Agile Modelling with Formal Methods

Richard Ashworth

Exeter College
University of Oxford

A dissertation submitted for the
MSc in Software Engineering

Abstract

This dissertation has two aims:

1. To demonstrate that formal methods can add value in modelling business domains not traditionally associated with them.

2. To assess which aspects of Agile development realise the most benefit from formal modelling, and discuss how the associated techniques can be best applied.

A case study in the domain of Wholesale Credit Risk was conducted to tackle the first of these aims. Three successive models for Basel II default management were constructed using the Z notation, and then analysed using a range of tools and techniques. A discussion of the value that these models provide—and their effectiveness in an Agile development environment—follows, addressing the second aim.

Contents

1 Introduction

All software projects have a common goal: to deliver systems that accurately reflect their requirements, while minimising the time and cost needed to develop them. One obstacle that teams face in achieving this is the presence of defects, whether they originate from a system's requirements, architecture, or implementation. The cost of addressing these defects is affected by the phase of the development cycle in which they are introduced, and the amount of time before they are discovered:

> "The general principle is to find an error as close as possible to the time at which it was introduced. The longer the defect stays in the software food chain, the more damage it causes further down the chain." [McC04]

Providing the tools and practices to address defects as early as possible in the development life cycle is a key aspect of both Formal and Agile methodologies, although their approaches contrast sharply. Formal methods emphasise accuracy in defining a system's specification, and mathematical rigour in refining this into executable software. In their traditional application, formal methods are well-suited to a waterfall style of project management, in which requirements and design activities are performed before the implementation of a system begins. Only once the correctness and consistency of a set of requirements has been established may the mechanical process of turning these into software begin.

A number of high-profile projects have been delivered successfully using formal methods in this manner. For example, the Z notation was used in restructuring the CICS transaction processing system [HF87], and in verifying the security properties of a Smartcard product for eCommerce [SCW00]. For the development of critical systems, formal methods are a good fit: reasoning about the properties of a system before development begins means that issues with its requirements or design can be addressed early, and confidence can be achieved that the delivered system behaves exactly as specified. When lives are at stake, or security breaches could have disastrous consequences, the time and cost required to deliver such a system may be considered secondary to the correctness of its implementation. The overhead in constructing such a precise specification and performing proofs through each phase of refinement can therefore be justified in these cases.

For the majority of software projects, however, the constraints under which they must be delivered are different: time and cost may determine whether building a system is even viable, and the scope of a system's requirements may change during development to meet a particular deadline or budget. Agile methods are designed to support teams working in these conditions; changing requirements are a feature of most projects, and the Agile approach embraces this reality.

The first principle of the Agile Manifesto states that

> "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." [HF01]

To support this principle, Agile methods such as *Scrum* suggest that development is performed in short, time-boxed iterations [SB02]. This provides a feedback loop, giving developers and customers a regular opportunity to collaborate in refining and adjusting a system's requirements. Requirements themselves are often captured in the form of *user stories*, which provide a high-level overview of a proposed feature from the perspective of a particular user. Alistair Cockburn argues that these user stories should be viewed as promissory notes for a future conversation [Coc07], and it is through these conversations that the requirements for a system emerge, rather than being finalised up front.

One of the most difficult challenges for any development team is to ensure that they are building the *right* system, and it is only by working closely with their customers that this can be achieved. The quality of these conversations is therefore paramount to the success of a project, and the necessity of this collaboration is recognised as another principle in the Agile Manifesto:

> "Business people and developers must work together daily throughout the project."
> [HF01]

One activity that can increase the effectiveness of these conversations is sharing models of a system throughout its development. Modelling a system's problem domain provides a common language for development teams to hold these conversations effectively with their stakeholders. The models themselves may be expressed at different levels of abstraction, and constructed from a variety of mediums. For example, if a particular business flow is being analysed, then activity diagrams or *Business Process Modelling Notation* (BPMN) may be used to illustrate how different events and interactions will be sequenced in the system. More technical discussions around an implementation may be aided with *Unified Modelling Language* (UML), perhaps in the form of class or sequence diagrams that describe the mechanics of a particular solution.

While these techniques undoubtedly add value to the development process, what is missing in this approach is an overview of the *properties* of a system that can be referred to throughout the development life cycle. This overview is useful not only in understanding and clarifying the requirements of a system, but also in defining suitable test cases and writing documentation. Recent research efforts [BSK10, Liu10] propose the integration of modelling techniques from formal methods with the iterative and customer-centric approach of Agile development to address this deficiency. How effectively this integration can be achieved is the subject of this dissertation.

Although a plethora of different formal methods have been developed, they all share a common theme: the use of discrete mathematics to structure information about a system. This mathematical foundation furnishes users of formal methods with a language to reason about different aspects of their systems' behaviour at an appropriate level of abstraction. We will use the Z notation in this report to investigate how formal methods can be leveraged in an Agile enterprise environment. As a formal method, the Z notation has four distinct aspects [WD96]. The first of these concerns the notation itself and an accompanying 'mathematical tool-kit' that facilitates analysis on a system's specification—these elements of Z are built on the results of set theory and a first-order predicate calculus respectively. The second aspect is a schema language that can be used to capture a system's *state* as different operations are performed. This renders the Z notation particularly suitable for modelling large systems, as common components can be encapsulated in schemata and reused. The third aspect relates to the way that natural language is used to complement the mathematical elements of a specification. Although easily overlooked, this aspect is an essential component of Z: a well-written narrative provides the link between a model's formal notation and its associated business domain. The fourth aspect comprising the Z method is *refinement*. This provides a means for transforming an abstract specification into a concrete system, while ensuring that certain properties are preserved.

These attributes of Z make it a good candidate for evaluating the use of formal methods in a novel scenario. Built upon a general-purpose and accessible notation, Z can be learnt quickly by new team members, and can be used to model any state-based system naturally. Additionally, a range of tools include support for the Z notation, enabling type-checking, animation, theorem proving and model checking to be automated during a specification's analysis. This provides us with an opportunity to assess the use of different formal techniques within this report. To support our investigation, we will consider a modelling scenario from the industry of Wholesale Credit Risk: Basel II default data, a business domain in which projects must be delivered to strict regulatory deadlines, while ensuring that the legal frameworks governing their requirements are satisfied. Failure to comply with these constraints can result in serious reputational risk[1], and breaches carry severe financial penalties.

We shall begin with a summary of the business domain underlying this scenario, and consider some of the particular challenges faced by its constituent organisations and their systems. Three successive models of the requirements for systems managing Basel II defaults are then constructed using the Z notation. The value that these models contribute to different aspects of the software life cycle is evaluated, together with a discussion of how well suited they are to an Agile development environment. To guide this discussion, we will show how an initial model can be extended to support new or changing requirements, and adapted to facilitate conversations with stakeholders at different levels of abstraction. An account of how the models can be applied to various aspects of development is presented in Chapter 4, illustrating how animation, formal reasoning, model checking and prototyping can be used to amplify and sustain the dividends gained from the initial modelling activities.

Following this analysis, we consider ultimately whether the tools and techniques currently at our disposal are sufficient to support the integration of formal models in Agile methods, what limitations exist, and how future research might address these. It is hoped that the work presented in this report will encourage other teams faced with similar challenges to experiment further with the techniques discussed, and expose new synergies between the two disciplines.

[1]See [BBC14] for a real-world example.

2 Modelling Scenario

The business of Wholesale Credit Risk is an extremely complex field, involving many different facets and participants. As an industry, it has been subject to a great deal of scrutiny, particularly in the wake of the recent financial crisis. The failure of major institutions to successfully manage their exposure and risk appetite has had a profound impact upon thousands of people and businesses [The14], and rectifying this remains a key deliverable for those concerned. The Basel Accords [bas88, bas06, bas10] exist to encourage banks to mitigate these risks, and the second of these forms the business domain of our modelling scenario.

The Basel II legislation serves two purposes: to dictate the amount of regulatory capital a bank must hold in relation to its counterparty credit risk exposure, and to establish a culture of risk management and accountability through all levels of an organisation. A key aspect of this concerns the appropriate handling of counterparty default data. Under the Basel II definition[1], a *default* on a credit facility (such as a loan) is considered to have occurred if any of the following events take place:

- The bank considers that the obligor is unlikely to pay its credit obligations to the banking group in full, without recourse by the bank to actions such as realising security (if held).
- The obligor is past due more than ninety days on any material credit obligation to the banking group.

This definition is important because banks are required by law to report on these events:

> “456. A bank must record actual defaults on IRB exposure classes using this reference definition.” [bas06]

When a counterparty is deemed to have performed such an event, details of the surrounding circumstances are recorded in a document called a ‘default instance’. On receipt of the first such document for a particular counterparty, a ‘default case’ is opened against the obligor, which is effectively an inventory of the associated default instances. Once this has happened, the bank must ensure that no further lending is sanctioned to the counterparty. To verify that default instances are well-founded, a two-person approval process is enforced; default instances are not considered material until they been approved.

Once put into default, parties may take action to improve their credit strength. Where they are successful in doing so, their associated default instances may be considered no longer relevant. To handle these situations, default instances can be closed as well as opened, a process which is again subject to ratification. Once all of the default instances within a default case have been closed, the party is considered to be no longer in default, and the bank is then free to reopen any pre-existing lines of credit.

[1]See [bas06], section 452

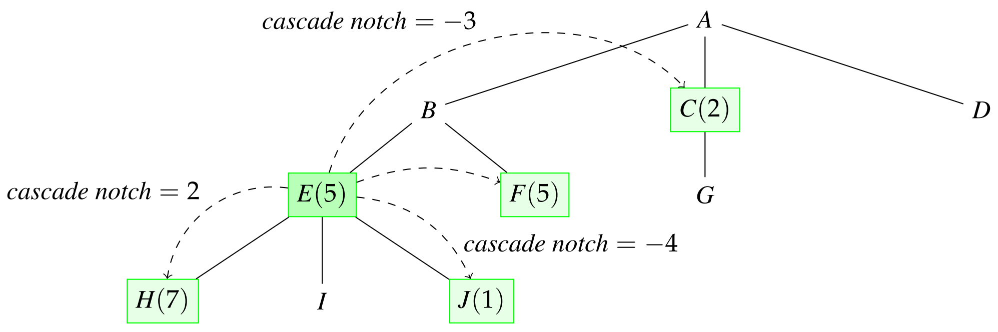


Figure 2.1: Cascading a credit grade

Up to this point, we have only considered default instances that are raised manually on the decision of Credit Officers. Notification of default events may also be received from other sources and systems, however. For certain counterparties, manual approval is required before default cases can be created, and so a suitable mechanism for handling default event triggers from other systems is therefore required. When a default event trigger is received for a counterparty that is not already in default, a special type of default instance—known as a 'deferred default'—is raised to acknowledge this, although no default case is created at this stage.

The presence of deferred defaults serves to alert users that opening a default case may be required, and if putting the party into default is indeed considered appropriate, a 'synthetic' default instance can be raised. Although synthetic defaults do not relate directly to any particular default event, they cannot be approved (and thus cause the party to default) unless at least one deferred default has already been received for the counterparty. When a synthetic default is approved, all deferred instances are added to the default case as 'indirect' defaults. On their approval—and with the party now in default—any further triggers get added automatically to the case, without the need for deferral or explicit approval.

Just as external systems send triggers to create default instances, they may also send notification for closing existing ones. On receipt of these 'closure triggers', any associated deferred defaults will be removed. Should the corresponding indirect default instances already have been promoted into the default case, they will be closed automatically with one exception: if the default instance being closed is the last material default in the case (i.e. its closure will take the party out of default), then a synthetic default should be raised immediately to enforce a review of closing the instance. Synthetic defaults can only be closed if all other default instances in the case are indirect; the presence of any direct default instances precludes their closure. If the last two remaining default instances in a case are comprised of one manual and one synthetic default, then closing the manual default will result in closure of the synthetic.

Further complexity in the business domain is introduced with the handling of 'cross default' instances. Counterparties are registered in a structure that reflects their legal hierarchy, and a default instance against one party may be propagated to others within the hierarchy, depending on their credit strength. These inherited default events are referred to as 'cross defaults'.

When a party within a legal hierarchy is graded, that grading may be cascaded to other parties within the group. An example of this is given in Figure 2.1. In this scenario, party E has been graded with a credit score of 5. Party F inherits this grade directly, perhaps because a formal guarantee has been made that party E will support them in the event of their default. Parties C, H and J are also deemed to have related credit strengths to the parent, but with circumstances that warrant an adjustment to the inherited grade: upwards by 2 notches in the case of party H, and a reduction of 3 and 4 for parties C and J respectively. Other parties in

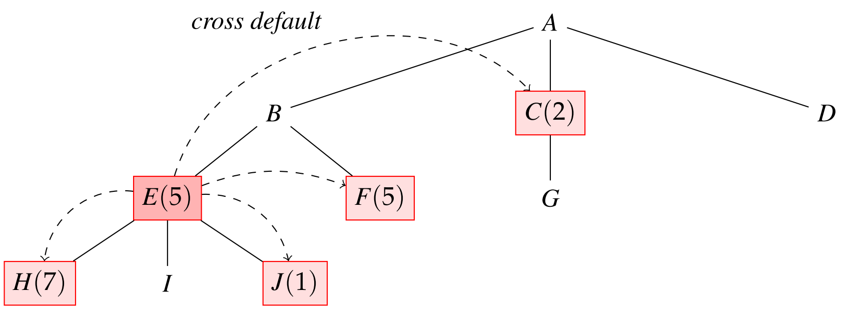


Figure 2.2: Cross defaults when the primary default is the cascade parent

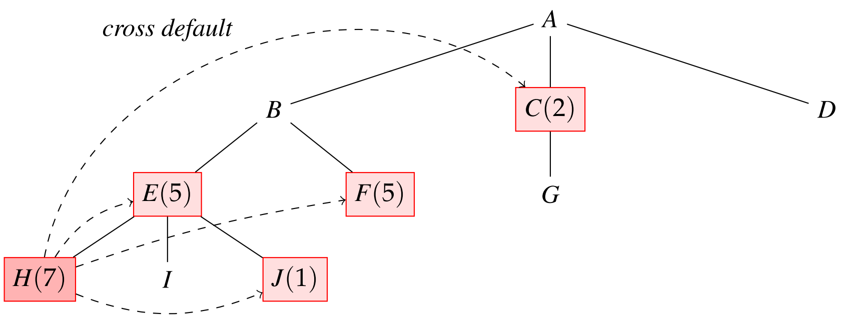


Figure 2.3: Cross defaults when the primary default is stronger than the cascade parent

the legal hierarchy are not considered to have a credit strength affected by party E (perhaps because their management is operationally independent), and will therefore need to be graded separately by a Credit Officer.

When a default event is recorded against one member of a grading hierarchy, other parties in the hierarchy may automatically default, depending on their relation to the defaulting party. If the parent of a cascade hierarchy defaults, then all children default. An example of this is illustrated in Figure 2.2. Similarly, if a cascade child with the same or better grade than the parent defaults, then the whole cascade hierarchy will also default; this is shown in Figure 2.3. The remaining scenario occurs when a cascade child with a worse grading than its parent defaults. This is shown in Figure 2.4, in which only weaker or equally-graded siblings will cross-default.

Although the preceding description of the business domain goes some way towards explaining what a system concerned with managing Basel II defaults must support, it suffices unaccompanied neither as a realistic set of requirements nor a practical specification for the system. Modelling is used to bridge the gap between a customer's account of a system's current or desired behaviour, and the understanding of those responsible for its implementation. To be successful in addressing this disconnect, the models—and techniques used in conducting their analysis—must satisfy a number of criteria.

An accurate description of existing functionality is required initially, so that we can reason about the impact of new requirements on the system, ensuring that any changes to current behaviour are expected. This will prevent development effort being wasted in delivering features that later need to be reverted as a result of their unintended consequences. The ability to explore specific use-cases allows the team to begin designing and validating high-level test cases early in the development life cycle. To facilitate this, the models must be accessible to

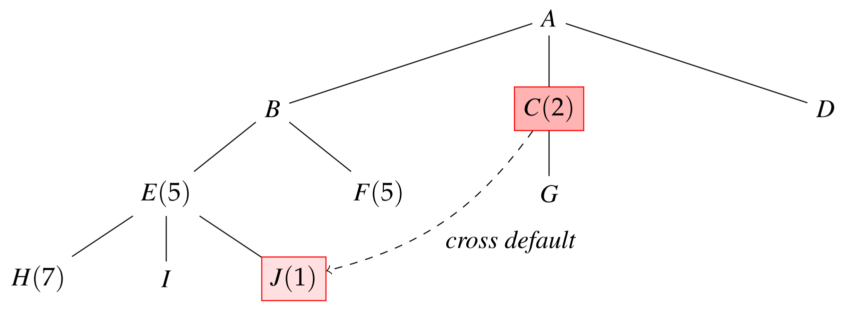


Figure 2.4: Cross defaults when the primary default is weaker than the cascade parent

all members of a team, regardless of any specialism they might have. The models should also serve to document the system's behaviour for people not already familiar with it, something that is particularly useful in an industry in which staff turnover is high, and access to expert domain knowledge limited. Finally, it is imperative that the modelling activities do not subvert the principles of collaborative and iterative development, nor add undue overhead to an environment that is driven by cost reduction and regulatory deadlines.

Having summarized the business domain, and presented some motivations and criteria for modelling it, we will proceed using this as a case study for examining how effectively formal methods can be applied in this context. We start by constructing an initial model for the proposal and approval of manual direct defaults using the Z notation, and then show how this can be extended to support the complexities introduced by deferred and cross defaults. The models themselves are presented in the next chapter, followed by an analysis of how various tools and techniques can be used to leverage their value through different phases of the system's life cycle. A more thorough discussion of the relative merits of these practices and their applicability to an Agile development environment is deferred until Chapter 5.

3 Models

3.1 Default Instances and Cases

An initial, high-level model to capture the association of default instances with obligor counterparties, and the management of their life cycle. Cascaded grading hierarchies and the handling of deferred defaults are out of scope at this stage.

3.1.1 Counterparty Defaults

As the central entity in our business domain, we begin with a definition for the *Default* type. Although in a particular implementation of the system, there may be a number of attributes pertaining to defaults, the only one relevant to a party's overall default status is the approval status of the default instance itself. We therefore define the *Default* type as follows:

$$\begin{array}{|l}\hline \textit{Default} \\ \textit{status} : \textit{ApprovalStatus} \\ \hline\end{array}$$

As a default instance is processed through the system, it will be associated with an approval status. These are enumerated with a free type:

$$\textit{ApprovalStatus} ::= \textit{PROPOSED} \mid \textit{APPROVED} \mid \textit{PROPOSED\_CLOSURE} \mid \textit{CLOSED}$$

Identifiers for the primary entities of our model are then declared, which will be used later in the bindings for the *DefaultSystem* state schema. There is no need to include a separate *Party* type here, since at this stage we are not concerned with the properties of counterparties themselves. Identifiers are sufficient to denote which defaults belong to different counterparties in our system, so we define these in our model with basic types:

$$[\textit{DefaultId}, \textit{DefaultCaseId}, \textit{PartyId}]$$

Default cases contain one or more default instances; parties for which no default events have been received do not have a default case. Our *DefaultCase* type therefore includes a non-empty set of references to particular default instances:

$$\begin{array}{|l}\hline \textit{DefaultCase} \\ \textit{defaultInstances} : \mathbb{P}_1\, \textit{DefaultId} \\ \hline\end{array}$$

Having introduced the primary entities of our model, we can now define the bindings between them in our *DefaultSystem* state schema.

$$\begin{array}{|l}\hline \textit{DefaultSystem} \\ \textit{parties} : \mathbb{P}\, \textit{PartyId} \\ \textit{defaults} : \textit{DefaultId} \pfun \textit{Default} \\ \textit{defaultCases} : \textit{DefaultCaseId} \pfun \textit{DefaultCase} \\ \textit{partyDefaultCase} : \textit{PartyId} \pinj \textit{DefaultCaseId} \\ \hline \operatorname{dom} \textit{partyDefaultCase} \subseteq \textit{parties} \\ \operatorname{ran} \textit{partyDefaultCase} \subseteq \operatorname{dom} \textit{defaultCases} \\ \hline\end{array}$$

A partial injective function is used for the *partyDefaultCase* binding, since our requirements preclude states in which multiple parties share the same default case.

3.1.2 Initialisation

The post-condition for starting up the system is expressed in the following initialisation schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{Init} \quad}} \\
\mathit{DefaultSystem}' \\
\hline
\mathit{defaults}' = \varnothing \\
\mathit{defaultCases}' = \varnothing \\
\mathit{parties}' = \varnothing \\
\mathit{partyDefaultCase}' = \varnothing \\
\hline
\end{array}$$

When the system is initialised, we assume that no counterparties or defaults have yet been registered. We therefore declare both $\mathit{parties}'$ and the $\mathit{defaults}'$ function as the empty set. Until the first party has been registered in the system, the only way to satisfy the state invariant

$$\mathrm{dom}\ \mathit{partyDefaultCase} \subseteq \mathit{parties}$$

is by declaring that no mappings exist from parties to default cases, hence $\mathit{partyDefaultCase}' = \varnothing$. Default cases that are not associated with a particular counterparty, although not explicitly forbidden in our requirements, have no bearing on the system from a Basel II perspective. We therefore complete the post-condition with the equality $\mathit{defaultCases}' = \varnothing$.

3.1.3 Counterparty On-boarding

With an initial state for the system defined, subsequent operations can be introduced in our model, beginning with the registration of new counterparties in the system:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{AddParty} \quad}} \\
\Delta \mathit{DefaultSystem} \\
\mathit{party}? : \mathit{PartyId} \\
\hline
\mathit{party}? \notin \mathit{parties} \\
\mathit{parties}' = \mathit{parties} \cup \{\mathit{party}?\} \\
\mathit{defaults}' = \mathit{defaults} \\
\mathit{defaultCases}' = \mathit{defaultCases} \\
\mathit{partyDefaultCase}' = \mathit{partyDefaultCase} \\
\hline
\end{array}$$

In practice, this operation represents a significantly more complex process, involving a number of different actions and checkpoints. We do not need to be concerned with the details of counterparty on-boarding here, however. It is sufficient in this model to provide a simple means of adding parties to the system so that they can later be associated with default events. Should we wish to model this process more precisely, perhaps to analyse some future requirement, this schema could easily be extended and refactored to support this.

3.1.4 Recording Default Events

One means by which default events can be recorded is a user manually proposing the creation of a default instance. There are two scenarios to consider: when the default event is the first to be associated with a particular counterparty, and when previous default events have already been received. We first extract common invariants for these cases into the *ProposeDefaultCommon* schema. These ensure that the default being proposed has the correct status, and does not already exist in the system.

$$\begin{array}{|l}
\textit{ProposeDefaultCommon} \\
\hline
\Delta DefaultSystem \\
party? : PartyId \\
defaultEvent? : DefaultId \times Default \\
\hline
(second\, defaultEvent?).status = PROPOSED \\
defaultEvent? \notin defaults \\
defaults' = defaults \cup \{defaultEvent?\} \\
party? \in parties \\
parties' = parties \\
\hline
\end{array}$$

When the proposed default instance is the first to be associated with a particular counterparty, a new default case is created. We include the condition that *newDefaultCase* $\notin$ *defaultCases* since parties cannot share the same default case, and add a proposed default instance directly into the case. Schema inclusion is used to define the *ProposeFirstDefault* operation, allowing elements from the *ProposeDefaultCommon* schema to be referenced and extended:

$$\begin{array}{|l}
\textit{ProposeFirstDefault} \\
\hline
ProposeDefaultCommon \\
newDefaultCase : DefaultCaseId \times DefaultCase \\
\hline
party? \notin \operatorname{dom} partyDefaultCase \\
newDefaultCase \notin defaultCases \\
(second\, newDefaultCase).defaultInstances = \{first\, defaultEvent?\} \\
defaultCases' = defaultCases \cup \{newDefaultCase\} \\
partyDefaultCase' = partyDefaultCase \cup \{party? \mapsto first\, newDefaultCase\} \\
\hline
\end{array}$$

Should a default case already have been created, further default instances will accumulate directly in the party's default case. This is captured in the *ProposeAdditionalDefault* schema:

$$\begin{array}{|l}
\textit{ProposeAdditionalDefault} \\
\hline
ProposeDefaultCommon \\
existingDefaultCase : DefaultCase \\
augmentedDefaultCase : DefaultCase \\
\hline
party? \in \operatorname{dom} partyDefaultCase \\
existingDefaultCase = defaultCases(partyDefaultCase\, party?) \\
augmentedDefaultCase.defaultInstances = \\
\qquad existingDefaultCase.defaultInstances \cup \{first\, defaultEvent?\} \\
defaultCases' = defaultCases \oplus \{partyDefaultCase\, party? \mapsto augmentedDefaultCase\} \\
partyDefaultCase' = partyDefaultCase \\
\hline
\end{array}$$

Having defined schemas for both of the scenarios in which default instances can be proposed, their disjunction forms a total operation, describing the system's change in state on proposal of a default:

$$ProposeDefault \mathrel{\widehat{=}} ProposeFirstDefault \lor ProposeAdditionalDefault$$

3.1.5 Reviewing Default Proposals

A key requirement of the system is that any manually-entered default instances must be reviewed before they are allowed to affect the overall status of a counterparty. To support this,

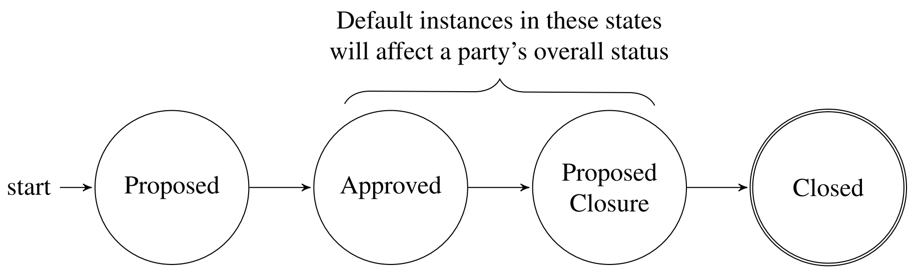

Default instances in these states
will affect a party's overall status
start
Proposed
Approved
Proposed Closure
Closed

Figure 3.1: Life cycle of a default instance

an approval process is followed, such that when the addition or closure of a default instance is proposed, another user must then approve or reject that proposal. Figure 3.1 describes the transitions followed by the status of a default instance through its life cycle. Proposed and closed instances do not have a material impact on a party's overall default status. We model this series of transitions as a sequence of the *ApprovalStatus* free type:

$$\mathit{DefaultLifeCycle} == \langle \mathit{PROPOSED}, \mathit{APPROVED}, \mathit{PROPOSED\_CLOSURE}, \mathit{CLOSED} \rangle$$

To ensure that the four-eye checks are appropriately enforced, we insist that all changes to a default instance pass through an intermediary 'proposed' state. Successive approvals for the opening and closing of default instances are precluded with the *ReviewEligible* predicate:

$$\mathit{ReviewEligible} \mathrel{\widehat{=}} [\mathit{Default} \mid 1 = \mathit{DefaultLifeCycle}^{\sim}(\mathit{status}) \bmod 2]$$

The use of modular arithmetic here means that reviews are effectively performed on every other transition of a default instance's status. This avoids the need for us to refer to particular states in our description of the eligibility criteria, yielding a more robust specification should new states be added to the default life cycle.

Default approvals can now be expressed in our model by advancing the status of a default instance through this sequence. The *ApproveTransitionLocal* schema describes the effect of performing this operation on the *Default* type:

```
ApproveTransitionLocal
  $\Delta \mathit{Default}$
  $\mathit{ReviewEligible}$
  ----
  $\mathit{status}' = \mathit{DefaultLifeCycle}(1 + \mathit{DefaultLifeCycle}^{\sim}(\mathit{status}))$
```

Since this schema only describes change in terms of the local *Default* state, we use the following promotion schema to link such operations with their effect on the global state:

```
Promote
  $\Delta \mathit{DefaultSystem}$
  $\Delta \mathit{Default}$
  $d? : \mathit{DefaultId}$
  ----
  $d? \in \operatorname{dom} \mathit{defaults}$
  $\mathit{defaults}' = \mathit{defaults} \oplus \{d? \mapsto \theta \mathit{Default}'\}$
  $\mathit{defaultCases}' = \mathit{defaultCases}$
  $\mathit{parties}' = \mathit{parties}$
  $\mathit{partyDefaultCase}' = \mathit{partyDefaultCase}$
  $\mathit{defaults}\, d? = \theta \mathit{Default}$
  $\mathit{defaults}'\, d? = \theta \mathit{Default}'$
```

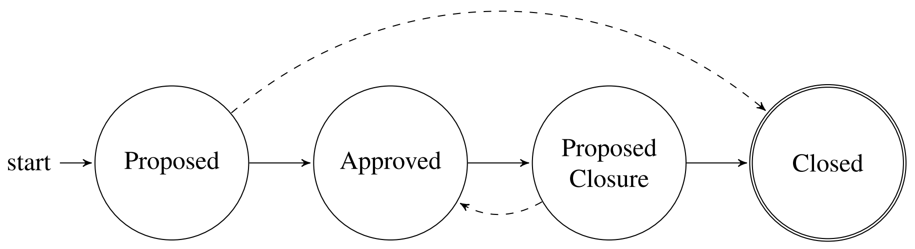


Figure 3.2: Revised life cycle of a default instance

Using a promotion schema adds flexibility to our model, allowing further operations on the *DefaultSystem* state schema to be added simply by providing suitable descriptions of changes to the local state. In this instance, the promotion is *free*: whatever the local state after promotion, a corresponding global state exists. More formally, we can write

$$(\exists\, Default' \bullet \exists\, DefaultSystem' \bullet Promote) \Rightarrow (\forall\, Default' \bullet \exists\, DefaultSystem' \bullet Promote)$$

The preceding schemas are then brought together in the *ApproveDefaultTransition* operation, modelling the approval of a proposed change to the default instance's status:

$$ApproveDefaultTransition \mathrel{\widehat{=}} \exists\, \Delta Default \bullet Promote \land ApproveTransitionLocal$$

Conversely, the user performing the review may decide that raising a default instance is not warranted on the evidence surrounding the default event, and will instead reject the proposed transition. Our current model for the life cycle of a default instance is inadequate to support this operation, so we extend this in the *DefaultLifeCycle′* type, where transitions can occur by advancing both forwards and backwards in the sequence. The resulting life cycle, in which proposals can now be rejected, is described in Figure 3.2.

$$DefaultLifeCycle' == \langle CLOSED \rangle \frown DefaultLifeCycle$$

The promotion schema defined earlier is reused here, with changes to the underlying default instances being described in the *RejectTransitionLocal* schema. The proposed transition of a default's status can only be rejected when it is eligible for review; as before, we include the *ReviewEligible* predicate in our local operation schema to enforce this.

RejectTransitionLocal

$\Delta Default$
$ReviewEligible$

$status' = DefaultLifeCycle'(DefaultLifeCycle'^{\sim}(status) - 1)$

$$RejectDefaultTransition \mathrel{\widehat{=}} \exists\, \Delta Default \bullet Promote \land RejectTransitionLocal$$

3.1.6 Closing Default Instances

When a counterparty has successfully resolved the issues surrounding its default, a Credit Officer may propose closure of the associated default instances. The *ProposeCloseDefault* operation, which again leverages the promotion schema defined previously, is used in our model to facilitate this action.

ProposeCloseDefaultLocal

$\Delta Default$

$status = APPROVED$
$status' = PROPOSED\_CLOSURE$

$$ProposeCloseDefault \mathrel{\widehat{=}} \exists\, \Delta Default \bullet Promote \land ProposeCloseDefaultLocal$$

3.1.7 Defaulted Counterparties

With a state schema and the primary operations now defined in our model, all that remains is to provide a means of identifying those counterparties that are in default for any given state of the system. Parties are considered to be in default when their associated default case contains at least one instance that has been approved, or whose closure is proposed. These default states are recognised in the *MaterialDefaultStatus* set:

$$\mathit{MaterialDefaultStatus} == \{\mathit{APPROVED}, \mathit{PROPOSED\_CLOSURE}\}$$

For any given state of the *DefaultSystem* schema, we can identify the corresponding set of defaulted counterparties. Where the system contains at least one default case, this is given by the following schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ShowDefaultedParties0}} \\
\Xi \mathit{DefaultSystem} \\
\mathit{defaultedParties}! : \mathbb{P}\, \mathit{PartyId} \\
\hline
\mathit{defaultedParties}! \subseteq \mathit{parties} \\
\forall p : \mathit{parties} \setminus \mathrm{dom}\, \mathit{partyDefaultCase} \bullet p \notin \mathit{defaultedParties}! \\
\forall p : \mathrm{dom}\, \mathit{partyDefaultCase} \bullet p \in \mathit{defaultedParties}! \Leftrightarrow \\
\quad (\exists d : (\mathit{defaultCases}(\mathit{partyDefaultCase}\, p)).\mathit{defaultInstances} \bullet \\
\qquad (\mathit{defaults}\, d).\mathit{status} \in \mathit{MaterialDefaultStatus}) \\
\hline
\end{array}$$

If a particular state does not contain any default cases, then none of its constituent parties should be considered in default:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ShowDefaultedPartiesNoneExist}} \\
\Xi \mathit{DefaultSystem} \\
\mathit{defaultedParties}! : \mathbb{P}\, \mathit{PartyId} \\
\hline
\mathit{defaultedParties}! = \varnothing \\
\mathrm{dom}\, \mathit{partyDefaultCase} = \varnothing \\
\hline
\end{array}$$

We combine these scenarios as a disjunction, declaring the *ShowDefaultedParties* schema as a total operation:

$$\mathit{ShowDefaultedParties} \mathrel{\widehat=} \mathit{ShowDefaultedParties0} \lor \mathit{ShowDefaultedPartiesNoneExist}$$

The specification we have presented in this section leaves us with a working model for the current behaviour of the system. In the next section, we build on these definitions, turning our attention to modelling future requirements.

3.2 Deferred Defaults

The requirement for handling deferred defaults is presented at a high level as follows:

> As the organisation's primary broker of Basel II defaults, I want to postpone the processing of default cases, and stop last default closures from closing the default case, so that the defaulted status of certain counterparties does not change automatically.

To model the changes required in our system to support this, we will focus solely on default events that are associated with a particular counterparty, and examine how the presence of synthetic defaults can be used to control their overall default status. We start by extending the *Default* type from the previous model, adding details of the default instance's inception.

$$
\begin{array}{|l}
\textit{Default} \\
\hline
\textit{status} : \textit{ApprovalStatus} \\
\textit{defaultType} : \textit{DefaultType} \\
\hline
\end{array}
$$

$$\textit{DefaultType} ::= \textit{DEFERRED} \mid \textit{DIRECT} \mid \textit{INDIRECT} \mid \textit{SYNTHETIC}$$

Direct defaults correspond to the manual instances in our previous model, whereas deferred defaults relate to events received from other systems. As such, they require further examination before being converted into indirect defaults, at which point they are added to the default case. To support a review and approval process for that conversion, synthetic default instances are raised directly in the system.

3.2.1 Party Default Data

In this model, we are concerned primarily with the handling of deferred default events and the creation of synthetic default instances, rather than the relations between parties and default cases. We adopt this level of abstraction in our state schema, and proceed to model the underlying business logic from the perspective of a single counterparty.

A party's default status is binary:

$$\textit{PartyStatus} ::= \textit{DEFAULTED} \mid \textit{NOT\_IN\_DEFAULT}$$

This overall default status is added to our state schema, and derived by evaluating the respective statuses of default instances within the case:

$$
\begin{array}{|l}
\textit{DefaultSystem} \\
\hline
\textit{defaults} : \textit{DefaultId} \mathrel{+\!\!\!\to} \textit{Default} \\
\textit{defaultCase} : \mathbb{P}\,\textit{DefaultId} \\
\textit{deferredTriggers} : \mathbb{P}\,\textit{DefaultId} \\
\textit{partyStatus} : \textit{PartyStatus} \\
\hline
\textit{defaultCase} \cup \textit{deferredTriggers} \subseteq \operatorname{dom} \textit{defaults} \\
\textit{partyStatus} = \textit{DEFAULTED} \Leftrightarrow \\
\qquad \{d : \textit{defaultCase} \mid (\textit{defaults}\, d).\textit{status} \in \textit{MaterialDefaultStatus}\} \neq \varnothing \\
\forall\, d_1, d_2 : \textit{defaultCase} \bullet \\
\qquad ((\textit{defaults}\ d_1).\textit{defaultType} = (\textit{defaults}\ d_2).\textit{defaultType} = \textit{SYNTHETIC} \land \\
\qquad \{(\textit{defaults}\ d_1).\textit{status}, (\textit{defaults}\ d_2).\textit{status}\} \subseteq \textit{MaterialDefaultStatus}) \Rightarrow d_1 = d_2 \\
\hline
\end{array}
$$

At any given time, there can be at most one synthetic default instance that determines the overall status of the counterparty. This is enforced through an invariant in the state schema. Since we are less concerned with the opening and closing of default cases in this model, we can relax the previously-enforced constraints around their creation here, initialising the system with an empty default case:

$$
\begin{array}{|l}
\textit{Init} \\
\hline
\textit{DefaultSystem}' \\
\hline
\textit{defaults}' = \varnothing \\
\textit{defaultCase}' = \varnothing \\
\textit{deferredTriggers}' = \varnothing \\
\textit{partyStatus}' = \textit{NOT\_IN\_DEFAULT} \\
\hline
\end{array}
$$

In contrast to our previous model, we will not use promotion to factor the operations here. Since the *DefaultSystem* global state schema now refers directly to properties of the local *Default* state, the promotion would be *constrained*, that is:

$$(\exists Default' \bullet \exists DefaultSystem' \bullet Promote) \not\Rightarrow (\forall Default' \bullet \exists DefaultSystem' \bullet Promote)$$

This means that operations may be defined on the local state that lead to inconsistencies in the global state following promotion. Rather than accept this reduced level of modularity in our model, we shall employ other techniques, such as schema inclusion and composition, to describe the operations associated with handling deferred and synthetic defaults.

3.2.2 Proposing Manual Defaults

As before, direct defaults can be proposed manually; on proposal, they are added immediately to the default case:

$$\begin{array}{|l}
\textit{ProposeManualDefault} \\
\hline
\Delta DefaultSystem \\
defaultEvent? : DefaultId \times Default \\
\hline
defaultEvent? \notin defaults \\
(second\ defaultEvent?).status = PROPOSED \\
defaults' = defaults \cup \{defaultEvent?\} \\
defaultCase' = defaultCase \cup \{first\ defaultEvent?\} \\
deferredTriggers' = deferredTriggers \\
\hline
\end{array}$$

$$ProposeDirectDefault \mathrel{\widehat=} [ProposeManualDefault \mid (second\ defaultEvent?).defaultType = DIRECT]$$

Where deferred triggers have been received against a party that is not yet in default, the user may wish to propose a synthetic default instance. Doing so is supported in our model with the following schema:

$$\begin{array}{|l}
\textit{ProposeSyntheticDefault} \\
\hline
ProposeManualDefault \\
\hline
partyStatus = NOT\_IN\_DEFAULT \\
deferredTriggers \neq \varnothing \\
(second\ defaultEvent?).defaultType = SYNTHETIC \\
\hline
\end{array}$$

3.2.3 Handling Default Event Triggers

In practice, indirect defaults may enter the system as the result of a number of different events—batch processing, web service calls etc. For the purposes of this model, however, we represent all of these events with a single operation, defined across two schemas. In both cases, the new default trigger is added to the set of default instances managed by the system:

$$\begin{array}{|l}
\textit{ReceiveDefaultTriggerCommon} \\
\hline
\Delta DefaultSystem \\
defaultTrigger? : DefaultId \times Default \\
\hline
defaultTrigger? \notin defaults \\
defaults' = defaults \cup \{defaultTrigger?\} \\
\hline
\end{array}$$

Where a party is not in default at the time a default event trigger is received, that trigger is added to the deferred trigger list, where it is subject to review. All default event triggers are considered eligible for deferral at this stage; should we wish to include the specific criteria for deferral in a subsequent refinement of our model, this could be achieved by extending the following *ReceiveDefaultTriggerSolventParty* schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ReceiveDefaultTriggerSolventParty}} \\
\mathit{ReceiveDefaultTriggerCommon} \\
\hline
\mathit{partyStatus} = \mathit{NOT\_IN\_DEFAULT} \\
(\mathit{second}\ \mathit{defaultTrigger}?).\mathit{status} = \mathit{PROPOSED} \\
(\mathit{second}\ \mathit{defaultTrigger}?).\mathit{defaultType} = \mathit{DEFERRED} \\
\mathit{deferredTriggers}' = \mathit{deferredTriggers} \cup \{\mathit{first}\ \mathit{defaultTrigger}?\} \\
\mathit{defaultCase}' = \mathit{defaultCase} \\
\hline
\end{array}$$

If a party is already in default at the time a trigger is received, that trigger should be added directly to the party's default case:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ReceiveDefaultTriggerDefaultedParty}} \\
\mathit{ReceiveDefaultTriggerCommon} \\
\hline
\mathit{partyStatus} = \mathit{DEFAULTED} \\
(\mathit{second}\ \mathit{defaultTrigger}?).\mathit{status} = \mathit{APPROVED} \\
(\mathit{second}\ \mathit{defaultTrigger}?).\mathit{defaultType} = \mathit{INDIRECT} \\
\mathit{defaultCase}' = \mathit{defaultCase} \cup \{\mathit{first}\ \mathit{defaultTrigger}?\} \\
\mathit{deferredTriggers}' = \mathit{deferredTriggers} \\
\hline
\end{array}$$

The *ReceiveDefaultTrigger* schema operation then defines a total operation, formed by the disjunction of the two previous scenarios:

$$\mathit{ReceiveDefaultTrigger} \mathrel{\widehat=} \mathit{ReceiveDefaultTriggerDefaultedParty} \lor \mathit{ReceiveDefaultTriggerSolventParty}$$

3.2.4 Handling Default Closure Triggers

Alternatively, triggers may be received that represent closures for existing default instances, perhaps because information has been obtained that suggests a party's credit strength has improved. If the underlying default instance is still deferred at the time a closure trigger is received, then it can simply be removed from the deferred trigger list:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CloseDeferredTrigger}} \\
\Delta \mathit{DefaultSystem} \\
\mathit{default}? : \mathit{DefaultId} \\
\mathit{newDefault} : \mathit{Default} \\
\hline
\mathit{newDefault}.\mathit{status} = \mathit{CLOSED} \\
\mathit{newDefault}.\mathit{defaultType} = (\mathit{defaults}\ \mathit{default}?).\mathit{defaultType} \\
\mathit{defaults}' = \mathit{defaults} \oplus \{\mathit{default}? \mapsto \mathit{newDefault}\} \\
\mathit{defaultCase}' = \mathit{defaultCasedefault}? \in \mathit{deferredTriggers} \\
\mathit{deferredTriggers}' = \mathit{deferredTriggers} \setminus \{\mathit{default}?\} \\
\hline
\end{array}$$

Closure triggers for indirect default instances are handled differently if their closure will affect the overall default status of the counterparty. This would occur if they are last remaining default instance in the case that is approved, or whose closure is proposed. The following schemas are used to determine whether closing a particular default instance will have a material impact on the party's overall default status:

$$
\begin{array}{|l}
\hline
\mathit{MaterialDefaultsInCase} \\
\mathit{DefaultSystem} \\
d? : \mathit{DefaultId} \\
\mathit{materialDefaults} : \mathbb{P}\,\mathit{DefaultId} \\
\hline
\mathit{materialDefaults} = \{x : \mathit{defaultCase} \mid (\mathit{defaults}\ x).\mathit{status} \in \mathit{MaterialDefaultStatus} \bullet x\} \\
\hline
\end{array}
$$

$\mathit{LastDefaultInCase} \mathrel{\widehat=} [\mathit{MaterialDefaultsInCase} \mid \mathit{materialDefaults} \subseteq \{d?\}]$
$\mathit{NotLastDefaultInCase} \mathrel{\widehat=} [\mathit{MaterialDefaultsInCase} \mid \{d?\} \subset \mathit{materialDefaults}]$

On closing an indirect default instance, a number of changes to the global state occur regardless of whether the instance being closed is the last in the default case. We capture these changes in the *CloseIndirectDefaultCommon* schema:

$$
\begin{array}{|l}
\hline
\mathit{CloseIndirectDefaultCommon} \\
\Delta \mathit{DefaultSystem} \\
d? : \mathit{DefaultId} \\
\mathit{updatedDefault} : \mathit{Default} \\
\hline
d? \in \mathrm{dom}\ \mathit{defaults} \\
(\mathit{defaults}\, d?).\mathit{status} = \mathit{APPROVED} \\
(\mathit{defaults}\, d?).\mathit{defaultType} = \mathit{INDIRECT} \\
\mathit{defaultCase}' = \mathit{defaultCase} \\
\mathit{deferredTriggers}' = \mathit{deferredTriggers} \\
\mathit{defaults}' = \mathit{defaults} \oplus \{d? \mapsto \mathit{updatedDefault}\} \\
\hline
\end{array}
$$

If the instance that is being closed is not the last material default in the case, then its closure simply results in a relevant update to its status:

$$
\begin{array}{|l}
\hline
\mathit{CloseNonFinalIndirectDefault} \\
\mathit{CloseIndirectDefaultCommon} \\
\mathit{NotLastDefaultInCase} \\
\hline
\mathit{updatedDefault}.\mathit{defaultType} = \mathit{INDIRECT} \\
\mathit{updatedDefault}.\mathit{status} = \mathit{CLOSED} \\
\hline
\end{array}
$$

Greater complexity is present on receiving a trigger for closing the last material default in a case: the system must postpone closure of the default case until an additional review has been performed. The default case is kept open by adding a 'synthetic' default instance to it, handled in our model by converting the last indirect default instance to a synthetic default upon its closure:

$$
\begin{array}{|l}
\hline
\mathit{CloseLastIndirectDefault} \\
\mathit{CloseIndirectDefaultCommon} \\
\mathit{LastDefaultInCase} \\
\hline
\mathit{updatedDefault}.\mathit{defaultType} = \mathit{SYNTHETIC} \\
\mathit{updatedDefault}.\mathit{status} = \mathit{APPROVED} \\
\hline
\end{array}
$$

We therefore arrive at a total operation for closing indirect default instances by writing the disjunction of the previous two schemas:

$\mathit{CloseIndirectDefault} \mathrel{\widehat=} \mathit{CloseNonFinalIndirectDefault} \lor \mathit{CloseLastIndirectDefault}$

3.2.5 Proposing Closure of Manual Default Instances

Just as the creation of direct and synthetic defaults can be proposed manually, so too can their closure. Common behaviour is extracted into the *ProposeCloseManual* schema:

$$
\begin{array}{|l}
\hline
\textit{ProposeCloseManual} \\
\Delta \textit{DefaultSystem} \\
d? : \textit{DefaultId} \\
\textit{updatedDefault} : \textit{Default} \\
\hline
d? \in \operatorname{dom} \textit{defaults} \\
(\textit{defaults}\, d?).\textit{status} = \textit{APPROVED} \\
\textit{defaultCase}' = \textit{defaultCase} \\
\textit{deferredTriggers}' = \textit{deferredTriggers} \\
\textit{updatedDefault}.\textit{status} = \textit{PROPOSED\_CLOSURE} \\
\textit{updatedDefault}.\textit{defaultType} = (\textit{defaults}\, d?).\textit{defaultType} \\
\textit{defaults}' = \textit{defaults} \oplus \{d? \mapsto \textit{updatedDefault}\} \\
\hline
\end{array}
$$

This schema is then extended to define separate operations for proposing closure of the two flavours of manual default:

$$
\begin{array}{l}
\textit{ProposeCloseDirect} \mathrel{\widehat=} [\textit{ProposeCloseManual} \mid (\textit{defaults}\, d?).\textit{defaultType} = \textit{DIRECT}\,] \\
\textit{ProposeCloseSynthetic} \mathrel{\widehat=} [\textit{ProposeCloseManual} \mid (\textit{defaults}\, d?).\textit{defaultType} = \textit{SYNTHETIC}\,]
\end{array}
$$

The disjunction of these schemas, together with the condition that synthetic default instances cannot be closed unless they represent the last material default in the case, forms the *ProposeDefaultClosure* operation:

$$
\textit{ProposeDefaultClosure} \mathrel{\widehat=} \textit{ProposeCloseDirect} \lor \textit{ProposeCloseSynthetic} \land \textit{LastDefaultInCase}
$$

3.2.6 Reviewing Proposed Changes to Default Instances

Proposals to add or close default instances may be rejected following a review of the circumstances surrounding their inception. We model this rejection by reversing the status of a default instance back through its life cycle.

$$
\begin{array}{|l}
\hline
\textit{RejectProposedTransition} \\
\Delta \textit{DefaultSystem} \\
d? : \textit{DefaultId} \\
\textit{updatedDefault} : \textit{Default} \\
\hline
d? \in \operatorname{dom} \textit{defaults} \\
(\textit{defaults}\, d?).\textit{defaultType} \in \{\textit{DIRECT}, \textit{SYNTHETIC}\} \\
(\textit{defaults}\, d?).\textit{status} \in \{\textit{PROPOSED}, \textit{PROPOSED\_CLOSURE}\} \\
\textit{defaultCase}' = \textit{defaultCase} \\
\textit{deferredTriggers}' = \textit{deferredTriggers} \\
\textit{updatedDefault}.\textit{defaultType} = (\textit{defaults}\, d?).\textit{defaultType} \\
\textit{updatedDefault}.\textit{status} = \textit{DefaultLifeCycle}'(\textit{DefaultLifeCycle}'^{\sim}((\textit{defaults}\, d?).\textit{status}) - 1) \\
\textit{defaults}' = \textit{defaults} \oplus \{d? \mapsto \textit{updatedDefault}\} \\
\hline
\end{array}
$$

Following a review, it may be decided that opening a default instance against a counterparty is indeed warranted. In this case, the default proposal is approved:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ApproveOpenDefaultCommon}} \\
\Delta \mathit{DefaultSystem} \\
d? : \mathit{DefaultId} \\
\mathit{updatedDefault} : \mathit{Default} \\
\hline
d? \in \operatorname{dom} \mathit{defaults} \\
(\mathit{defaults}\, d?).\mathit{status} = \mathit{PROPOSED} \\
\mathit{defaultCase}' = \mathit{defaultCase} \\
\mathit{deferredTriggers}' = \mathit{deferredTriggers} \\
\mathit{updatedDefault}.\mathit{defaultType} = (\mathit{defaults}\, d?).\mathit{defaultType} \\
\mathit{updatedDefault}.\mathit{status} = \mathit{APPROVED} \\
\mathit{defaults}' = \mathit{defaults} \oplus \{d? \mapsto \mathit{updatedDefault}\} \\
\hline
\end{array}$$

Adding a constraint on the default type yields the basis for our operation modelling the approval of a direct default:

$$\mathit{ApproveOpenDirect0} \mathrel{\widehat{=}} [\mathit{ApproveOpenDefaultCommon} \mid (\mathit{defaults}\, d?).\mathit{defaultType} = \mathit{DIRECT}]$$

Although associated deferred triggers may have been present at the time of proposal for a synthetic default instance, those triggers could have been closed by the time the instance is approved. We add a constraint to our model enforcing this requirement: all synthetic default instances must correspond to at least one active deferred trigger.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ApproveOpenSynthetic}} \\
\mathit{ApproveOpenDefaultCommon} \\
\hline
\mathit{deferredTriggers} \neq \varnothing \\
(\mathit{defaults}\, d?).\mathit{defaultType} = \mathit{SYNTHETIC} \\
\hline
\end{array}$$

The approval of a direct default renders any coexisting proposed synthetics redundant, since its approval entails the requisite assessment of the underlying counterparty. Any such synthetics should be closed immediately on the approval of a direct default instance:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CloseRedundantSynthetics}} \\
\Delta \mathit{DefaultSystem} \\
\mathit{syntheticDefaults} : \mathit{DefaultId} \pfun \mathit{Default} \\
\hline
\operatorname{dom} \mathit{syntheticDefaults} = \{d : \mathit{defaultCase} \mid (\mathit{defaults}\ d).\mathit{defaultType} = \mathit{SYNTHETIC}\} \\
\forall\, d : \mathit{syntheticDefaults} \bullet \\
\qquad (\mathit{second}\, d).\mathit{status} = \mathit{CLOSED} \land (\mathit{second}\, d).\mathit{defaultType} = \mathit{SYNTHETIC} \\
\mathit{defaults}' = \mathit{defaults} \oplus \mathit{syntheticDefaults} \\
\mathit{defaultCase}' = \mathit{defaultCase} \\
\mathit{deferredTriggers}' = \mathit{deferredTriggers} \\
\hline
\end{array}$$

Schema composition is used to ensure that redundant synthetic default instances are closed following the approval of a direct default:

$$\mathit{ApproveOpenDirect} \mathrel{\widehat{=}} \mathit{ApproveOpenDirect0} \mathbin{⨾} \mathit{CloseRedundantSynthetics}$$

Additionally, when a direct default instance is approved, all deferred triggers for the associated counterparty should be promoted into material default instances and added to the case. We use the *AddIndirectTriggers* schema to show this:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AddIndirectTriggers}} \\
\Delta \mathit{DefaultSystem} \\
d? : \mathit{DefaultId} \\
\mathit{indirectDefaults} : \mathit{DefaultId} \pfun \mathit{Default} \\
\hline
\mathrm{dom}\ \mathit{indirectDefaults} = \mathit{deferredTriggers} \\
\forall d : \mathit{indirectDefaults} \bullet \\
\qquad (\mathit{second}\, d).\mathit{status} = \mathit{APPROVED} \land (\mathit{second}\, d).\mathit{defaultType} = \mathit{INDIRECT} \\
\mathit{defaults}' = \mathit{defaults} \oplus \mathit{indirectDefaults} \\
\mathit{defaultCase}' = \mathit{defaultCase} \cup \mathrm{dom}\ \mathit{indirectDefaults} \\
\mathit{deferredTriggers}' = \varnothing \\
\hline
\end{array}
$$

Again, schema composition is used here to describe this in our model:

$$\mathit{ApproveOpenDefault} \mathrel{\widehat=} (\mathit{ApproveOpenDirect} \lor \mathit{ApproveOpenSynthetic}) \mathbin{;} \mathit{AddIndirectTriggers}$$

On approving the closure of a default instance, the system behaves in a similar fashion to our previous model. However, in the way that we did for default approvals, separate operations for the closure of direct and synthetic instances are given here, since additional constraints must be satisfied on closing synthetic defaults.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ApproveCloseDefaultCommon}} \\
\Delta \mathit{DefaultSystem} \\
d? : \mathit{DefaultId} \\
\mathit{updatedDefault} : \mathit{Default} \\
\hline
d? \in \mathrm{dom}\ \mathit{defaults} \\
(\mathit{defaults}\, d?).\mathit{status} = \mathit{PROPOSED\_CLOSURE} \\
\mathit{defaultCase}' = \mathit{defaultCase} \\
\mathit{deferredTriggers}' = \mathit{deferredTriggers} \\
\mathit{updatedDefault}.\mathit{defaultType} = (\mathit{defaults}\, d?).\mathit{defaultType} \\
\mathit{updatedDefault}.\mathit{status} = \mathit{CLOSED} \\
\mathit{defaults}' = \mathit{defaults} \oplus \{d? \mapsto \mathit{updatedDefault}\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{ApproveCloseDirect} \mathrel{\widehat=} [\mathit{ApproveCloseDefaultCommon} \mid (\mathit{defaults}\, d?).\mathit{defaultType} = \mathit{DIRECT}] \\
\mathit{ApproveCloseSynthetic} \mathrel{\widehat=} [\mathit{ApproveCloseDefaultCommon} \mid (\mathit{defaults}\, d?).\mathit{defaultType} = \mathit{SYNTHETIC}]
\end{array}
$$

The condition that a synthetic must be the last material default in the case for its closure to be proposed is enforced again on approval:

$$\mathit{ApproveCloseDefault} \mathrel{\widehat=} \mathit{ApproveCloseDirect} \lor \mathit{ApproveCloseSynthetic} \land \mathit{LastDefaultInCase}$$

3.3 Party Hierarchies and Cross-Defaults

Up to this point, our models have dealt only with *primary* default instances, in which each incoming default event relates directly to the party being defaulted. In practice, however, the credit strengths of multiple parties are often closely related: a default event recorded against one party may cause others to default. These so called ‘cross defaults’ can occur only between counterparties within the same organisational group, and a fundamental requirement of the system is to manage the legal hierarchies that reflect the respective seniority of different parties within these groups. The structure of these relationships forms the basis of our next model.

3.3.1 Legal Hierarchies

Legal hierarchies have a tree-like structure: given $\{A,B,C,D,E,F,G,H\} \subseteq \mathit{Party}$, the examples shown in Figure 3.3 are all valid.

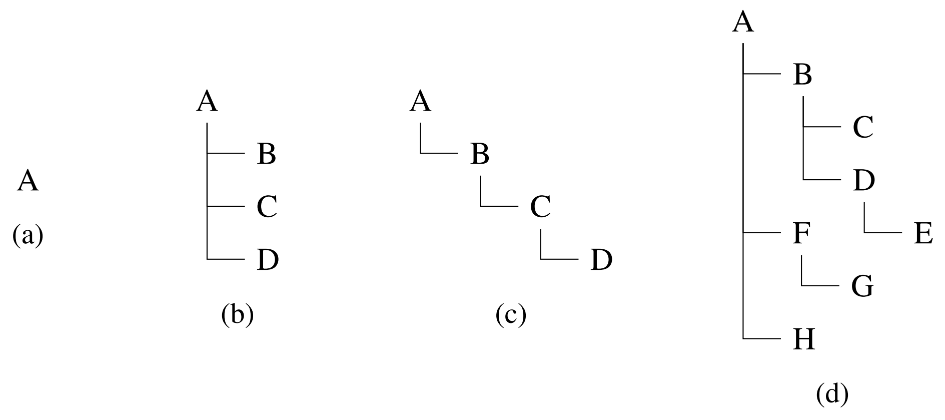


Figure 3.3: Examples of valid legal hierarchies

The Z notation can be used to model these structures naturally using a recursive type. Trees whose branches contain an arbitrary number of nodes are known as *rose trees*, and where their nodes are all of a particular type, their structure can be expressed concisely in Z using a free type. Legal hierarchies—whose nodes can be represented with the *Party* type—are therefore defined as follows:

$$\mathit{LegalHierarchy} ::= \mathit{Node}\langle\langle \mathit{Party} \times \mathbb{P}\,\mathit{LegalHierarchy} \rangle\rangle$$

We can now proceed to model operations on this type. For example, the number of parties within a particular legal hierarchy can be expressed as a function:

$$\begin{array}{|l}
\mathit{size} : \mathit{LegalHierarchy} \rightarrow \mathbb{N} \\
\hline
\forall p : \mathit{Party} \bullet \mathit{size}\,(\mathit{Node}(p, \varnothing)) = 1 \\
\forall p : \mathit{Party};\ l : \mathit{LegalHierarchy};\ \mathit{ls} : \mathbb{P}\,\mathit{LegalHierarchy} \bullet \\
\qquad l \in \mathit{ls} \Rightarrow \mathit{size}\,(\mathit{Node}(p, \mathit{ls})) = \mathit{size}(l) + \mathit{size}(\mathit{Node}(p, \mathit{ls} \setminus \{l\}))
\end{array}$$

A legal hierarchy can be ‘flattened’ to yield the set of parties it contains:

$$\begin{array}{|l}
\mathit{flatten} : \mathit{LegalHierarchy} \rightarrow \mathbb{P}\,\mathit{Party} \\
\hline
\forall a : \mathit{Party} \bullet \mathit{flatten}\,(\mathit{Node}(a, \varnothing)) = \{a\} \\
\forall a : \mathit{Party};\ \mathit{ts} : \mathbb{P}\,\mathit{LegalHierarchy} \bullet \\
\qquad \mathit{flatten}(\mathit{Node}(a, \mathit{ts})) = \bigcup\{t : \mathit{ts} \bullet \mathit{flatten}\,t\} \cup \{a\}
\end{array}$$

For any legal hierarchy, we can derive the party at its root:

$$\begin{array}{|l}
\mathit{ultimateParent} : \mathit{LegalHierarchy} \rightarrow \mathit{Party} \\
\hline
\forall a : \mathit{Party};\ \mathit{ts} : \mathbb{P}\,\mathit{LegalHierarchy} \bullet \mathit{ultimateParent}(\mathit{Node}(a, \mathit{ts})) = a
\end{array}$$

…and remove this to obtain the set of subsidiary parties, which we express using the *lambda notation* as follows:

$$\mathit{subsidiaries} == (\lambda\, l : \mathit{LegalHierarchy} \bullet \mathit{flatten}\,l \setminus \{\mathit{ultimateParent}\,l\})$$

A partial function that will give a party's immediate parent (where one exists) can be defined as:

$$\begin{array}{|l}
\mathit{parent} : \mathit{Party} \times \mathit{LegalHierarchy} \mathrel{+\!\!\!\!\to} \mathit{Party} \\
\hline
\forall p_1, p_2 : \mathit{Party};\ l_1, l_2 : \mathit{LegalHierarchy};\ ls_1, ls_2 : \mathbb{P}\,\mathit{LegalHierarchy} \bullet \\
\qquad \mathit{parent}(p_1, \mathit{Node}(p_2, ls_1 \cup \{\mathit{Node}(p_1, ls_2)\})) = p_2 \land \\
\qquad p_1 \in \mathit{flatten}\, l_1 \Rightarrow \mathit{parent}(p_1, \mathit{Node}(p_2, ls_1 \cup \{l_1\})) = \mathit{parent}(p_1, l_1)
\end{array}$$

A new legal hierarchy may be added as a child node of another with the below function, taking the new and existing trees, and a point of insertion as arguments:

$$\begin{array}{|l}
\mathit{addSubtree} : (\mathit{LegalHierarchy} \times \mathit{LegalHierarchy} \times \mathit{Party}) \mathrel{+\!\!\!\!\to} \mathit{LegalHierarchy} \\
\hline
\forall p_1, p_2 : \mathit{Party};\ t_1, t_2 : \mathit{LegalHierarchy};\ ts : \mathbb{P}\,\mathit{LegalHierarchy} \bullet \\
\qquad \mathit{addSubtree}(t_1, \mathit{Node}(p_1, ts), p_1) = \mathit{Node}(p_1, ts \cup \{t_1\}) \land \\
\qquad p_1 \in \mathit{flatten}\, t_2 \Rightarrow \\
\qquad\qquad \mathit{addSubtree}(t_1, \mathit{Node}(p_2, ts \cup \{t_2\}), p_1) = \mathit{Node}(p_2, ts \cup \{\mathit{addSubtree}(t_1, t_2, p_1)\})
\end{array}$$

Similarly, part of a legal hierarchy may be removed, by specifying the parent node of the subtree to be pruned:

$$\begin{array}{|l}
\mathit{removeSubtree} : (\mathit{Party} \times \mathit{LegalHierarchy}) \mathrel{+\!\!\!\!\to} \mathit{LegalHierarchy} \\
\hline
\forall p_1, p_2 : \mathit{Party};\ l : \mathit{LegalHierarchy};\ ls_1, ls_2 : \mathbb{P}\,\mathit{LegalHierarchy} \bullet \\
\qquad \mathit{removeSubtree}(p_1, \mathit{Node}(p_2, ls_1 \cup \{\mathit{Node}(p_1, ls_2)\})) = \mathit{Node}(p_2, ls_1) \land \\
\qquad p_1 \in \mathit{subsidiaries}\, l \Rightarrow \\
\qquad\qquad \mathit{removeSubtree}(p_1, \mathit{Node}(p_2, ls_1 \cup \{l\})) = \mathit{Node}(p_2, ls_1 \cup \{\mathit{removeSubtree}(p_1, l)\})
\end{array}$$

Having presented a number of functions to manipulate the structure of our *LegalHierarchy* instances, we can begin to consider the grading hierarchies that exist within them, and from which cross defaults are derived.

3.3.2 Grading Hierarchies

When a credit grading is cascaded to another counterparty, the reason for doing so is recorded, together with any adjustment—known as a 'notch'—made to the grade itself. A cascaded grading can therefore be modelled as a tuple comprising the party that is inheriting the grade, a cascade reason, and the notch (which will be zero where no adjustment is made):

$$\begin{array}{l}
[\mathit{CascadeReason}] \\
\mathit{Notch} == \mathbb{Z} \\
\mathit{CascadedGrading} == \mathit{Party} \times \mathit{CascadeReason} \times \mathit{Notch}
\end{array}$$

We can then construct the *GradingHierarchy* type by binding the initially-graded party to the set of cascaded gradings that relate to it.

$$\mathit{GradingHierarchy} == \mathit{Party} \times \mathbb{P}\,\mathit{CascadedGrading}$$

The following functions are used to extract specific information from the above types. Given a particular grading cascade, we can find the party involved:

$$\begin{array}{|l}
\mathit{cascadeParty} : \mathit{CascadedGrading} \to \mathit{Party} \\
\hline
\forall \mathit{party} : \mathit{Party};\ \mathit{reason} : \mathit{CascadeReason};\ \mathit{notch} : \mathit{Notch} \bullet \\
\qquad \mathit{cascadeParty}(\mathit{party}, \mathit{reason}, \mathit{notch}) = \mathit{party}
\end{array}$$

. . . or alternatively, retrieve the party from a grading hierarchy whose grade is being inherited:

$$\begin{array}{|l}
\mathit{cascadeParent} : \mathit{GradingHierarchy} \rightarrow \mathit{Party} \\
\hline
\forall \mathit{parent} : \mathit{Party};\ \mathit{cascadedGradings} : \mathbb{P}\,\mathit{CascadedGrading} \bullet \\
\qquad \mathit{cascadeParent}(\mathit{parent}, \mathit{cascadedGradings}) = \mathit{parent}
\end{array}$$

All recipients of the cascaded grading can be enumerated with the *cascadeChildren* function:

$$\begin{array}{|l}
\mathit{cascadeChildren} : \mathit{GradingHierarchy} \rightarrow \mathbb{P}\,\mathit{Party} \\
\hline
\forall p : \mathit{Party};\ \mathit{cascadedGradings} : \mathbb{P}\,\mathit{CascadedGrading} \bullet \\
\qquad \mathit{cascadeChildren}(p, \mathit{cascadedGradings}) = \{c : \mathit{cascadedGradings} \bullet \mathit{cascadeParty}\, c\}
\end{array}$$

Finally, for any grading hierarchy, we can evaluate the set of parties whose credit strength is related:

$$\begin{array}{|l}
\mathit{creditRelatedParties} : \mathit{GradingHierarchy} \rightarrow \mathbb{P}\,\mathit{Party} \\
\hline
\forall p : \mathit{Party};\ cs : \mathbb{P}\,\mathit{CascadedGrading} \bullet \\
\qquad \mathit{creditRelatedParties}(p, \mathit{cascadedGradings}) = \{p\} \cup \mathit{cascadeChildren}(p, \mathit{cascadedGradings})
\end{array}$$

With these initial definitions in place, we can finally introduce the state schema to show the relationship between legal and grading hierarchies:

$$\begin{array}{|l}
\hline
\mathit{GradingSystem} \\
\mathit{legalHierarchies} : \mathbb{P}\,\mathit{LegalHierarchy} \\
\mathit{cascades} : \mathbb{P}\,\mathit{GradingHierarchy} \\
\mathit{grade} : \mathit{Party} \nrightarrow \mathbb{Z} \\
\hline
\forall c : \mathit{cascades} \bullet \exists\, l : \mathit{legalHierarchies} \bullet \mathit{creditRelatedParties}\, c \subseteq \mathit{flatten}\, l \\
\hline
\end{array}$$

The constraint in this schema ensures that all parties that are involved in a grading cascade belong to the same legal hierarchy.

3.3.3 Cross Defaults

So far, the definitions we have introduced are of limited intrinsic use. The value they contribute to this model is in helping form a description of the conditions in which cross defaults can occur in the system. This description is given in the *ShowCrossDefaults* operation schema:

$$\begin{array}{|l}
\hline
\mathit{ShowCrossDefaults} \\
\Xi \mathit{GradingSystem} \\
\mathit{primary}? : \mathit{Party} \\
\mathit{crossDefaults}! : \mathbb{P}\,\mathit{Party} \\
\hline
\forall c : \mathit{cascades} \bullet \\
\qquad (\mathit{cascadeParent}\, c = \mathit{primary}? \lor \\
\qquad\ \mathit{primary}? \in \mathit{cascadeChildren}\, c \land \mathit{grade}\, \mathit{primary}? \geqslant \mathit{grade}(\mathit{cascadeParent}\, c)) \Rightarrow \\
\qquad\qquad \mathit{crossDefaults}! = (\mathit{cascadeChildren}\, c \cup \{\mathit{cascadeParent}\, c\}) \setminus \{\mathit{primary}?\} \\
\qquad \land \\
\qquad \mathit{primary}? \in \mathit{cascadeChildren}\, c \land \mathit{grade}\, \mathit{primary}? < \mathit{grade}(\mathit{cascadeParent}\, c) \Rightarrow \\
\qquad\qquad \mathit{crossDefaults}! = \{p : \mathit{cascadeChildren}\, c \bullet \mathit{grade}\, p \leqslant \mathit{grade}\, \mathit{primary}?\} \setminus \{\mathit{primary}?\} \\
\hline
\end{array}$$

Evaluating this operation for the scenarios described in Figures 2.2, 2.3 and 2.4 gives the expected sets of parties in cross default. For example, for the grading hierarchy and default event given in Figure 2.4, we have:

1. The primary default has a weaker credit strength than the cascade parent, that is,

   $$\mathit{grade}\, C < \mathit{grade}\, E$$

2. The *ShowCrossDefaults* schema then gives the following definition for the set of cross defaults:

   $$\begin{aligned}
   & \mathit{crossDefaults}! \\
   = & \qquad \text{[definition of } \mathit{crossDefaults}!\text{]} \\
   & \{p : \mathit{cascadeChildren}\, E \bullet \mathit{grade}\, p \leqslant \mathit{grade}\, \mathit{primary}?\} \setminus \{\mathit{primary}?\} \\
   = & \qquad \text{[party } C \text{ is the primary default]} \\
   & \{p : \mathit{cascadeChildren}\, E \bullet \mathit{grade}\, p \leqslant \mathit{grade}\, C\} \setminus \{C\} \\
   = & \qquad \text{[evaluating } \mathit{grade} \text{ for party } C\text{]} \\
   & \{p : \mathit{cascadeChildren}\, E \bullet \mathit{grade}\, p \leqslant 2\} \setminus \{C\} \\
   = & \qquad \text{[definition of } \mathit{cascadeChildren}\text{]} \\
   & \{p : \{C, F, H, J\} \bullet \mathit{grade}\, p \leqslant 2\} \setminus \{C\}
   \end{aligned}$$

3. Since only parties C and J have a grading whose value is less than or equal to that of the primary default, this yields

   $$\begin{aligned}
   & \mathit{crossDefaults}! = \{C, J\} \setminus \{C\} \\
   = & \qquad \text{[set difference]} \\
   & \{J\}
   \end{aligned}$$

   as expected.

In this chapter, we have presented three successive models for a system concerned with managing Basel II defaults. Beginning with a simple model, written at a high level of abstraction, we arrived at a description of default instances, cases, and their impact on the overall credit status of affected counterparties. This was then extended to model the introduction of deferred and synthetic defaults. A number of issues were identified in this activity, including the need to raise synthetic defaults automatically under certain conditions. The construction of this model represented the first time that these requirements had been considered in terms of their impact on the system's current behaviour, and the integration of elements from the first model served as a convenient means by which this impact could be assessed. A different approach was taken in the third model: rather than describing sequential transitions of the system's state as a set of operation schemas, focus was shifted to the structure of the legal hierarchies maintained by the system. This provided a basis for defining functions to manipulate these structures, leading ultimately to an accurate description of the conditions under which cross defaults are raised.

Further analysis was performed on the models given here, using a variety of tools, and with an emphasis on a lightweight, collaborative approach conducive to Agile development. An account of the techniques that were used to conduct this analysis are presented in the next chapter, together with a discussion of how they were most effectively applied to our modelling scenario.

4 Analysis of the Models

The act of constructing the models described in the previous chapter is valuable in itself: identifying the essential aspects of a system's behaviour and expressing these with mathematical clarity yields an understanding of the problem domain that would be difficult to acquire solely through reading a set of requirements documents. According to the Dreyfus model of skill acquisition, reaching proficiency in a particular field requires the learner to recognise the holistic implications of an action as well as its immediate effect:

> "Each whole situation, for the first time, has a meaning which is its relevance to the achievement of a long-term goal... [This information] is organized and stored in such a way as to provide a basis for future recognition of similar situations viewed from similar perspectives." [DD80]

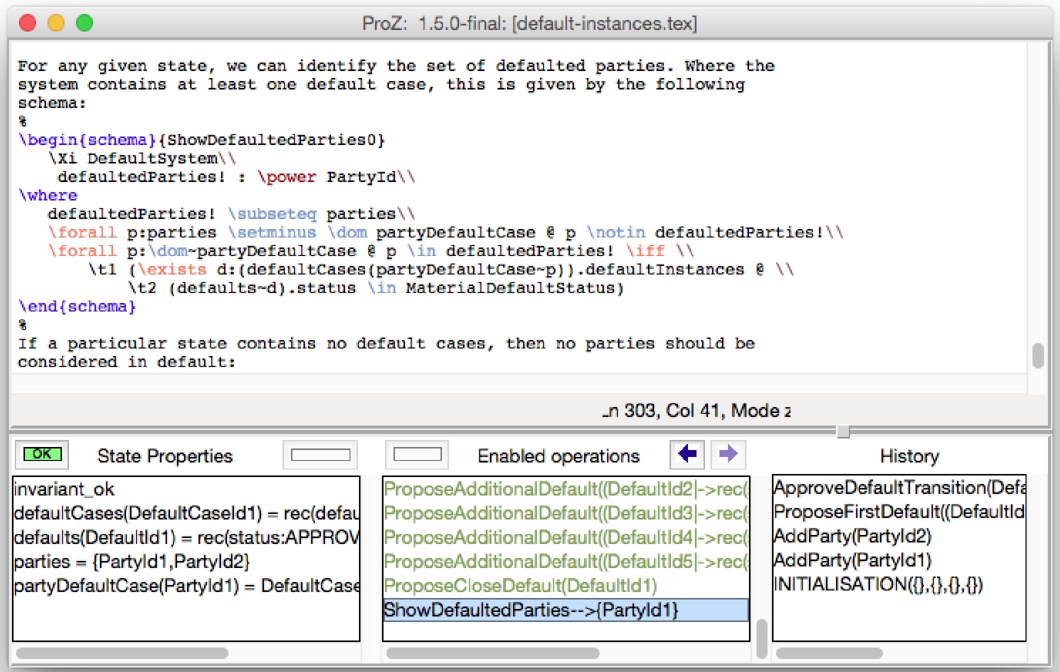
Describing operations in terms of a system's overall change in state accelerates this learning process, and encourages the team to consider problems at an appropriate level of abstraction. This helps to guide discussions of a system's requirements in a useful direction, and by reducing the initial emphasis on precision and detail, a model can focus the team's attention on what really matters: in our case, the impact of raising Basel II defaults on a party's overall credit status.

As well as helping to foster a deeper understanding of the problem domain within the development team, further benefits can be extracted from our models once they have been drafted. For models written in the Z notation, a number of different tools exist to support analysis on the specifications they describe. The *f*UZZ[1] type-checker can be used to give rapid feedback on any syntax errors in a specification, something that is particularly useful if the team is still getting familiar with the notational aspects of a model. For this benefit alone, it is worth investing in basic LaTeX skills; when errors emerge at this stage of development, the effort spent here can be handsomely rewarded. Once the initial specification has been typeset, feedback loops can be created by animating the model and performing analysis as a collaborative activity. This affords the team an early opportunity to analyse the system's current or proposed behaviour through particular use-cases.

Customer involvement is essential during the modelling activities: at this stage of development, any changes they propose making to a system's requirements and design can be accommodated relatively cheaply. The presence of such a feedback loop, which can be facilitated by tools such as ProB[2], is catalytic to the level of engagement between customers and the development team during a system's delivery. In this chapter, we will demonstrate a number of techniques being used to conduct analysis on the models we have constructed. We will cover animation, formal reasoning, model checking, and show how the axiomatic definitions we gave in the Z notation can be used as the basis for constructing a high-level prototype in a functional programming language like Haskell.

[1] See [Spi92]

[2] `http://stups.hhu.de/ProB/`, accessed 16 June 2014



```
For any given state, we can identify the set of defaulted parties. Where the
system contains at least one default case, this is given by the following
schema:
%
\begin{schema}{ShowDefaultedParties0}
   \Xi DefaultSystem\\
    defaultedParties! : \power PartyId\\
\where
   defaultedParties! \subseteq parties\\
   \forall p:parties \setminus \dom partyDefaultCase @ p \notin defaultedParties!\\
   \forall p:\dom~partyDefaultCase @ p \in defaultedParties! \iff \\
       \t1 (\exists d:(defaultCases(partyDefaultCase~p)).defaultInstances @ \\
           \t2 (defaults~d).status \in MaterialDefaultStatus)
\end{schema}
%
If a particular state contains no default cases, then no parties should be
considered in default:
```

```
invariant_ok
defaultCases(DefaultCaseId1) = rec(defau
defaults(DefaultId1) = rec(status:APPROV
parties = {PartyId1,PartyId2}
partyDefaultCase(PartyId1) = DefaultCase
```

```
ProposeAdditionalDefault((DefaultId2|->rec(
ProposeAdditionalDefault((DefaultId3|->rec(
ProposeAdditionalDefault((DefaultId4|->rec(
ProposeAdditionalDefault((DefaultId5|->rec(
ProposeCloseDefault(DefaultId1)
ShowDefaultedParties-->{PartyId1}
```

```
ApproveDefaultTransition(Defa
ProposeFirstDefault((DefaultId
AddParty(PartyId2)
AddParty(PartyId1)
INITIALISATION({},{},{},{})
```

Figure 4.1: Using the ProB animator to explore the model

4.1 Exploring the Models with the ProB Animator

The ProB animator and model checker includes an extension for working with models written in the Z notation [PL07]. We can use this tool to explore our model, and examine the effect of proposing and approving default instances on a party's overall credit status. Figure 4.1 shows ProB being used in this manner. The user interface displays the properties contained in the state schema as different operations are performed. An advantage of using this is that there is no requirement for people involved in this analysis to be familiar with the model's notation. Animators like ProB can therefore serve as a bridge between a formal specification and an understanding of the system it describes.

While graphical user interfaces can be built on top of the ProB tool to construct prototypes of a system, the purpose of our analysis here is to determine the conditions under which counterparties enter default. A simple view of the state schema's properties is therefore sufficient for our needs. There is also a risk in building prototypes whose interface resembles the front-end of a system, that these get confused for a description of *how* the system will be used in practice, and proceed to influence its design accordingly. The operations included in our models were not chosen to align precisely with the interactions between a system and its users; they exist solely as a means of traversing different states in the model.

ProB can also be used to extract a graph of the current bindings in the state schema. We do this to generate the diagram shown in Figure 4.2, in which two parties have been added to the system. A default instance has been approved for *Party1*, and proposed for *Party2*. Using the tool to evaluate the *ShowDefaultedParties* operation for this state confirms that only *Party1* is in default. These representations of the system's state are also useful in analysing the relationships between different entities of a model. We need to ensure that a number of constraints governing the association of default instances with cases are satisfied, and being

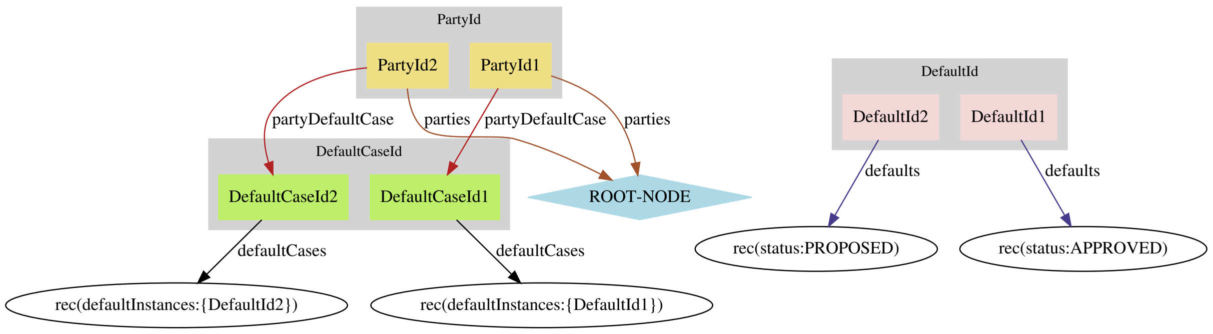


Figure 4.2: Graphical representation of a particular state

OPERATION	***defaults***
Init	$\varnothing$
ReceiveDefaultTrigger	$\{DefaultId1 \mapsto rec(defaultType: DEFERRED, status: PROPOSED)\}$
ProposeSyntheticDefault	$\{DefaultId1 \mapsto rec(defaultType: DEFERRED, status: PROPOSED),$ $DefaultId2 \mapsto rec(defaultType: SYNTHETIC, status: PROPOSED)\}$
ProposeDirectDefault	$\{DefaultId1 \mapsto rec(defaultType: DEFERRED, status: PROPOSED),$ $DefaultId2 \mapsto rec(defaultType: SYNTHETIC, status: PROPOSED),$ $DefaultId3 \mapsto rec(defaultType: DIRECT, status: PROPOSED)\}$
ApproveOpenDefault	$\{DefaultId1 \mapsto rec(defaultType: INDIRECT, status: APPROVED),$ $DefaultId2 \mapsto rec(defaultType: SYNTHETIC, status: PROPOSED),$ $DefaultId3 \mapsto rec(defaultType: DIRECT, status: APPROVED)\}$

Table 4.1: Sequence of operations leading to redundant default instances

able to validate this relationship for particular states provides confidence that our constraints have been selected appropriately.

As we have mentioned, the analysis facilitated by animating a model provides an efficient means of identifying a broad class of errors in a system's requirements. Two such issues emerged from our specification, the first of which concerns the automatic closure of synthetic defaults. In our second model, it was noticed that after a particular default case was closed, any remaining proposed synthetic defaults would persist in the case, despite them not contributing to the overall status of the counterparty directly. This scenario is demonstrated by the sequence of operations listed in Table 4.1, in which the corresponding sets of active default instances were extracted by evaluating the *defaults* relation in the state schema using ProB.

The system's original requirements did not describe how these default instances should be handled; rather than deferring this issue until it was discovered later (perhaps having wasted development effort, or introduced a bug into production), however, the *CloseRedundantSynthetics* schema was added to the model to clarify the intended behaviour as per the customer's advice. We can use ProB again to show that evaluating the same sequence of operations in the amended model now results in the correct set of defaults on approval of the direct instance:

$$\begin{aligned} defaults = \{&DefaultId1 \mapsto rec(defaultType : INDIRECT, status : APPROVED), \\ &DefaultId3 \mapsto rec(defaultType : DIRECT, status : APPROVED)\} \end{aligned}$$

A criterion of our modelling activities was that they should support customer engagement and iterative development; this example shows how animation can help achieve this during a model's analysis as well as its construction.

Another more serious issue in our specification was also discovered through analysing the model with ProB, and relates to the *ApproveOpenSynthetic* operation schema. Although associated deferred triggers may have been present at the time a synthetic default instance was proposed, these triggers could well have been closed by the time the instance comes to be approved. A trace showing the sequence of operations leading to this scenario is illustrated in Figure 4.3. In response to the issue highlighted by this feedback, we added a constraint to our model, preventing the synthetic default's approval in this scenario: all synthetic default instances must correspond to at least one active deferred trigger. The *ApproveOpenSynthetic* operation schema then becomes:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ApproveOpenSynthetic}} \\
\hline
\textit{ApproveOpenDefaultCommon} \\
\hline
\textit{deferredTriggers} \neq \varnothing \\
(\textit{defaults}\ d?).\textit{defaultType} = \textit{SYNTHETIC} \\
\hline
\end{array}$$

Once again, being in a position to address this before any code has been written presents the team with the best opportunity to implement the right solution; the fix is now embedded in the model, and can be incorporated directly into the system's design.

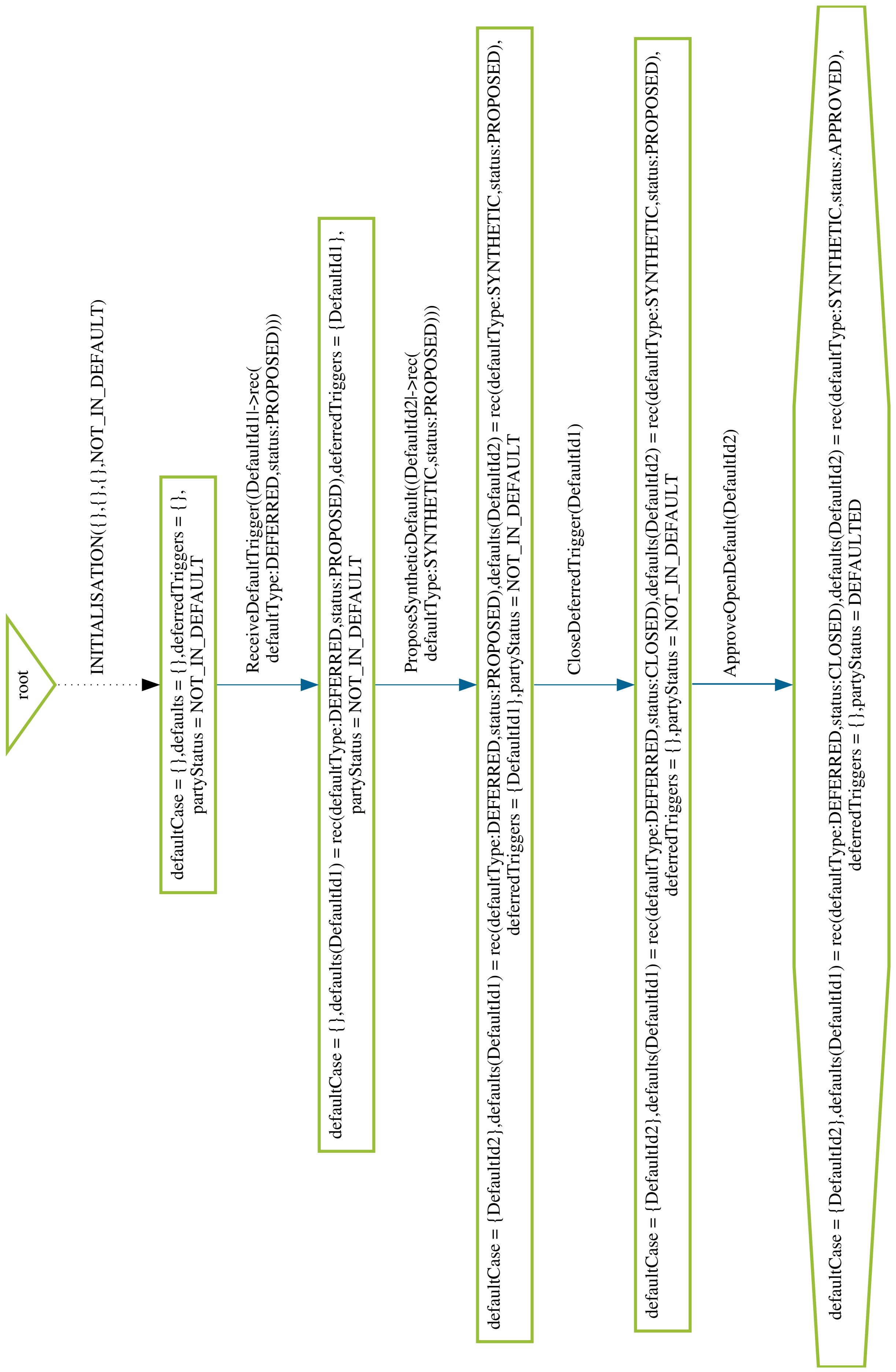
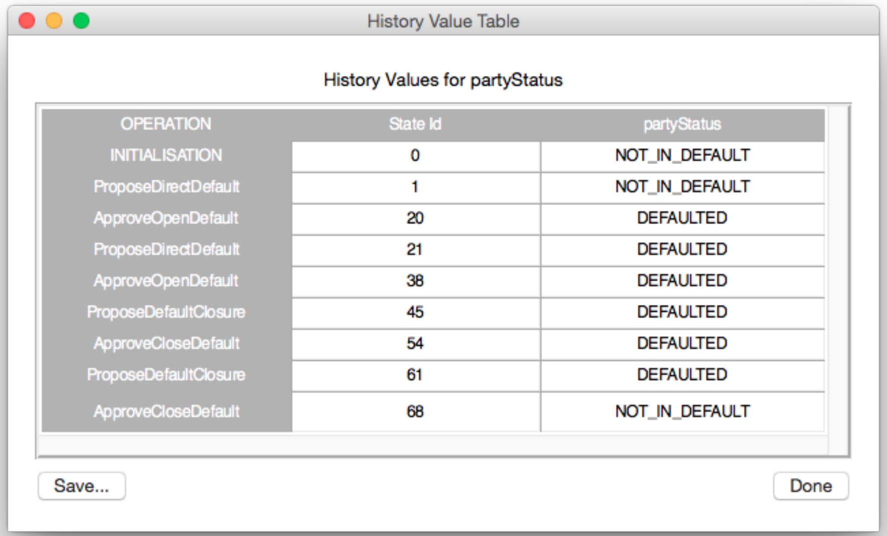

root
INITIALISATION({},{},{},NOT_IN_DEFAULT)
defaultCase = {},defaults = {},deferredTriggers = {}, partyStatus = NOT_IN_DEFAULT
ReceiveDefaultTrigger((DefaultId1|->rec(defaultType:DEFERRED,status:PROPOSED)))
defaultCase = {},defaults(DefaultId1) = rec(defaultType:DEFERRED,status:PROPOSED),deferredTriggers = {DefaultId1}, partyStatus = NOT_IN_DEFAULT
ProposeSyntheticDefault((DefaultId2|->rec(defaultType:SYNTHETIC,status:PROPOSED)))
defaultCase = {DefaultId2},defaults(DefaultId1) = rec(defaultType:DEFERRED,status:PROPOSED),defaults(DefaultId2) = rec(defaultType:SYNTHETIC,status:PROPOSED), deferredTriggers = {DefaultId1},partyStatus = NOT_IN_DEFAULT
CloseDeferredTrigger(DefaultId1)
defaultCase = {DefaultId2},defaults(DefaultId1) = rec(defaultType:DEFERRED,status:CLOSED),defaults(DefaultId2) = rec(defaultType:SYNTHETIC,status:PROPOSED), deferredTriggers = {},partyStatus = NOT_IN_DEFAULT
ApproveOpenDefault(DefaultId2)
defaultCase = {DefaultId2},defaults(DefaultId1) = rec(defaultType:DEFERRED,status:CLOSED),defaults(DefaultId2) = rec(defaultType:SYNTHETIC,status:APPROVED), deferredTriggers = {},partyStatus = DEFAULTED

Figure 4.3: State trace exhibiting the illegal approval of a synthetic default



History Values for partyStatus

OPERATION	State Id	partyStatus
INITIALISATION	0	NOT_IN_DEFAULT
ProposeDirectDefault	1	NOT_IN_DEFAULT
ApproveOpenDefault	20	DEFAULTED
ProposeDirectDefault	21	DEFAULTED
ApproveOpenDefault	38	DEFAULTED
ProposeDefaultClosure	45	DEFAULTED
ApproveCloseDefault	54	DEFAULTED
ProposeDefaultClosure	61	DEFAULTED
ApproveCloseDefault	68	NOT_IN_DEFAULT

Figure 4.4: Tracing the value of the *partyStatus* field through the test case

Another aspect of development in which formal models can be harnessed is testing. Agile methods have revolutionised the way that software is tested: it is now common for individual changes to be tested in parallel with their implementation, rather than deferring this to a later phase in the development life cycle. The 'eXtreme Programming' (XP) methodology [Bec00] takes this principle to its logical conclusion, advocating the practice of *Test-Driven-Development*. In this approach, tests are written before a feature is implemented, and are used to guide the design and behaviour of the delivered software. There is nothing preventing us from extrapolating this technique and applying the principles behind it to our use of formal methods. Using a formal model as a test oracle allows the team to draft high-level test cases, and once the behaviour they describe has been implemented, they can be used to check that certain properties of the system align with their expected values.

In addition to performing an ad-hoc exploration of the model and checking it for consistency, ProB can also be used in designing a system's acceptance tests. Figure 4.4 shows changes to the overall default status of a counterparty through a particular test case. The activity of writing acceptance tests broadens participation in a test-driven approach beyond those working directly on a system's codebase. Collaboration with business stakeholders and testers is essential in establishing a test suite that captures a system's intended behaviour accurately. Using a formal model to drive the design of test cases promotes this collaboration, by eliminating technical distractions from the conversation:

> "By working together to write these tests—*specifying collaboratively*—not only do the team members decide what behaviour they need to implement next, but they learn how to describe that behaviour in a common language that everyone understands." [WH12]

Tools such as *Cucumber*[3] enable acceptance tests to be written in natural language, and then executed through *step definitions* that translate the tests into operations on the system. Creating these step definitions for the operations in our formal model allows the test cases we

[3]`https://cucumber.io/`, accessed 01 March 2015

```
Feature: Managing a party's default status

    Scenario: Raise first default instance
        Given the party has "0" default instance(s)
        When a default instance is created
        Then the party's status should be "In Default"

    Scenario: Raise additional default instance
        Given the party has "1" default instance(s)
        When a default instance is created
        Then the party's status should be "In Default"

    Scenario: Close non-final default instance
        Given the party has "2" default instance(s)
        When a default instance is closed
        Then the party's status should be "In Default"

    Scenario: Close last default instance
        Given the party has "1" default instance(s)
        When a default instance is closed
        Then the party's status should be "Not In Default"
```

Listing 4.1: Acceptance tests for managing a party's default status

designed to be run automatically against the system. For example, in describing the effect of raising and closing default instances on party's overall default status, we can use the sequence of operations given in Figure 4.4 to arrive at the test cases shown in Listing 4.1. These tests support the team in building a system whose behaviour matches the customer's requirements, and enables non-technical members of the team to contribute in defining the system's tests.

In the traditional application of formal methods, a system's implementation is derived mechanically from its specification through successive iterations of *refinement*. In this paradigm, test-driven-development is redundant: every line of source code is governed entirely by the formal specification. When a system's design is derived through the refinement process, using test cases to validate its behaviour is unnecessary, since correctness will be verified through proof. In a more lightweight application of formal methods, however, a team can use the models to design acceptance tests, and then leverage these tests to maintain confidence in a system's functionality as they move from specification to code.

The practice of unit testing still has value in this approach, with lower-level tests helping to ensure both that individual components work as intended, and the system's design is ultimately driven from its requirements. The apparent disconnect between a system's specification and implementation is addressed by executing the acceptance tests, which can be extended and adjusted as requirements change. This complementary approach to testing a system, combining aspects of both Agile and Formal methodologies, extracts much of the value from a formal specification, without incurring the cost of performing lengthy proofs and refinement.

4.2 Reasoning about Legal Hierarchies

In the previous section, we demonstrated how animating the transitions of a model with a tool like ProB allows the team to explore state-based systems and validate the effects of particular operations. Part of the analysis required in our case-study involves reasoning about the struc-

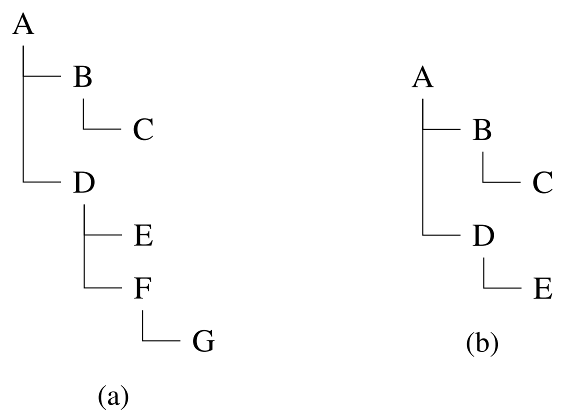


Figure 4.5: Applying the *removeSubtree* function to a legal hierarchy

ture of legal hierarchies, which underpins the business logic of cascaded gradings and cross defaults. Although ProB does provide rudimentary support for recursive definitions in Z, state-based animation is of limited use in reasoning about these structures. However, because the Z language has its roots in typed set theory, we have the rich toolset of discrete mathematics at our disposal when performing analysis on a specification's types.

Equational reasoning can be used to evaluate expressions involving the functions we defined previously on particular instances of the *LegalHierarchy* type. For example, if some legal hierarchy χ is represented by the tree (a) in Figure 4.5, then we can show that the expression $removeSubtree(F, \chi)$ evaluates to the tree represented in (b) as follows:

$$
\begin{aligned}
& removeSubtree(F, \\
& \quad Node(A, \{ \\
& \quad\quad Node(B, \{ \\
& \quad\quad\quad Node(C, \varnothing)\}), \\
& \quad\quad Node(D, \{ \\
& \quad\quad\quad Node(E, \varnothing), \\
& \quad\quad\quad Node(F, \{ \\
& \quad\quad\quad\quad Node(G, \varnothing)\})\})\})) \\
& = \hfill \text{[definition of } removeSubtree\text{]} \\
& Node(A, \{\{Node(B, \{Node(C, \varnothing)\})\} \cup \\
& \quad \{removeSubtree(F, Node(D, \{Node(E, \varnothing), Node(F, \{Node(G, \varnothing)\})\}))\}\}) \\
& = \hfill \text{[definition of } removeSubtree\text{]} \\
& Node(A, \{\{Node(B, \{Node(C, \varnothing)\})\} \cup \{Node(D, \{Node(E, \varnothing)\})\}\}) \\
& = \hfill \text{[properties of } \cup\text{]} \\
& Node(A, \{Node(B, \{Node(C, \varnothing)\}), Node(D, \{Node(E, \varnothing)\})\}) \\
& = \hfill \text{[identity]} \\
& Node(A, \{ \\
& \quad Node(B, \{ \\
& \quad\quad Node(C, \varnothing)\}), \\
& \quad Node(D, \{ \\
& \quad\quad Node(E, \varnothing)\})\}) \\
& \square
\end{aligned}
$$

As well as proving properties of *particular* hierarchies, we can construct proofs that hold for *all* members of the *LegalHierarchy* type, using the technique of structural induction. For example, we can prove that all legal hierarchies must contain at least one counterparty, that is, $\forall l : \mathit{LegalHierarchy} \bullet \mathit{size}\, l \geqslant 1$, as follows:

If P is some predicate defined as

$$P\_ : \mathbb{P}\, \mathit{LegalHierarchy}$$

then we may write the following structural induction principle[4] for the *LegalHierarchy* type:

$$\frac{\begin{array}{l}\forall p : \mathit{Party} \bullet P(\mathit{Node}(p, \varnothing)) \\ \forall p : \mathit{Party};\ l_1, \ldots, l_k : \mathit{LegalHierarchy} \bullet P(l_1) \wedge \ldots \wedge P(l_k) \Rightarrow P(\mathit{Node}(p, \{l_1, \ldots, l_k\}))\end{array}}{\forall l : \mathit{LegalHierarchy} \bullet P(l)}\ [\text{induction}]$$

This rule works because we can show that there can be no smallest instance of the *LegalHierarchy* type, L_{min}, for which P is false. If we were to assume on the contrary that

$$\exists L_{min} : \mathit{LegalHierarchy} \bullet \neg P(L_{min})$$

then by proving P for the base case, it follows that L_{min} must contain a non-empty set of subtrees; it cannot represent a standalone counterparty. Since by definition, none of these subtrees may contain the root node of L_{min}, they must be all be smaller than L_{min}, and hence P holds for each of them. By proving our inductive step, we show that since P holds for all subtrees of L_{min}, then $P(L_{min})$ must also be true, contradicting our assumption that there is some smallest legal hierarchy for which P is false. It follows that P holds for all members of the *LegalHierarchy* type.

In the context of analysing our model, we can use this principle to prove the statement

$$\forall l : \mathit{LegalHierarchy} \bullet \mathit{size}(l) \geqslant 1$$

by addressing the two antecedents of the rule as follows:

Base case: For any $p \in \mathit{Party}$,

$$\begin{array}{ll} \mathit{size}(\mathit{Node}(p, \varnothing) & \\ = & [\text{definition of } \mathit{size}] \\ 1 & \\ \geqslant & [\text{properties of } \geqslant] \\ 1 & \end{array}$$

Inductive step: To prove the case for $l = \mathit{Node}(p, \{l_1, \ldots, l_k\})$, where $p \in \mathit{Party}$, $l_i \in \mathit{LegalHierarchy}$ and $k > 0$, we assume that the theorem holds for each of the sub-hierarchies $l_1, \ldots, l_k$.

$$\begin{array}{ll} \mathit{size}\, l & \\ = & [\text{definition of } \mathit{size}] \\ \mathit{size}(l_1) + \ldots + \mathit{size}(l_k) + \mathit{size}(p, \varnothing) & \end{array}$$

[4]Note that this principle only holds for trees of finite depth; to prove properties for infinite-depth trees, we would need to use co-induction [Gor94]. Since in practice, the total number of counterparties within a legal hierarchy will always be finite (banks have only so many account managers), this principle is appropriate for proving properties of our *LegalHierarchy* type.

$=$ [definition of $\sum$]

$\sum_{i=1}^{k} size(l_i) + size(p, \varnothing)$

$=$ [definition of *size*]

$\sum_{i=1}^{k} size(l_i) + 1$

$\geqslant$ **[inductive hypothesis]**

$\sum_{i=1}^{k} 1 + 1$

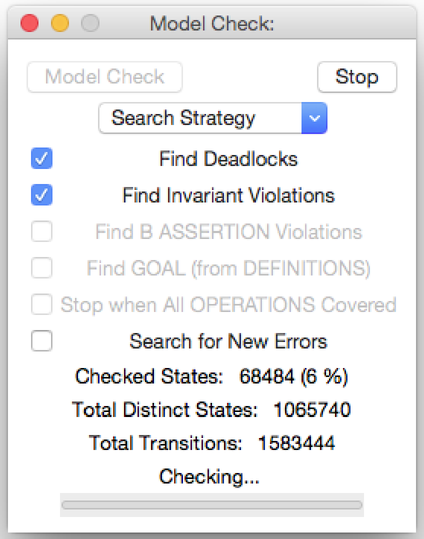
$=$ [properties of summation]

$k + 1$

$\geqslant$ [$k > 0$, by definition]

1

This completes the proof: from the induction principle given previously, we may conclude that the theorem holds for all $l \in LegalHierarchy$. □


Model Check:

Model Check | Stop

Search Strategy

[✓] Find Deadlocks
[✓] Find Invariant Violations
[ ] Find B ASSERTION Violations
[ ] Find GOAL (from DEFINITIONS)
[ ] Stop when All OPERATIONS Covered
[ ] Search for New Errors

Checked States: 68484 (6 %)
Total Distinct States: 1065740
Total Transitions: 1583444
Checking...

Figure 4.6: Model checking the specification with ProB

While formal reasoning provides a powerful means of analysing our mathematical models, the cost of performing more complicated proofs than the ones we have described can be prohibitive. Although the complete verification of a system's properties may be justified, essential even, for certain modelling scenarios, the usefulness it lends to the analysis in our case study does not warrant its cost of execution. A sufficient degree of confidence in the consistency of our specification can be achieved more cost-effectively by *model checking*.

Model checkers are available for a number of different languages, and the ProB tool supports this form of analysis for models written in Z. Figure 4.6 shows this tool being used to search for states that violate the constraints of our model for deferred defaults. Running this analysis for ten minutes on a five-year-old laptop lead to almost seventy thousand states being checked, none of which represent counterexamples to the constraints in our state schema. Compared to the amount of work required to *prove* this consistency each time a change to the model is introduced, performing the analysis in this manner delivers much of the same value at a fraction of the cost. Given the need for modelling activities to be as lightweight as possible, a judicious balance between formal reasoning—reserved perhaps for verifying critical aspects of a system—and other approaches (such as model checking) is therefore required in selecting the most appropriate techniques for analysis.

4.3 Prototyping with Haskell

Having animated our first two models with *ProB*, we can also prototype[5] and analyse the functions defined for legal hierarchies and cross defaults. As a high-level, functional programming language, Haskell is well-suited to this task: its support for algebraic datatypes provide a natural fit for the recursive structures defined in our model. For example, rose trees in Haskell can be declared in a similar way to the corresponding free type in Z:

```
data RoseTree a = Node {leaf :: a, subTrees :: [RoseTree a]}
```

An equivalent type is provided in the *Data.Tree* module of the community-maintained *containers*[6] package, and we will leverage the functions provided there for displaying these structures on the command-line. Since *Tree* is a type constructor in Haskell, we need to supply a type parameter to describe the nodes in our legal hierarchy. We do this with the *PartyInfo* type, which we will refine later in our prototype.

```
type LegalHierarchy = Tree PartyInfo
```

At this stage, our immediate concern lies with the *structure* of legal hierarchies; trees of strings therefore represent a suitable abstraction. We use a type synonym to declare this:

```
type PartyInfo = String
```

With these definitions in place, a simple legal hierarchy can be defined as follows:

```
simpleHierarchy = Node "parent" [
                        Node "child 1" [],
                        Node "child 2" [
                             Node "grandchild 1" []]]
```

Haskell's interactive environment, *GHCi*, can then be used to verify that particular instances of the *LegalHierarchy* type structurally resemble the legal hierarchies from the business domain:

GHCi

```
λ> putStrLn $ drawTree simpleHierarchy
parent
|
+- child 1
|
`- child 2
   |
   `- grandchild 1
```

The homology between specification and code extends to operations on the *LegalHierarchy* type. Functions corresponding to the *addSubtree* and *removeSubtree* definitions in our Z model can be implemented in Haskell with minimal effort required in their translation:

```
addSubtree                 :: Eq a ⇒ Tree a → a → Tree a → Tree a
addSubtree new x (Node y t)
```

[5]Complete listings for the code outlined in this section are provided in Appendix A.

[6]`https://hackage.haskell.org/package/containers-0.5.0.0`, accessed 30 October 2014

```
    | x ≡ y                                = Node y (new : t)
addSubtree new x (Node y (t : ts))
    | x ∈ flatten t                        = Node y ((addSubtree new x t) : ts)
    | x ∉ subsidiaries (Node y (t : ts))   = Node y (t : ts)
    | otherwise                            = addSubtree new x (Node y (ts ++ [t]))

removeSubtree                              :: Eq a ⇒ a → Tree a → Tree a
removeSubtree x (Node y (t : ts))
    | x ≡ root t                           = Node y (ts)
    | x ∈ flatten t                        = Node y ((removeSubtree x t) : ts)
    | x ∉ subsidiaries (Node y (t : ts))   = Node y (t : ts)
    | otherwise                            = removeSubtree x (Node y (ts ++ [t]))
```

Again, evaluating these functions in the console confirms that they behave as expected. This also provides an opportunity to get feedback on the model from stakeholders who might be more familiar with a visual representation of the domain objects:

GHCi

```
λ> let X = Node "A" [
λ|           Node "B" [],
λ|           Node "C" [
λ|             Node "D"[]]]
λ> let Y = Node "E" [
λ|           Node "F" [],
λ|           Node "G" []]
λ> let sumXYTree = addSubtree X "C" Y
λ> putStrLn $ drawTree sumXYTree
A
|
+- C
|  |
|  +- E
|  |  |
|  |  +- F
|  |  |
|  |  `- G
|  |
|  `- D
|
`- B

λ> putStrLn $ drawTree $ removeSubtree "C" sumXYTree
A
|
`- B
```

Similarly, the *size* function from our Z model can be written almost identically in Haskell:

```
size                  :: Tree a → Integer
size (Node x [])      = 1
size (Node x (t : ts)) = size t + size (Node x ts)
```

This pattern of recursion, however, can be expressed more concisely with the higher-order function *map*:

$$size' \ (Node\ x\ ts) = 1 + sum\ (map\ size'\ ts)$$

As well as writing code that closely resembles definitions in the Z model, we can also build on these functions and prototype new operations directly in Haskell. For example, composing the *addSubtree* and *removeSubtree* functions in the following manner yields a new operation for moving a subtree between two nodes in the same legal hierarchy:

```
moveSubtree                    :: Eq a ⇒ a → a → Tree a → Tree a
moveSubtree srcNode targetParent tree
   | srcNode ≡ targetParent = tree
   | otherwise                  = addSubtree (subtree srcNode tree)
                                      targetParent (removeSubtree srcNode tree)
```

Having created an elementary prototype for manipulating the structure of legal hierarchies, we can now turn our attention to refining the type used for their constituent nodes. In contrast to Z, cyclic types are permitted in Haskell. This means that a separate type to capture the structure of grading hierarchies is not required; this information is now encoded entirely between the *PartyInfo* and *Grading* types. To support this, the *PartyInfo* type is enriched with details of the counterparty's grading:

```
data PartyInfo = PartyInfo String Grading deriving (Eq)
```

As in our Z model, gradings can be empty, standalone or cascaded. We represent these distinct cases using value constructors in the *Grading* data type:

```
data Grading = Grading
   | Standalone Integer
   | CascadedGrading {cascadeParent :: PartyInfo, notch :: Integer} deriving (Eq)
```

So that we can represent individual legal hierarchies as trees on the command line using the *drawTree* function, we define an instance of the *Show* class for the *PartyInfo* type.

```
instance Show PartyInfo where
  show (PartyInfo p Grading)                    = p
  show (PartyInfo p (Standalone x))             = p ++ " [" ++ show x ++ "]"
  show (PartyInfo child (CascadedGrading parent n)) =
    let cascadedGrade = show $ fromJust $
         grade (PartyInfo child (CascadedGrading parent n))
      in child ++ " [" ++ cascadedGrade ++ "] - Cascaded from " ++
         party parent ++ " with notch = " ++ show n
```

Since our *LegalHierarchy* type has an instance of *Functor*, we can apply the *show* function to each node of a particular hierarchy using *fmap*. For convenience, we encapsulate this in the *formatHierarchy* function to display grading hierarchies as text on the command line. Input and output are side-effects—forbidden in a *pure* functional language like Haskell—so the *IO* monad is used here to write output to the terminal.

```
formatHierarchy    :: LegalHierarchy → IO ()
formatHierarchy x = putStrLn $ drawTree $ fmap show x
```

We can test this in GHCi using the *cascades* example hierarchy defined in Appendix A.3:

GHCi

```
λ> formatHierarchy cascades
Party A [5]
|
+- Party B [3] - Cascaded from Party A with notch = -2
|
+- Party C [6] - Cascaded from Party A with notch = 1
|   |
|   `- Party D
|
`- Party E [11]
    |
    `- Party F [14] - Cascaded from Party E with notch = 3
```

Having built a simple prototype of grading hierarchies, we can now use Haskell to explore the business logic for cross defaults. In order to do this, we need to be able to compare the credit grades of different parties within a particular legal hierarchy. Since not all parties may have been graded, the *grade* function defined below returns a *Maybe* type to handle this. The absence of *null* values in Haskell makes programs easier to reason about, since it can be inferred directly from a function's type whether a value will always be returned.

```
grade                                            :: PartyInfo → Maybe Integer
grade (PartyInfo p Grading)                      = Nothing
grade (PartyInfo p (Standalone x))               = Just x
grade (PartyInfo p (CascadedGrading parent notch)) = Just (parentGrade + notch)
  where parentGrade = fromJust (grade parent)
```

The identity of the party whose credit grade is being inherited is determined in the *parent* function using pattern matching. Again, the *Maybe* error monad is used to define the target type here, since standalone and ungraded parties are not associated with a parent:

```
parent                         :: Grading → Maybe PartyInfo
parent (CascadedGrading p _)   = Just p
parent _                       = Nothing
```

A list comprehension is used to determine the child parties of a grading cascade:

```
cascadeChildren       :: PartyInfo → LegalHierarchy → [PartyInfo]
cascadeChildren p h = [c | c ← flatten h, Just p ≡ parent (grading c)]
```

With these definitions in place, the *ShowCrossDefaults* operation from our Z model can be written in a similar fashion as a Haskell function, with both definitions using set or list comprehensions to describe the conditions under which cross default instances are raised:

```
crossDefaults                 :: PartyInfo → LegalHierarchy → [PartyInfo]
crossDefaults p h             = filter (≢ p) (crossDefaults' p h)
crossDefaults'                :: PartyInfo → LegalHierarchy → [PartyInfo]
crossDefaults' (PartyInfo c (CascadedGrading p n)) h
  | grade childInfo ⩾ grade p = p : [x | x ← cascadeChildren p h]
  | otherwise                 = [x | x ← cascadeChildren p h, grade x ⩽ grade childInfo]
  where childInfo = (PartyInfo c (CascadedGrading p n))
crossDefaults' p h            = cascadeChildren p h
```

Applying the *party* function to each member of the returned list yields a list of the parties that will be put into cross default. This function is analogous to the *ShowCrossDefaults* operation schema in our Z specification:

```
crossDefaultParties p h = map party (crossDefaults p h)
```

Returning to the grading hierarchy presented as an example in Chapter 2, we can now use our Haskell prototype to verify that the expected sets of cross default instances are generated for the same scenarios:

GHCi

```
λ> :l Examples.hs
Ok, modules loaded: Examples, CrossDefaults, LegalHierarchies.
λ> crossDefaultParties party_e chapter2Hierarchy
["Party H","Party J","Party F","Party C"]
λ> crossDefaultParties party_h chapter2Hierarchy
["Party E","Party J","Party F","Party C"]
λ> crossDefaultParties party_c chapter2Hierarchy
["Party J"]
```

In building this prototype, we have been concerned only with *what* the constituent functions do, not *how* their behaviour is implemented. The level of abstraction provided by this declarative approach concentrates our attention on the system's properties, rather than the prototype's execution. We have alluded previously to the danger in building models that closely resemble a finished product, and there is also a risk that the behaviour diverges from a specification where prototypes are defined in terms of their execution. The ability to translate—or even build—a model directly in Haskell gives us confidence that the resulting prototype accurately describes the system's intended behaviour. Confidence, rather than proof, is again sufficient here, given our modelling criteria. The approach we have chosen can be performed at a fraction of the cost of formally refining a specification into executable code.

Formal reasoning on this prototype can be conducted in much the same way as we did previously with Z: Haskell's referential transparency and support for algebraic datatypes make this significantly less arduous than with object-oriented or imperative languages. Furthermore, Haskell can also be used for performing analyses similar to the model checking we did with ProB, using the *QuickCheck*[7] library. Following from the same premise as our inductive proof that if some theorem is false, there will be a smallest counterexample to prove this is the case, programs can be written in Haskell to search for these counterexamples mechanically. In order to perform this search, a test must be able to access arbitrary instances of the underlying data structures. QuickCheck provides the *Arbitrary* class to support this, and to help us reason about the *LegalHierarchy* type, we instantiate this class to define the following generator:

```
type IntTree = Tree Int

instance Arbitrary IntTree where
  arbitrary = sized sizedArbIntTree

sizedArbIntTree   :: Int → Gen IntTree
sizedArbIntTree 0 = do
  c ← choose (minBound, maxBound)
  return $ Node c [ ]
sizedArbIntTree n = do
  c            ← choose (maxBound, minBound)
  subtreeCount ← choose (0, n − 1)
  subtrees     ← vectorOf subtreeCount (sizedArbIntTree (n − 1))
  return $ Node c subtrees
```

Again, since we are interested only in the structure of legal hierarchies at this point, trees of integers make suitable subjects for our analysis. The generator given above leverages the *sized* combinator, giving us a degree of control over the generated hierarchies through the *size* parameter. We can be sure that only finite trees will be generated, because with each recursive call of the *sizedArbIntTree* function, the maximum number of permitted subtrees will tend towards zero. With a suitable generator defined, properties of our model can now be tested. Previously, we used proof by induction to show that all legal hierarchies must contain at least one counterparty; we can now configure QuickCheck to test this property automatically:

GHCi

```
λ> let prop_size x = property $ size x >= 1
λ> quickCheckWith stdArgs {maxSize = 10} prop_size
+++ OK, passed 100 tests.
```

Although we cannot gain the same level of confidence from a test as we can from mathematical proof, the independence of tests from their subject means that we can make changes

[7]https://hackage.haskell.org/package/QuickCheck, accessed 29 December 2014

to the *size* function and re-run the tests to validate that the behaviour remains the same. Tests that are written during the initial modelling activities can be also be run during development. By testing the properties of a system, we can show that the behaviour exhibited by a model has been faithfully replicated in an implementation. Agile teams typically employ *continuous integration*, a practice in which tests are run every time the code changes. The inclusion of property-based tests in their continuous build provides the team with a feedback loop for their changes in terms of the impact they have on the system's constraints. As before, it is a question for the development team to determine which properties should be included in these tests, and which—if any—require formal proof. As Dijkstra acknowledged, "Program testing can be used to show the presence of bugs, but never to show their absence." [Dij72].

The results of the analysis presented in this chapter provide further evidence of the value that formal methods can bring to an Agile project. Leveraging the available tools, the key to maximising this value lies in the lightweight approach that was adopted throughout the analysis. Although this represents a significant departure from the traditional application of formal methods, it proved a good fit for Agile development. Returning to the core principles underlying Agile methods, the use of animation, model checking and prototyping enabled the team to respond quickly to change, and work more closely with their business counterparts throughout a system's delivery. The ability to accommodate change in this manner is recognised explicitly as a principle of Agile development, and was highlighted in Chapter 2 as an essential criterion in our use of formal methods:

> "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage." [HF01]

In the following chapter, we will discuss the integration of these two disciplines in greater depth, investigate what barriers still exist to the more widespread adoption of formal methods, and consider whether further research might help eliminate these barriers and illuminate new ways of applying formal techniques in Agile environments.

5 Discussion

5.1 Role of Formal Methods in Agile Environments

In the previous two chapters, we showed how the Z notation can be used to model a particular business domain at different levels of abstraction. We began by constructing a simple model, yielding a high-level yet accurate description of Basel II default instances and their effect on the credit status of associated counterparties. This provided a basis for understanding the role of default cases as containers for default instances, and the conditions under which they are opened and closed. This initial model was then extended to support the introduction of deferred and cross defaults, with Haskell being used to prototype functions defined on the underlying data structures. In this section, we will discuss the value that constructing and analysing these models brings to the development environment, and in particular, how the techniques of formal methods can best be applied to support the principles of Agile development.

Perhaps the most valuable contribution from the act of constructing the models themselves lies in helping the development team elicit requirements from the customer. Formal modelling allows discussion of the problem domain to be held in a common language, helping prevent a descent into jargon from either side of the table. As a universal language, mathematics serves well to describe the intended behaviour of a system: ambiguous requirements can be identified early, and any language barriers within a team need not put its members on an unequal footing. A case study conducted by the Institute for Software Research at Carnegie Mellon University had similar findings:

> "Customers, who were domain experts but neither seasoned programmers nor mathematicians, found it easier to understand the [formal] model and troubleshoot problems than code from previous systems." [JKM+09]

Using a language such as Z is particularly effective in distributed teams, where discussions around a system's requirements may take place in a number of different natural languages, depending on who is present. Recent research in sociology [Pag07] suggests that diverse teams tend to perform better than those more homogeneous in their constitution, and it is now common in large organisations for even the smallest teams to be distributed across geographies, languages and time-zones. The distinct semantic nuances of English and Hindi, for example—while contributing to their richness and expressivity—may mean that translation between the two yields a significant difference in meaning. Specifying some aspect of a system's requirements in terms of discrete mathematics resolves this issue effectively, without imposing one set of linguistic or cultural norms over another. This helps to maintain a higher level of engagement and a common understanding from all members of a team, regardless of how it is composed.

In our initial model, the system is described at a very high level of abstraction. Details about the entities themselves are not relevant at this stage, so identifiers serve adequately as placeholders for the counterparties managed in the system. Should we later wish to refine this abstraction by introducing a *Party* type in our model, this could easily be accomplished while maintaining consistency with the original specification. A more precise definition of a counterparty might be given in the following schema, which assumes that we have declared the *Name*, *Country* and *ContactDetails* types separately:

Party

$$
\begin{array}{l}
\mathit{name} : \mathit{Name} \\
\mathit{accountManager} : \mathit{Name} \\
\mathit{numberOfEmployees} : \mathbb{N} \\
\mathit{countryOfOperation} : \mathit{Country} \\
\mathit{countryOfIncorporation} : \mathit{Country} \\
\mathit{turnover} : \mathbb{Z} \\
\mathit{correspondence} : \mathit{ContactDetails} \\
\hline
\mathit{name} \neq \mathit{accountManager}
\end{array}
$$

While seeking to understand how default cases are managed, the level of detail provided in this schema adds no further value, however. This definition even introduces a risk that the limited time developers and customers have together is spent discussing whether this detail is correct, rather than achieving a mutual understanding of the mechanics underpinning default management. Accuracy, rather than precision, allows the most pertinent problems to emerge while discussing a system's requirements. Selecting the right level of abstraction is therefore essential to a model's value, whatever its notation or degree of formality.

A first draft of the model of default instances and cases took a couple of hours to construct, and was done initially with pen and paper. This activity is something that could easily be done as a team on a whiteboard, without the need for expensive tools or software packages. Once the fundamental properties of the model had been agreed upon, the specification was typeset so that it could be type-checked and explored with tools such as *f*UZZ and ProB. Involving stakeholders with the construction of a prototype leads to a shared sense of ownership of the model, further increasing the level of customer engagement in a system's delivery. "Customer collaboration over contract negotiation" [HF01] is a key aspect of Agile development, and the construction of formal models as a collaborative activity yields benefits from each of the two disciplines.

We have alluded previously to the value of short feedback loops in delivering systems that serve their intended purpose effectively. Animating a model to prototype some aspect of a system's functionality is an efficient means of creating such a loop. The ability to run through various scenarios with the customer and validate changes in the system's state allows issues with its requirements or design to be identified early. A key principle of Agile development is that when problems arise, they should be made visible to the team. Sandro Mancuso highlights the importance of this in *The Software Craftsman: Professionalism, Pragmatism, Pride*:

> "Narrowing the feedback loop helps us to make problems visible sooner, allowing us to inspect and adapt quickly." [Man14]

This visibility encourages a culture of openness, and helps to minimise the damage that errors cause to a project. As a result of the analysis described in Chapter 4, two issues in the specification were addressed, both of which could have lead to costly errors had they leaked into the system's implementation. In helping the team to identify and handle these issues quickly, a significant benefit of combining formal and Agile practices is made evident.

The models we created in this case study were also used in designing acceptance tests for the system. Interrogating them to determine the expected values for various properties of the system helps to ensure that the tests are consistent with the specification. This allows the team to begin writing high-level test cases for the system before development begins, giving confidence that the correct behaviour will be implemented in the delivered system. The acceptance tests that a team creates form a mould for the system's implementation, and ensuring that the tests describe the intended behaviour accurately goes some way towards establishing the relationship between requirements and code delivered traditionally by formal methodologies through refinement. Although the practice of test-driven-development might seem at odds with the ideals of formal methods, our goal is ultimately to deliver the right system under the constraints of an enterprise environment. This is more important than following the guidelines of a particular methodology for its own sake, and since adopting a more lightweight, hybrid approach to formal methods can support us in achieving this goal, the value in doing so is clearly apparent.

Agile methods have traditionally eschewed the use of substantial documentation, particularly as a tool for capturing a system's requirements or describing its implementation: "Working software over comprehensive documentation." [HF01]. User stories, containing *just enough* information to describe a particular requirement, are preferred to lengthy business requirements documents, and stand in stark contrast to traditional formal specifications. The value of maintaining separate artifacts to describe a system's behaviour is also challenged: "Good code is its own best documentation." [McC04]. This view, however, is based on an assumption that formal models and user stories will always play the same role in terms of documenting a system. We have seen in our case study that they can coexist: user stories serve best as a placeholder for discussing particular requirements with stakeholders, while formal models facilitate conversations around a system's properties with clarity and in a common language. In terms of documenting how a system works, it is true that the source code—if well written—performs this function with the greatest degree of accuracy: it is tautologous to say that a system's code cannot get out of sync with its implementation. However, a description of a system's properties is not always easy to extract directly from code. This difficulty becomes particularly acute for systems written in imperative languages, or those which are coded with the aim of achieving optimal performance over legibility.

We propose that the selective application of formal methods to create models describing these properties can address this deficiency effectively. In the system underlying our case study, a significant amount of effort would have been required in reading the codebase to extract the conditions under which default cases are created; for one thing, the logic is distributed across multiple services and layers in the system's architecture. The first formal model that we constructed exposes these properties at a much higher level of abstraction, and one that is vastly more useful in reasoning about the system's behaviour. As a description of *how* certain behaviour in a system is implemented, there is no substitute for reading the source code directly; as a description of *what* that behaviour is, however, formal models represent an extremely powerful tool.

Given the importance that Agile methodologies place on the collaborative nature of building software, any shared documentation that a team produces must be accessible to all of its members. A formal model, if written at an appropriate level of abstraction, can serve as an excellent source of this documentation. The results from the study conducted at Carnegie Mellon University support this assertion:

> "For several months the model served as the primary medium of communication among customers and developers." [JKM$^+$09]

In the place of traditional requirements documents, leveraging a system's models and acceptance tests provides an effective solution to describe both a system's expected behaviour and its

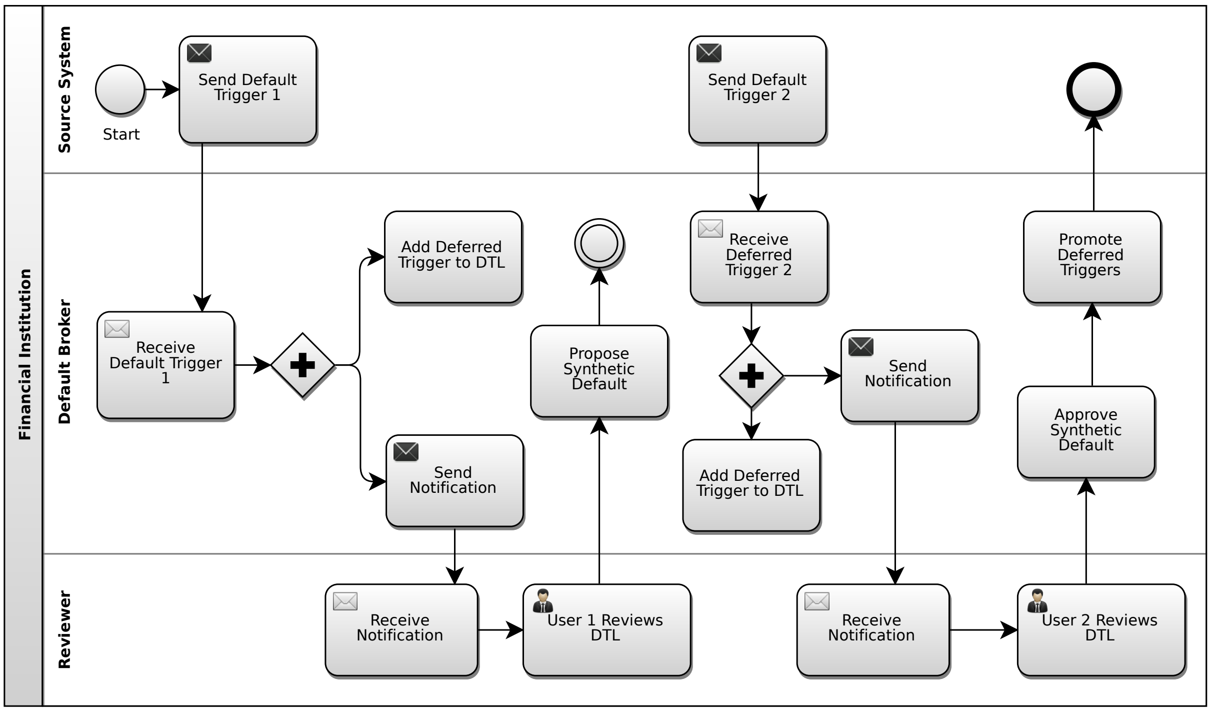


Figure 5.1: BPMN 2.0 model of a default deferral scenario

implementation, without incurring the overhead of checking and maintaining separate requirements documents.

Although as a device for capturing a system's properties, models constructed using formal methods are perhaps the most effective, this is not to suggest that they cannot be used alongside other approaches to modelling within a project. The diagram in Figure 5.1 uses the BPMN 2.0 notation to describe the roles of different actors in a particular business scenario. This is especially useful in understanding what input is expected from each actor, and how their interactions are orchestrated in the system. However, this model only describes a single use-case; we cannot deduce what would happen in this scenario were the default instances to be rejected, for example. By contrast, a formal model not only allows us to reason about new scenarios, but also provides a means to evaluate the system's state at each stage. We have shown how this can be used in designing test cases to confirm with the customer that the properties identified in a model do indeed capture their requirements for the system's behaviour.

As well as using formal methods to perform Agile practices, there are benefits to be gained from integrating the disciplines in the other direction: applying Agile principles to the practices of formal analysis. In our case study, this was particularly evident during the formal reasoning that was conducted. Fulfilling proof obligations is a central theme of formal methods in their traditional application, in which each successive refinement of a specification must be accompanied with mathematical proof of its correctness. Although these obligations no longer apply in our more lightweight approach, formal reasoning remains a valuable tool in deriving properties of the system from its constituent definitions.

Formal methods can therefore play an effective role in Agile environments, albeit one which differs significantly from their traditional application. With a reduced emphasis on proof and refinement, the use of mathematics as a modelling language—and the subsequent analysis that this supports—equips the development team with an extremely powerful set of tools in their efforts to deliver the right system to the customer. This lightweight, ad-hoc approach to using formal methods fits naturally in an Agile development environment; even more so than in the waterfall styles of project management with which they are commonly associated. Although a high-level model to derive the relationships between the primary entities of a system may be constructed before coding formally begins, it is neither necessary nor practical for all of the modelling activities to be completed up front. As requirements change, or a particular aspect of a system needs to be clarified, working in conjunction with the customer to construct a mathematical model provides the development team with an opportunity to reason about the system's properties, and react to any issues that emerge, with a degree of confidence that is beyond the reach of other methods.

5.2 Barriers to Widespread Adoption

Having seen the benefits that formal methods can bring to a project, it might seem surprising that their industrial application remains largely confined to a niche. There are a number of reasons for this, although few which represent genuine impediments to the introduction of formal methods in a project.

Perhaps the greatest barrier to their widespread adoption is the lingering perception of formal methods as a heavyweight and necessarily holistic approach to software development. A key factor in this perception is the association of formal methods with lengthy proofs, and the significant overhead to a project that this represents. Three incremental levels to which formal methods can be applied are identified in [BH95], beginning with their use solely as a specification language, adding verification and refinement, and culminating in machine-checked proofs. Using formal methods primarily as a means for eliciting and clarifying a system's requirements—the approach taken in the case study outlined in this report, and covered by the first level—does not preclude the use of techniques from subsequent levels, however. By revising the way that formal methods are applied, an Agile team can not only manage the overhead that these methods incur, they can even leverage the techniques of formal analysis to align their practices more closely with the principles of Agile development. Increased customer engagement and accurate, timely feedback are just two notable benefits to be gained from this approach.

We should be careful, however, in making the assumption that the use of formal methods in a project necessarily leads to longer delivery times. In their discussion of specific examples of this accusation being levelled at formal methods, Bowen and Hinchey note that

> "...these projects were not delayed due to the lack of ability of the formal methods specialists, but rather a lack of experience in determining how long development *should* take." [BH94]

One of the reasons for this myth's prevalence is the fact that formal methods can help make problems visible much earlier in a project's life cycle. If highlighting an issue with a system's design or requirements means that a particular feature is delivered later (because the team has decided to increase development time instead of compromising their product's quality), then this should be recognised as a benefit of using formal methods rather than a drawback.

Another perceived obstacle to the adoption of formal methods stems from the mathematical nature of their notations. Although a certain amount of expertise may be required to satisfy the proof obligations dictated by the traditional use of formal methods, none of the analysis described in this report presupposes any more mathematical proficiency than most software engineers draw from already in their daily work. Anyone who has worked with a collections library or written a simple SQL query will be familiar with the concept of a set, and the entire Z language (and supporting 'mathematical tool-kit') can be built solely from this concept

[Spi89]. Development teams that are using functional programming languages will recognise immediately the axiomatic definitions and set comprehensions present in a model; we showed in building the Haskell prototype described in Chapter 4 how easily Z definitions can be translated into a high-level functional programming language.

Difficulties a team faces in using a particular formal notation may be exacerbated, however, if the notation is not accompanied by a suitable narrative. This narrative is an essential component of any formal specification, and can be used to help explain design decisions as well as elucidating the mathematical elements of a model. Assessing the impact of their original report [BH95], Bowen and Hinchey re-emphasise the need for formal models to contain adequate documentation:

> "...we have always advocated augmenting formal specifications with natural language narrative. A proper paper trail is critical. Without it, the organisation loses the benefits of abstraction and might even lose useful information." [BH06]

The formal and informal elements of a specification must be carefully balanced. Mathematical notation represents an extremely powerful way to express an idea, removing the ambiguity and unnecessary detail that can hinder informal documentation. However, if this notation is impenetrable to a wider audience, a model's value will be limited. An accompanying narrative should therefore be used to help clarify and disseminate the ideas it contains.

It has been claimed that formal methods yield less value in projects concerned with large, complex systems: "Many of the most popular formal methods do not scale up to practical-size problems." [LG97]. When a system's requirements are unknown, or their scope means that the underlying business logic cannot easily be captured in a single model, it is true that attempts to write a comprehensive specification and refine this into executable code are more likely to fail. Even if the time needed to develop and prove a specification is available, systems whose every line of code must be derived by multiple layers of refinement are difficult to modify when requirements change. Modelling scenarios where these circumstances are acceptable are atypical, however, particularly in enterprise environments not concerned with the delivery of safety-critical systems. As we have shown in this case study, by aligning the application of formal practices more closely with Agile principles—and setting out realistic expectations for what they will accomplish—formal methods can be highly effective in helping teams to deliver the right solutions.

A final challenge that formal methods face in gaining widespread acceptance lies in generating the critical mass needed to bring them to the attention of a wider audience. Software engineering is an extremely fast-moving industry, and in order for techniques and methodologies to avoid becoming passing trends, their promised benefits need to be supported with evidence of their success. An action to take forward from this report will be to share the results of the case study, and suggest to other teams within the author's organisation that they consider adding formal methods to their development ecosystem. The adoption of Agile methods within this organisation required both time and a shift in the company's culture before gaining widespread adoption, and although the use of formal methods arguably represents a less dramatic change in terms of how development teams operate, the popularity of formal methods will depend directly on the advocacy of those who have gained value from them.

In this section, we have discussed a number of challenges faced by practitioners of formal methods looking to broaden their use. Although for the majority of projects, the promises of guaranteed correctness and error-free systems may be somewhat overstated, we have shown that there are a number of different ways in which formal methods can be usefully harnessed. Adjusting our expectations and ensuring that their application is suitable for the constraints of a particular project, shows that many of the misconceptions surrounding formal methods are unfounded, or rooted in an approach that is only relevant for a subset of modelling scenarios. As software professionals, it is our responsibility to ensure that these prejudices do not influence decisions around how problems are tackled; the right tools should be chosen regardless of what is understood to be common practice elsewhere. Given the evidence of their value and applicability to Agile development environments as outlined in this report, a serious reappraisal of formal methods' industrial use is long overdue.

5.3 Areas for Further Development

Having identified a significant gap between the perception of formal methods and the benefits they can bring to a project, we must consider what actions can be taken to address this, and facilitate further the integration of formal methods in Agile environments. We evaluate first the adequacy of support provided by the tools currently at our disposal, and scope for further development in this area.

With less emphasis on refinement and machine-checked proofs in our revised approach to formal methods, the overall reliance on tools is reduced. A significant amount of progress can be made using only the notational elements of a formal method to capture a system's requirements. No specific tooling is required to perform this activity, and it is true that more 'lo-fi' techniques can often be the most effective. Sophisticated tools introduce distractions, and can even dictate how a team operates, according to their idiosyncrasies or limitations. This is evidenced by the number of Agile teams who favour using sticky notes to record user stories over specialised bug-tracking software.

Despite their reduced significance in our revised approach, tools still played a useful role in our case study, particularly once the models had been typeset. Animating the specification provided an economical way to create a feedback loop, and presented a view of the models that was immediately recognisable to the customer. The ProB animator and model checker performed this function adequately for the modelling scenario in our case study. A number of further enhancements would still be beneficial, however, and it is encouraging to see that development of a new version[1] of the tool remains active. One particularly interesting feature that is currently under development is the introduction of a Java API. Given the almost ubiquitous nature of the Java platform in industry, this will allow teams to conduct model checking and animation using their existing technology stacks and skill sets. Creating web-based tools to interact with an animated model, for example, could prove especially useful in distributed teams, where it is not always possible to install software remotely on a customer's desktop.

In our case study, testing provided a number of opportunities to realise the benefits of combing formal and Agile practices. There are a number of ways in which this aspect of software engineering could be explored further, with respect to the application of formal methods. Extracting the key properties of a system from its model, and encoding these as executable tests, proved an effective technique for ensuring that a system's implementation behaves as described by its specification. Just as tools such as Cucumber and Fitnesse[2] can be used to execute tests written in natural language, a system's properties can also be used as the basis for tests [CH00]. We demonstrated this in Chapter 4 using the QuickCheck Haskell library. One way in which this technique of *property-based-testing* could be developed would be to extract test cases automatically from a formal specification, with developers providing the step definitions needed to translate operations from the model into invocations of a running system. This approach

[1] `https://github.com/bendisposto/prob2`, accessed 12 August 2015

[2] `http://www.fitnesse.org/`, accessed 12 August 2015

would mean that tests for the system's behaviour could be described at the level of abstraction provided by the model, yet feedback would be returned directly from the concrete system.

Assuming that the team develops a translation layer between a model's operations and the system's endpoints, another extension to the techniques we have described could be in using the animator to 'record' high-level test cases directly. Browser plug-ins have been developed to support a similar practice for testing websites [Ric10], and we saw from the examples in Chapter 4 how easily state traces can be extracted using ProB. The ability to involve the business more closely in designing a system's test cases allows the team to get feedback more quickly, and helps ensure that the acceptance tests themselves represent realistic business scenarios.

Haskell was instrumental not only in writing test cases, but also as a means to animate the model that we constructed for grading hierarchies and cross-defaults. Given the nature of this model, it was barely necessary to write the Z definitions first; many of the functions could have been written directly in Haskell without sacrificing the clarity and abstraction provided by the specification. This is one advantage of programming in a declarative style, in that a seamless transition can be made from specification to implementation. For the operations defined in the first two state-based models, however, it would have been more difficult to specify these directly in code, and the advantages of using Z are more obvious here. An alternative approach to using functional programming for animating such models, that further reduces the need for specialised tooling, is to use a monad to capture the model's global state. A Haskell-based implementation of this approach is described in [Goo95], and provides further evidence of the benefits to be gained from exploring new ways to use functional programming in conjunction with formal methods.

Aside from tool support, a number of efforts have been made to define a composite methodology, drawing elements from Formal and Agile methods into one unified framework [BSK10, SM14]. Shafiq and Minhas, for example, propose a system for deriving formal requirements specifications directly from user stories, and incorporating verification techniques into the established Agile practices of XP. For teams that are already using formal methods, adopting such a system would provide guidance in their migration to Agile development, allowing them to maintain their level of formal rigour while evolving towards a more iterative style. In the project described in our case study, however, we deliberately did not attempt to make wholesale changes to the team's operating rhythm in order to accommodate the introduction of formal techniques. This helped to minimise disruption to the team's deliveries, and ultimately led to an appreciation of formal methods as a *complementary* technique for modelling and analysing the system's requirements. For different modelling scenarios, however—particularly those for which performing a comprehensive verification of the system's properties is necessary—more significant changes to the structure of the team's iterations may be required to integrate formal practices harmoniously with Agile development patterns.

A final area for further investigation draws inspiration from recent trends in application design. It is becoming increasingly common for a single computer system to be built using multiple programming languages and paradigms—*polyglot* development—harnessing their respective strengths where they are most effective. In our case study, we were concerned primarily with modelling a state-based system; the Z notation proved to be an appropriate choice for this, and sufficient for the needs of its analysis. However, Z is less well-suited to modelling concurrency, for example, and there are other formal methods that serve this purpose more effectively. The *Communicating Sequential Processes* (CSP) method [Hoa85] and associated refinement checker, *FDR*[3], are designed specifically to help teams reason about concurrent and real-time systems. Although hybrid notations combining elements of Z and CSP have been

[3]`https://www.cs.ox.ac.uk/projects/fdr/`, accessed 19 September 2015

explored [Fis96], it may be more pragmatic in a large project to apply different formal methods independently to particular aspects of the system. In the context of our case study, CSP could be used to model the effects of multiple users proposing and approving default instances simultaneously for a given counterparty. Since we are not trying to produce a comprehensive specification for the entire system, this approach would allow different methods to be applied where they are most suitable.

The integration of formal methods and Agile development is therefore clearly a fertile ground for future research and development. Given that there are no insurmountable barriers to the immediate adoption of formal techniques—at least in a limited capacity—it is likely that some of the most useful results will also be gained from the successes and pitfalls of industrial experience.

6 Conclusions

Through the case study outlined in this report, we have confirmed our hypothesis: Formal methods can add significant value in modelling business domains not traditionally associated with them, and with a revised approach, they can be applied effectively in Agile development environments. In Chapter 2, we discussed the importance of computer systems in managing financial risk. In the wake of recent failures, it might no longer seem absurd to describe these systems as ‘critical’; their analysis and design warrants a level of rigour that has hitherto been lacking in some cases. We suggest that by combining the technical discipline of formal methods with the customer-centric approach of Agile development, progress can be made towards addressing this, and restoring trust in an industry that has been criticised for failing to protect its customers.

The use of formal methods in this case study proved most effective when we returned to the principles underlying Agile development, and used these to guide our application of formal practices. Understanding the need to work more closely with the business in specifying a system’s requirements helped to select an appropriate level of abstraction for the models, and encouraged the team to involve their customers directly in reasoning about future requirements. The relationship between Agile and Formal methodologies is therefore symbiotic: Agile principles help to ensure that formal practices are not conducted for their own sake, and customer input can be used to ensure that models correctly record the facts of the business domain. Likewise, the introduction of formal methods provides a mathematical language whose clarity and rigour transcends the ambiguities of an informal approach. Given the ultimate aim of delivering software that best serves the needs of its users, we have shown that formal methods represent a powerful set of tools when used appropriately.

A great deal of the value gained from using formal methods was obtained just by constructing models of the business domain, and analysing these with the available tools. Although for determining certain properties of a system, formal reasoning and proof remained useful, the effort required to perform these activities is significant with respect to the value that they deliver. A graph illustrating this logarithmic relationship between the cost and benefits of different formal techniques is given in Figure 6.1. It must be noted that some initial investment is likely to be required before the benefits of formal methods can be realised, particularly if the team needs to learn a new notation. Once this investment has been made, however—and the team is able to start building simple models of a system—it can be quickly repaid, in the form of rich feedback loops and a deeper understanding of the system’s intended behaviour. Meanwhile, investing further effort to perform subsequent activities, such as comprehensive proof and refinement, does not continue to return value at the same rate. The use of formal methods in an Agile environment should therefore be concentrated on modelling and lightweight analysis, with lengthy proofs only being conducted where necessary.

One of the most important findings of this dissertation was that in order for the integration of formal methods and Agile development to be successful, teams must avoid being dogmatic in their use of either methodology. Formal methods are not a panacea, and a number of teams

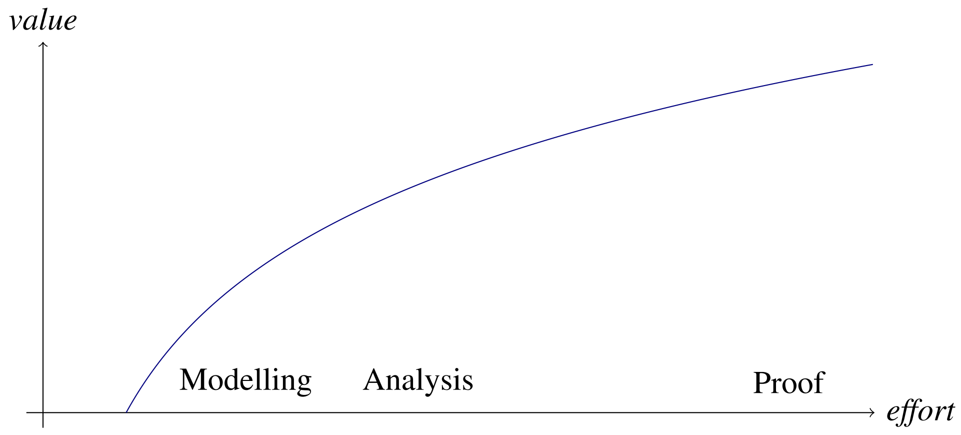


Figure 6.1: Value gained from different uses of formal methods

may have dismissed them simply because their benefits have been oversold. A core principle of Agile methods, however, is the need for technical quality: "Continuous attention to technical excellence and good design enhances agility." [HF01]. Formal methods provide an effective way for teams to enforce this technical excellence. The use of mathematics as a modelling language is an integral part of other engineering disciplines, and given how naturally mathematics can be used to describe the abstract nature of software, its absence here is difficult to justify. Rather than reject the use of mathematical language on the grounds that it is too costly or difficult to understand, we must accept that as professional engineers, it is not something we can afford to be without.

Although in terms of establishing how suitable formal techniques are for Agile development, this case study has delivered a number of useful findings, there are several aspects that could have been conducted more effectively. In order to help teams reflect on their progress and continue to evolve their practices, they will often gather metrics to provide an objective measure of their performance. By tracking their velocity over subsequent iterations, for example, this data can help identify any issues that are affecting their capacity, or can be used to conduct a quantitative assessment while evaluating new techniques or processes. Were we to repeat this case study, one activity that could be introduced would be to monitor the team's performance during their adoption of different formal techniques. For example, tracking the number of bugs created, and the time taken for them to be discovered, could have provided one such metric. Conducting the case study over a longer period of time would also be worthwhile. Although we saw that formal methods were successful in modelling a subset of the business domain from one project, their continued application would highlight whether there are other modelling scenarios to which formal methods are less well-suited, or their application requires a different approach.

The work presented in this dissertation builds upon a number of elements taught in the *Software Engineering Programme*[1]. Material from the *Software Engineering Methods* track was harnessed both in using the Z notation to construct our models, and as a starting point for the subsequent discussion of Agile and Formal methodologies. The *Software Engineering Mathematics*, *Specification and Design* and *Agile Methods* courses were particularly influential in this regard. Likewise, the use of Haskell to animate the models and derive code directly from a specification was inspired by material covered in the *Functional Programming* module from the *Software Engineering Tools* track. As well as leveraging specific skills from the

[1]`http://www.cs.ox.ac.uk/softeng/programme`, accessed 27 September 2015

modules, a number of common themes emerged while studying on the programme. One of the most prominent of these was the nature of Software Engineering as a fundamentally socio-technical discipline. For example, a key learning from the *Security Principles* course was that while cryptography and the mathematical analysis of protocols can aid in the design of secure systems, the social aspects of these systems must not be ignored: users—intended or otherwise—often present systems with their most challenging problems. This principle applied throughout the programme, and has been especially relevant during this case study. Although we have highlighted the merits of a number of different technical practices, their adoption proved most successful when they were performed collaboratively, and as a means to work more closely with others in delivering solutions.

As well as the immediate benefits given to the project described in our case study, the undertaking of this dissertation has provided an opportunity to reflect more generally on the way that software is engineered. We have shown that while Agile methods have revolutionised the entire development life cycle, the benefits of a reactive and iterative approach need not come at the expense of accuracy and rigour. Just as the techniques for coding and testing software have evolved to support Agile practices, the application of formal methods must also be adapted. As the mathematics of software engineering, formal methods represent the tools with which we can ensure the safety of our systems; and as technology connects more aspects of our lives than ever before, our trust and understanding of these systems' behaviour has never been more important.

Bibliography

[bas88] Basel I Capital Accord: International Convergence of Capital Measurements and Capital Standards. Technical report, Basel Committee on Banking Supervision, July 1988.

[bas06] Basel II Capital Accord: International Convergence of Capital Measurements and Capital Standards: A Revised Framework (comprehensive version). Technical report, Basel Committee on Banking Supervision, June 2006.

[bas10] Basel III: A Global Regulatory Framework for More Resilient Banks and Banking Systems. Technical report, Basel Committee on Banking Supervision, December 2010.

[BBC14] BBC News website, accessed 13 April 2015. RBS admits error in stress test data. `http://www.bbc.co.uk/news/business-30153633`, November 2014.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. An Alan R. Apt Book Series. Addison-Wesley, 2000.

[BH94] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods: Dispelling industrial prejudices. In *FME '94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 105–117. Springer Berlin Heidelberg, 1994.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, April 1995.

[BH06] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods... ten years later. *Computer*, 39(1):40–48, Jan 2006.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall series in computer science. Prentice Hall Europe, 1998.

[BSK10] Vieri Del Bianco, Dragan Stosic, and Joseph R. Kiniry. Agile formality: A mole of software engineering practices. In Stefan Gruner and Bernhard Rumpe, editors, *FM+AM*, volume 179 of *LNI*, pages 29–48. GI, 2010.

[CH00] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[Coc07] Alistair Cockburn. *Agile Software Development: The Cooperative Game*. The Agile Software Development Series. Prentice Hall, 2007.

[DD80] Stuart E. Dreyfus and Hubert L. Dreyfus. A five-stage model of the mental activities involved in directed skill acquisition. Technical report, DTIC Document, 1980.

[Dij72] Edsger W. Dijkstra. *Notes on Structured Programming*. Academic Press Ltd., London, UK, 1972.

[Fis96] C. Fischer. Combining CSP and Z. Technical report, University of Oldenburg, 1996.

[Goo95] Howard S. Goodman. The Z-into-Haskell tool-kit: An illustrative case study. In *ZUM '95: The Z Formal Specification Notation*, volume 967 of *Lecture Notes in Computer Science*, pages 374–388. Springer Berlin Heidelberg, 1995.

[Gor94] Andrew D. Gordon. A tutorial on co-induction and functional programming. In *Glasgow Functional Programming Workshop*, pages 78–95. Springer, 1994.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[HF87] Ian Hayes and Bill Flinn, editors. *Specification case studies*. Prentice-Hall International series in computer science. Englewood Cliffs, N.J. Prentice-Hall International, 1987.

[HF01] Jim Highsmith and Martin Fowler. The Agile Manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Jac96] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, New York, NY, USA, 1996.

[JKM$^+$09] Ciera Jaspan, Michael Keeling, Larry Maccherone, Gabriel L. Zenarosa, and Mary Shaw. Software Mythbusters Explore Formal Methods. *IEEE Software*, 26(6):60–63, October 2009.

[LFW10] Peter Gorm Larsen, John S. Fitzgerald, and Sune Wolff. Are Formal Methods Ready for Agility? A Reality Check. In Stefan Gruner and Bernhard Rumpe, editors, *FM+AM*, volume 179 of *LNI*, pages 13–25. GI, 2010.

[LG97] Luqi and Joseph A. Goguen. Formal methods: promises and problems. *IEEE Software*, 14(1):73–85, Jan 1997.

[Liu10] Shaoying Liu. An approach to applying SOFL for agile process and its application in developing a test support tool. *ISSE*, 6(1-2):137–143, 2010.

[Man14] S. Mancuso. *The Software Craftsman: Professionalism, Pragmatism, Pride*. Robert C. Martin Series. Pearson Education, 2014.

[McC04] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[Pag07] S.E. Page. *The Difference: How the Power of Diversity Creates Better Groups, Firms, Schools, and Societies*. Princeton University Press, 2007.

[PL07] Daniel Plagge and Michael Leuschel. Validating Z specifications using the ProB animator and model checker. In *Integrated Formal Methods*, pages 480–500. Springer, 2007.

[Ric10] Alan Richardson. *Selenium Simplified*. Compendium Developments, 2010.

[SB02] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Agile Software Development. Prentice Hall, 2002.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.

[SM14] Shagufta Shafiq and Nasir Mehmood Minhas. Integrating Formal Methods in XP—A Conceptual Solution. *Journal of Software Engineering and Applications*, 7(04):299, 2014.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Spi92] J. M. Spivey. The fuzz manual. *J. M. Spivey, Computing Science Consultancy*, 1992.

[The14] The Times website, accessed 14 April 2015. RBS faces class action over 'abuse' of small businesses. `http://www.thetimes.co.uk/tto/business/goingforgrowth/article4206485.ece`, September 2014.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[WH12] Matt Wynne and Aslak Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. The pragmatic programmers. Dallas, Tex. Pragmatic Bookshelf, 2012.

A Complete Code Listings

A.1 LegalHierarchies.hs

```haskell
{-# LANGUAGE FlexibleInstances #-}
module LegalHierarchies where

import Data.Tree
import Test.QuickCheck
import Test.QuickCheck.Arbitrary
import Test.QuickCheck.Gen
import Test.QuickCheck.Property

type IntTree = Tree Int

instance Arbitrary IntTree where
    arbitrary = sized sizedArbIntTree

sizedArbIntTree :: Int -> Gen IntTree
sizedArbIntTree 0 = do
    c <- choose (minBound, maxBound)
    return $ Node c []
sizedArbIntTree n = do
    c <- choose (maxBound, minBound)
    subtreeCount <- choose (0, n-1)
    subtrees <- vectorOf subtreeCount (sizedArbIntTree (n-1))
    return $ Node c subtrees

prop_size :: IntTree -> Property
prop_size t = property $ size t >= 1

size :: Tree a -> Integer
size (Node x []) = 1
size (Node x (t:ts)) = size t + size (Node x ts)

size' :: Tree a -> Integer
size' (Node x ts) = 1 + sum (map size ts)

root :: Tree a -> a
root (Node x ts) = x

childHierarchies :: Tree a -> [Tree a]
childHierarchies (Node x ts) = ts

subsidiaries = tail.flatten

subtree :: Eq a => a -> Tree a -> Tree a
subtree x (Node y ts)
    | x == y = Node y ts
```

```
subtree x (Node y (t:ts))
    | x `elem` flatten t = subtree x t
    | otherwise = subtree x (Node y ts)

immediateParent :: Eq a => a -> Tree a -> a
immediateParent x (Node y (t:ts))
    | x == root t = y
    | x `elem` flatten t = immediateParent x t
    | otherwise = immediateParent x (Node y ts)

removeSubtree :: Eq a => a -> Tree a -> Tree a
removeSubtree x (Node y (t:ts))
    | x == root t = Node y (ts)
    | x `elem` flatten t = Node y ((removeSubtree x t):ts)
    | x `notElem` subsidiaries (Node y (t:ts)) = Node y (t:ts)
    | otherwise = removeSubtree x (Node y (ts ++ [t]))

addSubtree :: Eq a => Tree a -> a -> Tree a -> Tree a
addSubtree new x (Node y t)
    | x == y = Node y (new:t)
addSubtree new x (Node y (t:ts))
    | x `elem` flatten t = Node y ((addSubtree new x t):ts)
    | x `notElem` subsidiaries (Node y (t:ts)) = Node y (t:ts)
    | otherwise = addSubtree new x (Node y (ts ++ [t]))

moveSubtree :: Eq a => a -> a -> Tree a -> Tree a
moveSubtree srcNode targetParent tree
    | srcNode == targetParent = tree
    | otherwise = addSubtree (subtree srcNode tree)
                    targetParent (removeSubtree srcNode tree)

path :: Eq a => a -> Tree a -> Maybe [a]
path x t
    | x `elem` flatten t = Just (path' [] x t)
    | otherwise = Nothing

path' :: Eq a => [a] -> a -> Tree a -> [a]
path' xs x (Node y t)
    | x == y = xs ++ [x]
path' xs x (Node y (t:ts))
    | x `elem` flatten t = xs ++ y:(path' xs x t)
    | otherwise = xs ++ path' xs x (Node y ts)
```

A.2 CrossDefaults.hs

```
module CrossDefaults where

import LegalHierarchies
import Data.Tree
import Data.Maybe

type LegalHierarchy = Tree PartyInfo

data PartyInfo = PartyInfo String Grading deriving (Eq)

data Grading = Grading
      | Standalone Integer
      | CascadedGrading {cascadeParent :: PartyInfo,
                               notch :: Integer} deriving (Eq)

instance Show PartyInfo where
  show (PartyInfo p Grading) = p
  show (PartyInfo p (Standalone x)) = p ++ " [" ++ show x ++ "]"
  show (PartyInfo child (CascadedGrading parent n)) =
    let cascaded_grade = show $ fromJust $
          grade (PartyInfo child (CascadedGrading parent n))
     in child ++ " [" ++ cascaded_grade ++ "] - Cascaded from " ++
          party parent ++ " with notch = " ++ show n

formatHierarchy :: LegalHierarchy -> IO ()
formatHierarchy x = putStrLn $ drawTree $ fmap show x

party (PartyInfo p _) = p

grading (PartyInfo _ g) = g

parent :: Grading -> Maybe PartyInfo
parent (CascadedGrading p _) = Just p
parent _ = Nothing

grade :: PartyInfo -> Maybe Integer
grade (PartyInfo p Grading) = Nothing
grade (PartyInfo p (Standalone x)) = Just x
grade (PartyInfo p (CascadedGrading parent notch)) = Just (parentGrade + notch)
  where parentGrade = fromJust (grade parent)

cascadeChildren :: PartyInfo -> LegalHierarchy -> [PartyInfo]
cascadeChildren p h = [c | c <- flatten h, Just p == parent (grading c)]

crossDefaults :: PartyInfo -> LegalHierarchy -> [PartyInfo]
crossDefaults p h = filter (/= p) (crossDefaults' p h)

crossDefaults' :: PartyInfo -> LegalHierarchy -> [PartyInfo]
crossDefaults' (PartyInfo c (CascadedGrading p n)) h
    | grade childInfo >= grade p = p : [x | x <- cascadeChildren p h]
    | otherwise = [x | x <- cascadeChildren p h, grade x <= grade childInfo]
  where childInfo = PartyInfo c (CascadedGrading p n)
crossDefaults' x h = cascadeChildren x h

crossDefaultParties p h = map party (crossDefaults p h)
```

A.3 Examples.hs

```haskell
module Examples where

import CrossDefaults
import Data.Tree

simpleTree = Node "parent" [Node "child 1" [],
                 Node "child 2" [Node "grandchild 1" []]]

deepTree = Node "a" [Node "b" [Node "c" [Node "d" [Node "e" []]]]]

parent_a = PartyInfo "Party A" (Standalone 5)
child_b  = PartyInfo "Party B" (CascadedGrading parent_a (-2))
child_c  = PartyInfo "Party C" (CascadedGrading parent_a 1)
child_d  = PartyInfo "Party D" Grading
parent_e = PartyInfo "Party E" (Standalone 11)
child_f  = PartyInfo "Party F" (CascadedGrading parent_e 3)

simpleHierarchy = Node parent_a [
                      Node child_b [],
                          Node child_c [
                          Node child_d []],
                      Node parent_e [
                          Node child_f []]]

party_a = PartyInfo "Party A" Grading
party_b = PartyInfo "Party B" Grading
party_c = PartyInfo "Party C" (CascadedGrading party_e (-3))
party_d = PartyInfo "Party D" Grading
party_e = PartyInfo "Party E" (Standalone 5)
party_f = PartyInfo "Party F" (CascadedGrading party_e 0)
party_g = PartyInfo "Party G" Grading
party_h = PartyInfo "Party H" (CascadedGrading party_e 2)
party_i = PartyInfo "Party I" Grading
party_j = PartyInfo "Party J" (CascadedGrading party_e (-4))

chapter2Hierarchy = Node parent_a [
                        Node party_b [
                            Node party_e [
                                Node party_h [],
                                    Node party_i [],
                                    Node party_j []],
                            Node party_f []],
                        Node party_c [
                            Node party_g []],
                        Node party_d []]
```