Constructive consistency checking for partial specification in Z

Eerke Boiten, John Derrick,
Howard Bowman and Maarten Steen
Computing Laboratory, University of Kent,
Canterbury, CT2 7NF, U.K.
(Phone: +44 1227 827615,
Email: E.A.Boiten@ukc.ac.uk) *

Revised version, December 1997

1 Introduction, goals and context

Partial specification, or specification by *viewpoints*, has arisen as a desirable method of specifying complex systems in several contexts, particularly in requirements engineering [24] and for example in Open Distributed Processing [29]. The central idea is that a specification consists of a collection of interlocking partial specifications, each of which describes the envisaged system from a different viewpoint. In particular the specification notation Z [39] is often advocated as a suitable language for this style of specification [1, 30, 37]. However, for collections of partial specifications to be meaningful, *consistency* between them has to be established. In the existing literature on viewpoint specification, no satisfactory general solution for this is given. This paper describes how to check consistency between partial specifications in Z, i.e. how to establish that different partial specifications of one system do not impose contradictory requirements. Using the traditional refinement relation in Z, we present techniques for constructing *unifications* (least common refinements) of partial specifications, which represent their combined requirements. Three relatively simple conditions on the partial specifications and the predicate that relates them characterise consistency.

The next subsections describe viewpoint specification and a formal framework for consistency checking for viewpoint specification. Section 2 is a brief introduction into Z, its "states-and-operations" specification style, and its refinement relation. Section 3 describes the parameters of viewpoint unification. In a naive approach, these are only the partial specifications themselves, but in non-trivial cases an extra parameter turns out to be necessary: a correspondence between types used in the various viewpoints. A complementary approach is to map the types explicitly to a new type, and finally it is shown how these extra parameters can be left implicit by designating default values for them. Section 4 then goes on to present the unification algorithm. Section 5 contains a proof that the unification is a least common refinement of the viewpoints – the conditions for consistency appear as extra assumptions necessary to complete this proof. Section 6 presents some variations and extensions to the simple unification algorithm, embedding it in a software development model. Section 7 then compares our work to related approaches and techniques for partial specification. Section 8 describes our conclusions and our ideas on how to proceed to make Z even more useful for partial specification.

This paper is based on [8], extending its results and significantly extending its context.

*This work was partially funded by the U.K. Engineering and Physical Sciences Research Council under grant number GR/K13035 and by British Telecom Labs., Martlesham, Ipswich, U.K.

1.1 Viewpoint specification

It is generally agreed that systems of a realistic size cannot be specified in single linear specifications, but rather should be decomposed into manageable chunks which can be specified separately. The traditional method for doing this is by hierarchical and functional decomposition. Nowadays, it is often claimed [31] that this is not the most natural or convenient (in relation to "perceived complexity") method – rather systems should be decomposed into different *aspects*. For each such *viewpoint* a specification of the system restricted to that particular aspect should be produced. Such *partial specifications* may omit certain parts of the system, because they are irrelevant to the particular aspect, and need not describe certain behaviours because they do not concern that specific viewpoint. Descriptions of this nature seem particularly appropriate for systems with various kinds of "users", each with their own view of the system.

> Imagine, for example, the views of a library system that library managers, loan officers, clients, system operators, and programmers of the system would have. In the library manager's view a book has a price which is essential in the operation of buying a book, but which none of the other views would be much interested in. The loan officer's view of lending out a book would include updating the library statistics, which would not appear in a client's view of lending a book.

Another reason which is often given for decomposing problems into aspects rather than subproblems is that this "horizontal" subdivision would give a more natural separation of concerns. In particular, it allows each aspect to use specialised specification languages, for example dataflow diagrams for control flow, process algebras for "behaviour", data definition languages, et cetera. A final argument in favour of viewpoint specification is that it supports fluid system development. The various viewpoint specifications can be gradually developed, often based on changes made to other viewpoints. To some extent this could even occur in parallel, in particular while the specification is completed to reflect all requirements.

One particular area in which viewpoint specification plays an important role, and our initial motivation to study viewpoint specification and consistency, is in *Open Distributed Processing* (ODP), an ISO/ITU standardisation framework. The ODP reference model [29] defines five viewpoints for the specification of open distributed systems: *enterprise*, *information*, *computational*, *engineering* and *technology*. These viewpoints are *static* in the sense that there is a fixed set of viewpoints, each targeting a predefined aspect of the system (as opposed to viewpoints in other methods). The use of formal description techniques in specifying these viewpoints is envisaged – in particular, Z is a strong candidate to be used in the information viewpoint [37]. For an overview of our project on the technical issues behind viewpoint specification for ODP, see [5].

The techniques described in this paper, however, are not specific to ODP specification. The techniques for Z could also be used to formalise the ad-hoc treatment of unification in [1], and in section 7.2 we demonstrate how our methods subsume some of those used for specification by "views" in [30]. Our general approach to consistency checking (as described in other papers and summarised in the next section) also applies to other viewpoint- or multiple paradigm specification styles (e.g. [33, 43, 28, 14]), in particular when the specifications languages involved are *formal* ones.

1.2 Consistency checking and unification

There is one serious technical problem in partial specification. Some elements (operations, variables, etc) of the envisaged system will be modelled in more than one viewpoint, and those descriptions will not in general be identical. Different viewpoints have different perspectives of the system, and they are likely to use different specification languages (for ODP the latter is a near certainty). This gives rise to an obligation to ensure that the partial specifications do not pose *contradictory* requirements: we need to check for *consistency*, potentially between descriptions in different languages and at different levels of abstraction.

However, first we need to define what it means for a collection of viewpoint specifications to be consistent. Viewing the specifications as predicates over some universe, the logical definition of consistency is that it is impossible to derive both some proposition and its negation from the combined viewpoints.

In the context of specification and development of a concrete system, however, this abstract logical approach does not seem too useful. What *is* the universe we are quantifying over, and how do we map our specification language(s) to predicates over that universe? Would not a common semantic basis for possibly multiple languages necessarily be at such a low level that performing any kind of consistency proof becomes extremely laborious [43]? Would it not make any arising inconsistencies hard to trace back to the original specifications? (For a more extensive discussion of these issues, cf. [6].) What do we mean by "the combined viewpoints", will it always just be the logical conjunction of their formal interpretations, or do we need a more complex operator for combining viewpoints? Our general answer to these questions is extensively described in [13] and summarised below – the concrete answer for Z specifications makes up the rest of this paper.

A more constructive view of consistency is one that is oriented towards system development. Instead of providing direct semantics for the specification languages, we encode our view of what specifications mean in *development relations*. Two specifications are in such a development relation if we consider one to be a correct (in the sense that it respects the requirements) development of the other on the way to an eventual implementation. A development relation may cross a language boundary, examples of such relations are semantics and translations, or it may not, in which case *refinement* relations and equivalences form the main example. Note that another view of these development relations is that they provide a development-based *semantics*: the meaning of a specification is the set of all specifications that can be developed from it.

Consistency checking is then defined as follows. Given a set of initial specifications, each with their associated development relation, does a viable implementation of all of them exist? That is, does a common image of each of these initial specifications under their respective development relations exist?

This definition of consistency gives little guidance on how to actually establish consistency between a collection of viewpoint specifications – generating the set of all implementations of each viewpoint and then computing emptiness of the intersection of those implementation sets is unrealistic if not impossible. We propose repeated *unification*[1] of pairs of specifications as a constructive method of consistency checking. The unification of two specifications should have all the requirements imposed by both specifications. Formally this means that it should be a common image of the viewpoints through their respective development relations – in other words, a *witness* to binary consistency between the viewpoints. Moreover, if the unification is to be used in consistency checking with a third viewpoint, it should impose no extra requirements besides those contained in the first two viewpoints, or else consistency checking might unnecessarily fail. Formally this amounts to choosing the most general unification – if the development relations involved induce an ordering, we need to choose the *least* unification, where "least" is understood in the sense of fewest development steps done, least detail added, etc. This guarantees that a unification of all viewpoints, if it exists, can be found through a series of binary consistency checks.

In practice it is often convenient to construct unifications in two steps. First one generates a *candidate* least common development, i.e. some specification that *is* the least unification *if* one exists, and then one performs some consistency tests on it to determine whether it actually *is* a least unification. We will call such candidates "unifications" as well (using the term in a slightly sloppy sense). Finally note that it is strictly speaking often incorrect to talk about *the* least unification of a collection of viewpoint specifications, since for most specification languages and development relations there will be many, often equivalent, ones.

[1] It could be argued that this term is too technically loaded, and that one should use a different term, e.g. *amalgamation*[1], however there are enough parallels for us to maintain this term.

2 Technical preliminaries

This section introduces some of the basic technical material on Z. The reader is referred to [39] for a complete description of Z. Here we will present first a brief overview of the main aspects of the "states-with-operations" specification style in Z, and then the refinement relation for it. The last two subsections, on equivalence relations and unary consistency, are concerned with less standard material, but are best treated here to avoid interrupting the flow of the story of later sections.

2.1 Z: states and operations

Although there is no fixed interpretation (apart from the semantics) of Z specifications, in practice usually the states-with-operations style of specification described here is used. The idea is that some schemas are state schemas representing a state space, and other schemas of a particular form represent operations on this state. A state schema with a collection of operations defined on it and an initialisation schema together form an *abstract data type* (ADT).

The general form of a schema is $Sname \mathrel{\widehat=} [\, Decls \mid Preds \,]$ or, graphically

$$\begin{array}{|l}
\hline Sname \\
Decls \\
\hline
Preds \\
\hline
\end{array}$$

where $Sname$ is an identifier denoting the schema's name; $Decls$ is a series of declarations of the form $x{:}S$ where x is the name of a component of the schema, and S is a set to which x should belong; $Preds$ is a list of predicates (whose meaning is the conjunction of them all). Actually, declarations may also be references to schemas, see below. If $Preds \equiv$ true, it may be omitted (including the line above it). The meaning of a schema is a set of records that have as labels all the components declared in $Decls$ (and all those imported through schema references), and whose values satisfy $Preds$.

Example 1 The meaning of

$$\begin{array}{|l}
\hline Num \\
x{:}\mathbb{N} \\
\hline
x \geq 2 \\
x \leq 4 \\
\hline
\end{array}$$

is the set (using Spivey's [39] unofficial notation for "bindings", i.e. labeled records) $Num{=}\{\langle\!| \ x{:}2 \ |\!\rangle, \langle\!| \ x{:}3 \ |\!\rangle, \langle\!| \ x{:}4 \ |\!\rangle\}$.

□

If one of the declarations is a schema reference, all its declarations and predicates are included as well.

Example 2 The meaning of

$$\begin{array}{|l}
\hline Squares \\
y{:}\mathbb{N} \\
Num \\
\hline
y = x * x \\
\hline
\end{array}$$

is the set $Squares{=}\{\langle\!| \ x{:}2, y{:}4 \ |\!\rangle, \langle\!| \ x{:}3, y{:}9 \ |\!\rangle, \langle\!| \ x{:}4, y{:}16 \ |\!\rangle\}$.

□

Schema references need not be just schema names. One can also use *decorations* to the name, like accents, subscripts, exclamation marks, and question marks. In a schema reference this returns the schema with the decoration applied to all its components and predicates. So for example Num' has component x' , and $Squares_1$ has components x_1 and y_1. When a schema reference is used as a predicate, this stands for the schema's predicate *including* all the restrictions on its components. In fact, a *calculus* of Z schemas exists, whose operators are the usual logical operators. Resulting schemas contain all the components, with their predicates combined according to the logical operators. Quantification results in hiding of components, for more details cf. [39].

The states-with-operations style has a particular interpretation for this. A schema with no decorated components (for example Num or $Squares$) is usually assumed to be a definition of a state space. A schema which contains both the state space and its primed decoration is an operation on this state space. The interpretation of an operation schema is that the primed state is the state after the operation, and the unprimed state is the state before.

Example 3

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Toggle\;}} \\
Num;\ Num' \\
\hline
x + x' = 6 \\
\hline
\end{array}
$$

Formally, $Toggle$ represents $\{[x{:}2\,,\,x'{:}4],[x{:}3\,,\,x'{:}3],[x{:}4\,,\,x'{:}2]\}$ but this is *interpreted* as an operation which changes x into $6-x$.

□

Operations may be non-deterministic (more than one possible x' for a particular x – for example if x' does not occur in the predicate) or partial (no possible x' for a particular x). As an abbreviation for $Sname;\ Sname'$ one can use $\Delta Sname$. Operations may have inputs, which are by convention all variables decorated with ?, and variables decorated with ! as outputs. Initialisation operations are often denoted as operations with no "before" state.

Various object oriented variants of Z exist [40], which encapsulate the ADT by (essentially) drawing a schema box around it.

Because of schema references, and because in a declaration of the form $x{:}S$ an *arbitrary* set S may be used instead of the type of x, schemas can be turned into equivalent ones by moving restrictions between the predicate and the declarations.

Example 4 Because $\mathbb{N} \subseteq \mathbb{Z}$, this is an alternative definition of Num:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;NumToo\;}} \\
x{:}\mathbb{Z} \\
\hline
x \geq 0 \\
x \geq 2 \\
x \leq 4 \\
\hline
\end{array}
$$

□

In fact, because in the Z type system $\mathbb{Z}$ is not included in a larger type, $NumToo$ is a canonical representation of Num. In general, replacing all schemas by equivalent ones such that all components have a "maximal" type is called schema *normalisation*. In a previous paper [19] on Z unification, we assumed that all schemas were normalised – in the current paper we do *not* make this assumption.

2.2 Pre- and postconditions

Unlike some other specification notations, specifications of operations in Z do not contain explicit pre- and postconditions. However, a unique characterisation of the precondition of an operation schema is possible.

Definition 5 (Precondition) If in an operation schema Op the state schema involved is $State$, and the output components are $x_i!:Out_i$ $(i = 1 \mathinner{..} n)$, then the precondition of Op is defined by

$$\operatorname{pre} Op = \exists\, State' \;;\; x_1!:Out_1 \;;\; \ldots \;;\; x_n!:Out_n \bullet Op$$

□

Thus, the precondition is a predicate[2] on $State$ and the input, characterising those situations where it is possible to find output and after state which relate to them by Op.

It is not possible in Z to give a similar characterisation of the postcondition of an operation, though a notation $\operatorname{post} Op$ for it exists. For a schema $Op \mathrel{\widehat=} [\,\Delta D \mid pred\,]$ which (to avoid some semantic problems) satisfies the condition $pred \Rightarrow \operatorname{pre} Op$, any condition P such that $\operatorname{pre} Op \land P \Leftrightarrow pred$ will do as "the" postcondition, in particular $pred$ itself. Thus any occurrence of $\operatorname{post} Op$ in the sequel should be taken to refer to *some* possible postcondition of Op.

2.3 Refinement

An abstract data type consists of a state schema, an initialisation schema for that state, and a collection of operation schemas on that state. Such an ADT can be *refined* by resolving some of the nondeterminism in the operations, and/or by extending the applicability of operations. The ADT we start from is usually called the *abstract* ADT, and the refined one the *concrete* ADT. For an extensive description of refinement in Z, cf. [42], which also covers backwards simulation based refinement – this paper considers forward simulation only.

Two types of refinement are distinguished, namely *operation refinement* which changes *only* one of the operations of the ADT, and *data refinement* which changes the state schema, and as a consequence also needs to replace all operations and the initialisation by ones operating on the new state.

2.3.1 Operation refinement

Operations can be refined in two ways: by extending their domain of definition (i.e. weakening their precondition), or by making them more deterministic. If AOp and COp are both operations on the same state $State$, both with input $x?:X$ and output $y!:Y$, then the conditions for COp to be an operation refinement of AOp[3] are the following:

termination COp should be defined ("guaranteed to terminate") everywhere where AOp is:
$\forall\, State \;;\; x?:X \bullet \operatorname{pre} AOp \Rightarrow \operatorname{pre} COp$

correctness wherever AOp is defined, COp should produce a result that AOp could have produced:
$\forall\, State \;;\; State' \;;\; x?:X \;;\; y!:Y \bullet \operatorname{pre} AOp \land COp \Rightarrow AOp$

2.3.2 Data refinement

In data refinement, the state schema is changed, and thus all operations and the initialisation need to be changed as well in order to operate on the new state. Assume (for simplicity) that the abstract state $AState$ and the concrete state $CState$ have no components with a common name. The abstract and concrete state spaces need to be linked up by a so-called retrieve relation, which is represented by a schema

[2]From the earlier discussion on schema calculus it should be clear that it is actually a *schema* – its components are those of Op except for $State'$ and $x_i!$.

[3]Strictly speaking, for an ADT containing an operation AOp to be refined by an ADT which replaces AOp by COp.

$$\begin{array}{|l}\hline \textit{Retr} \\ AState; CState \\ \hline Pred \\ \hline\end{array}$$

where $Pred$ determines how the elements of the two state spaces are connected. Data refinement is defined with respect to this retrieve relation (though it is often implicitly existentially quantified).

For an ADT $(AState, \{AOp_i \mid i \in I\}, AInit)$ to be refined by an ADT $(CState, \{COp_i \mid i \in I\}, CInit)$ using retrieve relation $Retr$ the following conditions need to hold.

initialisation every concrete initial state needs to match some abstract initial state:
$\forall CState' \bullet CInit \Rightarrow (\exists AState' \bullet AInit \wedge Retr')$

and for every pair of operations $AOp_i, COp_i, i \in I$,

input/output AOp_i and COp_i have the same inputs and outputs, w.l.o.g. assume that these are $x? : X$ and $y? : Y$;

termination COp_i should be defined on all representatives of $AState$ on which AOp_i is defined:
$\forall AState ; CState ; x? : X \bullet \mathrm{pre}\, AOp_i \wedge Retr \Rightarrow \mathrm{pre}\, COp_i$

correctness wherever AOp_i is defined, COp_i should produce a result related by $Retr$ to one that AOp_i could have produced:

$$\begin{array}{l}\forall AState ; CState ; CState' ; x? : X ; y! : Y \bullet \\ \quad \mathrm{pre}\, AOp_i \wedge COp_i \wedge Retr \Rightarrow \exists AState' \bullet Retr' \wedge AOp_i\end{array}$$

The refinement conditions imply that not all elements of the abstract type need to be related to some element of the concrete type, but just those elements which could be reached through the operations. As an extreme case, consider the situation where $Retr$ relates every point in the abstract space to one and the same point in the concrete space. All data refinement conditions hold trivially in that case (with COp_i the identity operation on that one point). Thus, the retrieve relation plays a crucial role in determining data refinement, it needs to be chosen sensibly for data refinement to have significance.

If the retrieve relation is a total function from concrete to abstract state spaces, the conditions become much simpler, cf. [39, 42].

The conditions given above only relate ADTs with matching sets of operations. A question one might ask as well (and one that we will need to ask ourselves later) is whether it is "allowed" to add operations to an ADT in refinement. There are two possible answers to this question:

- The first is based on the strict behavioural view of a Z ADT. From this point of view, adding operations to the "concrete" ADT is problematic, because it changes the behaviour of the ADT in its environment. Adding concrete operations that correspond to the *identity* operation on the abstract state may be less problematic, this depends on the interpretation of divergence. For a further discussion of this issue, which is central in the refinement of internal operations, cf. [16].

  An additional argument for sticking to this interpretation is that the refinement rules for Z were originally *derived* from just such a behavioural characterisation (cf. [42, 32]). If one strays from this view, the validity and usefulness of the existing refinement rules have to be re-examined.

- A second view, which fits better with our use of Z, is that a Z ADT describes a collection of *services* centered around a particular state. If the concrete ADT has an additional service available, this should make no difference to an environment expecting the collection of services of the abstract ADT only.

  Returning to the example of a library, the state of a Library ADT would be a collection of books with loan information for each of them. The Library ADT in the customer's view would have operations that change the loan information on books. However, the customer would not expect the library state to be immutable between his visits. The library manager in his Library ADT would probably have operations adding new books, for example. Adding such operations to a more global view would not invalidate the customer's view of things.

We will give unifications of ADTs matching both of these interpretations – clearly for the second one, a more liberal unification algorithm results.

2.4 Equivalence relations

In the sequel, we will often be discussing state schemas and ADTs which are "equivalent", in different ways.

One possibility for defining an equivalence relation is obvious. Data refinement is a partial order, so by intersection with its converse ("mutual refinement") we obtain an equivalence relation. From the preceding sections it should be clear that mutual refinement is an equivalence relation between ADTs. This implies that in general we need to look at the state schema *and* all the operations and the initialisation in order to establish mutual refinement.

Example 6 Consider

$$\begin{array}{|l} \multicolumn{1}{l}{Three} \\ \hline y{:}\{3\} \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{TInit} \\ \hline Three' \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{Skip} \\ \hline \Delta Three \\ \hline y = y' \\ \hline \end{array}$$

The ADT $(Three, TInit, \{Skip\})$ and the ADT with Num and $Toggle$ from example 3 and an appropriate initialisation are mutual refinements, in both directions with retrieve relation

$$\begin{array}{|l} \multicolumn{1}{l}{Retr} \\ \hline Num; Three \\ \hline x = y \\ \hline \end{array}$$

□

However, sometimes we want to say that two state schemas are "essentially the same" without having to consider them in the context of their collections of operations and initialisations. This equivalence relation we will call *state isomorphism*.

Definition 7 (State isomorphism) Two state spaces S and T are *state isomorphic* if a total injective function between them exists.

□

If S and T are isomorphic, they are essentially the same, modulo an injective relabelling of their elements. There is a clear relationship between state isomorphism and mutual refinement. If the state of an ADT is isomorphic to another state schema, then the operations of the ADT can be translated (using the total injective function) to create an ADT that is in the mutual refinement relation with the original ADT. The example above shows that the reverse is not true.

In summary, there are at least three possible equivalence relations between (state) schemas, which all imply each other in this order. The finest relation is syntactic equality. Then there is semantic equivalence, between schemas which have the same sets of bindings. We will generally even use this as an identity relation on schemas and call schemas "equal" or "identical" when they are "only" semantically equivalent. A slightly coarser one is isomorphism, essentially between schemas which have the same number of bindings. Mutual refinement is a relation between ADTs (rather than between state schemas) which is strictly weaker than state isomorphism.

2.5 Unary consistency in Z

We have discussed consistency *between* specifications, one might guess that this relates to the possibility of having consistency *of* a specification. It might even be the case that inconsistency between specifications shows up in their unification being inconsistent in itself. Unfortunately, this is hardly the case, as we will show later. However, for completeness' sake let us mention some of the ways in which a Z specification on its own could be inconsistent.

First, there are the direct contradictions, which all allow us to prove both P and $\neg P$ for some predicate P, or in other words which allow us to derive "false" from the specification. This is the simplest and most obvious definition of inconsistency in Z. The strong typing system of Z prevents quite a few classes of errors, but some kinds of contradictions can still be written, for example:

- Postulating that an empty set has an element:

  $$x : \varnothing$$

- Abusing the fact that a function is a set of pairs:

  $$\begin{array}{|l} f : \mathbb{N} \to \mathbb{N} \\ \hline f = \{(1,2),(1,3)\} \end{array}$$

  (of course similar examples exist for all the different types of functions, including sequences).

- Inconsistent free types (a lot has been written on this, see [39, 4, 38]), for example $T ::= atom\langle\!\langle \mathbb{N} \rangle\!\rangle \mid fun\langle\!\langle T \to T \rangle\!\rangle$.

It is clear that inconsistencies of this type will also be inconsistencies if they occur in partial specifications. However, these inconsistencies will not be generated by our unification techniques.

A different type of possible inconsistency occurs in the context of schemas with empty sets of bindings, for example (trivially)$D \mathrel{\widehat=} [\, x : S \mid \mathrm{false} \,]$. As long as we do not assert that we have a value from D, this is not an inconsistency in the sense used above. However, in the states-with-operations interpretation of Z, a schema with an empty set of bindings is a specification error. This is because for ADTs the so-called *Initialisation Theorem* needs to hold: the schema describing the initial state of an abstract data type should not be empty.

Except for checking the Initialisation Theorem, there will be no further need to discuss true unary inconsistencies in this paper. It will become clear that our unification method does not generate other internal inconsistencies for the language constructs considered in this paper[4].

3 Viewpoint unification: the parameters

In this section we will discuss the parameters of viewpoint unification. In a naive approach, these are only the viewpoint specifications themselves. However, when state components of different types need to be unified, we have no choice but to be explicit about the relation between those types. Such relations we will call correspondence relations. It turns out that these are related one-to-one with the state space in the unification. From that observation, it follows that an alternative approach is to specify the unified state space explicitly, in terms of the viewpoint state spaces.

3.1 Viewpoint specifications

Although viewpoint specifications could in principle be all kinds of Z specifications, containing other components besides state and operation schemas, we will concentrate on those two. We do not expect other Z constructs to cause extra complications. An additional reason for concentrating on states and

[4] Clearly unification of axiomatic declarations should have the possibility of generating internal inconsistencies.

operations is that these appear in the flattening to Z of specifications in object oriented variants of Z like ZEST [19].

Most of the effort will be in unifying state spaces, and thus we will not discuss operations much at this stage. This is because finding a least common *operation* refinement of two operations on the same state space ("operation unification") is relatively easy – effectively we factor the least common (data) refinement into two independent "least" data refinements and then possibly a least common operation refinement step. The construction for least common "operation refinement" of initialisations is a special case of the construction for operation unification. Adapting viewpoint operations to operate on a common state space first is harder, because it is a data refinement step. Data refinement is intrinsically more complicated, as it involves an implicit existentially quantified parameter: the retrieve relation involved. Choosing this retrieve relation in a sensible way indeed turns out to be the crucial issue in viewpoint unification.

Example 8 As an example of two state spaces that might need to be unified, consider the following.

$$\begin{array}{|l}\hline F_1 \\ x : Apple \\ \hline NotWormEaten\, x \\ \hline\end{array} \qquad \begin{array}{|l}\hline F_2 \\ x : Fruit \\ \hline NotRotten\, x \\ \hline\end{array}$$

where we assume that it follows from the rest of the specification that *Apple* is indeed a subset of *Fruit*.

□

3.2 Intuitive state unification, and the need for correspondences

In this subsection we will give an intuitive definition of state unification. This involves a particular interpretation of state schemas, but this interpretation will only be temporarily assumed in order to clarify the issue. Once the correspondence has been identified as a parameter to viewpoint unification, it can be used to pinpoint any desirable interpretation of state schemas in viewpoint unification. Thus, our intuitive definition may seem wrong, but there is enough generality in the eventual set up to encode any other interpretation.

So how do we unify the fruity state spaces given above? Let us assume that F_1 allows us to choose x from all apples, if we discard any worm-eaten ones. F_2 likewise offers us any fruit, provided it is not rotten. Our intuitive interpretation of a state schema is that the declarations give a range of choice, and the predicates give restrictions. Unification then should extend the range of choice, but combine the restrictions wherever they applied before. Looking at the schemas purely formally, this is an odd interpretation: predicates and subtypes are exchangeable, but we use disjunction on subtypes and (restricted) conjunction on the predicates. For the examples we have dealt with so far [18, 19] however, this default interpretation seemed to capture the intuition much better. In the fruity example, this would give

$$\begin{array}{|l}\hline D \\ x : Fruit \\ \hline NotRotten\, x \\ x \in Apple \Rightarrow NotWormEaten\, x \\ \hline\end{array}$$

Note that this interpretation also explains why we do not normalise state schemas (cf. section 2.1).

In the general case, let us assume we have been given state schemas (we will frequently refer to these names in the sequel)

$$\begin{array}{|l} \multicolumn{1}{l}{D_1} \\ \hline x : S \\ \hline pred_S \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{D_2} \\ \hline x : T \\ \hline pred_T \\ \hline \end{array}$$

coming from two different viewpoints. If we have to rely on implicit relations between the viewpoints, we should assume that everything that has the same name between two viewpoints should be unified. Let us also assume D_1 and D_2 were both originally called D, and thus need to be unified, but that they have been subscripted for disambiguation purposes. Types can be product types, so we can assume without loss of generality that every state space has only one component. According to the intuitive view given above, their unification should be [19]:

$$\begin{array}{|l} \multicolumn{1}{l}{D} \\ \hline x : S \cup T \\ \hline x \in S \Rightarrow pred_S \\ x \in T \Rightarrow pred_T \\ \hline \end{array}$$

However, this is not type correct in general: $S \cup T$ is an error unless S and T have the same (maximal) type. A *disjoint* union of S and T would not be right either, since then values that S and T have in common would be considered different. Is the general solution to take a disjoint union when S and T are unrelated, and set union otherwise?

Example 9 This example is based on a situation in a realistic case study [36] of a video telephony system. Given the following enumerated types in the different viewpoints

$$\begin{aligned} Status1 &::= idle \mid connected \\ Status2 &::= idle \mid connected \mid connecting \end{aligned}$$

how do we unify the following schemas?

$$\begin{array}{|l} \multicolumn{1}{l}{Stat1} \\ \hline x : Status1 \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{Stat2} \\ \hline x : Status2 \\ \hline \end{array}$$

Formally the types *Status*1 and *Status*2 are unrelated (though this could be considered a quirk in the Z typing system). Thus, the general solution we suggested earlier will unify these two to a type of five elements rather than to *Stat*2 as we would have hoped.

□

This last example illustrates another problem. What should happen if the unified state at some point (through an operation from the second viewpoint) evolves to a state where $x = connecting$? In particular, which of the operations of the first viewpoint should still be applicable at that point? None of them, modelling that *connecting* is some transient intermediate state during which all operations from the first viewpoint are disabled? Or should it be those which were applicable for $x = connected$ in the first viewpoint, making *connecting* a special case of *connected*, or similarly for *idle*? Such questions cannot usually be answered without extracting more information from the specifier.

This is where correspondence relations enter the picture. The ODP reference model [29] includes *correspondences* which relate the viewpoint specifications, but it is not very specific on what these correspondences could be. For the specific case of relating two Z viewpoint specifications, we can give a concrete characterisation of what correspondences are. Apart from the implicit links between schemas and their components which happen to have the same names across viewpoints, they also include *correspondence relations* between the types of linked components. If two values a and b for some component x are in such a correspondence relation, this represents the fact that operations on the first viewpoint can safely assume that $x{=}a$ when the second viewpoint maintains that $x{=}b$ and vice versa. Briefly jumping ahead, we can answer the problematic questions on $x{=}connecting$ above using the correspondence relation. If *connecting* is not in the correspondence relation, it is indeed a transient intermediary state. If (*connected*,*connecting*) is in the correspondence relation, *connecting* is a special case of *connected*, as far as the first viewpoint is concerned. In any case, the correspondence relation will probably include (*idle*,*idle*) and (*connected*,*connected*) in order to make explicit that these names were not accidentally identical.

As we will show, with examples, in the next subsection, introducing an explicit correspondence relation also means we do not have to assume the intuitive interpretation used above. We will return to the intuitive interpretation in subsection 3.5, where we show how we can avoid giving an explicit correspondence relation when it is "obvious" what it should be.

3.3 Correspondence relations and unified state spaces

In the previous subsection we have argued that it is in some cases necessary to provide an explicit correspondence relation between the types that a component has across the two viewpoints. In this subsection we will show that this is sufficient information to find a type for that component in the unification.

The crucial idea is to make the type in the unified state space a product of the types in the original state spaces. This idea originates in the method of specification by *views* [30] which we will discuss in section 7.2. The correspondence relation forms the kernel of this product type – however, some extra work is necessary for those values from the state spaces which are not in the domains of the correspondence relation.

There are two ways of explaining the construction of the type used in the unified state space. One is as a totalisation of the correspondence relation, the other is as a modification of a disjoint sum. We will give both explanations, because they may provide better insight on how correspondence relations are used, starting with the latter one.

Even though the result is contained in a product type, we start with the sum of the types involved. Assume the types are S and T as in the general example above, and their correspondence relation is $R \subseteq S \times T$. (In order to keep this explanation simple, we venture outside the Z typing system for a moment.) If $\mathbb{1}$ is a type with a single element not in S or T, let us call it $\perp$, then we could define the disjoint union of S and T by[5]

$$S + T = S \times \mathbb{1} \cup \mathbb{1} \times T$$

i.e. $S + T = \{(s,t) \mid (s \in S \wedge t{=}\perp) \vee (s{=}\perp \wedge t \in T)\}$. The smallest product set containing this set is $S_\perp \times T_\perp$, where $Q_\perp$ is the union of Q and $\mathbb{1}$. (Still a disjoint union, but of an appreciably simpler kind.) Now compute the state space as follows:

$states := S + T$
for each $(s,t) \in R$ **do** $states := (states \setminus \{(s,\perp),(\perp,t)\}) \cup \{(s,t)\}$

An interpretation of the disjoint union of S and T is that no element from S is considered equal to one in T. The interpretation of the correspondence relation is that it asserts that some s represents some t (and vice versa). If that is the case, two different elements $(s,\perp)$ and $(\perp,t)$ in the modified union need to be identified to one (s,t).

The second explanation is that the correspondence relation needs to be totalised. Not every element of S and T is in the left/right domain of R – so we add to R pairs $(s,\perp)$ for each $s \in S$ not in the left domain of R ($\mathrm{dom}\, R$), and pairs $(\perp,t)$ for each $t \in T$ not in the right domain of R ($\mathrm{ran}\, R$). Let us call the resulting set a *totalised correspondence relation*. Totalised correspondence relations

[5] This is probably the second-best known implementation of disjoint sum as a product, the better known one being $S + T = \{0\} \times S \cup \{1\} \times T$.

are linked in a one-to-one way with correspondence relations between S and T: for $tot\,R$ the totalised correspondence of R, we have $tot\,R = R \cup ((S \setminus \mathrm{dom}\,R) \times \mathbb{1}) \cup (\mathbb{1} \times (T \setminus \mathrm{ran}\,R))$, and $R = tot\,R \cap S \times T$.

Here ends our brief excursion outside the Z typing system; we now give the formal definitions in Z. The main differences arise from the need to use explicit injection functions (into free types) where we used set unions above. The one-to-one correspondence also holds in Z, it just looks a bit more complicated.[6]

Definition 10 (Type with bottom) For any type S, we define the type $S_\perp$ by the following free type definition:

$$S_\perp ::= \perp_S \mid justS\langle\langle S \rangle\rangle$$

For all such types, a partial injection $theS$ is defined as the inverse of the injection $justS$:

$$\begin{array}{|l}
theS : S_\perp \rightarrowtail\!\!\!\!\mapsto S \\
\hline
\mathrm{dom}\,theS = \mathrm{ran}\,justS \\
\forall x{:}S \bullet theS\,(justS\,x) = x
\end{array}$$

□

Definition 11 (Totalisation of a relation) The totalisation[7] $tot\,R$ of a relation R on two given types S and T is defined as follows:

$$\begin{array}{|l}
[S,T] \\
\hline
tot : (S \leftrightarrow T) \to (S_\perp \leftrightarrow T_\perp) \\
\hline
\forall R : S \leftrightarrow T \bullet \\
\quad tot\,R \;= theS \,;\, R \,;\, justT \\
\qquad\qquad \cup \{x : S \setminus \mathrm{dom}\,R \bullet (justS\,x, \perp_T)\} \cup \{y : T \setminus \mathrm{ran}\,R \bullet (\perp_S, justT\,y)\}
\end{array}$$

□

This definition is generic in the types S and T – thus, every occurrence of tot in this paper has, besides its relation parameter, two types as parameters. We leave these implicit, trusting that in the context it will be clear what they should be.

Totalised correspondences provide the possibility to specify anything between disjoint union (take the correspondence to be the empty relation) and union (take the correspondence to be the identity relation on the intersection). Moreover, they provide the opportunity to relate elements of types that cannot be directly related in Z even if they appear to be identical:

Example 12 (Union of enumerated types) Continuing example 9 we can form the union of these types by taking the correspondence relation to be $\{(connected, connected), (idle, idle)\} : \mathbb{P}(Status1 \times Status2)$. The totalised correspondence relation (abbreviating some names) is then the set

$$\{(just1\,conned, just2\,conned), (just1\,idle, just2\,idle), (\perp_1, just2\,conning)\}$$

which can be seen as a renaming of the set $\{connected, idle, connecting\}$.

[6] For an alternative formulation of this totalisation, using d'Inverno's optional construct [20], cf. [10].

[7] Note that this totalisation is *different* from the ones Woodcock and Davies [42] use in a similar context.

□

As well as for creating unified state spaces that are various types of unions of the viewpoint state spaces, correspondence relations can also be used to create state spaces that really feature two representations of one data type.

Example 13 Two viewpoints could have sets of numbers – one using the obvious representation $\mathbb{P}\,\mathbb{N}$ and the other one using a sequence without duplicates:

SetAsSet

$x : \mathbb{P}\,\mathbb{N}$

SetAsSeq

$x : \mathrm{seq}\,\mathbb{N}$

$\forall\, i,j : \mathrm{dom}\, x \bullet x\, i = x\, j \Rightarrow i = j$

and the correspondence relation between these two would be $R \subseteq \mathbb{P}\,\mathbb{N} \times \mathrm{seq}\,\mathbb{N}$ defined by

$$(a,b) \in R \Leftrightarrow a = \mathrm{ran}\, b$$

(or a subset of it restricted to sequences without duplicates).

□

In particular, one viewpoint may have a more abstract view of a data type and another viewpoint a more concrete one. The correspondence relation between those two types will then typically be the (predicate of the) retrieve relation between them. Effectively this extends viewpoint unification with data type implementation. Unlike in the other examples above, such correspondence relations will typically be non-functional (e.g. in example 13, a set of n elements corresponds to $n!$ different sequences according to R). Another use of non-functional correspondence relations is in the method of specification by *views* [30]. In section 7.2 we will show with some examples how correspondence relations and unification can be used to generalise that specification method. A more extensive account of the relation between views, data refinement and our viewpoint unification techniques can be found in [10].

3.4 Relabelling

If it was not already clear from the complicated definition of $tot\, R$, the last example clearly showed that the unified state space often looks more complicated than we would prefer. In many cases where we already know what the resulting state space should be, we end up making statements like the above: there is some isomorphism between the state space with bottoms and a simpler one. It is not always necessary for the result of unification to be an easily understandable specification. However, having a readable unification would certainly be helpful if we need to do additional unification with yet another viewpoint – if not for specifying the new correspondence relation, then for finding where any inconsistencies originated.

The solution to this is to include yet another parameter to the unification process: a relabelling. This relabelling should get us from $tot\, R$ to some (to be specified) goal type V. However, if the relabelling is going to be just that, this implies that we need the specifier to specify it in terms of $S_\perp$ and $T_\perp$, which does not reduce the necessary effort much. It seems much more natural to have the specifier only specify the mappings from S and T to the goal type. Thus the following definition.

Definition 14 (Relabelling) A *relabelling* for state schemas $D_1 \mathrel{\widehat=} [\, x{:}S \mid pred_S \,]$ and $D_2 \mathrel{\widehat=} [\, x{:}T \mid pred_T \,]$ with correspondence relation R consists of a *goal type* V and two injective functions $Q_S : S \rightarrow V$ and $Q_T : T \rightarrow V$ satisfying the conditions below:

$$\mathrm{dom}\, Q_S \supseteq \{x{:}S \mid pred_S\} \land \mathrm{dom}\, Q_T \supseteq \{x{:}T \mid pred_T\}$$

$$\forall\, s{:}S;\ t{:}T \bullet (s,t) \in R \Leftrightarrow Q_S\, s = Q_T\, t$$

□

The functions need to be injective to ensure that the relabelling is indeed a relabelling and does not identify elements that are different. The first condition (totality on a restricted domain) ensures that all elements of D_1 and D_2 can be renamed. The second condition has two aspects: from left to right it ensures that a unique relabelling can be found for each (s,t) pair, from right to left it also ensures that different elements do not get identified. A consequence of these conditions is that R needs to be functional in both directions.

When a relabelling is defined, the resulting state space consists of the goal type specified in the relabelling. As with totalised correspondence relations, any further restrictions on the unified state schema will appear as additional predicates (to be defined in section 4.1).

Thus, we have introduced relabelling as a possible extra parameter to unification. The way in which we have defined it ensures that no extra proof obligations are incurred by adding a relabelling (apart from showing that it *is* a relabelling): the resulting state schema is isomorphic to the one obtained without the relabelling.

However, there is something more to be said about the relabelling defined this way. The second condition, due to its shape, can also be read as an extensional definition of the correspondence relation R – in other words, the correspondence relation is *completely determined* by the choice of relabelling. Thus, we can actually *omit* the correspondence when a relabelling is specified, and just assume that the correspondence consists of those pairs of values which get renamed to the same value.

Example 15 We could have solved the problem of complicated naming and isomorphism in example 12 by not giving an explicit correspondence relation, but a relabelling instead. Let the goal type of the relabelling be $Status2$ (i.e., the type used in the second viewpoint). Define the relabelling function $Q_1 : Status1 \rightarrow Status2$ by

$$Q_1 = \{(idle, idle), (connected, connected)\}$$

and let Q_2 be the identity function on $Status2$. These relabelling functions are total and injective, and the correspondence relation that they implicitly define is the one we had before – it actually equals Q_1.

□

3.5 Default correspondence and default relabelling

We have established with examples that in some cases it is really necessary to provide an explicit correspondence relation. From our earlier remarks on "intuitive" state unification it should also follow that in some cases it is clear what the correspondence relation should be. In order to reduce the specification effort whenever possible, we define default correspondence relations and default relabellings. However, note that these defaults correspond to our interpretation of state schemas in unification, and can thus be viewed to be just as arbitrary as that.

The definition of a default correspondence relation is similar to the "general solution" we suggested (and discarded) in section 3.2: when the types of the viewpoint states are compatible, we take a set union.

Definition 16 (Default correspondence) The default correspondence relation on schemas $D_1 \mathrel{\widehat=} [\, x:S \mid pred_S \,]$ and $D_2 \mathrel{\widehat=} [\, x:T \mid pred_T \,]$ is $id_{S \cap T} = \{(x,x) \bullet x \in S \cap T\}$ if $S \cap T$ is a well-typed expression (i.e. S and T have a common supertype).

□

When the types are not compatible, their disjoint union is the only obvious candidate. However, it is not a useful one since it *guarantees* that no common refinement can be found. (Each viewpoint will want the initial value of the unified ADT to correspond to one of its initial values, and the correspondence is empty.)

In order to maintain *state consistency*, cf. section 5.1, it may sometimes be advisable to restrict R to values in $\{x{:}S \mid pred_S\} \times \{x{:}T \mid pred_T\}$.

This default correspondence indeed results in a union:

> If $R{=}id_{S\cap T}$, then dom $R{=}$ ran $R{=}S \cap T$. Thus, the three subsets of $tot\,R$ (cf. the definition) are $\{x : S \cap T . \bullet (justS\,x,justT\,x)\}$ which is isomorphic to $S \cap T$, $\{x : S \setminus (S \cap T) \bullet (justS\,x,\perp_T)\}$ which is isomorphic to $S \setminus T$, and $\{y : T \setminus (S \cap T) \bullet (\perp_S,justT\,y)\}$ which is isomorphic to $T \setminus S$. The isomorphic sets are disjoint, and together make up exactly $S \cup T$.

The situation gets even simpler when we consider a default *relabelling* as well.

Definition 17 (Default relabelling: union) The default relabelling on schemas $D_1 \mathrel{\widehat{=}} [\,x{:}S \mid pred_S\,]$ and $D_2 \mathrel{\widehat{=}} [\,x{:}T \mid pred_T\,]$ such that $S \cap T$ is a well-typed expression is defined as follows. The goal type $V{=}S \cup T$; the relabelling functions are $Q_S{=}id_S$; $Q_T{=}id_T$.

□

For reasons related to state consistency, it may sometimes be advisable to restrict Q_S to values satisfying $pred_S$, and similarly for Q_T.

The above definition allows us not to specify any correspondence relation or relabelling, and end up with the intuitive unification we proposed at the very beginning. This gives us "the best of both worlds": if the intuitive unification is the right one we can choose it without further ado; if it is not right we have a mechanism to specify what it should be.

4 Viewpoint unification: the algorithm

This section presents the algorithm for unifying two viewpoint specifications in the states-with-operations style. There are three aspects to this unification: first, state schemas that occur in both viewpoints need to be combined to unified state schema, then operations on those (including initialisations) need to be adapted to the unified state schema, and finally operations that occur in both viewpoints (including initialisations) need to be unified.

4.1 State unification

The correspondence relation and its totalisation form the main component of state unification. It only remains to account for the predicates in the original state schemas, and to create an actual state schema for the unified state.

If the correspondence relation is $R \subseteq S \times T$, the inhabitants of the unified state schema will be the tuples of $tot\,R$. To account for the predicate $pred_S$, we include a predicate that should hold whenever the $S_\perp$value is not $\perp_S$, and similarly for $pred_T$.

Definition 18 (Unified state by correspondence) Given schemas $D_1 \mathrel{\widehat{=}} [\,x{:}S \mid pred_S\,]$ and $D_2 \mathrel{\widehat{=}} [\,x{:}T \mid pred_T\,]$, their unification according to the correspondence relation $R \subseteq S \times T$ is

D
$x_1{:}S_\perp;\ x_2{:}T_\perp$
$(x_1,x_2) \in tot\,R$ $\forall\, x{:}S \bullet x_1{=}justS\,x \Rightarrow pred_S$ $\forall\, x{:}T \bullet x_2{=}justT\,x \Rightarrow pred_T$

□

This looks like we are actually maintaining two values for the state variable x; however, due to (x_1,x_2) being in $tot\,R$ it is the case that either exactly one of the two values is $\perp$ and thus invalid, or the two values are "equal" (since they are in R, and R only contains tuples of things we consider equal).

Example 19 (Union of enumerated types, ctd.) The unification of $Stat1 \mathrel{\widehat=} [\, x{:}Status1 \,]$ and $Stat2 \mathrel{\widehat=} [\, x{:}Status2 \,]$ from example 9 is fairly simple using the correspondence relation $R{=}\{(connected,connected),(idle,idle)\}$ with its totalisation (cf. example 12). The only predicate remaining is $(x_1,x_2) \in tot\ R$ which we have expanded below, the other two reduce to true.

$$
\begin{array}{|l}
\multicolumn{1}{l}{D} \\ \hline
x_1{:}Status1_{\perp};\ x_2{:}Status2_{\perp} \\ \hline
(x_1{=}justStatus1\ connected \wedge x_2{=}justStatus2\ connected) \vee \\
(x_1{=}justStatus1\ idle \wedge x_2{=}justStatus2\ idle) \vee \\
(x_1{=}\perp_{Status1} \wedge x_2{=}justStatus2\ connecting) \\ \hline
\end{array}
$$

which is isomorphic to $Stat2$.

□

For the following two examples we will use the default correspondence relation.

Example 20 The schemas

$$
\begin{array}{|l}
\multicolumn{1}{l}{D_1} \\ \hline
x{:}\mathbb{Z} \\ \hline
1 \leq x \leq 5 \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{D_2} \\ \hline
x{:}\mathbb{Z} \\ \hline
\exists\, z{:}\mathbb{N} \bullet x = z + z \\ \hline
\end{array}
$$

have the same type of component so their default correspondence relation is the identity relation on that type. The schema that results from unification is, after some simplifications, using $the\mathbb{Z}$ as the inverse of $just\mathbb{Z}$

$$
\begin{array}{|l}
\multicolumn{1}{l}{D} \\ \hline
x_1{:}\mathbb{Z}_{\perp};\ x_2{:}\mathbb{Z}_{\perp} \\ \hline
(x_1,x_2) \in tot\,\{(x,x) \mid x \in \mathbb{Z}\} \\
1 \leq the\mathbb{Z}\ x_1 \leq 5 \\
\exists\, z{:}\mathbb{N} \bullet the\mathbb{Z}\ x_2{=}z + z \\ \hline
\end{array}
$$

which is a complicated way of describing the schema $[x{:}\{2,4\}]$.

□

Example 21 Schemas $D_1 \mathrel{\widehat=} [\, x{:}S \,]$ where $S \mathrel{\widehat=} 1\,..\,5$ and $D_2 \mathrel{\widehat=} [\, x{:}T \,]$ where $T \mathrel{\widehat=} \{z{:}\mathbb{N} \bullet z+z\}$ have the identity relation on $S \cap T$ as the default correspondence relation, i.e. $\{(2,2),(4,4)\}$. The schema resulting from their unification is (first predicate expanded, last two reduce to true):

$$
\begin{array}{|l}
\multicolumn{1}{l}{D} \\ \hline
x_1{:}S_{\perp};\ x_2{:}T_{\perp} \\ \hline
(x_1{=}\perp_S \wedge x_2 \in \{z{:}\mathbb{N} \setminus \{1,2\} \bullet justT\ (z + z)\}) \vee \\
(x_1,x_2) \in \{z{:}\{2,4\} \bullet (justS\ z, justT\ z)\} \vee \\
(x_1 \in \{z{:}\{1,3,5\} \bullet justS\ z\} \wedge x_2{=}\perp_T) \\ \hline
\end{array}
$$

This schema is isomorphic to $D \mathrel{\widehat=} [\, x : S \cup T \,]$.

□

These two examples illustrate the effect of normalisation on state unification, the schemas in example

20 are the normalised versions of those in example 21, but their unifications are very different indeed. This difference is caused by different (default) correspondence relations being used.

Indeed, the default correspondence relations may be different for schemas that are semantically equal but syntactically different – as a function of the schemas, it is defined on their *syntax* rather than on their *semantics* (the latter being the more usual thing to do in the Z world). This does not point out a defect in our set up – the correspondence relation can and should always be chosen sensibly – but rather reflects our observation that the syntactical form does seem to matter for the intuitive interpretation of Z state schemas even when the semantics does not make a distinction.

Alternatively, if a relabelling is given, we can use that to determine the state unification.

Definition 22 (Unified state by relabelling) Given schemas $D_1 \mathrel{\widehat=} [\, x{:}S \mid pred_S \,]$ and $D_2 \mathrel{\widehat=} [\, x{:}T \mid pred_T \,]$, and a relabelling (V, Q_S, Q_T) their unification is

$$
\begin{array}{|l}
\multicolumn{1}{l}{D} \\
\hline
y{:}V \\
\hline
\forall\, x{:}S \bullet Q_S\; x = y \Rightarrow pred_S \\
\forall\, x{:}T \bullet Q_T\; x = y \Rightarrow pred_T \\
\hline
\end{array}
$$

□

Example 23 Using the default relabellings, the unification for example 20 is

$$
\begin{array}{|l}
\multicolumn{1}{l}{D} \\
\hline
y{:}\mathbb{Z} \\
\hline
\forall\, x{:}\mathbb{Z} \bullet x = y \Rightarrow 1 \leq x \leq 5 \\
\forall\, x{:}Z \bullet x = y \Rightarrow \exists\, z{:}\mathbb{N} \bullet x = z + z \\
\hline
\end{array}
$$

which can be simplified to $D \mathrel{\widehat=} [\, y{:}\mathbb{Z} \mid 1 \leq y \leq 5 \wedge \exists\, z{:}\mathbb{N} \bullet y = z + z \,]$. The unification for example 21 will be (both predicates reducing to true) $D \mathrel{\widehat=} [\, y : S \cup T \,]$.

□

The final example shows that a schema with a singleton set of bindings might fulfill a very useful role when we apply this state unification rule: modulo state isomorphism, it is the unit of state unification if we use the largest possible correspondence. Thus, we can formally treat the situation that a state only occurs in one of the two viewpoints by assuming it is defined to be the singleton state in the other viewpoint.

Example 24 (The singleton state) For the states $D_1 \mathrel{\widehat=} [\, x{:}\{1\} \,]$ and $D_2 \mathrel{\widehat=} [\, x{:}T \mid pred_T \,]$ the largest possible correspondence relation is the one that links 1 to every T. Its totalisation is $\{x : T \bullet (just1, justT x)\}$, and the resulting unified state is:

$$
\begin{array}{|l}
\multicolumn{1}{l}{D} \\
\hline
x_1{:}\{just1\} \\
x_2{:}T_\perp \\
\hline
x_2 \in \operatorname{ran} justT \\
\forall\, x : T \bullet x_2 = justT\; x \Rightarrow pred_T \\
\hline
\end{array}
$$

which, is clearly isomorphic to D_2.

□

In [8] we presented the *empty* state schema as the unit of state unification. This is a correct alternative, but not very useful, as an ADT with an empty state schema fails its Initialisation Theorem.

4.2 Operation adaptation

If a state schema has been unified with another one, the operations (including initialisation) in the viewpoint in which the first state schema resides will also need to be changed to operate on the new state. This amounts to choosing the least (in refinement order) data refinement of each operation where the retrieve relation is essentially the correspondence relation (see the proof in section 5.1 for the exact details of this). The adapted operation should be applicable whenever the relevant state component is not $\perp$ and the original operation's precondition holds, and it should return a unified state that represents the original operation's postcondition, which also implies that the relevant state component is not $\perp$.

Definition 25 (Operation adaptation: correspondence) Given schemas $D_1 \mathrel{\widehat=} [\, x{:}S \mid pred_S \,]$ and $D_2 \mathrel{\widehat=} [\, x{:}T \mid pred_T \,]$ with correspondence relation $R \subseteq S \times T$, an operation that was originally defined on the state D_1 by

$$\begin{array}{|l}
\multicolumn{1}{l}{Op_1} \\ \hline
\Delta D_1;\ Decl_1 \\ \hline
pred_1 \\ \hline
\end{array}$$

gets adapted to the new state schema by changing it to

$$\begin{array}{|l}
\multicolumn{1}{l}{AdOp_1} \\ \hline
\Delta D;\ Decl_1 \\ \hline
x_1 \in \operatorname{ran} justS \\
x_1{}' \in \operatorname{ran} justS \\
\mathbf{let}\ x == theS\, x_1\ ;\ x' == theS\, x_1{}' \bullet pred_1 \\ \hline
\end{array}$$

and similarly for operations on D_2.

A degenerate case of this is initialisation adaptation: initialisation scheme $Init_1 \mathrel{\widehat=} [\, D_1' \mid init_1 \,]$ gets adapted to

$$\begin{array}{|l}
\multicolumn{1}{l}{AdInit_1} \\ \hline
D' \\ \hline
x_1{}' \in \operatorname{ran} justS \\
\mathbf{let}\ x' == theS\, x_1{}' \bullet init_1 \\ \hline
\end{array}$$

and similarly for the other viewpoint's initialisation.

□

The last predicate in $AdOp_1$ can also be written as $pred_1[theS\, x_1 / x][theS\, x_1{}' / x']$. The situation is only slightly more complicated for operations which operate on multiple states – the rule above can then be applied repeatedly, and the only complication is the bookkeeping of which references to states have been updated to refer to changed states.

There is a variant to be used when the state has been unified via a relabelling rather than an explicit correspondence relation.

Definition 26 (Operation adaptation: relabelling) Given schemas $D_1 \mathrel{\widehat=} [\, x{:}S \mid pred_S \,]$ and $D_2 \mathrel{\widehat=} [\, x{:}T \mid pred_T \,]$ with relabelling (V, Q_S, Q_T), an operation that was originally defined on the state D_1 by

$$\begin{array}{|l}
\multicolumn{1}{l}{Op_1} \\ \hline
\Delta D_1;\ Decl_1 \\ \hline
pred_1 \\ \hline
\end{array}$$

gets adapted to the new state schema (cf. Definition 22) by changing it to

$$\begin{array}{|l}
\hline
AdOp_1 \\
\Delta D;\ Decl_1 \\
\hline
\forall x, x' : S \bullet Q_S\ x = y \wedge Q_S\ x' = y' \ \Rightarrow\ pred_1 \\
\hline
\end{array}$$

and similarly for operations on D_2.

For initialisation scheme $Init_1 \mathrel{\widehat=} [D_1' \mid init_1]$ we now get

$$\begin{array}{|l}
\hline
Init_1 \\
D' \\
\hline
\forall x' : S \bullet Q_S\ x' = y' \ \Rightarrow\ init_1 \\
\hline
\end{array}$$

and similarly for the initialisation of D_2.

□

4.3 Operation unification

The unification of two viewpoint operations (adapted to operate on the same unified state) should exhibit possible behaviour of each of the viewpoint operations in each situation where the viewpoint operation was applicable. This requirement can be formalised using pre- and postconditions. The unified operation should be applicable whenever one of the viewpoint operations is, i.e. its precondition should be the disjunction of the viewpoint operation preconditions. Moreover, when the unified operation is applied to a state satisfying one particular precondition, a state should result that satisfies the corresponding postcondition. Such an operation unification is also described by Ainsworth et al. [1], there called *union*, although they do not mention that the union may not exist. In the more abstract setting of binary relations used by Frappier et al [25] the same construct appears as the *demonic join*.

Definition 27 (Operation unification) The candidate least unification of operation schemas $AdOp_1$ and $AdOp_2$, both operating on the same state and having the same collection of inputs and outputs $Decls$, is given by[8]

$$\begin{array}{|l}
\hline
UnOp \\
Decls;\ \Delta D \\
\hline
\mathrm{pre}\, AdOp_1 \vee \mathrm{pre}\, AdOp_2 \\
\mathrm{pre}\, AdOp_1 \Rightarrow \mathrm{post}\, AdOp_1 \\
\mathrm{pre}\, AdOp_2 \Rightarrow \mathrm{post}\, AdOp_2 \\
\hline
\end{array}$$

□

That this schema only defines the desired unification under additional restrictions is shown in section 5.2.

The unification of two initialisations operating on the same state is a degenerate case of this. Because initialisations (obviously) have no preconditions, the result is a pure conjunction:

[8] Wim Feijen pointed out the similarity between the conditions in this schema and those in the w(eakest)p(recondition)-calculus for the guarded command $P_1 \rightarrow Op_1 \,\square\, P_2 \rightarrow Op_2$ where pre_i has the role of the guard.

Definition 28 (Initialisation unification) The candidate initialisation unification of two initialisations on the same state, $[\, D' \mid init_1 \,]$ and $[\, D' \mid init_2 \,]$ is the following:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ UnInit\ }} \\
D' \\
\hline
init_1 \wedge init_2 \\
\hline
\end{array}$$

□

For a meaningful unification, it needs to be established whether the Initialisation Theorem holds for the resulting ADT – i.e., in the above definition, whether $init_1 \wedge init_2$ is satisfiable. This property we will call *initialisation consistency*.

Definition 29 (Initialisation consistency) Two abstract data types are initialisation consistent with respect to a correspondence relation if the unification of their initialisation adaptations is satisfiable (i.e., satisfies the Initialisation Theorem of the candidate unified ADT).

□

4.4 The algorithm in full

The full algorithm for unifying two viewpoint specifications, using the unifications and adaptations described above, is now as follows.

> A viewpoint specification is assumed to consist of a collection of ADTs. For each ADT $(D_1, Init_1, Ops_1)$ in one viewpoint that corresponds to an ADT $(D_2, Init_2, Ops_2)$ in the other viewpoint, construct an ADT in the unification as follows:
>
> 1. Establish a correspondence relation R between (the component types of) D_1 and D_2, and construct the state unification of D_1 and D_2 based on that correspondence relation. This gives the state of the resultant ADT.
> 2. Check state consistency of D_1 and D_2 according to R; if it does not hold, the resultant ADT is likely not to be a common refinement, and a full refinement proof needs to be carried out in order to check this at the end.
> 3. Adapt all operations in Ops_1 and Ops_2 and the initialisations to the unified state. (These operations do *not* get added to the constructed ADT at this stage.)
> 4. Construct the initialisation unification of the adapted initialisations. If the resulting initialisation is satisfiable, it is the initialisation of the resultant ADT; if not, the whole unification process has failed.
> 5. For each pair of matching operations, check their operation consistency. If it fails, the whole unification process has failed. If it succeeds, construct their operation unification and add it to the resultant ADT.
> 6. It depends on the interpretation of ADTs (as discussed at the end of section 2.3.2) what happens to the remaining adapted operations:
>   - In the strict behavioural approach: for each adapted operation Op remaining from the first viewpoint, construct the operation adaptation $AdId_2$ of the second viewpoint's identity operation (ΞD_2). Then add the operation unification of Op and $AdId_2$ to the resultant ADT, provided they are operation consistent (if not, unification has failed). Analogously for adapted operations remaining from the second viewpoint.
>   - In the "services" approach: add all remaining adapted operations to the resultant ADT.

5 Proofs and consistency conditions

Here we present what amounts to a correctness proof for the unification rules given above. The proof will be in three steps: showing that the adapted operations with the unified state form data refinements of the viewpoints; showing that unified operations are (operation) refinements of the adapted operations; and finally a proof that the unification is a *least* common refinement. The proof given below imposes extra conditions on the viewpoint specifications in two places: one is *operation consistency* which is needed to prove the correctness of operation unification, the other is *state consistency* which follows from analysis of the preconditions of the adapted operations. Together with the initialisation consistency condition, these form the consistency conditions of the two viewpoints.

The proofs assume that the first viewpoint is an ADT with state $D_1 \mathrel{\widehat=} [\, x{:}S \mid pred_S \,]$ and an operation $Op_1 \mathrel{\widehat=} [\, \Delta D_1;\ Decls | pred_1 \,]$, the second viewpoint is an ADT with state $D_2 \mathrel{\widehat=} [\, x{:}T | pred_T \,]$ and an operation $Op_2 \mathrel{\widehat=} [\, \Delta D_2;\ Decls \mid pred_2 \,]$, with their unification according to correspondence relation R and adapted operations etc. as defined above. Often the contributions of input and output parameters to operations are ignored in order to simplify the formulas in the proofs – adding them would add no complication to the structure of the proofs, and no extra conditions.

5.1 Operation adaptation is data refinement

First we show that the unified state with the adapted operations form data refinements of the viewpoints with operations. For that purpose we have to link the state schemas using a *retrieve relation*. For the unified state schema D and the state schema D_1 of the first viewpoint the retrieve relation is given by the schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Retr}1} \\
\hline
D_1;\ D \\
\hline
x_1 = \textit{justS}\ x \\
\hline
\end{array}$$

Note that this retrieve relation reflects our intuitive view of how these specifications relate, in other words, if data refinement is established it is also a *meaningful* data refinement (cf. our earlier remarks in section 2.3.2). There are three conditions to prove that D with $AdOp_1$ is a valid data refinement of D_1 with Op_1. The initialisation condition is guaranteed to hold by construction of the initialisation adaptation, and the fact that any value of each original state space is represented in the unified state space. The remaining two conditions are, making any universal quantifications implicit:

1. $\operatorname{pre} Op_1 \land Retr1 \ \Rightarrow\ \operatorname{pre} AdOp_1$
2. $\operatorname{pre} Op_1 \land Retr1 \land AdOp_1 \ \Rightarrow\ \exists\, x' \bullet Retr1' \land Op_1$

The proof of the first property has a big hurdle in the middle of it. For simplicity we ignore the contribution of *Decls* to the predicate $AdOp_1$ since it makes the same contribution to Op_1. The term "translation" in the hints stands for the replacement of some quantified variables by new ones.

$$\begin{array}{ll}
& \operatorname{pre} AdOp_1 \\
\equiv & \quad \{ \ \text{definition of pre} \ \} \\
& \exists\, x_1{}';\ x_2{}' \bullet AdOp_1 \\
\equiv & \quad \{ \ \text{definition } AdOp_1 \ \} \\
& \exists\, x_1{}';\ x_2{}' \bullet D \land D' \land x_1 \in \operatorname{ran} \textit{justS} \\
& \qquad\qquad \land\ x_1{}' \in \operatorname{ran} \textit{justS} \land pred_1[\textit{theS}\ x_1/x][\textit{theS}\ x_1{}'/x'] \\
\equiv & \quad \{ \ \text{conjuncts independent of new state} \ \}
\end{array}$$

$$D \wedge x_1 \in \operatorname{ran} justS$$

$$\wedge \ \exists x_1' ; x_2' \bullet D' \wedge x_1' \in \operatorname{ran} justS \wedge pred_1[theS\, x_1/x][theS\, x_1'/x']$$

$\equiv$ { **WISH:** x_2' always exists here; translation $x' := theS\, x_1'$ (so $justS\, x' := x_1'$) }

$$D \wedge x_1 \in \operatorname{ran} justS \wedge \exists x' \bullet D_1[theS\, x_1'/x] \wedge pred_1[theS\, x_1/x]$$

$\equiv$ { definition of pre }

$$D \wedge x_1 \in \operatorname{ran} justS \wedge \operatorname{pre} Op_1[theS\, x_1/x]$$

$\Leftarrow$ { definition $Retr1$, substitution }

$$Retr1 \wedge \operatorname{pre} Op_1$$

Of course the crux of this proof is the step marked with **WISH**. It is clear that we need an extra condition here, the predicate really depends on x_2' through the conjunct D'. A correct x_2' may not exist in exactly one type of situation: $(x_1', x_2') = (justS\, s, justT\, t)$ and $(s,t) \in R$, $pred_S[s/x]$ holds but $pred_T[t/x]$ does not hold. That is to say, the output value of the operation is linked by the correspondence relation to an "illegal" value, whereas the input value is linked to a legal one (and thus not excluded from the translated precondition $Retr1 \wedge \operatorname{pre} Op_1$). At this point we will assume that the viewpoints are *state consistent* to prevent this problem:

Definition 30 The two state schemas $D_1 \mathrel{\widehat=} [\, x{:}S \mid pred_S \,]$ and $D_2 \mathrel{\widehat=} [\, x{:}T \mid pred_T \,]$ are *state consistent* with respect to the correspondence relation $R \subseteq S \times T$ iff

$$(s,t) \in R \Leftrightarrow (pred_S[s/x] \Leftrightarrow pred_T[t/x])$$

□

This is a sufficient, but not a necessary condition; for a further discussion of related properties, see section 5.4. The second property is more easily proved:

$$\exists x' \bullet Retr1' \wedge Op_1$$

$\equiv$ { definitions }

$$\exists x' \bullet D_1' \wedge D' \wedge x_1' = justS\, x' \wedge D \wedge D' \wedge pred_1$$

$\equiv$ { D and D' independent of x'; $theS$ is inverse of $justS$ }

$$(\exists x' \bullet D_1' \wedge theS\, x_1' = x' \wedge pred_1) \wedge D \wedge D'$$

$\equiv$ { one point rule for existential quantifier }

$$pred_S[theS\, x_1'/x] \wedge pred_1[theS\, x_1'/x'] \wedge D \wedge D'$$

$\Leftarrow$ { first conjunct follows from D'; property of substitution }

$$pred_1[theS\, x_1/x][theS\, x_1'/x'] \wedge x_1 = justS\, x \wedge D \wedge D'$$

$\Leftarrow$ { definitions $AdOp_1$ and $Retr1$, add conjunct }

$$\operatorname{pre} Op_1 \wedge AdOp_1 \wedge Retr1$$

Of course the proof for the second viewpoint is completely analogous.

5.2 Operation unification is refinement

The operation unification $UnOp$ of two operations $AdOp_1$ and $AdOp_2$ as defined in section 4.3 should be a refinement of each of the operations. In order for it to be a least common refinement, it should weaken the precondition no more than is necessary, which implies that the precondition of $UnOp$ should be the disjunction of the preconditions of $AdOp_1$ and $AdOp_2$. We will establish a condition for this to be true first.

We write pre_1 for $\mathrm{pre}\, AdOp_1$ etc for clarity in the following calculation, and assume for simplicity that the operations have no input or output:

$$
\begin{array}{ll}
 & \mathrm{pre}\, UnOp \\
\equiv & \quad \{ \quad \text{definition pre} \quad \} \\
 & \exists\, State' \bullet (\mathrm{pre}_1 \lor \mathrm{pre}_2) \land (\mathrm{pre}_1 \Rightarrow \mathrm{post}_1) \land (\mathrm{pre}_2 \Rightarrow \mathrm{post}_2) \\
\equiv & \quad \{ \quad \mathrm{pre}_1 \text{ and } \mathrm{pre}_2 \text{ do not refer to } State' \quad \} \\
 & (\mathrm{pre}_1 \lor \mathrm{pre}_2) \land \exists\, State' \bullet (\mathrm{pre}_1 \Rightarrow \mathrm{post}_1) \land (\mathrm{pre}_2 \Rightarrow \mathrm{post}_2) \\
\equiv & \quad \{ \quad \text{case analysis on } \mathrm{pre}_1 \text{ and } \mathrm{pre}_2; \\
 & \quad\quad\;\; \exists\, State' \bullet \mathrm{pre}_i \Rightarrow \mathrm{post}_i \text{ holds by definition of pre} \quad \} \\
 & (\mathrm{pre}_1 \lor \mathrm{pre}_2) \land \exists\, State' \bullet \mathrm{pre}_1 \land \mathrm{pre}_2 \;\Rightarrow\; \mathrm{post}_1 \land \mathrm{post}_2 \\
\equiv & \quad \{ \quad \mathrm{pre}_1 \text{ and } \mathrm{pre}_2 \text{ do not refer to } State' \quad \} \\
 & (\mathrm{pre}_1 \lor \mathrm{pre}_2) \land (\mathrm{pre}_1 \land \mathrm{pre}_2 \;\Rightarrow\; \exists\, State' \bullet \mathrm{post}_1 \land \mathrm{post}_2)
\end{array}
$$

In other words, the precondition of the union is *only* the disjunction of the preconditions if both postconditions can be satisfied when both preconditions are. This is an essential condition which will form part of our consistency check. In fact, it will turn out to be a condition for the union to be a common refinement of the operations, and it is useful to give it a name. The extension to include input and output parameters is straightforward.

Definition 31 Operations A and B, operating on the same state space $State$, both with input $x? : X$ and output $y! : Y$, are said to be *operation consistent* iff

$$\forall\, State\, ;\; x? : X \bullet \mathrm{pre}\, A \land \mathrm{pre}\, B \;\Rightarrow\; \exists\, State'\, ;\; y! : Y \bullet \mathrm{post}\, A \land \mathrm{post}\, B$$

□

In order to show that $UnOp$ is a common refinement of $AdOp_1$ and $AdOp_2$, it suffices to give only the half of the proof for one viewpoint. Because this step involves no change of state space, we only need to prove the two conditions for operation refinement, again omitting universal quantifications:

1. $\mathrm{pre}\, AdOp_1 \;\Rightarrow\; \mathrm{pre}\, UnOp$
2. $\mathrm{pre}\, AdOp_1 \land UnOp \;\Rightarrow\; AdOp_1$

The first condition is only true if the *operation consistency* condition holds, see the calculation of $\mathrm{pre}\, UnOp$ above (and then it is a one line proof). The second is easily proved using the fact that the predicate part of an operation schema A can be given as $\mathrm{pre}\, A \land \mathrm{post}\, A$.

5.3 Unification is least

The final step of the least common refinement proof is showing that the unification is a *least* common refinement. This will be done by showing that an *arbitrary* refinement of both viewpoints is necessarily a refinement of the unification.

Suppose an ADT with state schema E and operation schema Opp also form a (data) refinement of both viewpoint specifications $(D_1, Init_1, \{Op_1\})$ and $(D_2, Init_2, \{Op_2\})$, and that the state of E is given by the (fresh) variable y. This means that two retrieve relations exists, let us assume they are given by (i=1,2)

$$\begin{array}{|l}
\multicolumn{1}{l}{Retr_i} \\
\hline
D_i;\ E \\
\hline
retr_i \\
\hline
\end{array}$$

The assumption that these are data refinements translates into assumptions we can use in proofs:

1. $\text{pre}\, Op_i \wedge Retr_i \;\Rightarrow\; \text{pre}\, Opp$
2. $\text{pre}\, Op_i \wedge Retr_i \wedge Opp \;\Rightarrow\; \exists\, x' \bullet Retr_i{}' \wedge Op_i$

We now prove that, under these assumptions, $(E, EInit, \{Opp\})$ is a data refinement of $(D, UnInit, \{UnOp\})$. Thus we have to find some retrieve relation $RetrED$ such that[9]

1. $\text{pre}\, UnOp \wedge RetrED \;\Rightarrow\; \text{pre}\, Opp$
2. $\text{pre}\, UnOp \wedge RetrED \wedge Opp \;\Rightarrow\; \exists\, x_1{}';\ x_2{}' \bullet RetrED' \wedge UnOp$

Our choice for that retrieve relation is the following schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{RetrED} \\
\hline
D;\ E \\
\hline
retr_1[theS\, x_1/x] \vee retr_2[theT\, x_2/x] \\
\hline
\end{array}$$

(The main motivation for this particular choice is that it works.)

Now we prove the two properties. For the first we leave out universal quantification over y, the "concrete state".

$$\begin{array}{ll}
& \forall\, x_1;\ x_2 \bullet \text{pre}\, Opp \Leftarrow \text{pre}\, UnOp \wedge RetrED \\
\equiv & \quad \{ \text{ assuming operation consistency } \} \\
& \forall\, x_1;\ x_2 \bullet \text{pre}\, Opp \Leftarrow (\text{pre}\, AdOp_1 \vee \text{pre}\, AdOp_2) \wedge RetrED \\
\equiv & \quad \{ \text{ definition } RetrED \ \} \\
& \forall\, x_1;\ x_2 \bullet \text{pre}\, Opp \Leftarrow (\text{pre}\, AdOp_1 \vee \text{pre}\, AdOp_2) \wedge D \wedge E \\
& \qquad\qquad\qquad \wedge retr_1[theS\, x_1/x] \vee retr_2[theT\, x_2/x] \\
\Leftarrow & \quad \{ \text{ calculus } \} \\
& \forall\, x_1;\ x_2 \bullet \text{pre}\, Opp \Leftarrow (\text{pre}\, AdOp_1 \wedge D \wedge E \wedge retr_1[theS\, x_1/x]) \\
& \qquad\qquad\qquad \vee (\text{pre}\, AdOp_2 \wedge D \wedge E \wedge retr_2[theT\, x_2/x]) \\
\equiv & \quad \{ \text{ definition pre}\, AdOp_i \text{ (state consistency);} \\
& \quad\ \ \text{translation } (x_1, x_2) := (justS\, x, justT\, y) \ \} \\
& \forall\, x;\ y \bullet \text{pre}\, Opp \Leftarrow (\text{pre}\, Op_1 \wedge D_1 \wedge E \wedge retr_1) \\
& \qquad\qquad\qquad \vee ((\text{pre}\, Op_2)[y/x] \wedge D_2 \wedge E \wedge retr_2[y/x]) \\
\equiv & \quad \{ \text{ definition } Retr_i\text{; assumptions } \} \\
& \text{true}
\end{array}$$

[9]The contribution of the initialisations in "least" is omitted here – it should be obvious that the conjunction used in definition 28 indeed generates the "least" initialisation, given that the ordering used is implication.

The second proof is a quite complicated one. We are asked to prove that $\forall x_1;\, x_2;\, y \bullet P \Rightarrow (\exists x_1{}';\, x_2{}' \bullet Q)$ for certain predicates P and Q. The proof proceeds by first showing how $\exists x_1{}';\, x_2{}' \bullet Q$ can be rewritten as $\exists x' \bullet Q_1 \vee \exists x' \bullet Q_2$. Then we do a case introduction on P such that $P{=}(P_1 \vee P_2)$ and we show that $(i{=}1,2)$ $\forall x_1;\, x_2;\, y \bullet P_i \Rightarrow (\exists x' \bullet Q_i)$ follows from the assumption that E is a refinement of the i-th viewpoint, which then completes the proof.

$$
\begin{array}{ll}
 & \exists x_1{}';\, x_2{}' \bullet RetrED' \wedge UnOp \\
\equiv & \quad \{ \quad \text{definition } UnOp\text{, assuming operation consistency} \quad \} \\
 & (\exists x_1{}';\, x_2{}' \bullet RetrED' \wedge AdOp_1) \vee (\exists x_1{}';\, x_2{}' \bullet RetrED' \wedge AdOp_2)
\end{array}
$$

The simplifications of these disjuncts will be completely analogous so we show only one:

$$
\begin{array}{ll}
 & \exists x_1{}';\, x_2{}' \bullet RetrED' \wedge AdOp_1 \\
\Leftarrow & \quad \{ \quad \text{definition of } RetrED' \text{ and } AdOp_1 \quad \} \\
 & \exists x_1{}';\, x_2{}' \bullet D' \wedge E' \wedge (retr_1[theS\, x_1/x])' \wedge D \wedge x_1 \in \operatorname{ran} justS \\
 & \qquad\qquad \wedge\, x_1{}' \in \operatorname{ran} justS \wedge pred_1[theS\, x_1/x][theS\, x_1{}'/x'] \\
\equiv & \quad \{ \quad \text{assuming state consistency, translate } x_1{}' := justS\, x' \quad \} \\
 & \exists x' \bullet D_1{}' \wedge E' \wedge retr_1{}' \wedge D \wedge x_1 \in \operatorname{ran} justS \wedge pred_1[theS\, x_1/x] \\
\equiv & \quad \{ \quad \text{definition of } Retr_1 \quad \} \\
 & \exists x' \bullet Retr_1{}' \wedge D \wedge x_1 \in \operatorname{ran} justS \wedge pred_1[theS\, x_1/x]
\end{array}
$$

The antecedent (we called it P in the proof overview above) of the universal quantification can be rewritten in the form $P_1 \vee P_2$ as follows:

$$
\begin{array}{ll}
 & \operatorname{pre} UnOp \wedge RetrED \wedge Opp \\
\equiv & \quad \{ \quad \text{assuming operation consistency} \quad \} \\
 & \quad (x_1 \in \operatorname{ran} justS \wedge \operatorname{pre} AdOp_1 \wedge RetrED \wedge Opp) \\
 & \vee \;\; (x_2 \in \operatorname{ran} justT \wedge \operatorname{pre} AdOp_2 \wedge RetrED \wedge Opp)
\end{array}
$$

Now we show that each of the disjuncts in the antecedent (P_i) proves one of the disjuncts in the consequent (Q_i). Again these two proofs are completely analogous, so only one is given.

$$
\begin{array}{ll}
 & \forall x_1;\, x_2;\, y \bullet x_1 \in \operatorname{ran} justS \wedge \operatorname{pre} AdOp_1 \wedge RetrED \wedge Opp \\
 & \qquad\qquad \Rightarrow \; \exists x' \bullet Retr_1{}' \wedge D \wedge x_1 \in \operatorname{ran} justS \wedge pred_1[theS\, x_1/x] \\
\Leftarrow & \quad \{ \quad \text{assuming state consistency, translate } x_1 := justS\, x \quad \} \\
 & \forall x;\, y \bullet \operatorname{pre} Op_1 \wedge D_1 \wedge E \wedge retr_1 \wedge Opp \\
 & \qquad \Rightarrow \exists x' \bullet Retr_1{}' \wedge pred_1 \wedge D_1 \\
\equiv & \quad \{ \quad \text{definition } Retr_1\text{, assumption} \quad \} \\
 & \text{true}
\end{array}
$$

This concludes our proof that every common refinement of the viewpoints is a refinement of the unification, and thus the unification is indeed a least common refinement.

5.4 Concluding remarks on the algorithm

The proofs above have shown that our unification algorithm produces a least common refinement when three relatively simple conditions are satisfied: *initialisation consistency*, *operation consistency* and *state consistency*.

Operation consistency appears to be sufficient and necessary for a common refinement of two operations on the same state to exist. However, it can only be established once there is a unified state, so state unification really has to come first. Thus, indirectly operation consistency also depends on the choice of correspondence relation. The same holds for initialisation consistency, which can be viewed as a degenerate case of operation consistency.

State consistency, however, is certainly not a necessary condition. The following example demonstrates this.

Example 32 We return to the state schemas and correspondence rules used in example 23 and before. Unifying $D_1 \mathrel{\widehat=} [\, x{:}\mathbb{Z} \mid 1 \leq x \leq 5 \,]$ and $D_2 \mathrel{\widehat=} [\, x{:}\mathbb{Z} \mid \exists\, z{:}\mathbb{N} \bullet x = z + z \,]$ with default relabelling and implicit correspondence relation $id_{\mathbb{Z}}$ yields $D \mathrel{\widehat=} [\, y{:}\{2,4\} \,]$. This violates the state consistency condition, for example for x=1, (1,1) is in the correspondence relation, the predicate of D_1 holds, but that of D_2 does not. However, if the only operation defined on both viewpoints is

$$
\begin{array}{|l}
\multicolumn{1}{l}{Toggle_i} \\
\Delta D_i \\
\hline
x + x' = 6 \\
\hline
\end{array}
$$

then the unification *is* a refinement of both original specifications. This is because the new state space D is closed under the operations $Toggle_i$.

□

Apparently a condition weaker than state consistency would also suffice. The condition we are looking for is that *if* a before-state is linked to a unified state by the state unification's retrieve relation, a possible corresponding after-state should *also* be linked to the unified state by that retrieve relation. State consistency guarantees this condition, by making sure the correspondence relation does not link legal with illegal values. Another option would be to demand that all operations "respect" the correspondence relation, but this would give a quantification over all present and future operations. Also, that would make state unification dependent on operations, which seems to introduce a circular dependency.

So, now we know that state consistency is formally too strong, is it a problem to impose it as a condition on state unification? We should probably let our interpretation come to the rescue here. In general, in Z data refinement it is not necessary for every abstract state to be represented by a concrete state. However, in the examples we have considered so far, the data types defined in the viewpoints included *only* meaningful values that would be just as meaningful in the unification. For a unified state space *not* to represent some values of a viewpoint state space just seems wrong in our interpretation. This is exactly what state consistency prevents. Thus, state consistency may be *formally* too strong for checking that a unification is a refinement, in our *interpretation* it is the right condition even when it is not formally necessary. A methodological advantage of using the state consistency condition is that it simplifies the unification process: state unification can be done mostly independently of operation unification. A new operation may be added to both viewpoints at any later point without the possible consequence of invalidating state unification – however, if new operations fail their operation consistency checks, this may still indicate that the correspondence relation was not chosen correctly. Obviously a certain way of guaranteeing state consistency is to define R not on $S \times T$ but on its subset $\{x{:}S \mid pred_S\} \times \{x{:}T \mid pred_T\}$.

The fact that our unification is the least common refinement whenever it exists, and the many properties that hold of Z refinement as a partial order, strongly suggest that when the unification is not a refinement of the viewpoints, no common refinement satisfying the given correspondence relation exists, so an inconsistency between the viewpoints has been found.

6 Variations and extensions to the algorithm

We have given an algorithm in the previous sections, essentially to unify two viewpoint specifications, each of which consists of a number of state schemas with their operation schemas. In this section we describe some ways of extending and adapting this algorithm to make it usable in more situations and part of a multiple viewpoints software development model.

6.1 Deriving correspondences

Apart from giving explicit or default correspondence relations, there may in some cases be another way of establishing a correspondence relation. This method, similar to a common way of establishing (bi-)simulations between process algebraic specifications, starts from the requirement of initialisation consistency. In the case where each initialisation determine a unique initial value, these values need to be related by the correspondence for initialisation consistency to hold. Operation consistency demands that for matching operations, values linked by the correspondence before the operation need to result in values linked by the correspondence afterwards. The smallest set satisfying these properties for all operations is a sensible correspondence relation.

6.2 More than two viewpoints

The properties of Z data refinement, in particular transitivity and the existence of a least common refinement (as proved in section 5), guarantee that the method of finding a unification of multiple viewpoints by an arbitrary sequence of binary unifications (cf. section 1.2) will indeed work for Z viewpoints. However, there is one important issue to be addressed: what correspondence relations will be needed for establishing consistency between n viewpoints?

It is clear that state unification using the default correspondences and default relabellings on compatible types is associative, in other words, the schemas resulting from any bracketing of the unification are semantically identical. For example, the three state schemas

$$\begin{array}{l} D_1 \\ \hline x : S \\ \hline pred_S \end{array} \qquad \begin{array}{l} D_2 \\ \hline x : T \\ \hline pred_T \end{array} \qquad \begin{array}{l} D_3 \\ \hline x : V \\ \hline pred_V \end{array}$$

(assuming $S \cup T \cup V$ is well-defined) will be unified to

$$\begin{array}{l} D \\ \hline x : S \cup T \cup V \\ \hline x \in S \Rightarrow pred_S \\ x \in T \Rightarrow pred_T \\ x \in V \Rightarrow pred_V \end{array}$$

no matter which order of unification is taken. In fact, it appears that in this situation it might be profitable not to do operation adaptation and operation unification on the intermediate state, but only on the three-way state unification.

The general case, however, is not as easy. We cannot expect the specifier to come up with correspondence relations in terms of the intermediate state spaces (which involve bottoms etc.), so all we can assume is that correspondence relations between viewpoint state spaces exist. So the general situation for n=3 is that we have

$$\begin{array}{l} D_1 \\ \hline x : S \\ \hline pred_S \end{array} \qquad \begin{array}{l} D_2 \\ \hline x : T \\ \hline pred_T \end{array} \qquad \begin{array}{l} D_3 \\ \hline x : V \\ \hline pred_V \end{array}$$

with correspondence relations $R_{12} \subseteq S \times T$, $R_{13} \subseteq S \times V$, $R_{23} \subseteq T \times V$. In order to unify a state and a unified state, we have to derive a new correspondence relation between them. The most obvious way of doing this is to assume that e.g. (for the case where D_1 and D_2 are unified first) $R_{12,3}$ is the smallest relation satisfying

$$(justS\ s,x) \in tot\ R_{12} \wedge (s,v) \in R_{13} \Rightarrow ((justS\ s,x),v) \in R_{12,3}$$

$$(x,justT\ t) \in tot\ R_{12} \wedge (t,v) \in R_{23} \Rightarrow ((x,justT\ t),v) \in R_{12,3}$$

However, for arbitrary initial correspondence relations the resulting three-way state unifications arising from different orders of unification are not even necessarily isomorphic.

Example 33 In the general schema above, take all predicates to be true and let the sets be given by the one element free types $S ::= a$, $T ::= b$, $V ::= c$, and the correspondence relations by $R_{12}=\varnothing$, $R_{13}=\{(a,c)\}$ and $R_{23}=\{(b,c)\}$. Then $R_{12,3}=\{((justS\ a,\perp_T),c),((\perp_S,justT\ b),c)\}$ and its totalisation is a two element set, but $R_{1,23}=\{(a,(justT\ b,justV\ c))\}$ and its totalisation will be a one element set.

□

Clearly extra conditions on the various correspondence relations are necessary for n-way correspondence relations to make sense – maybe we could even call this *correspondence consistency*. The situation becomes a little simpler if we only allow the specification of a minimal number of these correspondence relations, with all others derived from those. This certainly seems realistic when the viewpoints can be viewed to be in a sequence (probably of increasing level of detail) where for each viewpoint only the correspondences to the ones adjacent to it are necessary.

6.3 Local state components

State unification via correspondence relations unfortunately does not model all possible ways of composing state spaces. In particular, it is not immediately obvious how to model that components of a state space are local to their viewpoint, i.e. cannot be changed by operations from outside that viewpoint. This seems to be a consequence of the fact that the unification is a *least* refinement more than anything else. The empty correspondence relation does model the (inconsistent!) situation where state spaces are *completely* unrelated, but there the viewpoints turn out to *completely* exclude each other in the unification. Now consider the situation

D_1
$x : S$
$pred_S$

D_2
$x : T$ $y : V$
$pred_{TV}$

where we really want only a correspondence between the two components labelled x. Given $R \subseteq S \times T$, how do we construct a sensible correspondence $R' \subseteq S \times (T \times V)$? The obvious solution is not to restrict the y component at all, i.e. to have

$$(s,(t,v)) \in R' \Leftrightarrow (s,t) \in R \wedge pred_{TV}[t/x, v/y]$$

However, an operation adapted from D_1 to the unified state will now change the x components, but also allow the y component to be changed to an arbitrary value satisfying $pred_{TV}$ – not necessarily the value of y before. This is an unavoidable effect of the fact that the unification is "least" cf. [10]. We are not aware of a correspondence relation which would come closer to allowing y to be a genuine local component. Given the correspondence relation R' and the state space it induces, there *are* some ways of varying the operation *adaptation* which will bring the "local component" interpretation a little closer. For simplicity, assume R is total, so we can assume the unified state space to be

$$\begin{array}{|l}
\multicolumn{1}{l}{D} \\ \hline
x_1 : S ;\ x_2 : T ;\ y : V \\ \hline
(x_1, x_2) \in R \\
pred_S[x_1 / x] \\
pred_{TV}[x_2 / x] \\ \hline
\end{array}$$

Possible alternative adaptations of an operation given by

$$\begin{array}{|l}
\multicolumn{1}{l}{Op} \\ \hline
\Delta D_1 \\ \hline
pred_1 \\ \hline
\end{array}$$

include one which forces the y component to change *only* if it needs to:

$$\begin{array}{|l}
\multicolumn{1}{l}{Ad} \\ \hline
\Delta D \\ \hline
pred_1[x_1 / x\ ,\ x_1{}' / x'] \\
(\exists\, z : T \bullet (x_1{}', (z, y)) \in R') \Rightarrow y' = y \\ \hline
\end{array}$$

which will be a refinement[10], but not a least refinement in general; or one which does not allow the y component to change at all:

$$\begin{array}{|l}
\multicolumn{1}{l}{Ad} \\ \hline
\Delta D \\ \hline
pred_1[x_1 / x\ ,\ x_1{}' / x'] \\
y' = y \\ \hline
\end{array}$$

which is not even a refinement if $pred_{TV}[x_2{}' / x]$ does not hold whenever $pred_{TV}[x_2 / x]$ does.

The best solution we can offer (in the above situation) is for the *second* viewpoint to include operations

$$\begin{array}{|l}
\multicolumn{1}{l}{Op} \\ \hline
\Delta D_2 \\ \hline
y' = y \\ \hline
\end{array}$$

for all operations Op from the *first* viewpoint which should not affect local variables y. The "strict behavioural" approach to extra operations (cf. section 2.3.2) amounts to stating that *all* variables of one viewpoint are "local" for operations that only the other viewpoint offers.

The last example of an alternative operation adaptation shows that the issue of local state components is closely related to the "framing" problem, which is: how to specify what an operation is allowed to change and what it is supposed to keep unchanged. Partial solutions to this problem, in the context of partial specification in Z, are also discussed in [30].

6.4 Partial specification of inputs and outputs

Due to the input/output condition on data refinement (cf. section 2.3.2), in operation unification we needed to assert that both operations had identical sets of inputs and outputs. This imposes a

[10] Provided state consistency holds.

limitation on partial specification: every specifier of (an aspect of) a particular operation needs to know *all* inputs and outputs of that operation, even those which are irrelevant and unused in that particular viewpoint. This might not be desirable or even realistic.

This problem can be removed by adopting a generalisation of data refinement, called IO-refinement [9]. This refinement relation allows adding inputs and outputs, provided the original outputs can be reconstructed from the new outputs. As a consequence of that, unification based on IO-refinement also allows different sets of inputs and outputs for the operations. The paper [9] also gives a formal motivation for IO-refinement: it is derived from the same abstract characterisation used in [42] to derive standard data refinement.

6.5 Consistency checking in a software development model

In this section we will sketch briefly how we envisage the use of constructive consistency checking through unification as a part of a software development model. More on this issue can be found in the extensive literature on the use of viewpoints in software engineering, e.g. [21, 24, 22] – though the emphasis there is on requirements engineering rather than on the development phase. Our particular approach to consistency handling in a development situation is described in more detail in [6].

Clearly, in the initial specification phase viewpoints need to be developed mostly independently. Occasionally consistency checks can already be done, and in particular establishing correspondence relations early on seems sensible (similar to having data dictionaries). Architectural semantics and specification styles could provide guidelines for this (cf. section 8).

In the development phase, there are essentially two extreme options. One is to unify all viewpoint specifications first, and then develop the resulting unification. This guarantees a common implementation will be found if one exists, and that only one consistency check needs to be done. However, this also eliminates all the advantages of viewpoint specification in this phase. No matter how sophisticated our unification techniques will become, it remains likely that unifications will be complex and more unwieldy than traditional complete specifications. (Though they *are* still likely to be more *correct* due to the separation of concerns that viewpoint specification allows.)

The other extreme is to *only* use unification for consistency checking, and to develop the viewpoint specifications independently as far as possible. Because every development step potentially introduces an inconsistency (by choosing a refinement that is not "common"), consistency checking needs to be done relatively often. That makes this method more suitable for situations where the overlap between the viewpoints is relatively small. Combining these two extremes, a rough guideline would be to unify early where there is much overlap between the viewpoints, and to develop independently where there is little. Additionally, there are approaches [24] which allow for development to continue in the presence of a (temporary) inconsistency.

Thus far we have only described a linear (or tree-like) development process, with success or failure at the end of it, depending on the outcome of the final consistency check. The discovery and resolution of inconsistencies will add iterations to the development process. Not every inconsistency discovered is a serious error in specification or development. For example, there may be no serious problem if an operation's precondition is restricted in the unified state space because of conditions imposed through another viewpoint ("restrictive co-refinement" in the terms of [2], cf. section 7.1). This will often be the desirable effect, and this can usually be resolved by restricting the viewpoint operation or even the state space it operates on in the initial specification, or by reducing the correspondence relation. Another example is given in the case study of our techniques in [7], where operation consistency holds only if a "free" constant of the specification has a trivial value. Such restrictions change the meaning of the initial specification, though in the light of the discussion in section 6.1 the correspondence relation may in some cases be viewed as derived from the specification.

7 Related issues and approaches

7.1 Viewpoint amalgamation and co-refinement

An approach very similar to ours is the one advocated by Ainsworth, Wallis, et al [1, 3, 2]. They use the term *amalgamation* for what we call unification, and *union* for what we call operation unification. Their state unifications are driven by ad-hoc reasoning [1] or by retrieve relations [2] – the latter are fairly close to our correspondence relations. This can be observed from the retrieve relations we used in the proofs in section 5: together these contain exactly the same information as the correspondence relation.

An important concept in their approach is *co-refinement* [3]. They claim that ordinary refinement (for example in Z) is too restrictive to be used in viewpoint specification, because it does not allow viewpoints to put restrictions on operations in other viewpoints. Using co-refinement instead of refinement amounts to maintaining a predicate which represents these restrictions, and which needs to be satisfiable for co-refinement to hold. Comparing this to our approach, part of these predicates would indeed show up as inconsistencies; others will be part of non-trivial correspondence relations. Having such a predicate also gives increased possibilities for incremental specification.

7.2 Specification by views: non-functional correspondence relations

The method of specification by *views* as advocated by Daniel Jackson [30] is very similar to viewpoint specification. The arguments in favour are similar to ours: separation of concerns, with a special emphasis on the possibility of having multiple co-existing representations of states. Such multiple representations are linked by invariants which fulfill the same role as our correspondence relations. The views are linked in a syntactically simple way: by defining a new state space consisting of the view state spaces restricted by the invariant. This has as a side effect that the combined views do not necessarily relate as well *semantically* to the original views. In terms of this paper, the combined state space is the correspondence relation rather than its totalisation, so when the invariant is not total some operations may not be refined because some of their after values have been excluded in view composition. An extensive comparison of these methods may be found in our paper [10], we will present part of an example here.

So far in this paper all correspondence relations (except in example 13) have actually been *injective functions*. We can incorporate (and generalise) Jackson's method, and in general incorporate data type refinement, by using non-functional correspondence relations. As an example, we will present some of the editor example as used by Jackson, based on [41]. For more details of the specification, cf. [30, 10]. This example will also point out the semantical difference between the two methods.

Example 34 (Two views of an editor) One way of specifying the state space of an editor is the following.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{File}} \\
\hline
\textit{left}, \textit{right} : \operatorname{seq} \textit{Char} \\
\hline
\end{array}
$$

The sequence $left$ describes all the characters to the left of the cursor, and $right$ all those to the right of it. One operation on this view is inserting a character:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{insertChar}} \\
\hline
\Delta \textit{File} \\
c? : \textit{Char} \\
\hline
\textit{left}' = \textit{left} \frown \langle c? \rangle \\
\textit{right}' = \textit{right} \\
\hline
\end{array}
$$

In the second view on editors, the *Grid* view, the state is a sequence of sequences of limited length, each constituent sequence representing a line, with a cursor position. This allows specification of operations like moving the cursor down.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Grid\;}} \\
lines : \mathrm{seq}\,\mathrm{seq}\,Char \\
x, y : \mathbb{N} \\
\hline
pred_{Grid}
\end{array}$$

where $pred_{Grid}$ ensures that $lines$ is a correctly wrapped sequence of lines of limited length with (x,y) a sensible cursor position in that grid. A non-immediate consequence of this predicate is that no word can be longer than the maximum line length, because it cannot then be correctly wrapped.

The predicate linking the views has both views in its signature, so one way of representing it is as the schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Editor\;}} \\
File; Grid \\
\hline
left \frown right = \frown / \, lines \land \\
\#left = x + \Sigma(i : 1 \,..\, y-1 \bullet \#lines[i])
\end{array}$$

This states that both views should represent the same text, and that the cursor positions should match: $left$ should be as long as all of the lines before line y together, plus all of line y up to column x.

In view composition, this is all the information we need. The schema *Editor* acts as the unified state space, to which we can now adapt the operations, for example

$$insertCharE \mathrel{\widehat{=}} [\, \Delta Editor \mid insertChar \,]$$

This ensures that a corresponding *Grid* for the new *File* will be found. However, using *Editor* as the unified state tacitly excludes some of the values from the original views. In particular, any *File* with a word longer than the maximum line length is excluded because there is no corresponding *Grid* for it. This has a serious semantic effect. *Editor* with *insertCharE* is *not* a data refinement of *Grid* with *insertChar*. A state where a word only just fits on a line is still in *Editor*, but the state after adding one more character is not, thereby excluding the former one from $\mathrm{pre}\, insertCharE$.

In viewpoint composition, this problem can be resolved. Because the invariant is not total on the *File*state space, some bindings of *File* will get linked to $\perp$, using the invariant as the correspondence relation. The resulting state space, after some renaming, is:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;FileandGrid\;}} \\
x_1 : File \\
x_2 : Grid_{\perp} \\
\hline
(x_1, theGrid\, x_2) \in R \lor (x_2 = \perp_{Grid} \land x_1 \in Longwordfiles)
\end{array}$$

where R is the relational representation of the invariant, and *Longwordfiles* the set of "forbidden" *File*s. The adaptation of *insertChar* to this state space *will* be a data refinement.

$\square$

In general, data refinements and other relations between state spaces can be incorporated in viewpoint unification, by taking the predicates involved as the basis of correspondence relations. This yields all the advantages of view composition and data refinement, often without introducing their disadvantages. A formal justification of this can be found in [10].

7.3 Demonic join and feature interaction

Desharnais, Frappier, Mili and others [11, 25, 26] study a calculus (lattice) of binary relations with a refinement relation which has great similarities to operation refinement in Z. In their framework, our "operation unification" appears as "demonic join", which is only defined if a consistency condition (our operation consistency) holds. They use the term "program construction by parts" [25] for what we call viewpoint- or partial specification. In their recent work [26] they demonstrate that this approach can also be used to investigate the problem of *feature interaction*. Features (e.g. of a telephone system) can only be combined without interaction when their demonic join is well-defined. In our approach, each feature would be a separate viewpoint (operation).

7.4 Conjunction as composition

Zave and Jackson describe in several papers [43, 44] a multiparadigm specification technique, with impressive applications in specifications of telephone switching systems. Their work is similar to ours in that it uses Z and other languages for partial specification. For consistency checking, they use a translation of all specifications to first order predicate logic. Composition of partial specifications is then "just" conjunction [43]. In our approach to unification in Z and between Z and other languages, at some level of interpretation composition is also conjunction – however, as we have argued in [6], we prefer not to work at this level for reasons of traceability.

A particular concern mentioned in [44]:

> "*There is no general method for establishing inter-specification consistency in the presence of shared state components.*"

we believe is one of the main issues that has been addressed, and partially solved, in the current paper.

7.5 Others

Approaches in which Z specifications are augmented with specifications in other formalisms can also be viewed as specifications with multiple viewpoints, which may have consequences similar to those described in our work on comparing viewpoints in LOTOS and Z [17, 15]. However, most methods that combine Z with some other language manage to avoid the consistency issue by the use of layering techniques, or by using the various languages in different stages of development [28, 35]. Kasurinen and Sere [33], for example, in their integration of Z and action systems use a layering technique, Z providing the types and operations to be used in the action systems descriptions.

Other viewpoint methods [22] generally do not base their notion of consistency on development relations. Partly this is due to the fact that they use languages which are less formal or development oriented than the ones we use. Consistency is often determined by explicit consistency relations on and between the viewpoints [23], based on overlap identification (akin to our correspondences) and similarity analysis. Unification, however, also seems a useful process for consistency checking in requirements engineering [14].

8 Future work

Promotion

An established method of combining state spaces and their operations in Z is that of framing and promotion. The actual promotion is where operations on components of a system get combined to form top level operations. Often so-called framing schemas are used in this, which ensure that uninvolved parts of the system remain unchanged. This technique can be profitably used for specifying viewpoints at different levels of abstraction, cf. the example of a telephone system in [30] or that of the dining philosophers in [18]. The latter example also shows that, provided it is used in a particular way (which describes one viewpoint's using "standard components" provided by another), consistency is almost guaranteed. We wish to further investigate how promotion-based specification styles can significantly reduce consistency checking obligations. The examples mentioned above, and the fact that promotion commutes with refinement [42] indicate this is a promising approach.

Behavioural interpretations

We want to be able to investigate consistency between descriptions of states and operations as in Z, and descriptions of behaviour as in process algebras, e.g. LOTOS. For this purpose, we need to impose a behavioural interpretation on Z specifications, and relate development relations used in the process algebra world to refinement and possibly other development relations for Z. First results of these investigations are reported in [17, 16, 15].

Acknowledgements

We would like to thank the referees for their comments, which were extremely helpful for improving the contents and presentation. We are particularly thankful to one referee for pointing out the important consequences of the initialisation of ADTs.

Ralph Miarka and Philipp Heuberger commented on earlier drafts of this paper. The LaTeX code for this paper was generated using the MathSPad editing tool (http://www.win.tue.nl/win/cs/wp/mathspad/) with special stencils for oz.sty.

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] M. Ainsworth, S. Riddle, and P.J.L. Wallis. Formal validation of viewpoint specifications. *Software Engineering Journal*, 11(1):58–66, January 1996.

[3] M. Ainsworth and P. J. L. Wallis. Co-refinement. In D. Till, editor, *Proc. 6th Refinement Workshop*, City University, London, 5th–7th January 1994. Springer-Verlag.

[4] R. D. Arthan. On free type definitions in Z. In Nicholls [34], pages 40–58.

[5] E. Boiten, H. Bowman, J. Derrick, and M. Steen. Issues in multiparadigm viewpoint specification. In Finkelstein and Spanoudakis [22], pages 162–166.

[6] E. Boiten, H. Bowman, J. Derrick, and M. Steen. Managing inconsistency and promoting consistency. Submitted for publication, September 1997.

[7] E. Boiten, H. Bowman, J. Derrick, and M. Steen. Viewpoint consistency in Z and LOTOS: A case study. 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 644–664. Springer-Verlag, September 1997.

[8] E. Boiten, J. Derrick, H. Bowman, and M. Steen. Consistency and refinement for partial specification in Z. In Gaudel and Woodcock [27], pages 287–306.

[9] E.A. Boiten, J. Derrick, H. Bowman, and M. Steen. IO-refinement in Z. Submitted for publication.

[10] E.A. Boiten, J. Derrick, H. Bowman, and M. Steen. Coupling schemas: data refinement and view(point) composition. In D.J. Duke and A.S. Evans, editors, *Northern Formal Methods Workshop*, Electronic Workshops In Computing. Springer, 1997.

[11] N. Boudriga, F. Elloumi, and A. Mili. On the lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.

[12] J.P. Bowen, M.G. Hinchey, and D.Till, editors. *ZUM '97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[13] H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency. Submitted for publication, 1997.

[14] H.S. Delugach. An approach to conceptual feedback in multiple viewed software requirements modeling. In Finkelstein and Spanoudakis [22], pages 242–246.

[15] J. Derrick, E.A. Boiten, H. Bowman, and M. Steen. Translating LOTOS to Object-Z. In D.J. Duke and A.S. Evans, editors, *Northern Formal Methods Workshop*, Electronic Workshops In Computing. Springer, 1997.

[16] J. Derrick, E.A. Boiten, H. Bowman, and M. Steen. Weak refinement in Z. In Bowen et al. [12], pages 369–388.

[17] J. Derrick, H. Bowman, E. Boiten, and M. Steen. Comparing LOTOS and Z refinement relations. In *FORTE/PSTV'96*, pages 501–516, Kaiserslautern, Germany, October 1996. Chapman & Hall.

[18] J. Derrick, H. Bowman, and M. Steen. Maintaining cross viewpoint consistency using Z. In K. Raymond and L. Armstrong, editors, *IFIP TC6 International Conference on Open Distributed Processing*, pages 413–424, Brisbane, Australia, February 1995. Chapman and Hall.

[19] J. Derrick, H. Bowman, and M. Steen. Viewpoints and Objects. In J. P. Bowen and M. G. Hinchey, editors, *Ninth Annual Z User Workshop*, LNCS 967, pages 449–468, Limerick, September 1995. Springer-Verlag.

[20] M. d'Inverno and M. Hu. A Z specification of the soft-link hypertext model. In Bowen et al. [12], pages 297–316.

[21] A. 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.

[22] A. Finkelstein and G. Spanoudakis, editors. *SIGSOFT '96 International Workshop on Multiple Perspectives in Software Development (Viewpoints '96).* ACM, 1996.

[23] A. Finkelstein, G. Spanoudakis, and D. Till. Managing interference. In Finkelstein and Spanoudakis [22], pages 172–174.

[24] 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.

[25] M. Frappier, A. Mili, and J. Desharnais. Program construction by parts. In B. Möller, editor, *Mathematics of Program Construction: Third International Conference*, volume 947 of *Lecture Notes in Computer Science*, pages 257–281. Springer-Verlag, 1995.

[26] M. Frappier, A. Mili, and J. Desharnais. Defining and detecting feature interactions. In R.S. Bird and L. Meertens, editors, *IFIP TC2 WG 2.1 International Workshop on Algorithmic Languages and Calculi*, pages 212–239. Chapman & Hall, 1997.

[27] M.-C. Gaudel and J. Woodcock, editors. *FME'96: Industrial Benefit of Formal Methods, Third International Symposium of Formal Methods Europe*, volume 1051 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1996.

[28] M.G. Hinchey. JSD, CSP and TLZ. In *Methods Integration Workshop*, Leeds, 1996.

[29] ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.

[30] D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4), October 1995.

[31] D. Jackson and M. Jackson. Problem decomposition for reuse. *Software Engineering Journal*, 11(1), January 1996.

[32] He Jifeng and C.A.R. Hoare. Prespecification and data refinement. In *Data Refinement in a Categorical Setting*, number PRG-90 in Technical Monograph. Oxford University Computing Laboratory, November 1990.

[33] V. Kasurinen and K. Sere. Integrating action systems and Z in a medical system specification. In Gaudel and Woodcock [27], pages 105–119.

[34] J. E. Nicholls, editor. *Z User Workshop, York 1991*, Workshops in Computing. Springer-Verlag, 1992.

[35] F. Polack and K. C. Mander. Software quality assurance using the SAZ method. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 230–249. Springer-Verlag, 1994.

[36] M. Rizzo, M. Steen, H. Bowman, J. Derrick, I. Utting, et al. A case study in ODP viewpoint specification: a video telephone system. Work in progress, 1997.

[37] R. O. Sinnott and K. J. Turner. Specifying ODP computational objects in Z. In E. Najm and J.-B. Stefani, editors, *Proc. 1st IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 375–390. Chapman & Hall, March 1996.

[38] A. Smith. On recursive free types in Z. In Nicholls [34], pages 3–39.

[39] J. M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 1989.

[40] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, 1992.

[41] B.A. Sufrin. Formal specification of a display-oriented text editor. *Science of Computer Programming*, 1:157–202, 1982.

[42] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.

[43] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.

[44] P. Zave and M. Jackson. Where do operations come from? A multiparadigm specification technique. *IEEE Transactions on Software Engineering*, 22(7):508–528, July 1996.