VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Viewpoint Specification and Z

M.Ainsworth*, A.H.Cruickshank*, L.J.Groves† and
P.J.L.Wallis*

Technical Report CS-TR-94/7
January 1994

Abstract

We discuss the use of viewpoint specifications, a technique which concentrates on making large specifications more understandable. Rather than specifying the whole system at once, a system is described using several self-contained partial specifications, which may then be amalgamated to give a description of the complete system. Amalgamation is taken to be a composite process in which the data and operations of the constituent viewpoints are separately considered. The approach is illustrated in terms of Z specifications.

Publishing Information

Appeared in *Information and Software Technology* **36**, 1 (January 1994), *pp*43–51.

*School of Mathematical Sciences, University of Bath, Claverton Down, Bath BA2 7AY, United Kingdom. Email: `ma,ahc,pjlw@maths.bath.ac.uk`.

†Email: `lindsay@comp.vuw.ac.nz`.

1 Introduction

Modern computer-based systems are highly complex because computers are efficient processors of large quantities of detailed information. Human beings are not so good at this processing: we prefer to understand complicated systems in terms of a few ideas at a time. This difference between the information-processing capabilities of people and of computers is a major cause of *scale problems* in software specification. Such problems manifest themselves when an attempt to develop a formal specification for a complex system leads to a document that leaves the poor human reader quite unable to see the forest for the trees. The thesis of this work is that we should try to alleviate scale problems in formal software specification by setting out to mimic the techniques that people find useful in thinking about complicated systems in an intuitively appealing way.

A computer user trying to understand a modern software system such as a text-processing package will build up a number of areas of knowledge about particular aspects of the facilities offered, such as typefaces, page layout, character set and hyphenation. Each individual item of knowledge acquired is quite straightforward, the combination of such items reflecting the user's current knowledge of the system. A large system is relatively easy to understand if a new item of knowledge (e.g. the presence of a new font) can be understood in terms of a simplification of the remainder of the system. Our suggested approach to software specification is to enable each separate item of knowledge about the system to be specified in as succinct a manner as possible; to borrow a phrase from the study of software reuse, we wish to be able to construct a specification by combining many specifications of low *perceived complexity* [BW88].

The idea that we are suggesting is that of *viewpoint* specification: a software specification should be constructed by taking simple specifications of *viewpoints* of the system and *amalgamating* them to produce the final system specification. Viewpoint techniques have been quite widely used in requirements analysis [Lei89, Rob89, FF89, FKN$^+$92], but here they are applied to the development of formal specifications. Our approach is to develop a generic technology for viewpoint amalgamation, and then to apply this to the manipulation of Z specifications. The Z specification language is a particularly interesting vehicle for such an experiment, since the Z *schema calculus* [Woo89] already provides language features that go some way towards the satisfactory support of specification techniques using viewpoints.

This paper introduces current research aimed at a general understanding of viewpoint techniques in software specifications and their application to the development of Z specifications. We begin by discussing *viewpoints* and their *amalgamation* to build specifications in quite general terms, and then introduce the idea that amalgamation may be usefully divided into three consecutive processes called *preparation*, *co-refinement* and *coalescence*. These ideas are then applied to the development of a Z specification.

2 Viewpoints and Amalgamation

Viewpoints are partial formal descriptions of systems. Each viewpoint is a *partial* specification in that it only describes certain aspects of the system. Each viewpoint is also *self-contained* as far as possible, in that it requires minimal knowledge of any other viewpoint for it to be understood. Having described our system as a number of viewpoint specifications, we are then faced with the task of implementing the specification. At this point we need some way of amalgamating viewpoints and studying the resulting larger specifications. There are two main reasons for this: one is that it is easier to check for consistency if we can look at the entire specification at once, the other is that an arrangement of viewpoints which is suitable for the designers may not be the easiest for the programmers to work with.

In this section we discuss in general terms a technology for amalgamating viewpoints. The model is generic in the sense that it is independent of the language used to write the viewpoints. Our basic model for the amalgamation process is the MFD (Modular Formal Description) Model [Wal92, Wal93]. Here we consider only the amalgamation of pairs of viewpoints, since any number of viewpoints may be amalgamated by performing a sequence of pairwise amalgamations.

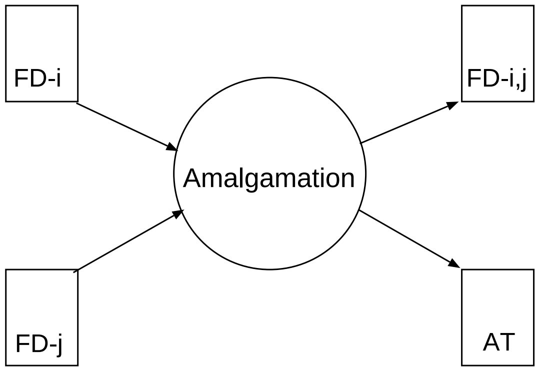

Figure 1: The Amalgamation Process

The basic shape of the MFD Model is shown in Figure 1 [Wal92]. The amalgamation process takes two formal descriptions (FD_i, FD_j) as inputs and combines them into one amalgamated description ($FD_{i,j}$). In doing so it also produces an *amalgamation trail* (AT). The amalgamation trail is a record of all the changes that were made during the amalgamation, and ensures that the amalgamation process is reversible. Eventually it is hoped that this reversibility might be exploited to allow, for instance, a set of formal descriptions to be combined into a specification of the whole system, and then split apart in a different fashion for implementation purposes. More detailed consideration of the amalgamation trail is outside the scope of the present paper.

2.1 Preparation, Co-Refinement and Coalescence

Previous applications of amalgamation have generally been in areas such as software requirements analysis, where the different viewpoints to be amalgamated may well have been developed by different teams using different formalisms. Amalgamation of such viewpoints often involves considerable effort in relating concepts between the different formalisms. For the purposes of this paper, where we are primarily concerned with specifications written in Z, we assume that viewpoints are written in the same language, and that it is possible to relate elements in different viewpoints using functional invariants. The use of invariants to describe relationships between elements in different viewpoints allows us to formally express our amalgamations using refinement techniques, and to provide proof obligations for the verification of the amalgamation.

We consider a viewpoint to be a set of elements, where an element might be a set of values (a type), a data object, an operation etc. Each viewpoint is written with respect to an underlying model which describes the type of each element. Two viewpoints may be inconsistent in various ways.

A *syntactic clash* occurs when an identifier is defined differently in different viewpoints. In this case, the two definitions need to be inspected to determine whether they represent the same idea. This requires human intuition — even if the two definitions are identical, they may actually represent different ideas. Two viewpoints are said to be *coherent* if there are no syntactic clashes.

A *semantic clash* occurs when an operation is defined differently in different viewpoints. Again, we need to inspect the two definitions to determine how they should be combined. Two viewpoints are said to be *contradictory* if there are semantic clashes, and *mutually feasible* (or *non-interfering*) if there are none.

The overall amalgamation process used in this paper is, therefore, divided into three main stages:

- *preparation* — in which we identify the elements to be amalgamated, and perform simple pre-processing.
- *co-refinement* — in which we identify and resolve clashes between viewpoints.
- *coalescence* — in which we merge the viewpoints.

This model of amalgamation is similar to that used in [Wal92] in general outline: the same ideas can be more formally presented in terms of co-refinement [AW94]. A full presentation of the MFD model and its relationship to the formal co-refinement relation is in preparation [Wal93].

2.2 Preparation

The preparation phase involves identifying which elements in the viewpoints are to be amalgamated, and may also involve some simple pre-processing of the viewpoints.

In practice, at this stage we might also do some restructuring of the viewpoints, such as moving some parts of the body of a specification into the bodies of separate functions. This *fragmentation* process is intended to make the eventual amalgamation easier.

2.3 Co-Refinement

The co-refinement phase is itself divided into two main processes: *data co-refinement* in which we resolve syntactic clashes, and *operation co-refinement* where we resolve semantic clashes. During data co-refinement we also identify situations where the same idea is introduced in both viewpoints using different names (e.g. `colour` in one viewpoint, and `color` in another). Although not strictly clashes, these situations would lead to redundancy in the amalgamated specification if left.

The basic form of the co-refinement is a repetitive process in which we successively locate a potential clash, determine whether it is an actual clash and if so, decide how to resolve the clash and rewrite the viewpoint accordingly. In the remainder of this section, we look at various kinds of clashes that might arise, and discuss possible ways of resolving them.

2.3.1 Planning Data Co-Refinement

If an existing system is subsequently split into viewpoints, each viewpoint will use the same underlying model, and hence each element will have the same type throughout the system.

When viewpoints are written independently, however, it is often the case that each viewpoint will use a different underlying model. For example a formal specification of a filing system might regard a file in terms of an access code in one viewpoint, and as a sequence of characters in another. In such cases, a common model must be constructed, and each viewpoint rewritten to use the common model, before any attempt can be made to amalgamate the viewpoints and study the combined system. This process of determining the combined model and rewriting the viewpoints is known as *data co-refinement*. In the data co-refinement stage we merge the *state* parts of the specifications.

If an element has different types in the two viewpoints, then we normally choose an amalgamated type which can represent all the possible values. Sometimes, however, the amalgamation does not need to (and cannot) represent all information that can be represented by each of the separate viewpoints. How the two types are combined depends on the information represented by those elements, and relationships between their types. For example:

- If the two elements represent the same attribute of some object, but describe it using different types, say t_1 and t_2, we will probably choose a type t_3 such that there is a one-to-one correspondence between elements of t_1 and some subset of t_3, and likewise with t_2. If t_1 and t_2 have the same base type (e.g. they are both sets of naturals), t_3 will just be their union.
- If the two elements represent different attributes of some object, we will probably take the cross-product of the two types. So, for our filing system above, the amalgamation might regard a file as an ordered pair comprising both an access code and a sequence of characters for the file's contents.

The ways in which types are amalgamated during data co-refinement are similar to the ways in which database schemas are combined during database integration: see for example [BLN86].

The data co-refinement process can be formalised in terms of data refinement [MG90]. If an element is represented using types x, y in the two viewpoints, and by type z in the amalgamation of those viewpoints, we require z to be a data refinement of both x and y. That is,

$$(x \preceq_I z) \wedge (y \preceq_I z)$$

for some suitable abstraction invariant (or retrieve relation) I. We will assume that I is functional from z to x and from z to y, and I will normally be adequate [Jon90] with respect to each of the initial viewpoints. If the amalgamation restricts the states of individual viewpoints, however, I will describe a *partial* data refinement. We can combine these separate relations together to form a single invariant which describes the relationships between the states used in both viewpoints and the state described in the amalgamation. Abstraction invariants are also recorded in the amalgamation trail, and can be used as the basis for verifying the amalgamation (these concepts are described in more detail in [AW94]).

2.3.2 Planning Operation Co-Refinement

Once we have all the viewpoints expressed in terms of a common coherent model, we can then amalgamate the operations described in the viewpoints. Different viewpoints might describe different aspects of the same operation. So a `save-file` operation in one viewpoint might produce an access code for the file, while in another viewpoint, `save-file` might write a stream of characters to a disc. The full operation would be a combination of both of these.

Operation co-refinement involves identifying semantic clashes between viewpoints, deciding precisely how the partial specifications of operations should be amalgamated, and if necessary rewriting viewpoints to remove clashes. While data co-refinement is a mainly *syntactic* process, operation co-refinement is a *semantic* process, and consequently can be more problematic.

Our basic aim in amalgamating two operations is to produce an operation which satisfies each of the original viewpoints. That is, given viewpoints A and B, their amalgamation C should satisfy

$$(A \preceq C) \wedge (B \preceq C)$$

for some suitable relation $\preceq$. In many cases $\preceq$ will be a refinement relation, but this may not be so in all cases.

We can use the principles of data refinement to verify the amalgamation of the operations. If we have a viewpoint operation Vop over a state V and an amalgamated operation Aop over a state A, together with some retrieve relation $Retr$ between V and A, then the amalgamated operation is a data refinement of the viewpoint operation if:

$$\begin{array}{l} \mathrm{pre}\ Vop \wedge Retr \Rightarrow \mathrm{pre}\ Aop \\ \mathrm{pre}\ Vop \wedge Retr \wedge Aop \Rightarrow \exists\, V \bullet Vop \wedge Retr' \end{array}$$

These two conditions are known as *applicability* and *correctness* [Jos88]. The first condition (applicability) states that if the original operation is valid in a state, then the new operation must also be valid for the corresponding state in the merged system. The second condition states that if we apply the new operation we must get a result which could have been obtained by the original operation[1].

If VI is an initialisation condition for the viewpoint operation and AI is an initialisation for the corresponding amalgamated operations, then we have an additional condition:

$$\exists\, A \bullet AI \wedge (\forall\, A \bullet (\exists\, V \bullet VI \Rightarrow AI \wedge Retr))$$

This is the *initialisation* condition, which states that the initial state of the amalgamation must be a data refinement of the initial state of the viewpoint.

In addition, we would have to show that any retrieve relation used for the operation co-refinement is consistent with the relation used in the data co-refinement stage. We require each of the retrieve relations used in the operation co-refinement to be a *conservative extension* of the retrieve relation used for the data co-refinement, where we define J to be a conservative extension of I if $J(x) = I(x)$ for any x in the domain of I.

The combination of specifications written in the Z specification language is of particular interest because many (but not all) of the necessary operations are available as a part of the Z schema calculus. The following are some of the main ways that viewpoint descriptions of operations might be amalgamated in Z [Spi89, PST91]: similar constructions can be derived in other specification languages.

[1] Note that the new operation may be more deterministic than the original.

Overriding If A is satisfied by B (i.e. B is a refinement of A), then we can simply use B as the amalgamation, and *vice versa*.

In the more general case where neither viewpoint is a refinement of the other, we must produce a new specification from the original viewpoints which satisfies both viewpoints simultaneously.

Conjunction The most obvious way of combining two schemas is to use schema conjunction

$$C \mathrel{\widehat{=}} A \wedge B$$

However, the precondition of C would then be

$$\text{pre } C = \text{pre } A \wedge \text{pre } B$$

which is stronger than the preconditions of the original viewpoints. This sort of amalgamation restricts the states of the original viewpoints. Whether this is regarded as a refinement or not depends on our definition of satisfaction for viewpoints.

Union If we wish to preserve a wider range of possible behaviours in our amalgamation we can define a new operator. We refer to this as a *union*, denoted $\cup$.

In terms of pre- and post-conditions, this is expressed as:

$$\text{pre } A \cup B = \text{pre } A \vee \text{pre } B$$
$$\text{post } A \cup B = (\text{pre } A \Rightarrow \text{post } A) \wedge (\text{pre } B \Rightarrow \text{post } B)$$

If both pre-conditions are true, then the union will satisfy both post-conditions. If only one pre-condition is true, then only the relevant post-condition has to be met.

2.4 Coalescence

Once the co-refinement phase has been carried out, we should be left with two non-interfering viewpoints and a script for their amalgamation (the *coalescence plan*). The only remaining task is to gather together the elements described in the various viewpoints into a single specification. This should be a fairly straightforward process that could be mechanised, for example techniques for amalgamating non-interfering programs are already available [HPR89].

3 Ward Monitoring Example

In this section an example of viewpoint amalgamation using the model-based specification language Z [Spi89, PST91] is given. The use of Z for this example illustrates the point that not all the operations required are directly supported by the Z schema calculus. Our example concentrates on the amalgamation phase of co-refinement, comprising data co-refinement and operation co-refinement.

When performing data co-refinement, it is not generally possible to use the Z schema calculus to describe the overriding or merging of two types or to describe the rewriting of other elements within the viewpoints, and further notation will have to be introduced.

In the coalescence phase the Z schema calculus can be used to describe some of the merging operations, but by no means all of them. In particular, the schema calculus operations can only be applied to schemas and schema calculus statements. They cannot be used to describe the merging of free type branches or axiomatic definitions, for example. In addition to this, there is no overriding operator[2] in the schema calculus. Hence when describing the coalescence operation being performed, several extensions to the Z schema calculus must be used. The actual coalescence operations are beyond the scope of this paper, though they are intuitive to some degree, and in the following examples only the final amalgamated specification will be given.

The viewpoints used are self-contained and small, and only describe a part of the total system. Hence a reader of the specification only needs to look at and understand the part of the system of current interest, without the need to filter out unnecessary details. This means that the *perceived complexity* of individual viewpoints is greatly decreased.

In addition, the breakdown of the system into viewpoints is largely intuitive and not constrained by artificial structuring techniques which apply to the computer more than the specifier. This should lead to viewpoint specifications which are easier to write as well as easier to understand.

3.1 Overview of Ward Monitoring System

The system to be specified is a ward monitoring system, which maintains a continuous check on patients, alerting the duty nurse if any of the patients is in danger. The example is based on that found in [Fea89]; it was also presented less formally in [ACGW93].

The system is specified using three viewpoints which detail the following:

1. patient details: admission and monitoring of their conditions.
2. device details: adding a new device to the available devices and monitoring their performance.
3. links between patients and devices: attaching a device to a patient and reading the correct value.

3.2 Viewpoint 1: Monitoring Patients

In the first viewpoint we consider a patient monitoring system which only knows about patients and a status value which describes how "well" they are. A warning is given if a patient's status value becomes unsafe.

We have a set of patients and a set of status values. The values represent relevant status information about the patients.

$$[PATIENT, VALUE]$$

A warning must be given if a patient is in danger (i.e. with a status value exceeding a stated limit), otherwise a report of "ok" is generated.

$$\begin{aligned} REPORT ::= {} & ok\langle\!\langle PATIENT\rangle\!\rangle \\ | {} & patient\_warning\langle\!\langle PATIENT\rangle\!\rangle \end{aligned}$$

[2] I.e., there is no way of replacing one schema with a different version.

We have a set of patients who are currently in the ward. In addition to this information we need to determine the status value for each patient, and this is modelled by a partial mapping of patients to status values.

$$
\begin{array}{|l}
\hline
\textit{Patients} \\
\textit{inpatients} : \mathbb{P}\, \textit{PATIENT} \\
\textit{pvalue} : \textit{PATIENT} \nrightarrow \textit{VALUE} \\
\hline
\operatorname{dom} \textit{pvalue} = \textit{inpatients} \\
\hline
\end{array}
$$

Initially a ward is empty.

$$
\begin{array}{|l}
\hline
\textit{InitWard} \\
\textit{Patients}' \\
\hline
\textit{inpatients}' = \varnothing \\
\textit{pvalue}' = \varnothing \\
\hline
\end{array}
$$

The initial value for new patients is given by the following definition.

$$
\begin{array}{|l}
\textit{init\_value} : \textit{VALUE}
\end{array}
$$

Adding a patient to the ward is specified as follows. Patients can only be added to the ward if they are not already inpatients.

$$
\begin{array}{|l}
\hline
\textit{AddPatient} \\
\Delta \textit{Patients} \\
\\
\textit{new\_patient}? : \textit{PATIENT} \\
\hline
\textit{new\_patient}? \notin \textit{inpatients} \\
\\
\textit{inpatients}' = \textit{inpatients} \cup \{\textit{new\_patient}?\} \\
\textit{pvalue}' = \textit{pvalue} \cup \{\textit{new\_patient}? \mapsto \textit{init\_value}\} \\
\hline
\end{array}
$$

Whether the patients' status values mean that they are in danger or not is described by a mapping from a value to *true* or *false*. If a value maps to *false*, then a warning should be given.

$$
\begin{array}{|l}
\hline
\textit{Safety} \\
\textit{safe} : \textit{VALUE} \longrightarrow \textit{Boolean} \\
\hline
\end{array}
$$

The monitoring station outputs a set of reports, one report for each patient. If a patient's status is unsafe, then a warning report is given, otherwise an "ok" report is generated.

$$
\begin{array}{|l}
\hline
\textit{Monitor} \\
\Xi \textit{Patients} \\
\Xi \textit{Safety} \\
\\
\textit{p\_reports}! : \mathbb{P}\, \textit{REPORT} \\
\hline
\textit{p\_reports}! = \{p : \textit{PATIENT} \mid p \in \textit{inpatients} \land \textit{safe}(\textit{pvalue}(p)) = \textit{True} \bullet \\
\qquad\qquad\qquad\qquad \textit{ok}(p)\} \cup \\
\qquad\qquad \{p : \textit{PATIENT} \mid p \in \textit{inpatients} \land \textit{safe}(\textit{pvalue}(p)) = \textit{False} \bullet \\
\qquad\qquad\qquad\qquad \textit{patient\_warning}(p)\} \\
\hline
\end{array}
$$

3.3 Viewpoint 2: Monitoring Devices

In the second viewpoint we consider a device monitoring system which only knows about devices. If a device fails, then a warning is given.

The type of a device is not important, and is modelled by a given type.

$$[DEVICE]$$

A warning must be given if a device fails, otherwise a report of "ok" is generated.

$$\begin{aligned} REPORT ::= {} & ok\langle\!\langle DEVICE \rangle\!\rangle \\ \mid {} & device\_warning\langle\!\langle DEVICE \rangle\!\rangle \end{aligned}$$

We have a set of devices which are currently available and a mapping from each device to a boolean value. The mapping describes whether a device has failed or is still functioning correctly.

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{Devices}} \\ \hline devices : \mathbb{P}\ DEVICE \\ failed : DEVICE \leftrightarrow Boolean \\ \hline \mathrm{dom}\ failed = devices \\ \hline \end{array}$$

Initially there are no devices available.

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{InitDevices}} \\ \hline Devices' \\ \hline devices' = \varnothing \\ failed' = \varnothing \\ \hline \end{array}$$

We need operators to add new devices, delete old devices, repair devices, etc, but here we will only consider adding a device. We assume that only working devices are added to the available devices.

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{AddDevice}} \\ \hline \Delta Devices \\ new\_device? : DEVICE \\ \hline new\_device? \notin devices \\ devices' = devices \cup \{new\_device?\} \\ failed' = failed \cup \{new\_device? \mapsto False\} \\ \hline \end{array}$$

The monitoring station outputs a set of reports, one report for each device. If a device fails then a warning is given, otherwise an "ok" report is generated.

$$
\begin{array}{|l}
\hline
\textit{Monitor} \\
\Xi Devices \\
d\_reports! : \mathbb{P}\ REPORT \\
\hline
d\_reports! = \{ d : DEVICE \mid d \in devices \wedge failed(d) = True \bullet \\
\qquad\qquad device\_warning(d) \} \cup \\
\qquad \{ d : DEVICE \mid d \in devices \wedge failed(d) = False \bullet \\
\qquad\qquad ok(d) \} \\
\hline
\end{array}
$$

3.4 Viewpoint 3: Linking Patients and Devices

In the third viewpoint we consider the links between patients and devices.

We have a set of patients, a set of devices and a set of values. The values represent the status information about the patients.

$$[PATIENT, DEVICE, VALUE]$$

We need to determine the associated value for each patient, and this is modelled by a mapping of patients to values.

$$
\begin{array}{|l}
\hline
\textit{Patients} \\
pvalue : PATIENT \pfun VALUE \\
\hline
\end{array}
$$

For each device we need to determine whether it has failed, and if not, the value it is giving.

$$
\begin{array}{|l}
\hline
\textit{Devices} \\
dvalue : DEVICE \pfun VALUE \\
failed : DEVICE \pfun Boolean \\
\hline
(\forall d : DEVICE \bullet \neg failed(d) \Rightarrow d \in \mathrm{dom}\ dvalue) \\
\hline
\end{array}
$$

Links between patients and devices are described by a partial injection. This adds the constraint that only one device is linked to each patient.

$$
\begin{array}{|l}
\hline
\textit{Links} \\
Patients \\
Devices \\
link : DEVICE \pinj PATIENT \\
\hline
\mathrm{ran}\ link \subseteq \mathrm{dom}\ pvalue \\
\hline
\end{array}
$$

To attach a device to a patient we amend the links. The patient cannot already have a device attached ($patient? \notin \mathrm{ran}\ link$) and the device cannot already be attached to a patient ($device? \notin \mathrm{dom}\ link$). In addition, a failed device cannot be attached to a patient.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AttachDevice \quad}} \\
\Delta Links \\
Devices \\
\\
patient? : PATIENT \\
device? : DEVICE \\
\hline
patient? \notin \operatorname{ran} link \\
device? \notin \operatorname{dom} link \\
failed(device?) = False \\
\\
link' = link \cup \{device? \mapsto patient?\} \\
\hline
\end{array}
$$

The monitor examines the links between patients and devices. A device's value is the status value of the patient it is linked to, unless the device has failed.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Monitor \quad}} \\
\Xi Patients \\
\Xi Devices \\
\Xi Links \\
\hline
(\forall d : DEVICE \mid d \in \operatorname{dom} link \bullet \\
\qquad dvalue(d) = pvalue(link(d)) \lor \\
\qquad failed(d) = True) \\
\hline
\end{array}
$$

3.5 Amalgamation of Viewpoints 1 and 2

Viewpoints 1 and 2 are amalgamated to produce a monitoring system which outputs a single set of reports on patients and the devices attached to them.

When discussing elements from the viewpoints in the text, we will attach a subscript to the names that identifies the corresponding viewpoint. Elements in the amalgamation of viewpoints i and j will be subscripted with (i, j).

3.5.1 Data Co-Refinement

We begin by identifying clashes and rewriting parts of the viewpoints so as to resolve them.

Clash 1: Free Type $REPORT_1$ & Free Type $REPORT_2$.

This clash will be resolved by merging the two free types during the coalescence phase.

Clash 2: Branch ok_1 of $REPORT_1$ & Branch ok_2 of $REPORT_2$.

This clash only occurs because the clash between $REPORT_1$ and $REPORT_2$ is resolved by merging. We resolve it by renaming ok_2 as $working_2$.

This will require rewriting the predicate in $Monitor_2$ to use the new name introduced to resolve the previous clash:

$$\begin{aligned} d\_reports! = \{&d : DEVICE \mid d \in devices \wedge failed(d) = True \bullet \\ &\qquad device\_warning(d)\} \cup \\ \{&d : DEVICE \mid d \in devices \wedge failed(d) = False \bullet \\ &\qquad working(d)\} \end{aligned}$$

3.5.2 Operation Co-Refinement

Clash 3: Schema $Monitor_1$ & Schema $Monitor_2$.

The next clash is between $Monitor_1$ and (the rewritten) $Monitor_2$. This clash will be resolved at the coalescence phase by merging the two schemas.

Clash 4: Component $p\_reports!_1$ in $Monitor_1$ & Component $d\_reports!_2$ in $Monitor_2$.

A *redundancy* is noted between the components $p\_reports!_1$ and $d\_reports!_2$, which we want to be combined into a single report in the amalgamated specification. To do this we will rename both $p\_reports!_1$ and $d\_reports!_2$ to $reports!$.

3.5.3 Coalescence

The coalescence phase collects the various components resulting from the previous phases. In this case, the prepared components mentioned above are combined with the components of the initial viewpoints that were not altered.

The given sets from the two viewpoints are collected:

$$[PATIENT, VALUE, DEVICE]$$

The free types $REPORT_1$ and $REPORT_2$ are merged to form a new free type as follows.

$$\begin{aligned} REPORT ::= &\ ok\langle\langle PATIENT\rangle\rangle \\ \mid &\ patient\_warning\langle\langle PATIENT\rangle\rangle \\ \mid &\ working\langle\langle DEVICE\rangle\rangle \\ \mid &\ device\_warning\langle\langle DEVICE\rangle\rangle \end{aligned}$$

The schemas $Monitor_1$ and $Monitor_2$ are merged by combining their declaration and predicate parts to give $Monitor_{(1,2)}$ as follows.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Monitor \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Xi Patients \\
\Xi Safety \\
\Xi Devices \\
reports! : \mathbb{P}\ REPORT \\
\hline
reports! = \{p : PATIENT \mid p \in inpatients \land safe(pvalue(p)) = True \bullet \\
\qquad\qquad\qquad\qquad ok(p)\} \cup \\
\qquad\qquad \{p : PATIENT \mid p \in inpatients \land safe(pvalue(p)) = False \bullet \\
\qquad\qquad\qquad\qquad patient\_warning(p)\} \cup \\
\qquad\qquad \{d : DEVICE \mid d \in devices \land failed(d) = True \bullet \\
\qquad\qquad\qquad\qquad device\_warning(d)\} \cup \\
\qquad\qquad \{d : DEVICE \mid d \in devices \land failed(d) = False \bullet \\
\qquad\qquad\qquad\qquad working(d)\} \\
\hline
\end{array}
$$

Note that some of the amalgamations required can be performed using the Z schema calculus, but others cannot. Renaming component names, as in *Monitor*, can be done using the schema calculus; renaming branches in free types and combining free types, as in *REPORT*, cannot.

3.5.4 Verifying the amalgamation

To show that the amalgamated specification properly combines the information represented by both the initial viewpoints, we need to show that the amalgamation of views 1 and 2 is a data refinement of each of these viewpoints. We do this by showing that each schema in the amalgamation is a data refinement of the corresponding schema(s) in the initial viewpoints.

In this case, all of the schemas in the amalgamation, except *Monitor*, are identical to a single schema in one of the initial viewpoints. They are trivial data refinements of both viewpoints, with the identity retrieve relation for the viewpoint containing that schema, and the empty retrieve relation for the other.

In the case of *Monitor* we can define two functional retrieve relations:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Retr1 \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
Monitor_1 \\
Monitor_{(1,2)} \\
\hline
p\_reports!_1 = \{r : REPORT_{(1,2)} \mid \\
\qquad\qquad\qquad r \in reports!_{(1,2)} \land \\
\qquad\qquad\qquad (\exists\, p : PATIENT \bullet r = ok(p) \lor r = patient\_warning(p)) \bullet \\
\qquad\qquad r\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Retr2 \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
Monitor_2 \\
Monitor_{(1,2)} \\
\hline
d\_reports!_2 = \{d : DEVICE \mid device\_warning(d) \in reports!_{(1,2)} \bullet \\
\qquad\qquad\qquad\qquad\qquad device\_warning(d)\} \cup \\
\qquad\qquad \{d : DEVICE \mid working(d) \in reports!_{(1,2)} \bullet \\
\qquad\qquad\qquad\qquad\qquad ok(d)\} \\
\hline
\end{array}
$$

We could then use these retrieve relations to show formally that $Monitor_{(1,2)}$ is a refinement of $Monitor_1$ by proving that:

$$\text{pre } Monitor_1 \wedge Retr1 \Rightarrow \text{pre } Monitor_{(1,2)}$$

$$\text{pre } Monitor_1 \wedge Retr1 \wedge Monitor_{(1,2)} \Rightarrow \\ \exists\, Patients_1', Safety_1', p\_reports!_1 \bullet Monitor_1 \wedge Retr1'$$

since *Patients* and *Safety* are unchanged by *Monitor*, this second conditions simplifies to

$$\text{pre } Monitor_1 \wedge Retr1 \wedge Monitor_{(1,2)} \Rightarrow \exists\, p\_reports!_1 \bullet Monitor_1 \wedge Retr1'$$

(Similar conditions can be derived with respect to $Monitor_2$). When the retrieve relations are functional, this type of proof is trivial.

3.6 Amalgamation of Viewpoints (1,2) and 3

Viewpoints (1,2) and 3 are amalgamated to produce a monitoring system which uses the values of the devices to determine whether a patient is in danger or not. A single set of reports on the patients and devices is produced.

3.6.1 Data Co-Refinement

Clashes 1,2 & 3: Given Types.

In the first three cases the two given types represent the same idea, so the clashes can be resolved by merging at the coalescence phase.

Clash 1: Given Type $PATIENT_{(1,2)}$ & Given Type $PATIENT_3$.
Clash 2: Given Type $DEVICE_{(1,2)}$ & Given Type $DEVICE_3$.
Clash 3: Given Type $VALUE_{(1,2)}$ & Given Type $VALUE_3$.

Clash 4: Schema $Patients_{(1,2)}$ & Schema $Patients_3$.

$Patients_{(1,2)}$ and $Patients_3$ are already coherent, and can be easily amalgamated during the coalescence phase.

Clash 5: Schema $Devices_{(1,2)}$ & Schema $Devices_3$.

Similarly with the clash between $Devices_{(1,2)}$ and $Devices_3$. A new *Devices* schema can be created using schema conjunction.

3.6.2 Operation Co-Refinement

Clash 6: Schema $Monitor_{(1,2)}$ and Schema $Monitor_3$.

When amalgamating the underlying models, an adjustment to $Monitor_{(1,2)}$ must be made. The values for patients under the new model are accessed using the link to devices, and hence the schema *Links* must be included and the predicate adjusted. The function *dvalue* is used in place of *pvalue*, and the patient information for *ok* and *patient_warning* is accessed using the link from the device to the patient.

The modified part of the predicate is as follows:

$$
\begin{aligned}
reports! = \{ & d : DEVICE \mid d \in \operatorname{dom} link \land safe(dvalue(d)) = True \bullet \\
& \qquad ok(link(d))\} \cup \\
\{ & d : DEVICE \mid d \in \operatorname{dom} link \land safe(dvalue(d)) = False \bullet \\
& \qquad patient\_warning(link(d))\} \cup \\
& \ldots
\end{aligned}
$$

3.6.3 Coalescence

The coalescence phase again collects all the resulting components. The only elements that are not taken directly from one of the parent viewpoints are *Devices* and *Monitor*, which are as follows:

$$
\begin{array}{|l}
\hline
\textit{Devices} \\
devices : \mathbb{P}\ DEVICE \\
\\
failed : DEVICE \rightarrow\!\!\!\!\!\!+\; Boolean \\
dvalue : DEVICE \rightarrow\!\!\!\!\!\!+\; VALUE \\
\hline
\operatorname{dom} failed = devices \\
(\forall d : DEVICE \bullet \neg failed(d) \Rightarrow d \in \operatorname{dom} dvalue) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Monitor} \\
\Xi Patients \\
\Xi Safety \\
\Xi Devices \\
\Xi Links \\
\\
reports! : \mathbb{P}\ REPORT \\
\hline
(\forall d : DEVICE \mid d \in \operatorname{dom} link \bullet \\
\quad dvalue(d) = pvalue(link(d)) \lor \\
\quad failed(d) = True) \\
\\
reports! = \{d : DEVICE \mid d \in \operatorname{dom} link \land safe(dvalue(d)) = True \bullet \\
\qquad\qquad ok(link(d))\} \cup \\
\qquad \{d : DEVICE \mid d \in \operatorname{dom} link \land safe(dvalue(d)) = False \bullet \\
\qquad\qquad patient\_warning(link(d))\} \cup \\
\qquad \{d : DEVICE \mid d \in devices \land failed(d) = True \bullet \\
\qquad\qquad device\_warning(d)\} \cup \\
\qquad \{d : DEVICE \mid d \in devices \land failed(d) = False \bullet \\
\qquad\qquad working(d)\} \\
\hline
\end{array}
$$

Note that the addition of $Links_3$ to $Monitor_{(1,2)}$ can be described using the schema calculus, but the change in the predicate cannot.

3.6.4 Verifying the amalgamation

Again, most of the schemas in the amalgamation are identical to a single schema in one of the initial viewpoints, so they are trivial data refinements.

The relationship between $Devices_{(1,2)}$, $Devices_3$ and $Devices_{((1,2),3)}$ is defined by the following retrieve relation:

$$
\begin{array}{|l}
\hline
\textit{Retr\_Device} \\
\textit{Devices}_{(1,2)} \\
\textit{Devices}_3 \\
\textit{Devices}_{((1,2),3)} \\
\hline
\textit{devices}_{((1,2),3)} = \textit{devices}_{(1,2)} \\
\textit{dvalue}_{((1,2),3)} = \textit{dvalue}_3 \\
\textit{failed}_{((1,2),3)} = \textit{failed}_{(1,2)} = \textit{failed}_3 \\
\hline
\end{array}
$$

Again, this can be turned into a pair of retrieve relations that are functional from $Devices_{((1,2),3)}$ to $Devices_{(1,2)}$ and $Devices_3$, respectively.

The relationship between $Monitor_{(1,2)}$, $Monitor_3$ and $Monitor_{((1,2),3)}$ can also be defined using a retrieve relation. In this case, we have a partial data refinement, since the final monitor can produce either $ok(p)$ or $patient\_warning(p)$ if the device attached to patient p has failed. The required relation is similar to that in the previous amalgamation, but slightly more complex.

4 Conclusions

The starting-point for this work was the idea of making scale problems more tractable by using viewpoint amalgamation. This allows the construction of complicated formal descriptions by combining a number of simpler descriptions of lower perceived complexity, which appears to mirror one way people habitually organise their own understanding of complicated systems.

We began by showing how viewpoint amalgamation can be defined formally, particularly with reference to refinement. These ideas are currently being investigated further, and should eventually lead both to a deeper understanding of the mathematical properties of amalgamation, and to techniques for proving the correctness of amalgamations. By writing information such as the abstraction invariants into the amalgamation trail, we can produce not only a development history, but also a list of proof obligations for the amalgamation. We are currently investigating how these techniques may be generalised to a situation where viewpoints are written using widely different formalisms.

We then gave a detailed example of viewpoint specification in Z. While viewpoint techniques have a natural ability to reduce the perceived complexity of large systems, this will not help with industrial-scale problems unless the amalgamation process can be mechanised. Work is currently in progress to determine how much of the amalgamation process can be mechanised, and to develop amalgamation tools.

We are currently exploring details of amalgamation processes for a variety of specification techniques and how these amalgamations may be expressed and verified formally. This will improve our general understanding of viewpoint technology and allow a full evaluation of this overall approach to combating the insidious effects of scale problems in the lucid formal description of complex systems.

5 Acknowledgements

The assistance of the referees is gratefully acknowledged. Mike Ainsworth and Tony Cruickshank are supported by UK SERC Research Studentships.

References

[ACGW93] M Ainsworth, AH Cruickshank, LJ Groves and PJL Wallis. Formal specification via viewpoints. In J Hosking (ed) *Proc. 13th New Zealand Computer Conference*, Auckland, New Zealand (18–20 August 1993), New Zealand Computer Society, pp 218–237.

[AW94] M Ainsworth and PJL Wallis. Co-refinement. *Proc. 6th Refinement Workshop.* City University, London (5–7 January 1994) (To appear).

[BLN86] C Batini, M Lenzerini, and SB Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[BW88] MF Bott and PJL Wallis. Ada and software reuse. *Software Engineering Journal*, 3(5):188–183, September 1988.

[Fea89] MS Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198–208, 1989.

[FF89] A Finkelstein and H Fuks. Multi-party specification. In *Proc. 5th Int. Workshop on Software Specification and Design*, pages 185–195, Los Alamitos, CA, 1989. IEEE Computer Soc. Press.

[FKN+92] A Finkelstein, J Kramer, B Nuseibeh, L Finkelstein, and M Geodicke. Viewpoints: A framework for integrating multiple perspectives in system development. *Int. J. on Software Engineering and Knowledge Engineering* 2(1):31–57, 1992.

[HPR89] S Horwitz, J Prins, and T Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[Jon90] CB Jones. *Systematic Software Development using VDM.* International Series in Computer Science. Prentice-Hall, second edition, 1990.

[Jos88] MB Josephs. The data refinement caclulator for Z. *Information Processing Letters* 27:29–33, 1988.

[Lei89] JCSP Leite. Viewpoint analysis: a case study. In *Proc. 5th Int. Workshop on Software Specification and Design*, pages 111–119, Los Alamitos, CA, 1989. IEEE Computer Soc. Press.

[MG90] C Morgan and PHB Gardiner. Data refinement by calculation. *Acta Informatica* 27(6):481–503, 1990.

[PST91] B Potter, J Sinclair, and D Till. *An Introduction to Formal Specification and Z.* International Series in Computer Science. Prentice-Hall, 1991.

[Rep89] T Reps. Demonstration of a prototype tool for program integration. Technical Report 819, Department of Computer Sciences, University of Wisconsin, Madison, WI, 1989.

[Rep91] T Reps. Algebraic properties of program slicing. *Science of Computer Programming*, 17:139–215, 1991.

[Rob89] WN Robinson. Integrating multiple specifications using domain goals. In *Proc. 5th Int. Workshop on Software Specification and Design*, pages 219–225, Los Alamitos, CA, 1989. IEEE Computer Soc. Press.

[Spi89] M Spivey. An introduction to z and formal specifications. *Software Engineering Journal*, 4(1):40–50, 1989.

[Wal92] PJL Wallis. A new approach to modular formal description. Technical Report 92-57, University of Bath, 1992.

[Wal93] PJL Wallis. Modular formal description, 1993. (Submitted)

[Woo89] JCP Woodcock. Structuring specifications in Z. *Software Engineering Journal*, 4(1):51–66, 1989.

[Woo92] JCP Woodcock. *Using Z – Specification, Refinement and Proof.* 1992. (In preparation).