulations 2016 [HMR16]. Section 4 outlines the process of authorised donor intermediaries: (i) creating and delivering Gift Aid declarations to charities on behalf of donors, and (ii) processing cancellations of Gift Aid; see [WDSM20] for the Rodin Event-B models. Section 5 records the positive impact of the formal analysis, avoiding otherwise costly functionality. We conclude with a discussion of related work in Sect. 6 and future work in Sect. 7.

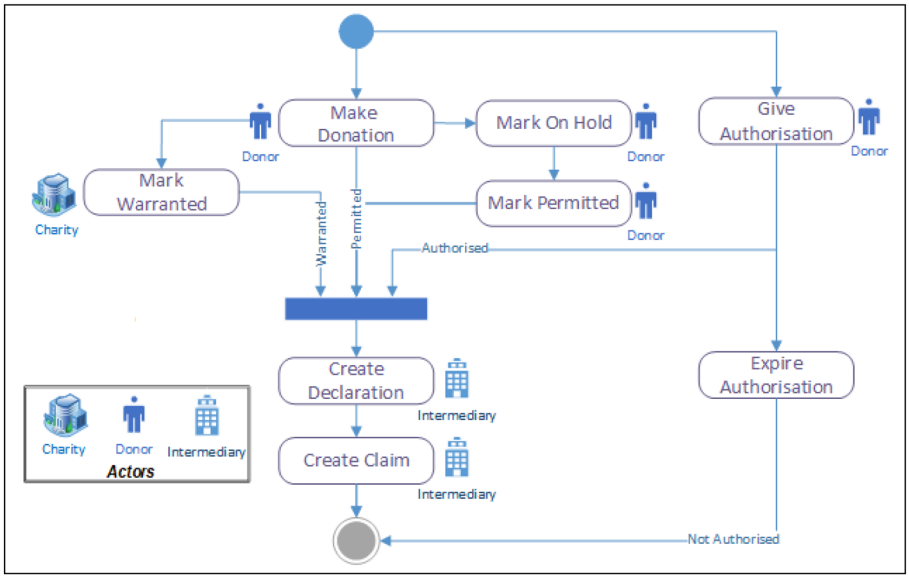


**Fig. 1.** The key activities in the Swiftaid system

## 2. Development process

Our software development approach combined elements of agile and formal methods using Gherkin as a semi-formal intermediary specification language to aid the iterative development of Event-B machines to meet legislated Gift Aid requirements. This section begins by summarising Gift Aid legislation, introduces the Given-When-Then syntax of Gherkin and concludes with an Event-B overview.

### Legislation

The Donations to Charity (Gift Aid Declarations) Regulations 2016 [HMR16] are the statutory instrument specifying criteria for reclaiming income tax paid on donations made by UK taxpayers. Charities can claim an additional 25% from HMRC on each qualifying donation at no additional cost to the donor. Higher and additional rate taxpayers may also claim back the difference between the basic rate of tax claimed on their donation and the higher rate of tax the donor actually pays. The donor must provide a Gift Aid declaration that states they have paid at least the same amount in Income Tax or Capital Gains Tax in that tax year and that they agree to Gift Aid being claimed. The declaration must also include a description of the gift, the name of the charity and the name and address of the donor. Typically, this information is provided in writing by the donor by completing a standard Gift Aid declaration form each time they make a donation.

Swiftaid must satisfy such requirements. As a donor intermediary, donors can authorise Streeva to create and give Gift Aid declarations on their behalf. Figure 1 illustrates the typical flow of Swiftaid activities. A Gift Aid declaration can be created (and a claim made) only if the intermediary has received authorisation and only if a donation was received and confirmed by the charity and donor to be a qualifying donation.

Regulation 6 (see Fig. 2) specifies the requirements for declarations given by a donor intermediary, Regulation 7 imposes record keeping requirements on a donor intermediary, while Regulation 8 (see Fig. 3) specifies the circumstances under which a donor intermediary must provide the donor with an annual statement. Regulation 6 requires Gift Aid declarations given by a donor intermediary to contain the donor's name and address, name the charity, identify the gift being Gift Aided and, importantly, confirm that the identified gift is a qualifying donation. Qualifying donations are specified within Section 416 of ITA 2007; Section 424 of ITA 2007 must have been explained to the donor in advance of the donor intermediary receiving authorisation to give Gift Aid declarations on the donor's behalf. Gift Aid cannot be claimed on donations that are payment for goods or services. For example, Gift Aid cannot be claimed when giving money to enter a raffle, participate in an auction or for tickets to attend an event.

Regulation 6. Gift Aid declaration given by a donor intermediary

1. A Gift Aid declaration given by a donor intermediary on behalf of a donor under regulation 4(1)(b) must
   (a) contain the name and home address of the donor,
   (b) name the charity (or be made in circumstances where the charity is identified),
   (c) identify the gift to which the Gift Aid declaration relates, and
   (d) confirm that the identified gift is to be a qualifying donation for the purposes of section 416 of ITA 2007.
2. Before giving a Gift Aid declaration a donor intermediary must have—
   (a) been authorised by the donor to give a Gift Aid declaration on behalf of the donor, and
   (b) given an explanation of the effect of section 424 of ITA 2007 to the donor

   in order for the Gift Aid declaration to have effect for the purposes of the Income Tax Acts (subject to paragraph (5) and regulation 10).
3. A donor intermediary must obtain the authorisation referred to in paragraph (2)(a) and provide the explanation referred to in paragraph (2)(b) in every tax year in which it gives a Gift Aid declaration on behalf of the donor. This is subject to paragraph (4).
4. Paragraph (3) does not apply in respect of a tax year if the donor intermediary obtained the authorisation referred to in paragraph (2)(a) and gave the explanation referred to in paragraph 2(b) on or after 1st March in the immediately preceding tax year.
5. A donor is entitled to cancel the authorisation referred to in paragraph (2)(a) by giving notice to the donor intermediary.
6. The notification under paragraph (5) may be given in writing or orally, including the use of written or oral methods of electronic communications.
7. Where a donor notifies a donor intermediary under paragraph (5), the authorisation ceases to have effect from the date on which the donor intermediary receives that notification, or if the notice specifies a later date, from that date.

**Fig. 2.** Donations to charity (Gift Aid Declarations) Regulations 2016—Regulation 6 [HMR16]

Regulation 8. Annual statement to be provided by donor intermediaries

1. ... following the end of a tax year during which a donor intermediary has given a gift aid declaration on behalf of a donor under regulation 6, the donor intermediary must send the donor a statement in writing, or details as to how the donor can access a statement in writing .....
2. Paragraph (1) does not apply to a tax year during which
   (a) the aggregate value of the gift aided donations is £20 or less, or
   (b) only one gift aided donation is made.

**Fig. 3.** Donations to charity (Gift Aid Declarations) Regulations 2016—Regulation 8 (*selected text*) [HMR16]

When you receive something in return for your donation, it is not recognised as a 'freewill' gift. Likewise, a donation does not qualify for Gift Aid if a friend gives the donor cash for them to donate on their behalf. As not all donations qualify, Swiftaid should allow for donations to be marked as ineligible for Gift Aid, whether by the donor or the charity.

The few regulations presented in Figs. 2 and 3 already begin to indicate events and state variables one should expect to be included within system design and its formal model. Regulation 6(2)(a) suggests that some status of *authorised* is required before a declaration can be given by a donor intermediary; conversely a declaration cannot be given with the status *notAuthorised*. The status shall transition from *notAuthorised* to *authorised* upon the occurrence of a *giveAuthorisation* event. Regulation 6(3) suggests that each authorisation pertains to a single tax year, thus the authorisation status transitions back to *notAuthorised* upon the expiry date of the authorisation, typically the final day of the tax year. Regulation 6(5), 6(6) and 6(7) allow for a donor to cancel the authorisation by giving notice to the intermediary, suggesting a *cancelAuthorisation* event may occur when *authorised*, with the effect of bringing forward the occurrence of an *expireAuthorisation* event. Similarly, Regulation 8 requires an annual statement to be provided to each donor that has given an aggregate value of gift aided donations in excess of £20 over the tax year and has gift aided at least two donations in that time. This alludes to the system design and formal model needing to record the amounts donated by each donor and account explicitly for the passing of days.

## Gherkin scenarios

Behaviour-Driven Development (BDD) is used to discover, illustrate, and verify desired system behaviour during development [Sma15]. Our first step towards formal development was to specify desired system behaviour as Gherkin features and scenarios within a language that could be easily understood by all stakeholders including senior management, software architects, developers, testers and external stakeholders (in our case, HMRC). Our Gherkin scenarios subsequently aided the construction of the Event-B models by following simple patterns that could be used by both development and modelling teams.

Within Behaviour-Driven Development using Gherkin, concrete example scenarios are used to specify system behaviour and form the basis for automated tests. Gherkin allows scenarios to be specified in a Given-When-Then form. *Given* some precondition holds, *when* some specified triggering event occurs, *then* some desirable behaviour is witnessed, which changes the state of the system such that some post-condition holds. For example, the following scenario relates to a tutorial in which a small code fragment is developed that can figure out whether it's Friday yet [Cuc19a]. Within the feature 'Is it Friday yet?' everybody wants to know whether today is Friday. If asked on a Friday, then the system under test should return 'TGIF' else if it is Sunday or any day other than Friday then the system should respond 'Nope.'

**Scenario: The one where today is or is not Friday**

**Given** today is `<day>`
**When** I ask whether it's Friday yet
**Then** I should be told `<answer>`

**Examples**:

| `day` | `answer` |
|---|---|
| Friday | TGIF |
| Sunday | Nope |
| anything else! | Nope |

Example-driven scenarios often underspecify system behaviour, by failing to capture an exhaustive set of examples. Formal models more precisely capture the system behaviour helping to identify gaps or ambiguities left by such examples. This is useful for resolving issues arising from underspecification or misinterpretation ahead of development effort; such edge cases also provide useful test cases. Formal modelling helps validate and verify that the Gherkin scenarios faithfully capture the intended behaviour expressed in the legislated requirements and that the system design meets the behaviour specified within the Gherkin scenarios.

Given-When-Then statements can be mapped to the $\{P\}C\{Q\}$ constructs of Floyd-Hoare logic, where $P$ denotes some precondition, $C$ denotes some triggering event and $Q$ denotes the resultant postcondition [Car17]. Hoare's $\{P\}C\{Q\}$ triple and its accompanying axioms [Hoa69] provide a means to reason about program correctness and is fundamental to state-based formalisms such as Z [WD96], (classical)-B [Abr96], and Event-B [Abr10]. In [Car17] the authors posit that the mapping enables the creation of a (classical) B-Method model to explore the inherent system behaviour and identify gaps in requirements and test plans. BDD tools (e.g., cucumber [Cuc19b], Behave [Beh19], and jBehave [jBe19]) enable the execution of each specified scenario by *driving* the system under test and checking assertions regarding its behaviour.

The Gherkin scenarios we specified provided a clear foundation for agreement between the formal modelling team and the system design and software development/testing teams.

## Event-B machines

Event-B [Abr10] is a state-based formalism founded on set theory and first order logic. In Event-B, a system is typically presented by a context and machine. The context represents the static part of a model where data values are defined, i.e., carrier sets and constants that are constrained by axioms. The machine represents the dynamic and functional behaviour of the Event-B model, which is composed of variables describing

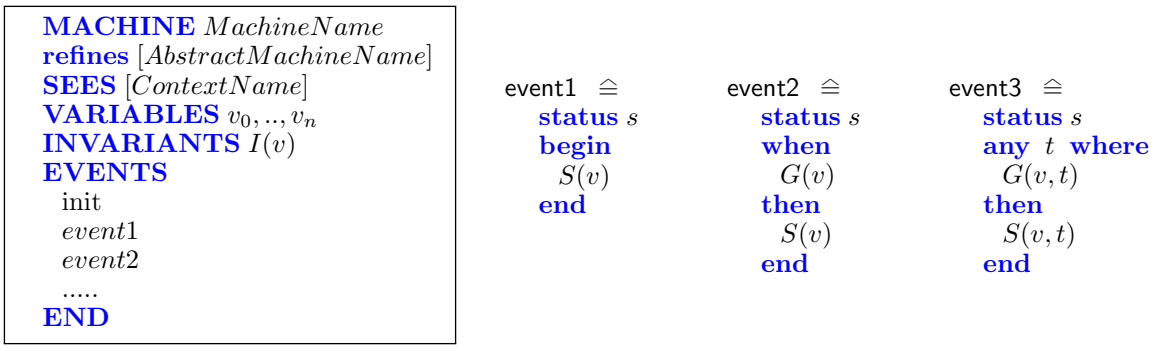


**Fig. 4.** Event-B machine and different event body samples

the system states, invariants constraining variables and events handling machine behaviour, as illustrated in Fig. 4. The machine is linked to the context by the **sees** relationship. Invariants $I(v)$ are necessary predicates that assert specific properties of the variables (the state) before and after the occurrence of each event. Each event includes a guard $G(v)$ and an action $S(v)$. Guards are necessary conditions for an event to be enabled, and actions describe how the state variables change when the event is triggered. Events may also contain local parameters $t$, which are declared in the guard $G(v,t)$ and may be used in the action $S(v,t)$ to update the state variable $v$. Every machine must have an initialisation event, which sets the initial state of each state variable; this special event has no guards or parameters.

Development in Event-B typically begins with developing a high level abstract model of the system and refining the model over a series of steps to incrementally develop a more detailed specification. As presented in [Abr10, HLP13, STW14] and [HSTW16], Event-B provides an effective and flexible refinement strategy, allowing events to be added, merged, forked and extended in steps. Refinement rules may extend functionality, add data values, strengthen event guards and introduce invariants. The **refines** notation indicates that a machine refines another and is also applicable to events. Each Event-B model is mathematically validated through a set of proof obligations to maintain correctness and consistency between refinement levels. Consistency between the behaviour of the refined machine and the machine being refined must be maintained and the proof obligations (e.g., feasibility FIS_REF, guard strengthening GRD_REF and invariant preservation INV_REF) for each event must be proven to hold [Abr10]. Rodin [ABH+10] facilitates Event-B modelling and verification via the generation and automated discharging of such proof obligations.

## 3. Expressing legislated requirements as Gherkin scenarios

In this section we express requirements garnered from the Donations to Charity (Gift Aid Declarations) Regulations 2016 into Gherkin scenarios. Gherkin was selected as an intermediary semi-formal language that could consistently be understood and used by the distinct design, modelling and development teams. The concrete examples of the scenarios specify system behaviour via a Given-When-Then syntax. Given some precondition, when some event occurs, then the state of the system is updated.

Regulation 6(1)(d) of the Donations to Charity (Gift Aid Declarations) Regulations 2016 (see Fig. 2) requires that the gift is confirmed to be a qualifying donation for the purposes of section 416 of ITA 2007. Swiftaid allows both the charity and the donor to influence whether the donation qualifies for Gift Aid. The following conditions must be satisfied for a Gift Aid declaration to be given for a donation:

- A donor must have given authorisation for the intermediary to give Gift Aid declarations on their behalf
- A donor must have made a donation that they have not subsequently marked as ineligible
- The charity should have confirmed that they know of no reason that the donation does not qualify

The donor must take action if they wish some time to consider whether it qualifies (*markOnHold* transitions the donor eligibility from *permitted* to *onHold*) or if they wish to permanently mark the donation as ineligible for Gift Aid (via *markCancelled*). Scenario F02S02 specifies this behaviour, which is later illustrated in Fig. 9 (see Sect. 4.1)

**F02S02** The one where the donor changes the eligibility of a donation

**Given** a donation that is marked `<fromState>` and `<checkDeclared>`
**When** a donor performs `<action>`
**Then** the donation is marked `<newState>`

**Examples**:

| `fromState` | `checkDeclared` | `action` | `newState` |
|---|---|---|---|
| *permitted* | is not declared | *markOnHold* | *onHold* |
| *permitted* | is not declared | *markCancelled* | *cancelled* |
| *permitted* | is declared | *markCancelled* | *cancelled* |
| *onHold* | is not declared | *markCancelled* | *cancelled* |
| *onHold* | is not declared | *markPermitted* | *permitted* |

*Only state changes listed are possible, e.g., markOnHold cannot occur when a declaration exists.*

Conversely, the charity must explicitly mark the donation as *warranted* to confirm that they have no reason to doubt that the donation may qualify (F02S03). The charity cannot return the *charityEligibility* to *notWarranted*; a design decision made to manage the complexity of Swiftaid's first release. Section 5 elaborates on how the modelling helped to appreciate and mitigate complexity arising from design decisions.

**F02S03** The one where the charity confirms the eligibility of the donation

**Given** a donation that is not marked *warranted*
**When** a charity performs *markWarranted*
**Then** the donation is marked *warranted*

F02S04 specifies that declarations may only be created for qualifying donations on behalf of donors that have authorised the donor intermediary to do so (those marked *permitted*, *warranted* and *authorised*).

**F02S04** The one where the intermediary tries to create a declaration from a donation

**Given** a donation that is marked `<donorEligibility>`
And the donation is marked `<charityEligibility>`
And the intermediary is `<authorisation>` to create declarations for the donor
And there is no existing declaration linked to the donation
**When** the intermediary tries to create a declaration
**Then** a declaration is `<result>`

**Examples**:

| `donorEligibility` | `charityEligibility` | `authorisation` | `result` |
|---|---|---|---|
| *permitted* | *warranted* | *authorised* | *created* |
| *onHold* | * | * | *notCreated* |
| *cancelled* | * | * | *notCreated* |
| * | *notWarranted* | * | *notCreated* |
| * | * | *notAuthorised* | *notCreated* |

*We use * to denote a wildcard of any arbitrary value*

Feature F02 is a special case; its scenarios capture more of the Swiftaid system design than the legislation mandates. Regulation 6(1)(d) requires a donation to qualify, whereas F02 specifies how Swiftaid ensures the donation qualifies. Other regulations were more directly captured in Gherkin, such as the following examples.

Regulation 6(7) of the Donations to Charity (Gift Aid Declarations) Regulations (see Fig. 2) requires the donor to be able to cancel authorisation from a specified date by giving notice to the donor intermediary.

**F04S01: The one where notice is given to cancel the authorisation on a specified date**

**Given** a donor gave authorisation on 2018-02-28 set to expire on 2018-04-06
**When** a donor intermediary receives notice from the donor
to cancel their authorisation on 2018-03-01
requested to expire on `<requestedExpiryDate>`
**Then** the authorisation is set to expire on `<expiryDate>`

**Examples**:

| `requestedExpiryDate` | `expiryDate` |
|---|---|
| 2018-03-01 | 2018-03-01 |
| 2018-03-08 | 2018-03-08 |
| 2018-01-01 | 2018-03-01 |
| 2019-01-01 | 2018-04-06 |

In the first two examples, the date at which the authorisation ceases to have effect is brought forward to the requested expiry date. The other examples capture what must happen if (i) the requested expiry date is before the notice is received or (ii) the requested expiry date is after the current expiry date. In (i) the authorisation ceases to have an effect from the date the notice is received; in (ii) the expiry date is not changed. Recall from Sect. 3 that example-driven scenarios are likely to underspecify the system behaviour, by failing to capture an exhaustive set of examples. In F04S01, just a few representative examples are given. Numerical data values often give rise to such underspecification, so we delayed the introduction of such values until the final Event-B machine in the refinement chain (see Sect. 4). In F02S04, we used a wildcard '*' to denote an arbitrary value to capture all circumstances in just a few lines.

F06S03 also specifies just a few examples using numerical values to represent all possible values. Regulation 8 of the Donations to Charity (Gift Aid Declarations) Regulations (see Fig. 3) requires the donor intermediary to send the donor a statement unless the aggregate value of the gift aided donations in that year is £20 or less, or only one gift aided donation was made. Various circumstances meeting or failing to fulfil the criteria are expressed via concrete scenario examples. In the first three examples, an annual statement needn't be provided by the intermediary due to Regulation 8(2). In the first example no donation has been made by the donor, in the second the aggregate value is insufficient, while the third has insufficient gift aided donations. Finally, Regulation 8(1) *does* apply to the fourth scenario, so a statement *must* be provided.

**F06S03** The one where an annual statement will or will not be provided to the donor

**Given** the donor has `<count>` declarations that total `<amount>` in this tax year
**When** the tax year ends
**Then** a statement `<will-or-will-not>` be provided to the donor

**Examples**:

| `amount` | `count` | `will-or-will-not` |
|---|---|---|
| £0 | 0 | will not |
| £20 | 3 | will not |
| £500 | 1 | will not |
| £20.01 | 2 | will |

In total, 34 Gherkin scenarios over 7 features were specified to capture Swiftaid's required behaviour. Each scenario was annotated with the associated items of legislation (regulation). For example, F02S02 was labelled '@2016/1195/6.1.d' providing a link back to Regulation 6(1)(d) [HMR16]. There was a many-to-many relationship between regulations and scenarios. Sometimes a regulation was best expressed in several scenarios, while scenarios were sometimes refactored to capture aspects of multiple regulations. The Event-B machines were similarly commented, such that an item of legislation could be linked to the set of scenarios that captured its required behaviour, which could in turn be linked to each line in the Event-B.

## 4. Modelling system behaviour via Event-B machines

Event-B modelling was conducted prior to software development and supported refinement of the system requirements and design. Once an initial set of Gherkin scenarios had been specified, regular meetings were held between the modelling and development teams to refine the models, design and requirements, as necessary. This section documents the final Event-B models [WDSM20] delivered within the Swiftaid project [UKR19]. We architected a series of refinements involving the following three Event-B machines, illustrated in Fig. 5:

- $M_0$ **Core Functionality**—the initial Event-B machine incorporates the essential Swiftaid system behaviour concerning the authorisation of the donor intermediary to create declarations on behalf of the donor, receiving notification of a donation, creating a declaration for a qualifying donation and creating a Gift Aid claim for a set of declared donations. It abstracts away details concerning quantities; it excludes the amount donated and does not account explicitly for the passing of time.
- $M_1$ **With Numerical Values**—our second Event-B machine refines $M_0$ by providing supplementary state information and new events regarding the amount donated and the amount of donations made, necessary for certain legislated requirements. For example, Regulation 8 (see Fig. 3) requires that an annual statement be provided to a donor only if the aggregate value of their gift aided donations exceeded £20 and at least two gift aided donations were made; $M_1$ tallies the total aggregate value of each donor's gift aided donations and counts the number of donations each donor has made.
- $M_2$ **Finer Grained Time**—our final Event-B machine further refines the system behaviour by explicitly accounting for the passing of days. Certain aspects of the requirements are date-dependent. For example, Regulation 6(4) (see Fig. 2) allows the donor intermediary to obtain authorisation for the forthcoming tax year on or after 1st March in the current (soon to end) tax year. A donor is also entitled to cancel the authorisation on any date by giving notice to the donor intermediary.

As well as illustrating the three Event-B machines in the refinement chain, Fig. 5 also shows their composition of the following six groups of events (each of which are discussed in more detail later in this section):

- **Time**—As certain Gift Aid regulations are time constrained, our models should include the passing of time. In $M_0$ and $M_1$ we account only for the passing of tax years and maintain a temporal ordering of events; a finer grained treatment of time is deferred until $M_2$, which accounts for the passing of days to enable the consideration of requirements that concern specific dates of the year.
- **Authorisation**—A donor must provide a donor intermediary with authorisation to create Gift Aid declarations on their behalf. A donor typically provides authorisation for the remainder of the tax year but may also give notice to cancel an authorisation, which shall expire on the date given. Towards the end of the tax year a donor may provide advance authorisation for the duration of the subsequent year.
- **Donation**—Once Swiftaid is notified of a donation then a Gift Aid declaration can be made on behalf of the donor only if the donor and charity agree that it is a qualifying donation. Each machine incorporates actions within the Swiftaid system design associated with the donor and charity confirming whether or not the donation qualifies for Gift Aid (by marking the donation *permitted* and *warranted*, respectively).
- **Declaration**—Assuming that the donor and charity agree that it is a qualifying (eligible) donation, i.e., the donation is marked as *permitted* and *warranted*, and the donor intermediary has been authorised by the donor, then a Gift Aid declaration for this donation may be created on behalf of the donor.
- **Claim/Overclaim**—Finally, Gift Aid claims can be created on behalf of the charity that include each donation for which a Gift Aid declaration has been created on behalf of the donor. Occasionally, a donor may cancel a declaration for which Gift Aid has already been claimed; an overclaim is then required to refund HMRC the appropriate amount.

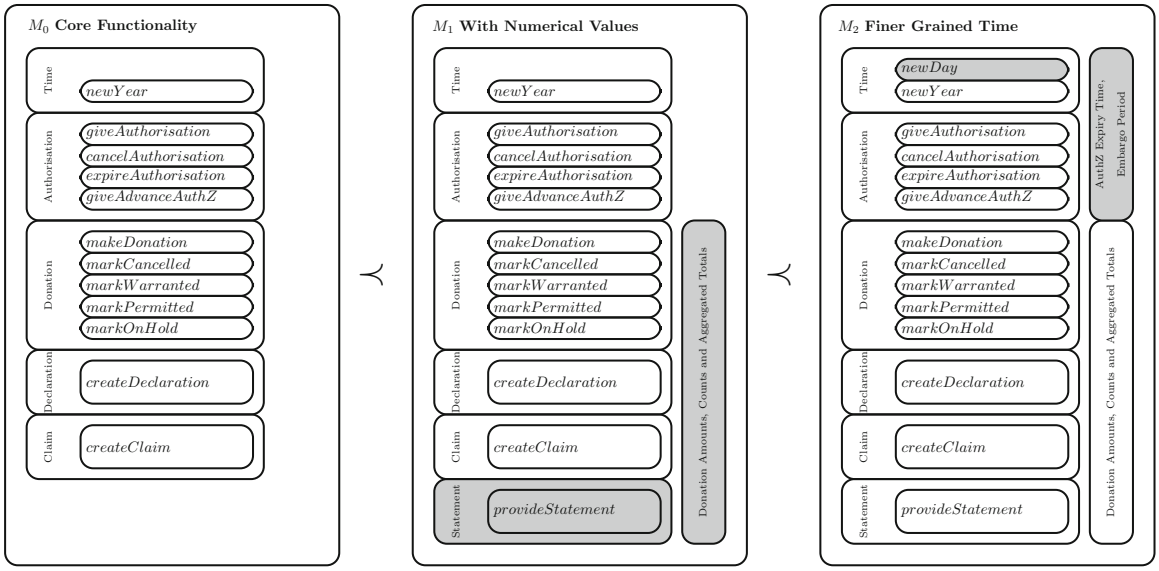


**Fig. 5.** Event-B stepwise refinement of Swiftaid

- **Statement**—At the end of the tax year donor intermediaries are required to provide annual statements to donors. Statements are to be provided to all donors that have donated more than once in the tax year and the donations amounted to more than £20. In $M_1$ we introduce the amount donated, count the number of donations made and sum the total amount donated.

The remainder of this section describes aspects of the three Event-B models ($M_0$, $M_1$ and $M_2$) that are most pertinent to the following three User Stories[1]. Processing a donation (through to creating Gift Aid declarations and claims), and allowing cancellation of a declaration at any time, are essential aspects of Gift Aid. Our description focuses on these and other representative modelling aspects; other details are omitted for brevity.

- **Processing a Donation**: As a donor, I want Swiftaid to process my donation so that the charity can claim Gift Aid on my behalf. According to the regulations, only qualifying (eligible) donations should be declared and subsequently included in a claim.
- **Cancelling a Donation/Declaration**: As a donor, I want Swiftaid to cancel the processing of a donation so that either: (i) Gift Aid is not claimed; or (ii) Gift Aid is refunded to HMRC if already claimed, i.e., an over claim is recorded in a subsequent claim.
- **Reversing a Cancellation**: As a donor, upon changing the status of a previously cancelled donation to permitted, I want Swiftaid to process my donation so that the charity can claim Gift Aid on my behalf.

The third user story is not a functionality included in the final model or system design. It was a user story considered during early stages of the system design, but it was deemed extraneous as a result of the formal analysis. The added value provided by the formal model in communicating the increased complexity that arose from the inclusion of the feature is reserved for independent discussion in Sect. 5.

## 4.1. Processing a donation

The primary Swiftaid user story is for a donation to be processed such that the charity can claim Gift Aid with minimal effort to the donor. This first requires an annual *authorisation* step in which the donor authorises the

[1]The user stories presented in this section provide structure to our explanation of the modelling of the Swiftaid system behaviour. Scenarios were not derived from the user stories; the Gherkin scenarios were specified directly from Gift Aid legislation.

donor intermediary to give Gift Aid declarations on their behalf. Thereafter, when notified that a donation has been made, Swiftaid generates a declaration for the donation (if it qualifies for Gift Aid) and periodically creates a Gift Aid claim for a set of declared donations on behalf of the charity. This section presents the step-wise refinement in terms of the key aspects of the model concerning processing of a donation. Figure 1 illustrates this typical process.

### $M_0$—*Core Functionality*

Our first Event-B machine, $M_0$, represents the core abstract behaviour of the Swiftaid system for handling Gift Aid declarations and claims, but without incorporating precise amounts or dates. As shown in Fig. 5, $M_0$ incorporates five key groups of events: (i) Time; (ii) Authorisation; (iii) Donations; (iv) Declarations; and (v) Claims. $M_0$ excludes the amount donated and does not account for the passing of days. $M_0$ does, however, account for the passing of tax years and maintains the temporal ordering of events.

**Time**—While accounting for precise dates within each tax year is deferred until $M_2$, the initial machine at least accounts for the passing of tax years. This is so that our initial model can incorporate abstract notions of an authorisation expiring on or before the end of a tax year and advance authorisation being provided for the subsequent tax year. The event **newYear** marks the end of one tax year and the beginning of the next.

**Authorisation**—Regulation 6 (see Fig. 2) of the Donations to Charity (Gift Aid Declarations) Regulations 2016 [HMR16] specifies that a donor intermediary must be authorised by the donor to give a Gift Aid declaration on their behalf. A donor typically provides authorisation for the remainder of the tax year but may also give notice to cancel an authorisation, which shall expire on the date given. Towards the end of the year a donor may provide advance authorisation for the duration of the subsequent tax year.

Four events update the state of *donorAuthorisation*, the variable that records whether a given donor has authorised the donor intermediary to give Gift Aid declarations on their behalf (*authorised*) or not (*notAuthorised*). **giveAuthorisation** models a donor giving authorisation to an intermediary to process Gift Aid on their behalf for the remainder of the current tax year. It switches the *donorAuthorisation* from *notAuthorised* to *authorised*. The initial state of the *donorAuthorisation* variable is *notAuthorised*.

Three other events in $M_0$ can also change the *donorAuthorisation* state. **expireAuthorisation** switches *donorAuthorisation* to *notAuthorised* upon reaching the date that authorisation expires (this is typically the end of the tax year, but could otherwise be at an earlier date specified by the donor). **cancelAuthorisation** shall later enable the donor to specify a date at which the authorisation shall expire. In this initial machine, which does not keep track of specific dates, *skip* indicates that this event does not change the state of the system at this level of abstraction. Finally, **giveAdvanceAuthorisation** allows a donor to grant an advance authorisation for the next tax year when within thirty days of the next tax year. Since accounting for the passing of days is deferred until $M_2$ the execution of this event is not restricted in the initial machine.

Figure 6 documents selected aspects regarding donor authorisation within the initial machine $M_0$, while Fig. 7 illustrates the possible authorisation (and advance authorisation) states for a single donor. A donor starts without having given the donor intermediary authorisation, nor advance authorisation. A typical cycle through the statespace would be for authorisation to be given (via *giveAuthorisation*) at the start of each tax year, which expires at the end of the year (i.e., upon *newYear*). Giving authorisation when the donor intermediary is already authorised does not change the state of the system at this initial abstract, as witnessed by the *giveAuthorisation* self loops. In a subsequent refinement, once the passing of days is accounted for, giving authorisation until the end of the current tax year having previously brought forward the authorisation expiry date to earlier in the year via *cancelAuthorisation* would change the state; the *expiryDate* would be updated to be the last day of the current tax year. The event *expireAuth* models the expiry of authorisation at a date earlier than the end of the year, otherwise *newYear* expires the authorisation. If advance authorisation has been given, then upon the start of a new tax year (*newYear*) then the *donorAuthorisation* is set to *authorised* and the *donorAdvanceAuthorisation* is reset to *notAuthorised*.

**Donation**—Our model includes an event modelling the Swiftaid system receiving notification that a donation has been made and events for actions that may be taken by the donor or charity ahead of a declaration being created on behalf of the donor. **makeDonation** models the notification that a donor has made a donation; it associates an identifier of the donation and a donor (and an amount, but not until the first refinement). Subsequently, there are five events that switch the overall state of a donation once a donation is received.

Founded in Regulation 6(1)(d) of the Donations to Charity (Gift Aid Declarations) Regulations 2016 (see Fig. 2), F02S02 requires that the donor must take action if they wish some time to consider whether it qualifies (see Sect. 3); **markOnHold** transitions the *donorEligibility* from *permitted* to *onHold*. Having

**MACHINE** $M_0$
**SEES** $[C_0]$
**VARIABLES** $donorAuthorisation, donorAdvanceAuthorisation, .....$
**INVARIANTS**
$donorAuthorisation \in DONORS \rightarrow AUTHORISATION\_STATES$
$donorAdvanceAuthorisation \in DONORS \rightarrow AUTHORISATION\_STATES$
.....
**EVENTS**

init
**begin**
$donorAuthorisation \coloneqq \{x \mapsto y \mid x \in DONORS \land y = notAuthorised\}$
$donorAdvanceAuthorisation \coloneqq \{x \mapsto y \mid x \in DONORS \land y = notAuthorised\}$
.....
**end**

giveAuthorisation $\widehat{=}$
**status** *ordinary*
**any** *donor* **where**
$donor \in DONORS$
**then**
$donorAuthorisation(donor) \coloneqq authorised$
**end**

expireAuthorisation $\widehat{=}$
**status** *ordinary*
**any** *donor* **where**
$donor \in DONORS$
$donorAuthorisation(donor) = authorised$
**then**
$donorAuthorisation(donor) \coloneqq notAuthorised$
**end**

cancelAuthorisation $\widehat{=}$
**status** *ordinary*
**any** *donor* **where**
$donor \in DONORS$
$donorAuthorisation(donor) = authorised$
**then**
*skip*
**end**

giveAdvanceAuthorisation $\widehat{=}$
**status** *ordinary*
**any** *donor* **where**
$donor \in DONORS$
$donor \in dom(donorAdvanceAuthorisation)$
**then**
$donorAdvanceAuthorisation(donor) \coloneqq authorised$
**end**

newYear $\widehat{=}$
**status** *ordinary*
**begin**
$donorAuthorisation(donor) \coloneqq \{x \mapsto y \mid x \in DONORS \land y = donorAdvanceAuthorisation(x)\}$
$donorAdvanceAuthorisation(donor) \coloneqq \{x \mapsto y \mid x \in DONORS \land y = notAuthorised\}$
.....
**end**

.....
**END**

**Fig. 6.** Donor authorisation variables, invariants and events in the first machine-$M_0$

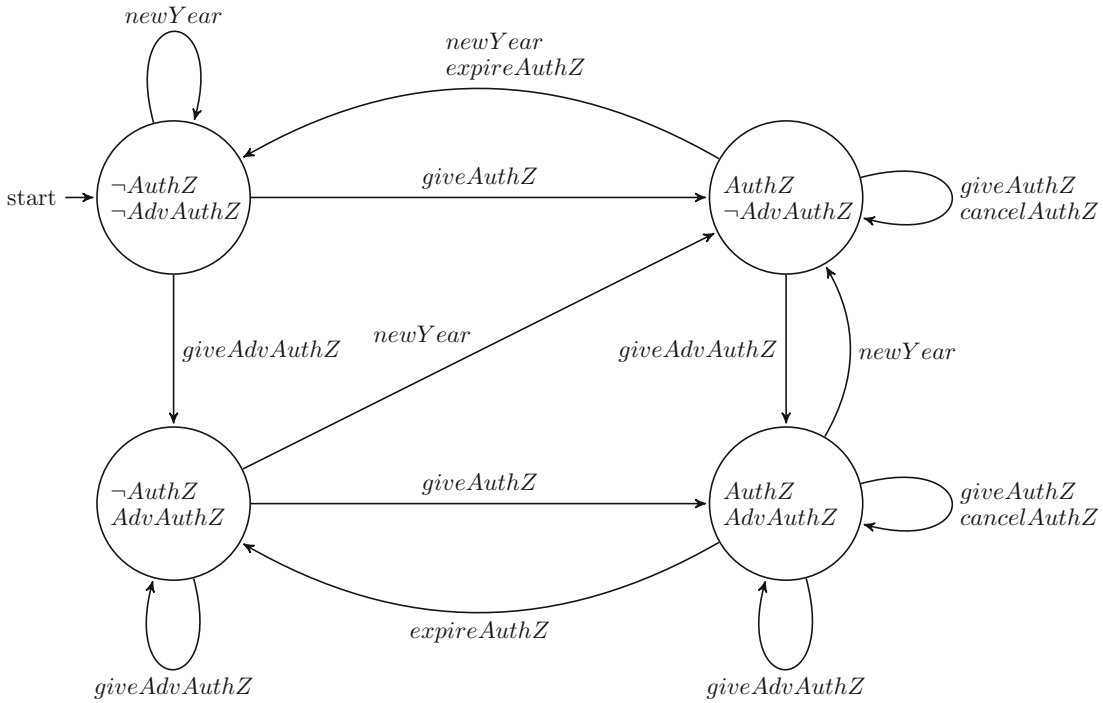


**Fig. 7.** The state machine for authorisation (AuthZ) and advance authorisation (AdvAuthZ) of a single donor

taken time to consider whether the donation qualifies in the *onHold* state, a donor may confirm that it can be gift aided by again marking it as *permitted* via **markPermitted**. The donor may otherwise wish to permanently mark the donation as ineligible for Gift Aid, but we defer discussion of **markCancelled** until Sect. 4.2. Figure 8 documents aspects regarding *donorEligibility* within $M_0$, while Fig. 9 illustrates the behaviour of $M_0$ in terms of those events concerning *donorEligibility* for a single donation. F02S03 requires the charity to explicitly mark the donation as eligible to confirm that they have no reason to doubt that the donation may qualify; **markWarranted** transitions the *charityEligibility* from the *notWarranted* state to *warranted* confirming that the donation was not, for example, in return for good or services.

As the initial machine does not account for days passing to determine the precise date within the year, an auxiliary variable called *yearCreated* classifies donations based on the recency of their creation (i.e., *thisYear* or *somePreviousYear*). This is necessary to ensure that only donations made in the current tax year (in which the donor intermediary holds authorisation) can have declarations created. The *yearCreated* is set to *thisYear* upon the receipt of notification that a donation has been made (via *makeDonation*); the *newYear* event sets all donations received as having been received in *somePreviousYear*.

**Declaration**—Assuming that the donor and charity agree that the donation qualifies for Gift Aid, i.e., the donation is marked as *permitted* and *warranted*, then a Gift Aid declaration for this donation may be created on behalf of the donor. **createDeclaration** models the creation of a Gift Aid declaration of a single donation. A donation can only be declared if specific properties are satisfied, adhering to the system requirements (see F02S02, F02S03 and F02S04 in Sect. 3).

The *donorEligibility* for this particular donation must be *permitted* and the *charityEligibility* must be *warranted*. Importantly, the donor intermediary must be authorised to give declarations on behalf of the donor, i.e., the *donorAuthorisation* state of the *donor* giving the *donation* must be *authorised* for a declaration to be created (via *createDeclaration*). Finally, a donation can only be declared when this donation is received in the current tax year (*thisYear*) and not some previous year (*somePreviousYear*). Should all such requirements be satisfied, i.e., all guards of the *createDeclaration* event hold, then *createDeclaration* adds an association between the *donation* and a unique *declarationId* to the set of all declarations (*Declarations*) and the *DeclarationState* for this particular declaration (*declarationId*) is set irreversibly to *declared*.

**MACHINE** $M_0$
.....
**EVENTS**
.....

markPermitted $\widehat{=}$
**status** *ordinary*
**any** *donor*, *donation* **where**
$donor \in DONORS$
$donation \in DONATIONS$
$(donation \mapsto donor) \in Donations$
$donorEligibility(donation) = onHold$
$donation \notin dom(Declarations)$
**then**
$donorEligibility(donation) := permitted$
**end**

markOnHold $\widehat{=}$
**status** *ordinary*
**any** *donor*, *donation* **where**
$donor \in DONORS$
$donation \in DONATIONS$
$(donation \mapsto donor) \in Donations$
$donorEligibility(donation) = permitted$
$donation \notin dom(Declarations)$
**then**
$donorEligibility(donation) := onHold$
**end**

markCancelled $\widehat{=}$
**status** *ordinary*
**any** *donor*, *donation* **where**
$donor \in DONORS$
$donation \in DONATIONS$
$(donation \mapsto donor) \in Donations$
$donorEligibility(donation) \neq cancelled$
**then**
$donorEligibility(donation) := cancelled$
$DeclarationState := DeclarationState \Leftarrow (Declarations[\{donation\}] \times \{reversed\})$
**end**

createDeclaration $\widehat{=}$
**status** *ordinary*
**any** *donor*, *donation*, *declarationId* **where**
$donor \in DONORS$
$donation \in DONATIONS$
$Donations(donation) = donor$
$donorEligibility(donation) = permitted$
$charityEligibility(donation) = warranted$
$yearCreated(donation) = thisYear$
$donorAuthorisation(donor) = authorised$
$declarationId \in DECLARATIONS$
$declarationId \notin ran(Declarations)$
$donation \notin dom(Declarations)$
**then**
$Declarations := Declarations \cup \{donations \mapsto declarationId\}$
$DeclarationState := DeclarationState \cup \{declarationId \mapsto declared\}$
**end**

.....
**END**

**Fig. 8.** Donor eligibility variables, invariants and events in the first machine-$M_0$

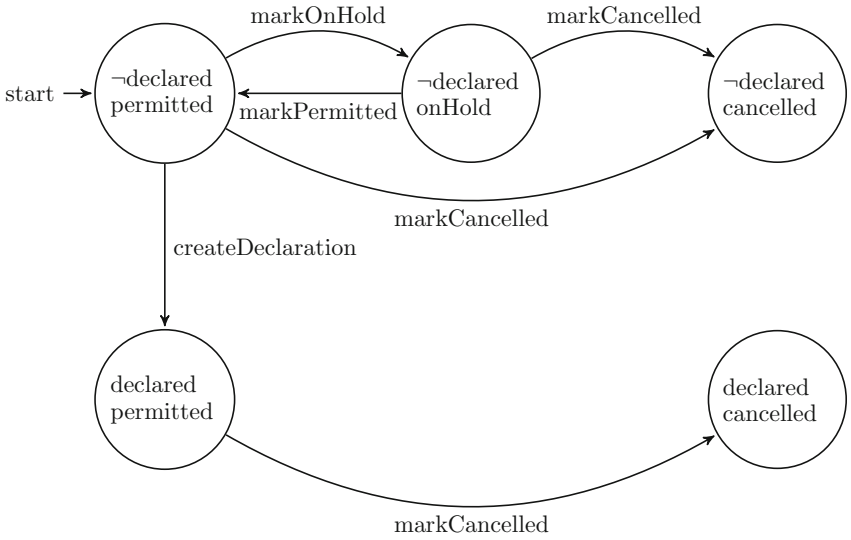


**Fig. 9.** The state machine for donor eligibility for a single donor

**Claim**—Finally, charities may claim Gift Aid on those donations for which they have received Gift Aid declarations. In our model *Claims* maps a *declarationId* of a declared donation to a *claimId*. The **createClaim** event generates a claim for all declared donations that are yet to be claimed. The event results in associating the set of all donations that are both *declared* and *permitted* (but not yet claimed) to a new *claimId*. Donations marked cancelled prior to having been declared shall be precluded from any subsequent declaration/claim. See Sect. 4.2 for discussion on cancelling a donation/declaration.

### $M_1$ — *With numerical values*

**Statement**—the first refinement, machine $M_1$, introduces the handling of numerical data to aggregate the total donated by each donor and count how many donations were made by each donor. A new event **provideStatement** is introduced to model the creation of a statement for a donor regarding their declared donations over the current tax year guarded by two constraints. Firstly, a *donor* must have a total amount of declared donations in the current year greater than 2000 (pence). Secondly, a donor must have had multiple Gift Aid declarations given on their behalf. Annual statements are to be provided around the end of the tax year, but as $M_1$ abstracts away from accounting for the passing of days; consideration of when in the year a statement is to be provided is deferred to $M_2$. Providing a statement is not intended to change the system state, which is modelled as *skip* within the action section of the event. Figure 10 illustrates the introduction of the new event *provideStatement* within the first refinement, $M_1$.
**Donation/Declaration**—for $M_1$ to record the data necessary to provide annual statements, events capturing (i) Swiftaid's receipt of a notification that a donation has been made and (ii) the creation of associated declarations, are extended to account for the amount being donated. An *amount* parameter is added to the **makeDonation** event allowing any donation amount within a defined range (*valueRange* specified within the context). A new relation *donationAmount* records the amount of each donation, mapping a *donationID* to the amount donated. Only once a donation is declared are the aggregate value donated and the number of donations count increased. Figure 10 illustrates how *makeDonation* is extended in this manner. In the **createDeclaration** event, once a declaration (*declarationId*) is created for a given donation (*donation*) made by a certain donor (*donor*), the newly added variable *donorDonationTotal* is incremented by the donation amount. Likewise, *donorDonationCount* is incremented by one for that donor.

```
MACHINE M1
refines [M0]
SEES [C1]
.....
EVENTS
  .....
  makeDonation  ≙
    status ordinary
    any donor, donation, amount where
      donation ∈ Donations
      donation ∉ dom(Donations)
      donor ∈ DONORS
      donor ∈ dom(donorAuthorisation)
      amount ∈ donValueRange
      donation ∉ dom(donationAmount)
    then
      Donations(donation) := donor
      donorEligibility(donation) := permitted
      charityEligibility(donation) := notWarranted
      yearCreated(donation) := thisYear
      donationAmount := donationAmount ∪ {donation ↦ amount}
    end

  provideStatement  ≙
    status ordinary
    any donor, donation where
      donor ∈ DONORS
      donor ∈ dom(donorDonationTotal)
      donor ∈ dom(donorDonationCount)
      donorDonationTotal(donor) > 2000
      donorDonationCount(donor) > 1
    then
      skip
    end
  .....
END
```

**Fig. 10.** Assorted new and refined events in machine $M_1$

### *$M_2$—Finer grained time*

The final machine in the refinement chain illustrated in Fig. 5, $M_2$, incorporates a finer grained notion of time into our Event-B model to account for the passing of days and handle requirements constrained by specific dates within the year. In $M_2$ we model time as a global clock via the *time* variable which incrementally counts the number of days since the beginning of the tax year.

A new convergent event *newDay* is introduced to mark the passing of days, incrementing *time* by one day until the final day of the current (tax) year. Convergent events must be proven to only occur some finite number of times before some non-convergent event occurs. To prove convergence a variant must be specified as an expression that yields a natural number, which strictly decreases upon the occurrence of any convergent event. Our variant is the integer value recording the number of days until the next tax year, calculated by subtracting the number of days into the current tax year, *time*, from the total number of days in the tax year, *totalNewDays* (see Fig. 11). Each *newDay* decreases the variant by one; the guard of *newDay* precludes the occurrence of the event if this value is less than two. On the final day of the tax year, the next tax year is one day away and *newDay* cannot fire. A distinct event *newYear*, which may only occur on the final day of each tax year, marks the end of one tax year and the beginning of another. *newYear* resets the number of days left in the tax year.

As the *time* variable is included in this machine to account for the passing of days, a number of events predefined in the previous machines can be extended by adding new parameters, guards or actions pertaining to *time*. Firstly, in the *makeDonation* event, the current *time* must be captured in the *DonationTime* set once a given donation is added to *Donations* set in order to monitor a one day embargo period to be handled

**MACHINE** $M_2$
**refines** $[M_1]$
**SEES** $[C_2]$
.....
**VARIANT** $totalNewDays - time$
**INVARIANTS**
$\forall donor.(donor \in dom(DonorAuthzExprTime) \implies$
$(DonorAuthzExpTime(donor) = 0 \Leftrightarrow donorAuthorisation(donor) = notAuthorised))$
.....
**EVENTS**
.....

cancelAuthorisation $\widehat{=}$
**status** *ordinary*
**any** $donor$, $exprTime$ **where**
$exprTime \in 0..totalNewDays$
$donor \in dom(DonorAuthzTime)$
$exprTime < DonorAuthzExpTime(donor)$
$exprTime \geq time$
**then**
$DonorAuthExpTime(donor) := exprTime$
**end**

expireAuthorisation $\widehat{=}$
**status** *ordinary*
**any** $donor$ **where**
$donor \in DONORS$
$donorAuthorisation(donor) = authorised$
$donor \in dom(DonorAuthExpTime)$
$DonorAuthExpTime(donor) = time$
**then**
$donorAuthorisation(donor) := notAuthorised$
$DonorAuthExpTime(donor) := 0$
**end**

newDay $\widehat{=}$
**status** *convergent*
**when**
$time = totalDays$
$\forall donor.(donor \in dom(DonorAuthzExpTime) \implies DonorAuthzExpTime \neq time)$
**then**
$time := time + 1$
**end**

.....
**END**

**Fig. 11.** Assorted new and refined events in machine $M_2$

in the *createDeclaration* event. The brief embargo period is introduced into the system design to leave some time for a donor to mark a donation as *onHold* or *cancelled* before creating a declaration on their behalf. The *createDeclaration* event can be enabled only if the current time is equal to or less than the total time of donation time and predefined embargo period.

To fulfil F06S03, in accordance with Regulation 8 of the Donations to Charity (Gift Aid Declarations) Regulations 2016, *provideStatement* models the provision of *donor* statements at the end of each tax year. Since in this machine the passing of days is now explicitly modelled, a guard can be introduced that checks if the current *time* reaches *totalNewDays* (i.e., the final day of the tax year). Time is also important for the fulfilment of scenarios concerning each donor's authorisation of the donor intermediary, in accordance with Regulation 6 (see Fig. 2). A new guard is added to *expireAuthorisation* monitoring current *time* with respect to the authorisation expiry time for a particular donor, *DonorAuthzExpTime*. If the current time has reached the donor's authorisation expiry time, the *donorAuthorisation* is set to *notAuthorised* and the *DonorAuthzExpTime* variable is reset for that donor. *cancelAuthorisation* is extended to more faithfully model the donor giving notice of when the authorisation should cease to have effect. This event is augmented with a new parameter *exprTime*, which refers to the earlier expiry time of authorisation being requested by

the donor. Once all guards of this event are satisfied, the event brings forward the *DonorAuthzExpTime* for a particular donor to the new *exprTime*.

To fulfil F03S02, *giveAdvanceAuthorisation* must be allowed only in the last thirty days of the current year. In previous machines, this event was always enabled without restriction. In this machine, we simply add a new guard to check that the current *time* is within the last 30 days of the current tax year.

The specification of the Event-B model enables us to check that the behaviour of the model is correct and consistent with the formal constraints of the system. The Rodin tool [ABH+10] automatically generates proof obligations (PO) which must be discharged, typically using Rodin's integrated mathematical provers. Using Rodin, we discharged POs for each of the three machines modelling the Swiftaid system behaviour, almost all of which were discharged automatically although some required the addition of missing statements or predicates. Due to the complexity of set comprehension being used, some POs in $M_1$ and $M_2$ were not fully discharged automatically. As the remaining proofs by hand were incidental, the relative risk of introducing errors at this stage was determined to be very low in relation to further holding back implementation. Greater benefit was obtained from the formal models uncovering significant unforeseen complexity in other aspects of the initial design (see Sect. 5).

## 4.2. Cancelling a donation/declaration

Recall from Sect. 2 that once a donation is received by the Swiftaid system the donor may postpone the creation of a declaration if they wish to consider whether it qualifies; *markOnHold* transitions the *donorEligibility* from *permitted* to *onHold*. A donor may decide that a donation does not qualify and may wish to permanently mark the donation as ineligible for Gift Aid; *markCancelled* transitions the *donorEligibility* to *cancelled* from *permitted* or *onHold*. Unlike a donation marked *onHold*, a donation marked *cancelled* cannot return to a *permitted* state. Once a declaration has been created for a donation then the donor can no longer postpone the creation of a declaration, but may still cancel the donation. Figure 9 illustrates the possible changes to the *donorEligibility*.

In $M_0$ the *markCancelled* event has the effect of either precluding a declaration or claim from being created for that donation or forcing the declaration to be included in a subsequent overclaim. As $M_1$ introduces numerical data to account for the aggregate amount donated and total number of donations made by each donor in a year, *markCancelled* also has the effect of reducing these amounts. No special attention to cancelling donations/declarations is required within the second refinement $M_2$, which introduces finer grained notion of time, as cancellation can occur at any time.

### $M_0$—*Core functionality*

**markCancelled** occurs when a donor wishes to permanently mark a donation as ineligible for Gift Aid. Any donation that has not already been marked as cancelled can be marked as such via *markCancelled*; once a donation has been cancelled it cannot be marked as *onHold* or *permitted*. Cancellation precludes the creation of a declaration for an as yet undeclared donation. Cancellation shall also preclude the inclusion within a claim of a declared but as yet unclaimed donation. Finally, if the declared donation has already been included in a claim, the cancellation will require it to appear as an overclaim in a subsequent claim by the charity. In the model, two distinct variables are defined for *Claims* and *Overclaims*. An invariant ensures that only those declarations that have previously been claimed can be overclaimed. The *createClaim* event generates a single claim listing both the recently declared donations and the previously declared but recently cancelled donations. The event results in mapping all declarations that are not yet claimed (nor cancelled), to a new *claimId* passed by the event parameter. Similarly, all declarations of donations that have already been claimed, and have since been cancelled but not yet overclaimed, are assigned the new *claimId*.

### $M_1$—*With numerical values*

The first refinement introduces numerical values to enable the aggregation of the total donated by each donor and the counting the donations made by each donor in a tax year. $M_1$ was extended to appropriately update the two variables *donorDonationCount* and *donorDonationTotal* upon the cancellation of a donation/declaration. In $M_1$ the *markCancelled* event is extended to subtract the appropriate amount from the aggregate total when cancelling a donation/declaration and decrement the total amount of donations by one.

## 5. Reversing a cancellation

An earlier version/iteration of the design/model allowed a donor (resp. charity) to toggle the state of a donation's *donorEligibility* (resp. *charityEligibility*) ad infinitum. Due to our analysis, it was decided to limit this behaviour to avoid significantly more complexity resulting from a cascading effect on the declaration and claims processes. Allowing a donation to be cancelled at any time, and allowing a cancellation to be reversed at any time after a cancellation, gave rise to a significantly more complex system, as explained below. By identifying that this feature would require additional resources for development, testing and maintenance, we enabled Streeva to make the informed decision to preclude this feature in the first release.

As in the final version of the model, the earlier version allowed for a donation to be marked as *onHold* or *cancelled* by the donor, if it had not yet been declared (the earlier version also allowed the charity to mark a donation as *notPermitted* in case they had previously marked it *permitted* by mistake). This would preclude a declaration from being created for the donation unless it was subsequently marked permitted again. The earlier version of the model also allowed a donation to be marked *onHold* or *cancelled*, if it had been declared but not yet claimed. In this case the donation was marked accordingly, but the declaration also needed to be marked as having been cancelled to ensure that the declared donation was not included in a subsequent claim. This could also be reversed leading to a claim finally being made (if it was not cancelled again in the meantime). Finally, as in the final version, the earlier version allowed for a claimed donation to be *cancelled* leading to the declaration to be included in a subsequent overclaim. However, the ability to cancel and reverse the cancellation of a declared donation ad infinitum raised questions on how this should be recorded and communicated. Should a new declaration (with a new declaration ID) be created each time, or should a declaration previously marked cancelled have the *cancelled* marking removed? The latter may lead to the same declaration appearing in two separate claims both claiming Gift Aid for the same donation (although an overclaim should have refunded the first of these). While Streeva could have incorporated the feature to toggle between cancelling a donation/declaration and reversing a cancellation (and may do so in the future) the added complexity may have slowed early delivery of working software. It would have also complicated the process of creating Gift Aid claims, raising questions over which declarations are included in which claims/overclaims and why they appear multiple times.

From the Event-B model, it became apparent that allowing what seemed (superficially at least) a simple feature of toggling between *onHold* and *permitted* (resp. *notWarranted* and *warranted*) required additional state information to be recorded to maintain a growing history of a donation's various eligibility, declarations, claims, cancellations and reversals. The Event-B modelling surfaced the additional overhead that would be incurred in managing this added complexity and enabled Streeva to judge that the business case for this was not strong enough; it is expected that donors/charities will mistakenly cancel a donation only very infrequently. Disallowing cancellations to be reversed is a conservative design choice, ensuring that Swiftaid adheres to the legislation. It does not preclude the standard manual process of a donor providing a Gift Aid declaration for a donation that was mistakenly/prematurely cancelled within the automated Swiftaid system. In contrast, not allowing for cancellations at all would not have been compliant with the legislation.

The added complexity was identified in the initial machine, enabling the business decision to be made at a very early stage of the software development life cycle. Using Rodin and ProB, we interactively stepped through traces of the model with Streeva, to appreciate the cascading effects of toggling a donation as cancelled at various stages (i.e., as yet undeclared, declared but as yet unclaimed, and claimed). In the final model/design, the state transitions of *DonorEligibility* were limited to those illustrated in Fig. 9. Guards of *markOnHold* and *markPermitted* preclude either event from occurring once a donation is marked cancelled. To put a donation on hold (*markOnHold*), the donation must be permitted and not yet declared. To mark a donation permitted again (*markPermitted*) it must be *onHold* and not yet declared.

Precluding a donor/charity from reversing a cancellation was the most significant change to the design resulting from our Event-B modelling and analysis. However, other edge cases were identified by this approach that were less consequential to the system design but significant nonetheless. The formal modelling also helped identify gaps or ambiguities in the specification of Gherkin scenarios helping to ensure that they were sufficiently robust prior to their use in driving the development effort. The primary contribution of the formal modelling to the software development was the development amongst all stakeholders of a strong understanding of the inherent behaviour of the system design enabling its complexity to be appreciated, leading to proposals of how this could be mitigated.

## 6. Related work

Behaviour-Driven Development (BDD) is an agile software development approach which focuses on analysis and testing of software specifications (i.e., an expected system behaviour). Since BDD was proposed by North [Nor06] to support test-driven development in the software testing phase, BDD has broadened to addresses to enhance software development in all life cycle stages [Sma15]. However, most of BDD-based works appear to focus only on supporting the implementation and testing phases in a software project neglecting planning and analysis phases [SW11]. Few attempts begin to address Behaviour-Driven Development in specification analysis to bridge the gap between stakeholders and domain experts and also to augment system requirements to be highly matched the desired system behaviour required by stakeholders. Snook et al. [SHD+18] take the advantage of the BDD approach to apply it specifically on the process of formal model development and validation for a given system using Event-B. Their proposal attempts to enable the modeller to provide a robust and rigorous formal model of the system (i.e., precise and reliable specifications) along with a set of desired test traces that will be used in integration and testing stage. In [SdSS17], the authors propose to involve the Event-B formal modelling tool in the BDD approach specification analysis using a structured natural language based on the Semantics of Business Vocabulary and Business Rules.

[Car17] enumerates mismatches between agile and formal software development using the Agile manifesto [Man19a] as a framework. Whereas the agile manifesto promotes individuals and interactions over processes and tools, formal methods appear to shift focus from people to tools. The agile manifesto also advocates working software over comprehensive documentation; formal specifications may be viewed as excessive documentation impeding production of working software. The agile manifesto promotes responding to change over following a plan; modelling the entire system before development delays the delivery of early milestones. Despite these fundamental differences BHive authors believe a formal model integrated with the agile approach (namely (classical) B and Behaviour-Driven Development) helps to resolve some of these deeply entrenched issues.

## 7. Discussion and future work

Significant benefits were realised by adopting Event-B modelling within an agile software development process. An early iteration of the modelling identified significant complexity in terms of the additional state information that would have been required in order to exhibit the specified behaviour. This gave an early indication of otherwise unanticipated development and maintenance overheads that were the result of providing convenient but non-essential features for donors and charities. The features in question concerned: (i) a donor's ability to create a new declaration for a donation after having cancelled a previous declaration for the same donation and (ii) a charity's ability to unwarrant a donation that they had previously warranted. The decision was made (by Streeva's senior management team, in conjunction with the software development team) to remove these features in the initial Swiftaid release. This represented a significant contribution of the formal development approach to the overall system design.

Additionally, the formal development approach identified a number of interesting edge cases that were not otherwise addressed in legislation/requirements that have informed the additional specification to manage such cases should they arise. For example, the model identified the interesting case in which the one day embargo period, which delays donations from being immediately declared to allow for donors to possibly mark them on *onHold* or *cancelled* prior to declaration, is due to end after the donor intermediary authorisation has expired. This enabled Streeva to further consider and specify how such a case should be handled by the system. The model has been used to identify notable traces of expected behaviour through the system (and simulation thereof) to validate the requirements and define future integration test plans.

Our formal approach proved useful in identifying and validating necessary assumptions required to address ambiguities in the legislation and aided direct communication with HMRC. The development process documented within this paper has helped to ensure adherence of the smart contracts with tax law, which adds trust to the system, removes reliance on audit and is a step towards real-time compliance. A formal modelling approach, backed by a shared ledger and run-time verification, may enable real-time assurance of the system. As the captured requirements are human-readable and the interactive model provides clarity of the combined effects of scenarios, it provides an economical road to an HMRC approved solution.

The Gherkin scenarios were developed by Michael and Chorley, who lead the Streeva development team. Darwish, Dupressoir and Williams, from the academic modelling team, also participated directly in the

specification of the Gherkin scenarios. HMRC were consulted to clarify aspects of the legislation, but in retrospect we would have had their direct participation in the specification of the Gherkin scenarios, which would have further accelerated this process. Involving all key stakeholders when interpreting and capturing requirements specified in legislation helps avoid misinterpretation and mitigates the impact of making misplaced assumptions.

The whole modelling in Event-B model took around 45 working days, with the model passing through a series of modifications as the assumptions changed. The main modeller (Darwish) is a qualified security expert but did not have previous experience with B or Event-B. Darwish was supported by Schneider and Williams with 25 years and 7 years respective experience in formal methods and reasoning in B and Event-B. The difficult decisions in the modelling centred on precisely capturing the legislated requirements and the explication of inherent assumptions.

Following the formal modelling and reasoning, the Streeva development team used GoDog, a Cucumber framework for golang, for acceptance testing. GoDog generated boilerplate code from the Gherkin requirements, which was then populated with gRPC code to call an instance running on Hyperledger Fabric.

Swiftaid is a step towards realising UK Government aim to make it easier for individuals and businesses to get their tax right and keep on top of their affairs under the banner 'Making Tax Digital'. Future work shall aim to address key issues in the current method of VAT collection, which accounts for 35% of the tax gap [HMR18c], i.e., the difference between the tax due and tax paid to HMRC. Unpaid VAT places compliant organisations at an unfair disadvantage and deprives the Exchequer of monies needed to effectively fund vital public services. The failure of overseas sellers to charge VAT on online sales is estimated to have cost the UK £1.5 billion in 2016 alone, which a digital solution to Split Payments could address [HMR18a].

Within the Swiftaid project we established a development process to capture/express legislated requirements using Gherkin and systematically develop formal models of the system design using Event-B. Our formal analysis identified unnecessary complexity during system design helping us avoid undue implementation, testing and maintenance costs. This process may be improved by further integrating the formal modelling by automating systematic aspects of the modelling and generating interactive visualisations to enable broader stakeholder engagement with the models produced. In particular, we aim to create interactive visualisations of our Event-B models using BMotionWeb, an extension of ProB for rapid creation of formal prototypes [LL16], or its successor VisB [WL20]. Further extensions of this work would be to utilise alternative means of capturing requirements and specifying behaviour that aids communication amongst all stakeholders while embedding Event-B modelling within a Behaviour-Driven Development, including Linear Temporal Logic (LTL) [HSTW16], and the iUML-B diagrammatic front-end for Event-B [SHD+18].

The adoption of formal methods within an agile software development process helped avoid misinterpretation of requirements and supported close collaboration between all stakeholders. Stepping through sequences of events with the models, via interactive Rodin and ProB simulations, enabled assumptions to be validated. Furthermore, by annotating each scenario with the precise items of legislation (regulation) whose behaviour the scenario specified and, similarly, by annotating each Event-B event with the scenarios that they modelled, we established a useful basis for understanding and appreciating inherent consequences arising from requirements change. We are able to efficiently evaluate the full impact of changes in requirements/design.

By allying agile and formal software development processes, we seek to make commercial use of formal methods mainstream. Where formal methods are already applied, (e.g., defence, aerospace, transport), making them more effective and efficient will enable further advances in academic research that supports these sectors. Moreover, the allying agile and formal software will enable exploitation of formal methods in new domains enabling rapid development of robust software across a broader set of industry sectors and academic disciplines. Sectors that immediately stand to benefit are those that would value streamlined payments or supply-chains, e.g., finance, retail, manufacture and construction.

## Acknowledgements

This work was funded by UKRI through the Innovate UK Swiftaid Project (Grant Number 133294). We thank Chris Chorley and Dr. François Dupressoir for contributing to the specification of Gherkin scenarios and Event-B modelling and we thank Dr. Benjamin Aziz and Dr. Philip Godsiff for insightful comments on

an early draft of this paper. We also thank the anonymous reviewers for providing their essential critical insight.



## References

[ABH+10] Abrial J-R, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. Int J Softw Tools Technol Transf 12(6):447–466

[Abr96] Abrial J-R (1996) The B-book: assigning programs to meanings. Cambridge University Press, Cambridge

[Abr10] Abrial J-R (2010) Modeling in EventB: system and software engineering. Cambridge University Press, Cambridge

[Beh19] Behave: behavior-driven development, python style. https://github.com/behave/behave. Accessed 24 May 2019

[Car17] Carter J (2017) BHive: behaviour-driven development meets B-method. Ph.D. thesis, The University of Guelph

[CC17] Common criteria for information technology security evaluation. V3.1r5. Technical report, Common Criteria (2017)

[Cuc19a] Cucumber: 10 minute tutorial. https://cucumber.io/docs/guides/10-minute-tutorial/. Accessed 24 May 2019

[Cuc19b] Cucumber: a tool that supports behaviour-driven development. https://github.com/cucumber/cucumber. Accessed 24 May 2019

[FLDL+13] Falampin J, Le-Dang H, Leuschel M, Mokrani M, Plagge D (2013) Improving railway data validation with ProB, pp 27–43. Springer, Berlin

[Ghe19] Gherkin Reference: Cucumber. Accessed 24 May 2019

[GJ13] Gmehlich R, Jones C (2013) Experience of deployment in the automotive industry, pp 13–26. Springer, Berlin

[HLP13] Hallerstede S, Leuschel M, Plagge D (2013) Validation of formal models by refinement animation. Sci Comput Program 78(3):272–292

[HMR16] HMRC (2016) The donations to charity (gift aid declarations) regulations 2016. Statutory instruments

[HMR18a] HMRC (2018a) Alternative method of VAT collection—split payment. Summary of responses

[HMR18b] HMR (2018b) Charitable giving and gift aid. HMRC research report 482

[HMR18c] HMRC (2018c) Measuring tax gaps 2018 edition. An official statistics release

[Hoa69] Hoare CAR (1969) An axiomatic basis for computer programming. Commun. ACM 12(10):576–580

[HSL16] Hansen D, Schneider D, Leuschel M (2016) Using B and ProB for data validation projects. In: Proceedings ABZ 2016, pp 167–182. Springer International Publishing

[HSTW16] Hoang TS, Schneider S, Treharne H, Williams DM (2016) Foundations for using linear temporal logic in Event-B refinement. Formal Asp Comput 28(6):909–935

[ILL+13] Ilić D, Laibinis L, Latvala T, Troubitsyna E, Varpaaniemi K (2013) Deployment in the space sector, pp 45–62. Springer, Berlin

[jBe19] JBehave: a framework for behaviour-driven development. https://jbehave.org/. Accessed 24 May 2019

[LB03] Leuschel M, Butler M (2003) ProB: A model checker for B. In: FME 2003: formal methods, pp 855–874. Springer, Berlin

[LFW10] Larsen PG, Fitzgerald JS, Wolff S (2010) Are formal methods ready for agility? a reality check. Technical report no. CS-TR-1218, Newcastle University

[LL16] Ladenberger L, Leuschel M (2016) BMotionWeb: A tool for rapid creation of formal prototypes. In: Software engineering and formal methods—14th international conference, SEFM 2016, Held as part of STAF 2016, Vienna, Austria, July 4–8, 2016, Proceedings, pp 403–417

[Man19a] Manifesto for agile software development. https://agilemanifesto.org/. Accessed 24 May 2019

[Man19b] Principles behind the agile manifesto. https://agilemanifesto.org/principles.html. Accessed 24 May 2019

[Nor06] North D (2006) Introducing behaviour-driven development (BDD). Better Software

[RT13] Romanovsky A, Thomas M (2013) Industrial deployment of system engineering methods providing high dependability and productivity. Springer, Berlin

[SdSS17] Siqueira FL, de Sousa TC, Silva PSM (2017) Using BDD and SBVR to refine business goals into an Event-B model: a research idea. In: 2017 IEEE/ACM 5th international FME workshop on formal methods in software engineering (FormaliSE), pp 31–36. IEEE

[SHD+18] Snook C, Hoang TS, Dghyam D, Butler M, Fischer T, Schlick R, Wang K (2018) Behaviour-driven formal model development. In: Formal methods and software engineering, pp 21–36. Springer

[Sma15] Smart JF (2015) BDD in action: behavior-driven development for the whole software lifecycle. Manning
[Str19] Streeva. Swiftaid. https://swiftaid.co.uk/. Accessed 09 July 2019
[STW14] Schneider S, Treharne H, Wehrheim H (2014) The behavioural semantics of Event-B refinement. Formal Asp Comput 26(2):251–280
[SW11] Solis C, Wang X (2011) A study of the characteristics of behaviour driven development. In: 2011 37th EUROMICRO conference on software engineering and advanced applications, pp 383–387. IEEE
[UKR19] UKRI gateway to research: Swift Aid project reference 133294. https://gtr.ukri.org/projects?ref=133294. Accessed 24 May 2019
[WD96] Woodcock J, Davies J (1996) Using Z: specification, refinement, and proof. Prentice Hall, Upper Saddle River
[WDSM20] Williams DM, Darwish S, Schneider S, Michael DR (2020) Swiftaid rodin event-b models. Zonodo https://10.5281/zenodo.3715494
[WKW+13] Wieczorek S, Kozyura V, Wei W, Roth A, Stefanescu A (2013) Business information sector, pp 63–79. Springer, Berlin
[WL20] Werth M, Leuschel M (2020) VisB: A lightweight tool to visualize formal models with SVG graphics. In: Proceedings ABZ 2020, LNCS

*Received 16 October 2019*
*Accepted in revised form 9 April 2020 by Michael Butler*