Exploring UML Refinement through Unification

Eerke Boiten and Marius Bujorianu

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Email: E.A.Boiten@kent.ac.uk.)

Abstract. One way of making UML more suitable for the development of critical systems is to define a formal notion of development (or *refinement*) for it. We explore refinement indirectly: through *unification*.
Different UML diagrams may contain information on the same system element, which may or may not be contradictory. Such diagrams may be part of the same UML model, or taken from different models representing "viewpoints". A representation of the combined information of diagrams is a *unification*. Implicit in this is a notion of "information content" which needs to be formalised. A unification is not only a representation of combined information, it also witnesses consistency between the models. The theory of consistency and unifications for viewpoint specification is well-developed for formal methods. In general, such unification methods are parameterised by a notion of refinement (i.e., how to compare information content), and a notion of correspondence (relating the information between specifications). In particular, in Z all of these can be expressed syntactically, and a variety of refinement relations have been developed inspired by different styles of viewpoint specification.
This paper considers a number of small UML models, their intuitive "unifying" diagrams, and how these would relate to the unifications of formalisations of the original diagrams. In this way, desirable properties for a formal development notion in UML emerge.

1 Motivation: Formalists' Pamphlet

For the use of UML in the development of critical systems, we believe that an increased degree of formality is necessary. Although the UML community as a whole may not ever agree on the interpretation of UML, within development teams such agreement is essential. Formal interpretations (plural intended!) of UML provide an economy of scale for such shared interpretations.

In the absence of a formal interpretation, there are a number of reasonable questions that one *might* ask about a UML specification, but which are not answerable. These questions are listed here, with theorems describing how they are interrelated. In essence, they are so closely related that an answer to one of these questions solves all or most of them.

Conformance If we have made the effort of writing a UML model, either before or after writing the program, can we actually check that the specification and the program match? In other words, given a UML model S, and a program P, does P conform to S? Let us write this as $Conf(P, S)$.

Consistency The various diagrams in UML describe systems from different angles, but they are not entirely orthogonal. So we may ask whether, even if each diagram by itself is consistent, their combination imposes constraints that cannot be satisfied. Let us denote the consistency of UML descriptions $S_1, \ldots, S_n$ as $\mathit{Consist}(S_1, \ldots, S_n)$.

Theorem 1 (Consistency for free (1)). *Given conformance, consistency can be defined by*

$$\mathit{Consist}(S_1, S_2) =_{def} \exists P \bullet \mathit{Conf}(P, S_1) \wedge \mathit{Conf}(P, S_2)$$

Refinement Imagine we have written a first UML specification, and obtained agreement from all parties involved that it correctly reflects all their requirements. Unfortunately, it does not quite describe the system's workings in detail – we would like to change the UML description to reflect such extra detail. Can we determine whether the resulting description bears any relation to the original requirements without renewed consultation? I.e., is our second specification S_2 a correct refinement of S_1, denoted $\mathit{Ref}(S_2, S_1)$?

Theorem 2 (Refinement for free (1)). *Given conformance, refinement can be defined by*

$$\mathit{Ref}(S_2, S_1) =_{def} \forall P \bullet \mathit{Conf}(P, S_2) \Rightarrow \mathit{Conf}(P, S_1)$$

Theorem 3 (Consistency for free (2)). *Given refinement, consistency can be defined by*

$$\mathit{Consist}(S_1, S_2) =_{def} \exists S_3 \bullet \mathit{Ref}(S_3, S_1) \wedge \mathit{Ref}(S_3, S_2)$$

Also, if we have a notion of "implementation specification" in UML, i.e., specifications which correspond one-to-one to actual implementations (call this $\mathit{ImpSpec}(S, P)$), then conformance also follows from refinement:

Theorem 4 (Conformance for free). *Given refinement and implementation specifications, conformance can be defined by*

$$\mathit{Conf}(P, S) =_{def} \exists S' \bullet \mathit{Ref}(S', S) \wedge \mathit{ImpSpec}(S', P)$$

Semantics Maybe this question is slightly less reasonable than the earlier ones. Nevertheless . . . we might ask what the acceptable *models* of a UML description S are. Let us denote the set of all of these as $\mathit{Models}(S)$.

Theorem 5 (Refinement for free (2)). *Given a model-based semantics, refinement can be defined by*

$$\mathit{Ref}(S_2, S_1) =_{def} \mathit{Models}(S_2) \subseteq \mathit{Models}(S_1)$$

We might avoid trivialisation by additionally insisting that $\mathit{Models}(S_2) \neq \varnothing$.

Theorem 6 (Consistency for free (3)). *Given a model-based semantics, consistency can be defined by*

$$Consist(S_1, S_2) =_{def} Models(S_1) \cap Models(S_2) \neq \varnothing$$

Obviously, semantics is "for free" when we have conformance (all conformant programs as models) or refinement (models are defined as the set of all possible refinements).

The theorems provide extensive possibilities for defining one of the notions in terms of the other; in particular, defining *refinement* is a strong basis for a full UML formalisation.

The theory represented here, briefly hinted at in [7], is inspired by our earlier work on viewpoint consistency [3, 5]. In that work, another useful notion is that of a *unification*.

Unification Given two specifications, one might create a single specification that contains (exactly) the information contained in both. These we call *(least) unifications*, let us denote the unification of S_1 and S_2 as

$$S_1 \oplus S_2$$

Least unifications can be characterised in terms of each of the preceding notions:

$$\begin{array}{l} Conf(P, S_1 \oplus S_2) \Leftrightarrow Conf(P, S_1) \wedge Conf(P, S_2) \\ Consist(S_1, S_2) \Leftrightarrow Consist(S_1 \oplus S_2) \\ Ref(S_3, S_1 \oplus S_2) \Leftrightarrow Ref(S_3, S_1) \wedge Ref(S_3, S_2) \\ Models(S_1 \oplus S_2) = Models(S_1) \cap Models(S_2) \end{array}$$

The second equation holds for any unification; the first and third become implications for unifications which are not "least". The last equality needs to be weakened to an inclusion if the image of *Models* is not closed under intersection.

Unifications are most useful if they can be constructed syntactically; it depends on the properties of the specification language whether this is possible. In the state based language Z this is indeed possible [4]; in process algebra, typically the process structuring is lost in unification [25].

In the next section we will argue why consistency and unification are, by themselves, relevant issues for UML, and discuss some of the approaches taken in the literature.

2 Consistency and Viewpoints for UML

The UML combines multiple notations into a single modelling language. This gives modellers the opportunity to apply appropriate notations for different aspects of the system; however, the various UML notations are not orthogonal or disjoint in scope. As a consequence, UML has a *consistency* issue: different diagrams may put contradictory constraints on some element of the described system. This is in addition to structural consistency requirements (or better: well-formedness conditions) of UML models, which may be represented as OCL constraints in the UML meta-model [14].

For architectural descriptions in UML in particular, a *viewpoints* [16] method has been proposed [11]. In such a context, consistency is an even more prevalent issue [15]. However, the problem of determining consistency between two UML "viewpoint" models is not fundamentally different from that of determining consistency within a single UML model. In either case, the source of any inconsistency will be a collection of diagrams (or OCL constraints, etc.) which, between them, put unsatisfiable restrictions on some element of the system. In a "viewpoints" situation, the possibility increases that these are diagrams of the same type, e.g., state charts describing the same class.

Most approaches to consistency checking for UML rely on a formalisation of all UML diagrams in a single semantic framework, which may be CSP [6, 12], algebraic specification [2], CSP-OZ [23], etc. UML consistency is then reduced to consistency of the underlying framework. Such approaches to consistency checking are non-constructive: they provide no feedback in terms of *UML* indicating whether consistency held or not. The Promela based approach of Lilius [20], where model checking is used to generate a counter-example of consistency, is nominally constructive, as it generates UML sequence diagrams.

In a fully constructive approach to consistency, for example the one for Z [4], the underlying semantic model remains hidden. Inconsistencies are thus reported in terms of the specifications themselves. Moreover, also the result of a successful consistency check is represented as a specification, which is called the *unification*. Unifications can also be used as the basis for incremental consistency checking of $n > 2$ specifications. See [3, 5] for a full discussion of consistency checking based on unifications.

In this paper, we explore some possibilities for constructive consistency checking in UML. The completeness of a method for this would, obviously, require a complete formal semantics for UML, which is not our concern at this point. Instead, we consider intuitive (maybe even naive!) formalisations of UML fragments whose meaning seems non-controversial, and explore whether these formalisations can substantiate, again, intuitive "unifications" in UML. These unifications are in the same spirit as rules for "joining" in Catalysis [10]. Unsurprisingly, we will use Z for the formalisation of UML, as the constructive method for Z may lead to unifications that may have direct counterparts in UML. Indeed, the aim of our work is not to show adequacy of Z as an underlying semantic framework for UML (see [18, 13] and others), but the degree to which UML itself can have constructive unification.

In the next section we present two small UML models, and consider an intuitive unification of them. Then, in Section 4 we translate the example into Z, apply the Z unification techniques to the translation, and evaluate the result. Section 5 presents some conclusions.

3 Example: Flight Booking System

In this example, we consider two views of a flight booking system. In both cases, the UML model consists of a class diagram and a state diagram.

In the first viewpoint, reservations can be made, but not cancelled. Moreover, the flight itself is not subject to cancellation. The class structure of this model (Figure 1) comprises three classes:

- BookingS describes a basic booking system: openFlight for opening a new flight, close for closing a flight and reserve for reserving a seat on a flight;
- Flight contains a basic attribute of a flight: the number of free seats seats noFreeSeats;
- Seat represents for a seat, whether it is reserved or free.

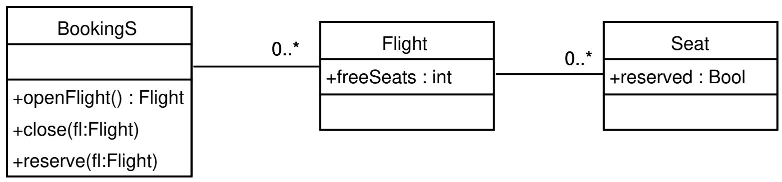

Fig. 1. First viewpoint: class diagram

Associations are defined between the classes: multiple instances of class Seat are associated with an instance of Flight, and multiple flights are associated with the BookingS class.

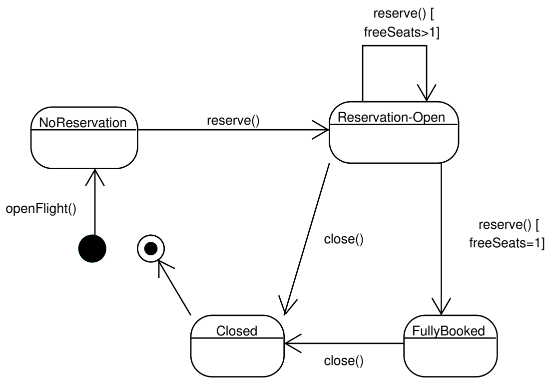

Fig. 2. First viewpoint: state chart

The state diagram (Figure 2) contains four states, apart from the initial and final ones: NoReservation (corresponding to the system state immediately after the flight was opened), ReservationOpen (the state where the flight has seats booked and the booking process is still ongoing), FullyBooked (the state where all seats are booked) and Closed (the seats booking process has stopped – for example two hours before take off). Observe that a flight can be closed even if not all seats are reserved (the transition from ReservedOpen to Closed).

In the second viewpoint, flights and reservations may be cancelled.

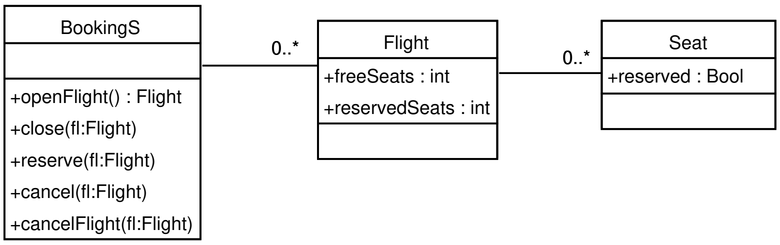

Fig. 3. Second viewpoint: class diagram

The class diagram (Figure 3) has a very similar structure to that of the first viewpoint. Two more methods have been added to the Book_Sys class (corresponding to BookingS): cancel for cancelling a seat reservation and cancelFlight for cancelling a flight. Also, rather than recording just the number of free seats, we also record the number of reserved seats. The state diagram (Figure 4), com-

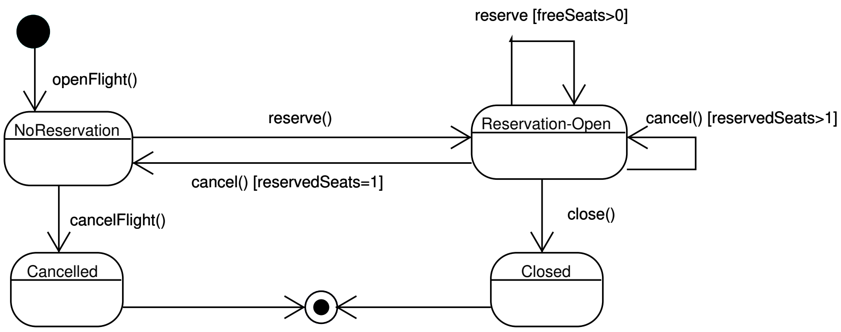

Fig. 4. Second viewpoint: state chart

pared to the previous one, has an extra state for cancelled flight, but does not identify fully booked flights. It also adds transitions corresponding to cancellations of seats or entire flights.

An intuitive unification of the two viewpoints consists of the second class diagram (Figure 3, which contains more information than the first), and a state diagram (Figure 5) which represents all the states and transitions available in the previous ones.

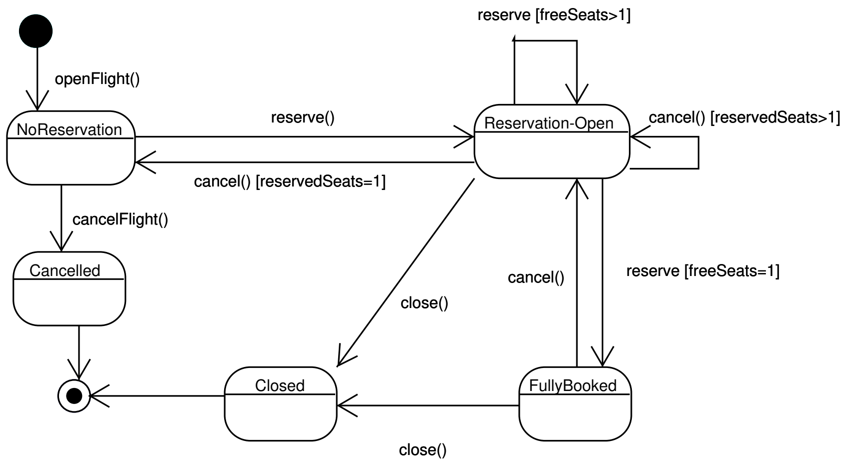

Fig. 5. Unified state chart

4 Unification from Z

Here, we first formalise the UML diagrams in Z. We will concentrate on the behaviour of an individual flight. The translation is similar in spirit to that by France et al. [18], formalising the behaviour of an individual object in Z. In order to also model aggregations, associations, object creation, etc., one would either need to create a layer of object instances on top of this as in [18], or move to an object-oriented variant of Z such as Object-Z [24]. Other formalisations in Z of UML, such as by Evans et al. [13], model UML concepts (such as objects, classes, etc.) as Z schema types – that kind of formalisation is less useful here, as we are not interested in refining UML *concepts* but UML models.

After formalising the models, we derive a unification of these formalisations, illuminating desirable properties for UML refinement relations as they appear.

4.1 First viewpoint

The class diagram of the first viewpoint (Figure 1) tells us it has an attribute $freeSeats : int$; according to the state diagram (Figure 2), it can also be in a number of states. We abbreviate their names in the Z specification.

$$States_1 ::= NoRes \mid ResOpen \mid Full \mid Closed$$

As there are no transitions leading into the initial state, nor labelled transitions into the final state, we can omit those for now. The transition leading out of the initial state corresponds to the initialisation.

$$\begin{array}{|l}\hline \textit{Flight}_1 \\ s_1 : \textit{States}_1 \\ \textit{freeSeats} : \mathbb{Z} \\ \hline\end{array} \qquad \begin{array}{|l}\hline \textit{Init}_1 \\ \Delta \textit{Flight}_1 \\ \hline s_1' = \textit{NoRes} \\ \hline\end{array}$$

When doing such formalisations "by hand", it is very tempting to introduce extra information which is not present in the UML though evident from our "understanding" of the real-life situation represented. However, we do not include any constraint on $\textit{freeSeats}'$ in *Init* as none is implied in the diagrams.

Operations correspond to the disjunctions of *all* transitions in the state diagram which have the same label.

$$\begin{array}{|l}\hline \textit{Reserve}_1 \\ \Delta \textit{Flight}_1 \\ \hline (s_1 = \textit{NoRes} \land s_1' = \textit{ResOpen}) \lor \\ (s_1 = \textit{ResOpen} \land \textit{freeSeats} > 1 \land s_1' = \textit{ResOpen}) \lor \\ (s_1 = \textit{ResOpen} \land \textit{freeSeats} = 1 \land s_1' = \textit{Full}) \\ \hline\end{array}$$

Again, it is tempting to include a predicate like $\textit{freeSeats}' = \textit{freeSeats} - 1$, or to assume $\textit{freeSeats}' > 0$ for the initial state, but it is not actually implied.

$$\begin{array}{|l}\hline \textit{Close}_1 \\ \Delta \textit{Flight}_1 \\ \hline s_1 = \textit{ResOpen} \lor s_1 = \textit{Full} \\ s_1' = \textit{Closed} \\ \hline\end{array}$$

In our class diagrams, all attributes were declared public. This was done, however, because the decision on visibility should be postponed until a later stage. In order to fully represent the attributes being public, we would need to include Z operations representing accessors and mutators on all attributes as well.

4.2 Second viewpoint

There are four states in the second state chart (Figure 4), with the initial and final states omitted for the same reason as above.

$$\textit{States}_2 ::= \textit{NoRsv} \mid \textit{RsvOpen} \mid \textit{Clsd} \mid \textit{Cancld}$$

Note that we use different abbreviations for the state names, in order to distinguish between the two viewpoints.

$$\begin{array}{|l}\hline \textit{Flight}_2 \\ s_2 : \textit{States}_2 \\ \textit{free}, \textit{reserved} : \mathbb{Z} \\ \hline\end{array} \qquad \begin{array}{|l}\hline \textit{Init}_2 \\ \Delta \textit{Flight}_2 \\ \hline s_2' = \textit{NoRsv} \\ \hline\end{array}$$

The operations are once again disjunctions of transitions with the same label.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Reserve_2 \quad}} \\
\Delta Flight_2 \\
\hline
(s_2 = NoRsv \land s_2' = RsvOpen) \lor \\
(s_2 = RsvOpen \land free > 0 \land s_2' = RsvOpen) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Cancel \quad}} \\
\Delta Flight_2 \\
\hline
s_2 = RsvOpen \\
reserved > 1 \Rightarrow s_2' = RsvOpen \\
reserved = 1 \Rightarrow s_2' = NoRsv \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CancelFlight \quad}} \\
\Delta Flight_2 \\
\hline
s_2 = NoRsv \land s_2' = Cancld \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Close_2 \quad}} \\
\Delta Flight_2 \\
\hline
s_2 = RsvOpen \land s_2' = Clsd \\
\hline
\end{array}
$$

4.3 Unification

For the states-and-operations style in Z, the process of unification is described in detail in [4]. Lacking space to repeat all details, we illustrate this process by example.

This sample unification approximates a development relation for UML in the following way. Unification in Z is defined in terms of *refinement*: it is the "least" (-prescriptive) common refinement of two specifications. A hierarchy of refinement relations in Z exist, see [9], leading to slight differences in the rules for unification. Viewing this in reverse: we should select the parameters for Z unification of the UML formalisations in such a way that we end up with a Z formalisation of the combined state diagram above. In this way, we establish desirable properties for UML refinement relations, which we will highlight as they appear. This we consider to be the main contribution of the paper.

State unification As it will turn out later, the crucial step in the unification is the identification of its state space. If the state spaces of the two views are identical, this step can be avoided. However, the schemas $Flight_1$ and $Flight_2$

are different. In terms of the required refinement relation, this implies that we need *data refinement*, rather than the simpler *operation* or *algorithmic* refinement. In other words, in refinement we should not just remove underspecification (non-determinism) from the operations, but also allow change of the internal detail (names and types of private attributes) of the classes.

> **Requirement 1:** UML refinement needs to include *data* refinement, to allow for hidden state with abstraction from internal details.

In order to continue unification, we need to establish the relationship between the variables in the two state schemas. This will be documented in a combined schema, which we call a *correspondence*. We define this in a number of steps, starting with the non-controversial relations between the states.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{Corr}_0 \quad}} \\
\mathit{Flight}_1 \\
\mathit{Flight}_2 \\
\hline
\mathit{free} = \mathit{freeSeats} \\
s_1 = \mathit{NoRes} \Leftrightarrow s_2 = \mathit{NoRsv} \\
s_1 = \mathit{Closed} \Leftrightarrow s_2 = \mathit{Clsd} \\
\hline
\end{array}
$$

Next, we are faced with the problem that the second viewpoint has a variable *reserved* that does not occur in the first viewpoint. We might reasonably assume that *free*+*reserved* remains constant in the second view; let us name this constant *capacity*. As reservation is possible from the initial state, its value should be at least 1.

$$
\begin{array}{|l}
\mathit{capacity} : \mathbb{N}_1
\end{array}
$$

The next issue is with the state *RsvOpen*, from which $\mathit{Reserve}_2$ may or may not succeed, depending on the value of *free*; this does not quite match the state *ResOpen* from which $\mathit{Reserve}_1$ is always defined. This is addressed by stating that the state *RsvOpen* is represented by either *ResOpen* (when there are free seats) or *Full* (when there are none). This completes the correspondence relation[1]:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{Corr} \quad}} \\
\mathit{Corr}_0 \\
\hline
\mathit{free} + \mathit{reserved} = \mathit{capacity} \\
s_1 = \mathit{Full} \Leftrightarrow s_2 = \mathit{RsvOpen} \land \mathit{free} \leq 0 \\
s_1 = \mathit{ResOpen} \Leftrightarrow s_2 = \mathit{RsvOpen} \land \mathit{free} > 0 \\
\hline
\end{array}
$$

Note that we could have chosen many other correspondences, but the current one represents our intuition about the specifications, including the supposition

[1] Inclusion of a Z schema's name in a new schema implies inclusion of all its declarations *and* predicates.

that they are actually consistent. Unlike in the mechanical process of formalising the UML diagrams, it *is* necessary to insert knowledge about the intention of the specifications at this stage. In general, (automatic) unification or consistency checking requires explicit linking of concepts between specifications more sophisticated than identification by name: identifying *RsvOpen* and *ResOpen* would have led to the detection of an inconsistency.

Totalisation A correspondence relation like *Corr* might serve as the state space for the unification already. However, in this case it will not be sufficient: there is no mention of the state *Cancld*. *Corr* represents a kind of dependent product of the two state spaces, in relational data base terms: a "theta-join". What is required for the combined state space (in order to guarantee refinement) is an "outer join" which contains representations of every member of both viewpoint state spaces; one might also view this as a *totalisation* of the correspondence relation. In this case, the outer join would link *Cancld* to a "null" value for $Flight_1$; because the variables *free* and *reserved* play no rôle in *Cancld*, we present a simpler equivalent schema where only $States_2$ is extended with a null value.

$$States_1 ::= NoRes \mid ResOpen \mid Full \mid Closed \mid Null$$

The full state schema for the unification is then (using the latest definition of $States_1$):

$$\begin{array}{|l}
\multicolumn{1}{l}{TCorr} \\
\hline
Corr \\
\hline
s_1 = Null \Leftrightarrow s_2 = Cancld \\
\hline
\end{array}$$

Adaptation Once we have defined a common state schema, all viewpoint operations can be "adapted" to operate on this common state. Technically, this is a data refinement step, determining the least refinement [8] for the retrieve relation which injects the viewpoint state space into the common one. Syntactically, it involves little more than a change of the state name. The first viewpoint's operations become

$$\begin{array}{|l}
\multicolumn{1}{l}{AdReserve_1} \\
\hline
\Delta TCorr \\
\hline
(s_1 = NoRes \land s_1' = ResOpen) \lor \\
(s_1 = ResOpen \land freeSeats > 1 \land s_1' = ResOpen) \lor \\
(s_1 = ResOpen \land freeSeats = 1 \land s_1' = Full) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{AdClose_1} \\
\hline
\Delta TCorr \\
\hline
s_1 = ResOpen \lor s_1 = Full \\
s_1' = Closed \\
\hline
\end{array}$$

where additional requirements are hidden in $\Delta TCorr$ – in particular, variables from $Flight_2$ are constrained to preserve the correspondence. The operations of the second viewpoint are adapted analogously.

Operation unification Any data refinement can be factored into a "least" data refinement and an operation refinement – operation adaptation handled the former, so we can restrict ourselves to operation unification now. This is a well-known technique, first described for Z by Ainsworth et al [1]. In Catalysis, it is used for "joining" pre/post-specifications, but [10] does not draw the link with refinement.

Two operations Op_1 and Op_2 defined in terms of ΔS are consistent, and can thus be successfully unified [1, 4] whenever

$$\text{pre } Op_1 \wedge \text{pre } Op_2 \Rightarrow \text{pre}(Op_1 \wedge Op_2)$$

i.e., whenever they are both applicable, a result consistent with both exists. In that case, their unification is

$$\begin{array}{|l}
\multicolumn{1}{l}{Op_1 \oplus Op_2} \\
\Delta S \\
\hline
\text{pre } Op_1 \vee \text{pre } Op_2 \\
\text{pre } Op_1 \Rightarrow Op_1 \\
\text{pre } Op_2 \Rightarrow Op_2 \\
\hline
\end{array}$$

This definition covers the situation where we assume that a precondition represents a "contract": within the given area, the operation should deliver the right result, and outside this area, we do not care. This is the common assumption for Z; however, for behavioural interpretations we may also view the precondition as a "guard": outside the precondition the operation should be disabled. In this interpretation, we additionally require that $\text{pre } Op_1 = \text{pre } Op_2$ for consistency, which means that the unification reduces to $Op_1 \wedge Op_2$. For a detailed discussion of "guards" vs. "contracts", see [9].

The preconditions of the two adapted *Reserve* operations coincide: either there are no reservations yet, or reservation is open and there are still free seats. The latter condition is encoded in the state in the first view, and in the variable *free* in the second view. Their outcomes are also consistent, and lead to

$$\begin{array}{|l}
\multicolumn{1}{l}{AdReserve_1 \oplus AdReserve_2} \\
\Delta TCorr \\
\hline
(s_1 = NoRes \wedge s_1' = ResOpen) \vee \\
(s_1 = ResOpen \wedge freeSeats > 1 \wedge s_1' = ResOpen) \vee \\
(s_1 = ResOpen \wedge freeSeats = 1 \wedge s_1' = Full) \\
\hline
\end{array}$$

where all the effects of $AdReserve_2$ are already implied by the combination of $AdReserve_1$ and $TCorr'$.

$AdClose_1$ and $AdClose_2$ are also equivalent to each other, and thus also to their unification.

The first viewpoint has no further operations; the second has also *Cancel* and *CancelFlight*. Formally, we can treat this by the operations being defined with a precondition of *false* in the first viewpoint; the unifications would then be *Cancel* and *CancelFlight*. The fact that we would unify two operations with different preconditions shows that we do need to take the "contract" interpretation of preconditions.

> **Requirement 2**: UML refinement should allow widening of preconditions.

As we could not actually distinguish in a state diagram between an operation that is never applicable (appearing as 0 arrows labelled with its name), and one that does not exist, we also have:

> **Requirement 3**: UML refinement should allow introduction of extra operations.

For a solid grounding of this requirement in an OO context, see the "constraint rule" of Liskov and Wing [21].

Finally, observe that the unified Z specification is indeed equivalent to the intuitive formalisation of the state diagram we presented earlier (Figure 5). There are five states, corresponding to the four states of the first view plus the *Cancld* state of the second view. The *RsvOpen* state of the second view is split into two states, depending on whether $free > 0$. The only information that the formal unification adds, compared to Figure 5, is that $freeSeats = 0$ in state FullyBooked. It is not surprising that this information is absent from the intuitive UML unification, as it was propagated from one state into the other via the correspondence relation (which we established during unification only). For more detailed models with OCL constraints specifying the effect of the operations on the given states, there would not have been similar "surprises". All in all, in this simple case study, the UML unification is consistent with the Z unification once the correspondence relation has been accounted for.

5 Concluding Comments

This small case study has substantiated a number of intuitions about the nature of possible refinement relations in UML. In their nature, they should be similar to refinement for Z (and B, and VDM, etcetera). However, in accordance with the Catalysis approach, we believe that a more complicated case study would have introduced additional requirements, leading to a more liberal notion of refinement.

First, the operations we formalised had no inputs or outputs. UML practice appears to allow for inputs and outputs to be elided when they are not relevant – as a consequence, we would need to allow *interface refinement* [22], which is a sound generalisation of normal refinement, see [9, Chapter 10].

Second, decomposition of (inter)actions plays an important rôle in UML. This could be addressed by having unification based on a notion of *action refinement*, which is an independent and compatible generalisation as well [9].

Clearly, issues of consistency, unification, and refinement could have been addressed on the basis of a different formalisation as well. In particular, a CSP formalisation [6, 12] seems an obvious candidate. However, defining the correct refinement relation on the CSP side is non-obvious, for example because most CSP refinement relations do not allow widening of guards, or because unifications in process algebra are likely to be expressible semantically but not syntactically [25].

References

1. M. Ainsworth, A. H. Cruickshank, P. J. L. Wallis, and L. J. Groves. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, February 1994.
2. E. Astesiano and G. Reggio. An attempt at analysing the consistency problems in the UML from a classical algebraic viewpoint. In *Recent Trends in Algebraic Development Techniques, Selected Papers of the 15th International Workshop WADT'02*, Lecture Notes in Computer Science. Springer, 2003. To appear.
3. E.A. Boiten and J. Derrick. Integration of specifications through development relations. In H. Ehrig, B.J. Krämer, and A. Ertas, editors, *IDPT 2002*. SDPS, 2002. Extended version to appear in Journal of Design and Process Science, September 2003.
4. E.A. Boiten, J. Derrick, H. Bowman, and M. Steen. Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75, September 1999.
5. H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency. *Formal Methods in Systems Design*, 21:111–166, 2002.
6. J. Davies and C. Crichton. Concurrency and refinement in the Unified Modelling Language. *Formal Aspects of Computing*, 2003. Special issue on REFINE'02, to appear.
7. J. Derrick, D. Akehurst, and E.A. Boiten. A framework for UML consistency. In Kuzniarz et al. [19], pages 30–45.
8. J. Derrick and E.A. Boiten. Calculating upward and downward simulations of state-based specifications. *Information and Software Technology*, 41:917–923, July 1999.
9. J. Derrick and E.A. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. FACIT. Springer Verlag, May 2001.
10. D.F. D'Souza and A.C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
11. A. Egyed and N. Medvidovici. Extending architectural representation in UML with view integration. In France and Rumpe [17], pages 2–16.
12. G. Engels, J.M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioural models. In V. Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 186–195. ACM Press, 2001.

13. A. Evans, R.B. France, K. Lano, and B. Rumpe. The UML as a formal modelling notation. In Jean Bézivin and Pierre-Alain Muller, editors, *UML 1998*, volume 1618 of *Lecture Notes in Computer Science*, pages 336–348. Springer, 1999.
14. A.S. Evans and S. Kent. Meta-modelling semantics of UML: the pUML approach. In France and Rumpe [17], pages 140–155.
15. A.C.W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.
16. A.C.W. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal on Software Engineering and Knowledge Engineering, Special issue on Trends and Research Directions in Software Engineering Environments*, 2(1):31–58, March 1992.
17. R. France and B. Rumpe, editors. *UML'99*, volume 1723 of *Lecture Notes in Computer Science*. Springer, 1999.
18. R.B. France, J.-M. Bruel, and M.M. Larrondo-Petrie. An integrated object-oriented and formal modeling environment. *Journal of Object Oriented Programming*, 10(7):25–34, 1997.
19. L. Kuzniarz, G. Reggio, J.L. Sourrouille, and Z. Huzar, editors. *UML'02 Workshop on Consistency Problems in UML-based Software Development*, Research Report 2002:06. Blekinge Institute of Technology, 2002.
20. J. Lilius and I.P. Paltor. vUML: a tool for verifying UML models. In R.J. Hall and E. Tyugu, editors, *Proceedings of ASE'99*, pages 255–258. IEEE Computer Society, 1999.
21. B. Liskov and J. M. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
22. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Application and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 82–101. Springer-Verlag, September 1997.
23. H. Rasch and H. Wehrheim. Consistency between UML classes and associated state machines. In Kuzniarz et al. [19], pages 46–60.
24. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
25. M. W. A. Steen, H. Bowman, and J. Derrick. Composition of LOTOS specifications. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing and Verification, XV*, pages 73–88, Warsaw, Poland, 1995. Chapman & Hall.