Formal Proof from UML Models

Nuno Amálio, Susan Stepney, and Fiona Polack

Department of Computer Science, University of York, York, YO10 5DD, UK
{namalio,susan,fiona}@cs.york.ac.uk

Abstract. We present a practical approach to a formal analysis of UML-based models. This is achieved by an underlying formal representation in Z, which allows us to pose and discharge conjectures to analyse models. We show how our approach allows us to consistency-check UML models, and model analysis by simply drawing snapshot diagrams.
Keywords: UML, Z, model analysis, formal proof, consistency checking.

1 Introduction

This paper describes a practical approach to the formal analysis of models of sequential systems. Our models are UML-based, yet they are amenable to formal analysis. This is achieved by an underlying formal representation in Z, which allows us to pose and discharge conjectures to analyse models. This formal analysis is supported by the Z/Eves theorem prover [1].

UML is the *defacto* standard modelling notation among software engineers. One feature of UML often forgotten (or perhaps unknown in some circles) is that the language has a multi-interpretation nature; the semantics of the language's constructs vary with the application domain. When using UML, modellers ought to make explicit the interpretation being followed, but this is seldom done.

Moreover, UML has many limitations that preclude rigorous (or sound) development. UML models are imprecise and cannot be formally analysed in the UML context. This brings the following consequences: (a) UML models result in ambiguous descriptions of software systems; (b) UML models cannot be checked for consistency, which means that one may produce unsatisfiable models for which no implementation may possibly exist; and (c) there are no means for checking whether certain desired system properties hold in a UML model.

Formal specification languages (FSLs), on the other hand, allow sound development. They yield precise descriptions of software systems that are amenable to formal analysis. However, these languages require substantial expertise from developers, and they are criticised for being *unpractical*, as substantial effort is involved in formally model and analyse systems.

In our modelling approach we aim at addressing these issues. We want (a) to deal with the multi-interpretation nature of UML, (b) to introduce soundness into UML-based development, and (c) to make sound development more approachable to developers and substantial more practical for use in wide engineering domains.

To address these issues we propose *modelling frameworks*. Modelling frameworks are environments for building and analysing models that are tailored to problem domains. Each modelling framework comprises a set of UML notations, a semantics for those notations and an analysis approach for models built using the framework. The semantics of the UML notations is expressed by using a FSL; the analysis approach is based on the analysis means of the FSL being used. The components that make-up the framework (definitions and FSL) are appropriate to the problem domain being targeted by the framework.

We use *templates* to define modelling frameworks. Templates are used to describe meta-level properties, such as, the formal representation (or semantics) of UML modelling elements (e.g. class), and proof-obligations (or meta-conjectures). Our templates are parametric descriptions of phrases in a FSL (here Z), describing the essential structure of a phrase. A template instantiation involves providing names for the template's parameters; when properly instantiated a template yields a FSL phrase.

By using the templates of a framework, we can generate conjectures to formally analyse a UML-based model. Some of these conjectures are simply *true by construction* because our meta-level representations based on templates allows us to do *meta-proofs* of properties (i.e., the property is proved at the meta-level), which, once proved, are true for many instantiations of a particular form. Other conjectures can be automatically generated just by drawing diagrams; this allows formal model analysis by simply drawing simple and intuitive diagrams.

The following terminology is used to refer to the results of proving conjectures:

- *true by construction* — it is guaranteed to be true, so there is no need to do a proof, essentially, a meta-proof has been done;
- *trivially true* – Z/Eves can do it automatically;
- and *provable* – can be proved but Z/Eves needs some help: we are working on patterns/tactics for this class, to make them *trivial*.

Here we present and illustrate the use of a modelling framework for sequential systems based on UML and Z. The illustration is a simplified library system, comprising a library catalogue, copies of catalogued books for borrowing, tracking and renewal of loaned copies, and recalling of books. Section 2 presents the framework. Sections 3 to 6 construct and analyse the model of the library system.

2 A Modelling Framework for Sequential Systems

We have developed a modelling framework to construct and analyse models of sequential systems by using Z as the FSL [2]. Models and associated analysis conjectures are built by instantiating templates of the framework [3].

The models of our framework comprise UML class, state and snapshot diagrams with an underlying Z representation. Snapshot diagrams are a feature of Catalysis [4]; they are essentially object diagrams, illustrating either one system state or one system state transition. In the framework, most properties are

expressed diagrammatically, but the modeller needs to resort to Z to express: invariants (or model constraints), and operation specifications. These properties would be expressed in OCL in a UML-only development.

Z has proved to be an appropriate language to represent abstract UML models. Z has semantics that is mathematical rather than computational. This makes the language flexible, powerful and extensible, and allows structuring based on different computational models. We have developed a model for Z based on the OO paradigm to represent abstract UML models. This results in well-structured and natural Z specifications. Another Z feature that is important in our approach is Z conjunction, which allows us to assemble the local pieces of state and behaviour into global ones. This gives us modular and conceptually clear Z.

In the following subsections, we discuss the semantic model used to represent UML diagrams, and the analysis strategy of the framework.

2.1 The semantic model

A semantic model is required to formally represent UML models. This is defined by using our OO Z structuring [2], which can be used to represent UML-based models and to construct Z specifications in a OO style.

Our OO structuring extends those reviewed in [5]. One of the novel features of our structuring is that it is views based, following a views structuring approach for Z [6]. The need for a views structuring approach came from the observation that not all properties of class models would fit elegantly into a single view.

In our structuring, a class has a dual meaning. *Class intension* defines the properties shared by the objects of that class (e.g, *Member* of figure 1 has properties *name*, *address*, and *category*). *Class extension* defines the class in terms of currently existing objects (e.g., *Member* is $\{oRalph, oJohn, oAlan\}$).

The main modelling entities are represented in Z in a state and operations style. Each class intension, class extension, association, subsystem and ultimately the whole system is represented as a Z abstract data type (ADT), consisting of a state description, an initialisation, and operations upon the state.

2.2 Analysis Strategy

The models of our framework are analysed through proof in Z. Our OO structuring based on ADTs across views, makes the task of demonstrating the consistency of our modelling entities easier, we just have to follow Z's consistency checking approach (proving initialisation theorems and calculating preconditions of operations). We also generate conjectures from diagrams, either to check the consistency of a diagram or to validate the system model. It is important to emphasise our approach to model validation, which is based on drawing snapshot diagrams; once we have these diagrams we can then generate conjectures, by instantiating templates, which are either trivially true or provable.

3 Modelling State

Modelling of state involves: building class models, describing the main entities of the system and the relationships between them; formally representing the class model and adding the model's static constraints (or invariants); and drawing snapshots of system state to validate the overall model of state.

3.1 UML Class Model

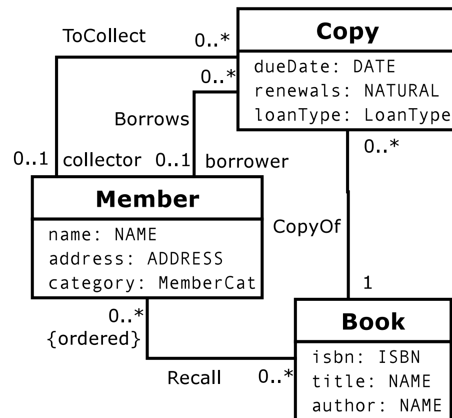

Fig. 1. Class diagram of library system.

Figure 1 is the abstract UML class model of the library system. *Book* represents the entries in the library catalogue, comprising an *isbn*, the *title* and *author* of the book. *Copy* represents the actual instances of book; the attributes record the *dueDate* of a loaned copy, the number of *renewals* that have been made, where each renewal would set a later due date, and the *loanType*. *Member* represents the library members that are entitled to borrow and recall copies of books; the attributes record the *name* of a member, the *address* and the member *category*.

Structurally, *CopyOf* associates each Copy object with one Book object; there may be many copies of each book. A Member object *Borrows* zero, one or many copies; at any time each Copy object may or may not be borrowed. Members may issue a *Recall* against a catalogued Book object for which no copy is available; a Book object may have many unserved recalls at a time. The *Recall* association is subject to an *ordering* constraint, because, when a copy of a book for which there is a recall is returned to the library, the copy is offered to the member who made the earliest unmet recall. There is inevitably a period of time between the copy of the recalled book becoming available and the recalling member collecting the copy; *ToCollect* expresses the association between the recalling member and the copy whilst the book is awaiting collection.

3.2 Formal Representation of UML Class Model and Constraints

Here we show how the class diagram is formally represented by instantiating templates. We also show how we can add model constraints that do not have a diagrammatic representation. In representing the different modelling elements we follow systematic naming conventions; extensional definitions are preceeded by $\mathbb{S}$, and association-related ones by $\mathbb{A}$.

The formal representation is illustrated for each view of our structuring, namely, *structural*, *intensional*, *extensional*, *relational* and *global*. In the structural view we represent objects. In the intensional and extensional views we represent the intensional and extensional meaning of classes, respectively. In the relational view we represent associations. The global view groups classes and associations into subsystems and ultimately the system.

Structural View. Objects are represented as atoms. We consider that there is a given set of all objects ($OBJECT$), of which objects are members. Each class has an object set, which is a subset of the set of all objects ($CopyOs$ is the object set of $Copy$).

$[OBJECT]$

$| \; CopyOs : \mathbb{P}\, OBJECT$

Intensional View. Here we represent state intensions and their initialisations. A state intension comprises class attributes and a class invariant. An initialisation makes an assignment of values to class attributes. The declaration part of *Copy* (below) is generated by template instantiation from the class and state models; the user provides the predicate part of the schema (intension invariant). *CopyInit* (below) is generated by template instantiation from the state model (indicates the initial value of *state*, see Fig. 4) and initial values provided by the user.

Copy

$$dueDate : DATE$$
$$loanType : LoanType$$
$$renewals : \mathbb{N}$$
$$state : CopyState$$
$$renewals \leq renewalLimit$$

CopyInit

$$Copy'$$
$$loanType? : LoanType$$
$$dueDate' = nullDate$$
$$renewals' = 0$$
$$loanType' = loanType?$$
$$state' = shelved$$

Extensional View. Class state extension defines the set of all existing class objects (a subset of the class' object set), and a mapping between object atoms and their state intensions. We represent state extension as a Z generic ($\mathbb{S}Class$ below), actual state extensions are instantiations of this generic. The $\mathbb{S}Copy$ state extension (below) and its initialisation are generated from the class model by instantiating their corresponding templates[1]; if there were extension invariants they would have to be provided by the user.

$\mathbb{S}\text{Class}\,[OSET, OSTATE]$

$$objs : \mathbb{P}\, OSET$$
$$objSt : OSET \rightarrowtail OSTATE$$
$$\text{dom}\, objSt = objs$$

$$\mathbb{S}Copy == \mathbb{S}\text{Class}[CopyOs, Copy][copys/objs, copySt/objSt]$$

[1] The renaming in the definition of $\mathbb{S}Copy$ is done to avoid name clashing.

Relational View. An association denotes a set of tuple pairs, where each tuple describes the pair of objects being linked. Association state defines a mathematical relation between the object sets of the classes being associated (e.g. association *Borrows* is defined as $borrows : MemberOs \leftrightarrow CopyOs$). Association static representations (state and initialisation) in Z are mostly generated from the class diagram information by instantiating proper templates; again only association invariants, if they exist, need to be added by the user. We have templates to handle special kinds of associations, such as those with the *ordered* constraint.

Global View. Subsystem state groups classes and association of the subsystem, includes consistency constraints between classes and associations, and system constraints that cross modelling elements. The system state of the library problem is given below, with its three classes and four associations. In the predicate, each name refers to another Z schema – *Consistency* predicates refer to consistency constraints on associations, and *Constraint* predicates to system-wide constraints. The initialisation of system state (*SysInit* below) consists of initialising each system component (class or association); this is defined by conjoining the initialisations of class extensions and associations. Both state and initialisation schemas are generated from templates, where most information comes from the class model; the user adds the *Constraint* schemas.

System

$\mathbb{S}Member;\ \mathbb{S}Copy;\ \mathbb{S}Book;\ \mathbb{A}Borrows;\ \mathbb{A}ToCollect;\ \mathbb{A}CopyOf;\ \mathbb{A}Recall$

$Consistency\mathbb{A}Borrows \land Consistency\mathbb{A}ToCollect$

$Consistency\mathbb{A}CopyOf \land Consistency\mathbb{A}Recall$

$ConstraintNoCommonCopiesBorrowsToCollect$

$ConstraintBorrowingLimit$

$$SysInit == System' \land \mathbb{S}MemberInit \land \mathbb{S}CopyInit \land \mathbb{S}BookInit \land \mathbb{A}BorrowsInit \land \mathbb{A}ToCollectInit \land \mathbb{A}CopyOfInit \land \mathbb{A}RecallInit$$

3.3 State Snapshots

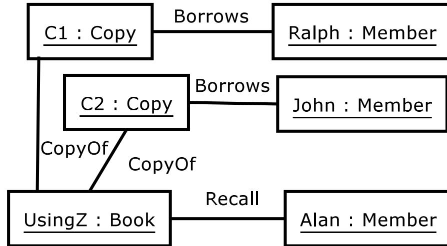

Fig. 2. Snapshot of a desired system state.

State snapshots are UML object diagrams describing one specific state of the system.

Figure 2 is a valid snapshot of the system state, comprising a book, *UsingZ*, with two copies, $C1$ and $C2$. There are three library members, two of whom are borrowing the copies of *UsingZ*; and the third of whom has issued a recall. Informally, one can see that this snapshot illustrates a valid state of our class model.

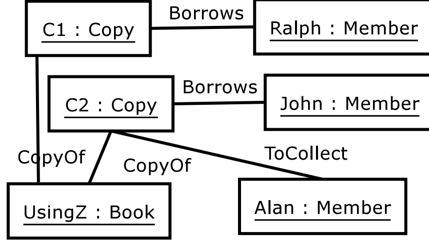

Fig. 3. Snapshot of an undesired system state.

Figure 3 is an invalid state snapshot. Here, copy $c2$ is being borrowed by *John* but also being kept awaiting collection for *Alan*. We want the system to disallow this – as soon as a copy of a recalled book is returned, its *Borrows* link should be replaced by a *ToCollect* link with a recalling member.

We considered placing a *disjoint* constraint between *Borrows* and *ToCollect*, with the intention of making the snapshot in Figure 3 invalid. Formal analysis shows that such a constraint would leave the snapshot valid, as the tuples in the snapshot are disjoint: *Borrows* comprises $\{(Ralph, C1), (John, C2)\}$ and *ToCollect* comprises $\{(Alan, C2)\}$. We express this in a Z constraint schema (included in the *System* schema, above), by stating that the tuples of those associations may not have copies in common (the *ranges* of relations *Borrows* and *ToCollect* must be disjoint).

$$\begin{array}{l} \textbf{schema } ConstraintNoCommonCopiesBorrowsToCollect \\ \quad \mathbb{A}Borrows;\ \mathbb{A}ToCollect \\ \hline \quad \text{disjoint}\langle \operatorname{ran} borrows, \operatorname{ran} toCollect \rangle \end{array}$$

3.4 Formally Representing Snapshots

Each snapshot is formalised as a specific instance of the system, by equating each general concept to a specific set of instances. The representation of the snapshot of Fig. 2 is given below; this is fully generated from the snapshot diagram by instantiating templates (names are declared axiomatically).

$$\begin{array}{l} \textbf{schema } LibrarySnapshot1 \\ \quad System \\ \hline \quad books = \{oUZ\} \land bookSt = \{oUZ \mapsto uz\} \land copys = \{oC1, oC2\} \\ \quad copySt = \{oC1 \mapsto c1, oC2 \mapsto c2\} \land members = \{oR, oJ, oA\} \\ \quad memberSt = \{oR \mapsto ralph, oJ \mapsto john, oA \mapsto alan\} \\ \quad copyOf = \{oC1 \mapsto oUZ, oC2 \mapsto oUZ\} \\ \quad borrows = \{oR \mapsto oC1, oJ \mapsto oC2\} \\ \quad recall = \{oA \mapsto oUZ\} \land ordRecall = \{oUZ \mapsto \langle oA \rangle\} \land toCollect = \varnothing \end{array}$$

4 Analysing State

State analysis involves: checking the consistency of the class model; and checking the class model against state snapshots.

4.1 Class Model

To demonstrate the consistency of state, we prove *initialisation* theorems. An initialisation theorem is of the form, $\vdash \exists State' \bullet StateInit$. This says that there is some valid instance of *State* satisfying its initialisation (hence the model of state is satisfiable). We need to prove initialisation theorems for the various system components, including, class intensions and extensions, associations, and the whole system.

For the example (and most that we studied), with correctly-instantiated templates,

- the initialisation theorem on class extensions and associations are true by construction;
- the initialisation theorem on the class intensions are trivially true;
- the initialisation theorem on the whole system, which ensures that all the constraints hold in the initial state, is (easily) provable.

In our example, the *Copy* intension initialisation, $\vdash \exists Copy' \bullet CopyInit$, is automatically discharged (proved true) by Z/Eves. Since *Copy* has an invariant stating that the number of renewals of a copy loan must be no more than the maximum number of renewals allowed ($renewals \leq renewalLimit$), the proof confirms that this constraint is satisfiable. Suppose that *Copy* state has the contradictory invariant, $renewals < 5 \land renewals > 10$. Z/Eves would now reduce the initialisation conjecture to *false*; no initialisation can be made in which this invariant is satisfied.

For the example, we proved the initialisation theorem for the whole system, $\vdash \exists System' \bullet SysInit$. This means that the overall system constraints are consistent, and that the specification is a satisfiable model.

4.2 State Snapshots

A state snapshot is a witness of the system model. Analysis demonstrates whether the snapshot is a valid witness or not. This involves proving an *existence conjecture*. As for the initialisation, if this conjecture is true then there is a state of the specified system that satisfies the state described in the snapshot. One can also perform negative model validation by using deliberately invalid snapshots (see section 6.3).

The instantiated template conjectures that validate the two state snapshots in Figures 2 and 3 are, respectively,

$$\vdash \exists System \bullet LibrarySnapshot1 \qquad \vdash \exists System \bullet LibrarySnapshot2$$

The first is provable in Z/Eves, confirming the validity of the snapshot; the second reduces to *false*, so is an invalid state of the whole system (as required), and we can conclude that our model disallows this state.

5 Modelling Behaviour

Modelling behaviour in our framework involves: describing the reactive behaviour of relevant classes with *state diagrams*; describing changes of system state in the context of system operations with *operation snapshots*; specifying operations in Z based on operation snapshot diagrams.

5.1 UML State Diagrams

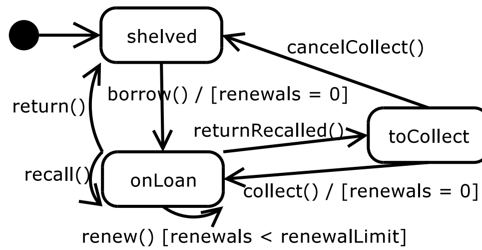

Fig. 4. State diagram of *Copy*

UML state diagrams describe the permitted state transitions of objects of each class. Figure 4 is the state diagram of the *Copy* class. We follow a Catalysis [4] convention, that each event is described in the form, $event[guard]/[postcondition]$, where *event*, the name of the event, equates to the name of a class operation; *guard* is an optional predicate stating the precondition for the transition; *postcondition* is an optional predicate to express what the transition is required to establish.

Here, the initial state of a copy is *shelved*. The loan of a copy by a member, *borrow*(), results in a transition from *shelved* to *onLoan*. When in the *onLoan* state, copies may be renewed, *renew*(), and recalled, *recall*(). The return of a copy, *return*(), results in a transition either back to *shelved*, or, if there is a recall on the copy's book, *returnRecalled*(), to *toCollect*. When the relevant member collects the copy, *collect*(), there is a transition from *toCollect* to *onLoan*. If the recalling member indicates that they no longer require the copy awaiting collection, *cancelCollect*(), there is a transition from *toCollect* to *shelved*. We might expect a fourth state, for copies which are being borrowed but which are subject to a recall; such a state would not allow *renew*() operations. However, in our domain recalls are made upon books, not copies; a recall is active until one of the copies of the recalled book is returned; as copies are not aware of the state of other copies, such a state would require a message to each copy on loan to unset the recall. Instead, in our model, there is a system level renewal operation which takes as input a copy instance, and checks whether there are any *Recall* links from the *Book* to which the copy is linked; only if there are no *Recall* links can the *copy.renew*() operation be executed.

5.2 Formal Representation of UML State Diagrams

State diagrams are represented in the intensional view, as part of the intensional definition of the class to which they are associated. Such a representation involves defining the class' state (to indicate the current state an object is in according

to the state model), initialisation and operations, according to the contents of the state model. This is achieved by instantiating templates.

In our example, *Copy*'s state schema includes a component *state* of type $\{shelved, onloan, toCollect\}$ (the possible states of *Copy* objects); the initialisation sets *state* to *shelved*, the initial state of the state model (see section 3.2 for schemas *Copy* and *CopyInit*); the operations establish the state transitions of the state model, for example, the operation *CopyBorrow* (see below) is associated with the state transition *shelved* to *onLoan*, the specification has a precondition stating that *state* is *shelved*, and a post-condition stating that *state* is *onLoan* and *renewals* is 0 (post-condition of transition).

5.3 Operation Snapshots

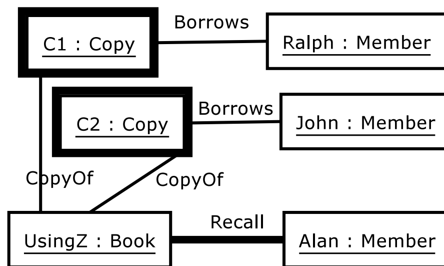

Fig. 5. Operation snapshot in the context of *Recall*.

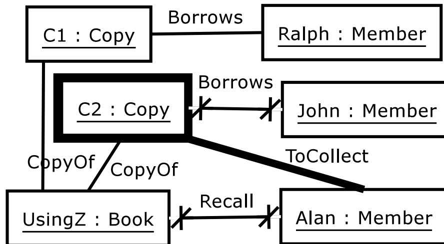

Fig. 6. Operation snapshot in the context of *Return*.

Operation Snapshots describe one system state transition in the context of an operation. These are valuable in understanding and describing operations, and in validating their specifications.

Figure 5 is a valid snapshot in the context of the *Recall* operation, invoked when *Alan* issues a recall on *UsingZ*. Highlighting denotes the change of state after execution – a link between the member that issued the recall and the recalled book is formed. The changes to $c1$ and $c2$ represent the resetting of due dates imposed when a recall is received.

Figure 6 is a snapshot in the context of the *Return* operation, invoked when *John* returns $C2$. Since *UsingZ* has an active recall, a link is formed between *Alan* and $C2$; as the recall is satisfied, the *Recall* link is deleted.

The snapshots increase confidence that the class and state diagrams and operation specifications are sensible. Later we see how we can use them to validate the model.

5.4 Operation Specifications

Operations are specified in Z from operation snapshot diagrams.

Class operations have intensional and extensional definitions. The intension specifies the required state changes of one class object. The extension *promotes* the intensional operation to all existing objects of a class. Below, *CopyRenew* is the intensional definition of *Copy.renew*() and $\mathbb{S}$*CopyRenew* the extensional one (each defined in intensional and extensional views respectively); $\mathbb{S}$*CopyRenew* is defined by using Z promotion [7], where $\Phi\mathbb{S}$*CopyUpdate* is the update promotion framing schema, and *CopyRenew* the intensional operation being promoted. These definitions have templates that capture their structure; *CopyRenew* is generated with information coming from the state diagram (*state* predicates), and provided by the user; $\mathbb{S}$*CopyRenew* can be automatically generated.

CopyRenew

$\Delta Copy$

$$state = onLoan$$
$$state' = onLoan$$
$$renewals' = renewals + 1$$
$$dueDate' = nextDueDate\ (loanType, todayD)$$
$$loanType' = loanType$$

$$\mathbb{S}CopyRenew == \exists\, \Delta Copy \bullet \Phi\mathbb{S}CopyUpdate \wedge CopyRenew$$

System operations are defined in the global view by conjoining a framing schema[2], class extensional operations, association operations, and predicates. This structure is captured by a template. The system operation *Renew*() is generated from this template with parameters provided by the user to yield,

$$SysRenew == \Psi SysRenew \wedge \mathbb{A}IsBorrowing \wedge \neg\, \mathbb{A}IsCopyRecalled \wedge \mathbb{S}CopyRenew$$

which states that the renewal of a *Copy* involves: making sure that the member is indeed borrowing the *Copy* being renewed ($\mathbb{A}$*IsBorrowing*), making sure that there is no active recall upon the *Copy* ($\neg\, \mathbb{A}$*IsCopyRecalled*), and calling the renew operation upon the *Copy* being renewed ($\mathbb{S}$*CopyRenew*), which performs a change of state in the *Copy* being renewed.

6 Analysing Behaviour

Analysis of behaviour consists of: checking the consistency of state diagrams against class models and operation specifications; pre-condition investigation of operations; and using operation snapshots to validate the model.

[2] Indicates what parts of system state change with the operation; this kind of framing schemas are preceeded by Ψ, to distinguish them from Φ promotion framing schemas.

6.1 State Diagrams

A state diagram sets requirements upon the initialisation and operations of a class intension. To check the consistency of a state diagram against initialisation and operation specifications, we state and try to prove conjectures expressing what we wish to hold. This gives us one conjecture for the initial state, and two conjectures per state transition: one for the pre-condition of the operation associated with the transition and one for the post-condition. These conjectures are captured by templates that can be fully generated from the state diagram.

In our example, the initialisation conjecture:

$$CopyInit \vdash state' = shelved$$

assumes the initialisation to show that the initial value of state is set as described by the state model. This conjecture is true by construction.

The approach to check pre- and post-conditions of operations is similar. The precondition conjecture for $Copy.renew()$:

$$\mathrm{pre}\ CopyRenew \vdash state = onLoan \land renewals < renewalLimit$$

assumes the precondition of the operation to show that the before-state ($state = onLoan$) and the guard of the *Renew* state transition ($renewals < renewalLimit$) are satisfied by the operation's precondition. This is trivially true. However, if we had made the common mistake of simply using the relevant class invariant ($renewals \leq renewalLimit$) as the guard, the calculated precondition would no longer establish the consequent, and the conjecture would be *false* – in some cases, a further renewal would exceed the limit set in the state constraint.

The postcondition conjecture for $Copy.renew()$:

$$CopyRenew \vdash state' = onLoan$$

assumes the operation to show that the *Copy* is left in the after-state of the *Renew* state transition ($state' = onLoan$). This is true by construction in this example, as there is no explicit postcondition in the state model for this transition.

6.2 Precondition Investigation

The precondition of an operation describes the sets of states for which the outcome of the operation is properly defined. Precondition calculation is routine in formal analysis. Systematic precondition analysis of class-level operations can reveal flaws in the specification. For example, consider a specification of *Recall* that has a precondition that (a) the recall must must be done by a member, (b) the requested book must be a book in the library catalogue, and (c) there are no copies of the book available in shelves. Pre-condition analysis reveals that this is too weak, in that it allows the recall of a book that have no copies (the multiplicity of the *Copy* end of *CopyOf* is $0\mathrel{..}*$), the placing of two or more active recalls by the same member, and recall by a member who already has a copy of the book on loan or awaiting collection. Analysis allows the precondition to be strengthened to disallow such behaviour.

6.3 Operation Snapshots

Snapshots of system-level operations result in a dual formal representation: the before-state, and the after-state. Both are generated by instantiating templates with information coming from the diagram, and both need to be challenged.

The before-state of the snapshot is required to satisfy the operation's precondition. This results in a conjecture that assumes the formal representation of the snapshot's before-state, and establishes the precondition of the system operation. For instance, the snapshot in Fig. 5, representing the system-wide effect of a recall, would result in the following instantiated template conjecture:

$$LibraryBeforeRecallSnp \vdash \mathrm{pre}\, SysRecall$$

This is provable in Z/Eves.

For the after-state, we require that from the snapshot's before-state, the system-level operations establish the snapshot's after-state. The conjecture assumes the formal representation of the snapshot's before-state and the system operation, and establishes the snapshot's after-state. For the snapshot in Figure 5, the instantiated template conjecture is:

$$LibraryBeforeRecallSnp;\ SysRecall \vdash LibraryAfterRecallSnp$$

This is provable in Z/Eves. The pre- and post- conjectures for the snapshot in Figure 6 are similarly constructed and proved.

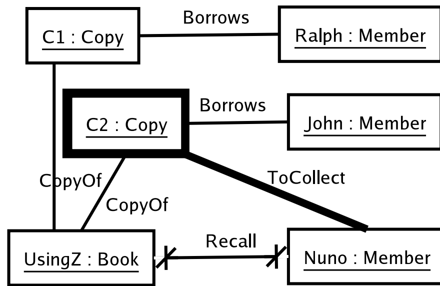

Fig. 7. State snapshot of an undesired behaviour in the context of *Return*.

We can also perform negative validation. Figure 7 illustrates a state transition, in the context of operation *Return*, that our model should disallow. The snapshot differs from that in Figure 6 in that the *Borrows* link between *John* and $C2$ is not deleted, violating the constraint added to the system state to prevent simultaneous *Borrows* and *ToCollect* links on the same copy.

If the post-condition conjecture on the formalised snapshot does not reduce to *false*, this indicates there may be missing preconditions or other constraints. Once determined, these can be added to the specification, and the conjecture reinvestigated, until it does reduce to *false*. (Clearly, other conjectures involving the same operation will also need to be reproved as part of this process.)

7 Discussion

Our template approach provides a Z formalisation of class, state, and snapshot diagrams, and, in addition, provides an analysis strategy for models composed

of these diagrams. Currently, our approach allows rigorous hand-generation of Z from diagrams by instantiating templates. Future work should address automation (tool development).

By using the templates of our framework, we can challenge a model by drawing snapshot diagrams; the associated formal representations and conjectures are generated by template instantiation (hence potentially automatable). The discharging of snapshot conjectures can also be partially automated; snapshots deal with model instances, which are more tractable by theorem provers. We can see snapshots as giving life to a model. Snapshot-based validation is a useful technique even if one builds Z-only specifications in a OO style.

The current drawback of our approach is that the discharging of some of the proofs in Z/Eves requires expertise in using the prover (mainly proofs dealing with system operations). We are looking at ways to mitigate this problem by using proof tactics and patterns that, potentially, could substantially simplify and perhaps fully automate some of the proofs.

Our approach of formalising UML class and state diagrams in Z with a state and operations style, using the schema calculus, enhances the analysis capabilities of our approach. Analysis in Z is tied to the state and operations style, where one is required to prove initialisation theorems and calculate pre-conditions.

Moreover, templates allow us to minimise the proof effort. We can prove properties at the meta-level that are applicable to all instantiations of a template, or to instantiations of particular form. We try to make as much proof as possible at the meta-level – a property is proved once, and then it becomes true when an instantiation relevant to the property occurs (true by construction).

The modelling framework presented here requires modellers that know both UML and Z. Our aim is to allow UML-only modellers to use our framework, hence, we need to make the underlying Z completely transparent to the user. This involves devising new notations (or using existing ones like OCL), for the expression of constraints and operation specifications, that can be easily represented in Z.

Modelling frameworks would greatly benefit from tool support. Currently we are working on the theoretical foundations. However, we envisage a tool that generates a formal representation of a UML model, checks that model for well formedness (through type-checking and other means), generates the associated conjectures (proof obligations) associated with the model, and then interacts with a prover to discharge those conjectures.

8 Conclusions and Future Work

Modelling frameworks and the associated template mechanism give us:

- Means to introduce soundness in UML-based development. We define interpretations of UML concepts in a FSL that has a formal semantics with analysis means. This allows us to build UML models that are precise and analysable.

– Means to make sound development more usable. Most Z of our framework's models is generated by template instantiation, hence potentially automatable. This may allow the construction of sound models composed of diagrams, opening sound development to UML-only modellers.
– Means to make sound development more practical. Our representations in the template form allows us to prove properties at the meta-level so that users don't have to actually do it.

We are looking at generalising conjectures coming from snapshot diagrams. This will allow us to prove properties that hold in our model for any model-instance, rather than a specific one.

Having generated an underlying formal model, we can exploit other forms of formal analysis. For example, the Alloy tool [8] uses a subset of Z; after some manipulation, we can use Alloy's Alcoa model checker. We have initial results verifying properties of class models [2], and we will also explore reachability analysis on state diagrams. We are also looking at other aspects of development within the formal transformational framework – Catalysis has an informal notion of model refinement based on diagrams, inspired by formal refinement. We want to combine the diagram refinement with the conventional formal refinement calculus, whilst retaining the developer-friendly approach.

Acknowledgements. This research was supported for Amálio by the Portuguese Foundation for Science and Technology under grant 6904/2001.

References

1. Saaltink, M. The Z/EVES system. In *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*. Springer-Verlag (1997)
2. Amálio, N., Polack, F., Stepney, S. A sound modelling framework for sequential systems I: Modelling. Technical Report YCS-2004, Department of Computer Science, University of York (2004)
3. Amálio, N., Stepney, S., Polack, F. Modular UML semantics: Interpretations in Z based on templates and generics. In Van, H. D., Liu, Z., eds., *FACS'03: Formal Aspects of Component Software, Int. Workshop, Pisa, Italy*, 284, pp. 81–100. UNU/IIST Technical Report (2003)
4. D'Sousa, D., Wills, A. C. *Object Components and Frameworks with UML: the Catalysis approach.* Addison-Wesley (1998)
5. Amálio, N., Polack, F. Comparison of formalisation approaches of UML class constructs in Z and Object-Z. In Bert et al. [9], pp. 339–358
6. Jackson, D. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4):365–389 (1995)
7. Stepney, S., Polack, F., Toyn, I. Patterns to guide practical refactoring: examples targetting promotion in Z. In Bert et al. [9], pp. 20–39
8. Jackson, D. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290 (2002)
9. Bert, D., et al., eds. *ZB 2003: Formal Specification and Development in Z and B, Int. Conference, Turku, Finland*, volume 2651 of *LNCS*. Springer-Verlag (2003)