CHAPTER 11

STEPWISE TRANSFORMATION OF SOFTWARE ARCHITECTURES

This chapter provides a further example of the use of a formal specification as the basis of architecture and design. The problem chosen is that of a file-handler and is thus close to the topic of database systems. The stepwise development given here starts with a simple specification; the development considers a realization on disk storage. The techniques (e.g. data type invariants, object transformation) of chapter 10 are illustrated. One new aspect of VDM which is illuminated here is the use of a formal specification in evolving the architecture of a system.

Thus the techniques of object and operation transformation are employed as a means of conquering architectual specification complexities, and the techniques of data structure invariance and object abstraction functions are employed as a means of studying the evolving architecture, as well as a means for proving correctness of transformation. The transformation stages can as well be seen as exemplifying realization stages of development, such as proven correct in chapter 10.

(The present chapter is based on [Bjørner 81a].)

CONTENTS

11.0 Introduction.....355
11.1 Top-Level (File Handler) Architecture.....(Stage 0).....357
11.2 Second-Level (File Handler) Architecture.....359
11.2.1 File Catalogue and Page Directories....(Stage 1).....359
11.2.2 Disks.....(Stage 2).....364
11.2.3 Storage and Disk.....(Stage 3).....365
11.3 Third-Level (File Handler) Architecture.....368
11.3.1 Storage and Disk File Sub-Systems.....(Stage 4).....369
11.3.2 "Flat" Storage and Disk.....(Stage 5).....374
11.3.3 Storage and Disk Space Management.....(Stage 6).....374
11.4 Summary.....376
11.5 Bibliography.....377

11.0 INTRODUCTION

Background

Sometimes software systems contain unnecessarily many, seemingly independent concepts. Occasionally a large number of such concepts are, however, necessary. Their presence being required in order to cope with a variety of "more-or-less" related concepts. (We think, here, of such things as functionality, efficiency, reliability, adaptability and extensibility, etc..) In all cases it is rather hard to grasp all the concepts, sort them out and interrelate them properly. In many cases this ability to dissect a software architecture into its many constituent notions is seriously hampered by opaque presentations of their interdependencies.

Proposed Remedies

Three possibilities for "solving" the apparent problem exist. Two extremes and a "middle road". Either, not to design such multi-concept systems at all; or, go on designing them in the old "hacker" fashion. Sometimes we shall choose the first extreme, sometimes the "middleroad" approach outlined below, but never the second "compromise".

Stepwise Development

By stepwise development of a software architecture we shall understand the following: first a model is established which exhibits, "as abstractly as deemed reasonable", the intrinsic concepts and facilities for which the software was intended in the first place. Then this model is subjected to object and/or operation transformations. We shall only illustrate object transformations. A sequence of such may be needed. Each step introduces further properties and/or details; none, some or all of which are exploited in exposing them to an external world. The order of the steps and their nature is dictated for example by technological and/or product-strategic considerations.

Overview of Example

Our example is that of a file handler system.

1. At the top level of architecture we focus our attention on files, file names, pages and page names as objects; and the creation and erasure of files, and writing, updating, reading, and deletion of pages. At this level files are named and consist of named pages.

At the top level no concession is made to the possible storing of files and their pages in such diverse storage media as foreground (fast access, "core"), or background (slow access, "disk") storage. The decision, which is hence recorded, of eventually implementing the storing of files and pages on disk-like devices, then predicates a need to be able to "look-up", reasonably fast, where, on possibly several disks, files and pages are stored.

2'. At the second level we therefore introduce first the notions of catalogues and directories, subsequently, as a further step of development, abstractions of the notions of main storage and disks.

Catalogues eventually record disk addresses of file directories, one per file. Directories eventually record disk addresses of pages. Our file system, at this level has one catalogue. We think, at the level of main storage and disks, of the one catalogue as always residing in main storage, whereas all directories are normally only stored on disks. To speed up access to disk pages we operate on main storage copies of directories. The intention to operate on a file is then indicated by its opening, an "act" which brings a disk directory copy into main storage. The intention to not operate further on a file is then indicated by its closing, an "act" which reverts the above copying.

2". Hence open and close operations are introduced.

They are file-related concepts primarily brought upon us by efficiency considerations. These efficiency concerns are rooted in insufficient technologies.

Neither at the top, nor at the second level of file handler architecting did we bother about the issue of reliability. We here define the reliability of our file handler as its ability to survive crashes. By a "crash" we restrictively mean anything which renders main storage information (catalogue and opened directories) useless. By total "survival" we mean the ability to continue (some time) after a "crash" as if no "crash" had occurred. (By "partial survival" we mean the ability to con-

tinue with at least a nonvoid sub-set of the files after a "crash" -- with the complement set of files being clearly identified.)

3. At the third level, building upon redundancies in catalogue-, directory and page recordings, we therefore introduce notions of checkpointing files and automatic recovery from "crashes".

11.1 TOP-LEVEL (File Handler) ARCHITECTURE (Stage 0)

Semantic Domains

Based on the immediately following English wording of what the 'state' Domain of our top-level file handler is we "derive" informally the formal Domain definitions.

The sole data structure of our file handler consists of a set of uniquely named files. Each file consists of a set of uniquely named pages. Let *Fn*, *Pn* and *PG* denote the further unspecified Domains of respectively file names, page names, pages. Then:

1. $FS = Fn \overset{m}{\rightarrow} FILE$
2. $FILE = Pn \overset{m}{\rightarrow} PG$

We have completed our first task: that of specifying the most important aspects first, namely the semantic Domains. No specific invariants need be expressed, that is:

3. $inv\text{-}FS(fs) \triangleq true$

Syntactic Domains

Five kinds of operations are applicable to our top-level file handler data: creation of new and erasure of old files, the write/update of new/old-, and the reading and deletion of existing, pages. Each of these operations can be thought of as being denoted by corresponding commands (or interface calls):

4. $Cmd = Crea \mid Eras \mid Put \mid Get \mid Del$

To create an initially empty file (of no pages) we need specify a new,

hitherto unused file name. To erase an existing file we need specify the name of a file already in the system. To put a page into a file we need specify the names of the file and page, and the page itself. To get a page from a file we need specify the names of the file and page. Finally to delete a page we need specify the same:

5. *Crea :: Fn*
6. *Eras :: Fn*
7. *Put :: Fn Pn PG*
8. *Get :: Fn Pn*
9. *Del :: Fn Pn*

No well-formedness constraints need be specified for commands, that is: *is-wf-Cmd[e]* $\triangleq$ *true*.

Semantic Functions

We believe the above informal syntactic outline plus the following formal semantics to be sufficiently self-explanatory not to warrant the kind of detailed annotations that we would otherwise provide:

```
10.   type Elab-Cmd: Cmd → (FS ~→ (FS | PG))
10.0  Elab-Cmd[c](fs)≜
  .1    cases c:
  .2      (mk-Crea(f)       → fs ∪ [f ↦ []],
  .3       mk-Eras(f)       → fs \{f},
  .4       mk-Put(f,p,pg)   → fs + [f ↦ fs(f) + [p ↦ pg]],
  .5       mk-Get(f,p)      → (fs(f))(p),
  .6       mk-Del(f,p)      → fs + [f ↦ (fs(f))\{p}])
10.7  pre: pre-Elab-Cmd[c](fs)

11.0  pre-Elab-Cmd[c](fs)≜
  .1    cases c:
  .2      (mk-Crea(f)       → f ¬∈ dom fs,
  .3       mk-Eras(f)       → f ∈ dom fs,
  .4       mk-Put(f,p,pg)   → f ∈ dom fs,
  .5       mk-Get(f,p)      → ((f ∈ dom fs) ∧ (p ∈ dom(fs(f)))),
  .6       mk-Del(f,p)      → ((f ∈ dom fs) ∧ (p ∈ dom(fs(f)))))
```

The type of *pre-Elab-Cmd* follows from the type of *Elab-Cmd*:

11. *type: pre-Elab-Cmd: Cmd → (FS → BOOL)*

Conclusion of Top-Level Definition

We have completely specified the basic, major functions of a simple file handler system. The abstraction is just that: we have abstracted from any concern with how actual input of commands, including input of pages, and of how output of pages take place. We have also abstracted "away" considerations of what kind of diagnostics to use in case of erroneous input -- we have only defined, in *pre-Int-Cmd* what we mean by erroneous input. We have abstracted from any representation of files, and, in fact, the entire file system. Finally, we have not been, and shall not, in this entire example, be concerned with what pages are.

11.2 SECOND-LEVEL (File Handler) ARCHITECTURE (Stages 1-2-3)

We divide this, the second level specification, into three stages. First we introduce the object notions of catalogue and directories, then the notion of disk, and finally the object notions of storage and disk. The single aim of this level is to introduce the operation notions of open and close.

11.2.1 File Catalogue and Page Directories (Stage 1)

Semantic Domains

We subscript now our Domain names according to the number of stage of development. The 0'th stage (which was the top-level) gave us:

1-2. $FS_0 = Fn \xrightarrow{m} (Pn \xrightarrow{m} PG)$

To each file we now associate a page directory. Each directory records where pages are stored. Directories are named, and these names are recorded in a catalogue.

12. $FS_1 :: CTLG_1 \; DIRS_1 \; PGS_1$
13. $CTLG = Fn \xrightarrow{m} Dn$
14. $DIRS_1 = Dn \xrightarrow{m} DIR_1$
15. $PGS_1 = Pa \xrightarrow{m} PG$
16. $DIR_1 = Pn \xrightarrow{m} Pa$

You may (justifiably) think of directories "translating" user-oriented page names into system-oriented page addresses, and PGS_1 to be a disk-like space within which all pages of all files are allocated. Let:

$$[f_1 \mapsto [p_{11} \mapsto g_{11}, p_{12} \mapsto g_{12}],\ f_2 \mapsto [p_{21} \mapsto g_{21}],\ f_3 \mapsto []]$$

be an abstract, FS_0, file system. Its counterpart in FS_1 is for example:

$$mk\text{-}FS_1([f_1 \mapsto d_1, f_2 \mapsto d_2, f_3 \mapsto d_3], [d_1 \mapsto [p_{11} \mapsto a_{11}, p_{12} \mapsto a_{12}], d_2 \mapsto [p_{21} \mapsto a_{21}], d_3 \mapsto []], [a_{11} \mapsto g_{11}, a_{12} \mapsto g_{12}, a_{21} \mapsto g_{21}])$$

Domain Invariant

Domain definitions (12.-16.) define too much. Not all combinations of catalogues, directories and pages go together. We must require (17.1) that there is a distinct directory in $DIRS_1$ for each file catalogued in $CTLG_1$; that (17.2) pages addressed in PGS_1 are actually recorded in directories; and that (17.3) every page, understood as page-address, is described in exactly one directory (that is belongs to exactly one file):

17.0 $inv\text{-}FS_1(mk\text{-}FS_1(c,ds,ps)) \triangleq$

.1 $(\underline{rng}\ c = \underline{dom}\ ds)$

.2 $\wedge(\underline{union}\{\underline{rng}\ d \mid d \in \underline{rng}\ ds\} = \underline{dom}\ ps)$

.3 $\wedge(\forall a \in \underline{dom}\ ps)(\exists! d \in \underline{dom}\ ds)(a \in \underline{rng}\ ds(d))$

The bijections of (13.), (14.) and (16.) already express that no two files have the same directory, that no two directories are identical, and that no two page names map to the same ("physical") page (-- by its address).

Abstraction: from FS_1 to FS_0

Given an FS_1 file system we can abstract a "corresponding" FS_0 from it. Abstraction (here called "retrieval") is a function:

18. *type: retr-FS$_0$: FS$_1$ → FS$_0$*

18.0 *retr-FS$_0$(mk-FS$_1$(c,ds,ps))*$\triangleq$

.1 [$f \mapsto$ [$p \mapsto ps((ds(c(f)))(p))$ | $p \in dom\ ds(c(f))$] | $f \in dom\ c$]

We can only retrieve well-formed file systems:

18.2 *pre: inv-FS$_1$(mk-FS$_1$(c,ds,ps))*

We always assume our functions to apply only to well-formed objects, that is (18.2) is superfluous.

Injection: from *FS$_0$* to *FS$_1$*

Given an abstract system we can inject it into any number of "corresponding systems". Injection is a relation.

Adequacy

We must ensure that to each abstract system there corresponds a non-trivial concrete which abstracts to it:

19. $(\forall s_0 \in FS_0)(\exists s_1 \in FS_1)(inv\text{-}FS_1(s_1) \supset (s_0 = retr\text{-}FS_0(s_1)))$

Syntactic Domains

No change to existing Domains, that is no refinements or transformations are necessary; and no new commands, that is no extension of the Command Domain.

Semantic Functions

We rewrite the *Elab-Cmd$_0$* of section 11.1. That is: for fixed syntactic Domains but changed semantic Domains we need re-express the semantics, now in terms of the new semantic Domains:

20. *type: Elab-Cmd$_1$: Cmd → (FS$_1$ ⇸ (FS$_1$ ⇸ (FS$_1$ | PG)))*

Instead of expressing *Int-Cmd$_1$* as one "monolithic" function we express it in terms of 5 sub-functions, one for each command category:

```
21.  type: Int-Crea1:  Crea → (FS1 ⇸ FS1)
22.  type: Int-Eras1:  Eras → (FS1 ⇸ FS1)
23.  type: Int-Put1:   Put  → (FS1 ⇸ FS1)
24.  type: Val-Get1:   Get  → (FS2 ⇸ PG)
25.  type: Int-Del1:   Del  → (FS2 ⇸ FS1)
```

We choose to "merge" the pre-condition checking into each function:

```
21.0 Int-Crea1[mk-Crea(f)](mk-FS2(c,ds,ps))≜
  .1    if f∈dom c
  .2       then undefined
  .3       else (let d∈Dn - dom ds in
  .4             mk-FS1(c∪[f↦d],ds∪[d↦[]],ps))

22.  Int-Eras1[mk-Eras(f)](mk-FS2(c,ds,ps))≜
  .1    if f∈dom c
  .2       then mk-FS1(c\{f},ds\{c(f)},ps\rng(ds(c(f))))
  .3       else undefined

23.0 Int-Put1[mk-Put(f,p,pg)](mk-FS2(c,ds,ps))≜
  .1    if f∈dom c
  .2       then if p∈dom(ds(c(f)))
  .3               then mk-FS1(c,ds,ps+ [(ds(c(f)))(p)↦pg])
  .4               else (let a ∈ Pa - dom ps in
  .5                     let ds' = ds+[c(f)↦ds(c(f))∪p↦a],
  .6                         ps' = ps∪ [a↦pg] in
  .7                     mk-FS1(c,ds',ps'))
  .8    else undefined

24.0 Val-Get1[mk-Get(f,p)](mk-FS1(c,ds,ps))≜
  .1    if (f ∈ dom c)∧(p ∈ dom(ds(c(f))))
  .2       then  ps(ds(c(f)))(p))
  .3       else undefined

25.0 Int-Del[mk-Del(f,p)](mk-FS2(c,ds,ps))≜
  .1   if (f ∈ dom c) ∧ (p ∈ dom (ds(c(f))))
  .2      then mk-FS1(c,ds+[c(f)↦(ds(c(f)))\{p}],
  .3                  ps\{(ds(c(f)))(p)})
  .4      else undefined
```

And finally:

```
20.0  Elab-Cmd1[c](fs1) ≙
  .1    (is-Create) → Int-Crea[c](fs1),
  .2     is-Eras(c) → Int-Eras1[c](fs1),
  .3     is-Put(c)  → Int-Put1[c](fs1),
  .4     is-Get(c)  → Val-Get1[c](fs1),
  .5     is-Del(c)  → Int-Del[c](fs1))
```

Correctness

Correctness of the above realization of $Elab\text{-}Cmd_0$ in terms of $Elab\text{-}Cmd$ with respect to the realization of FS_0 in terms of FS_1 is expressed by means of the abstraction function:

```
26.    type: retr-RES1: (FS1 | PG) → (FS0 | PG)
26.0   retr-RES1(r) ≙
  .1     (is-FS1(r) → retr-FS0(r), T→r)
```

and is:

```
27.0   thm1(∀c∈ Cmd)
  .1          (∀fs0∈FS0)
  .2            (∀fs1 ∈ FS1)
  .3            (((inv-FS1(fs1)∧ retr-FS0(fs1)=fs0)
  .4             ∧ pre-Elab-Cmd[c](fs0))
  .5             ⊃
  .6          (retr-RES1(Elab-Cmd2[c](fs2)) = Elab-Cmd1[c](fs0)))
```

We do not prove that the above theorem holds for our first stage realization.

Automatic Operation Transformation

In fact, we claim, without demonstrating it in this book, that given the following "input": FS_i, $inv\text{-}FS_i$, $Elab\text{-}Cmd_i$, FS_{i+1} $inv\text{-}FS_{i+1}$, $retr\text{-}FS_i$, and thm_i (see above), one can devise automatic means for transforming semantic functions $Elab\text{-}Cmd_i$ into $Elab\text{-}Cmd_{i+1}$. Since the transformed result has to satisfy rather "narrow" constraints, there are not very many choices of implementations left free.

11.2.2 Disks (Stage 2)

Semantic Domains

We are given:

12. $FS_1 :: CTLG_1 \; DIRS_1 \; PGS_1$
13. $CTLG_1 = Fn \xrightarrow{m} Dn$
14. $DIRS_1 = Dn \xrightarrow{m} DIR_1$
15. $PGS_1 = Pa \xrightarrow{m} PG$
16. $DIR_1 = Pn \xrightarrow{m} Pa$

The object transformation of this stage involves the "gathering" of directories and pages, that is of the above $DIRS_1$ and PGS_1 components of FS_1 into one component, called DSK_2 of FS_2. $DIRS_1$ and PGS_1 are modelled as maps, and DSK_2 will hence be a "merged" Domain of simiilar maps. Where before map domains were directory names, respectively page addresses, the "merged" (or "gathered") Domain will only have addresses in its map domain. We think of DSK_2 as modelling "actual" disks:

28. $FS_2 :: CTLG_2 \; DSK_2$
29. $CTLG_2 = Fn \xrightarrow{m} Adr$
30. $DSK_2 = Adr \xrightarrow{m} (DIR_2 \mid PG)$
31. $DIR_2 = Pn \xrightarrow{m} Adr$

Here addresses Adr (like file names, Fn, and page names, Pn, and pages, PG) are further undefined.

Domain Invariant

Again the Domain definitions (28.-31.) define too much. In addition to the invariants ["carried over" from the very similar definitions of FS_1], we must (first) make sure that directory addresses (listed in the catalogue) really denote directories on the disk, respectively that page addresses listed in directories really denote pages on the disk. Once this is established we can retrieve an FS_1 object from such a "tentatively well-formed" FS_2 object, and this abstracted object must satisfy the earlier stated constraint:

32. $inv\text{-}FS_2(fs_2) \triangleq (wf\text{-}AdrDen(fs_2) \wedge inv\text{-}FS_1(retr\text{-}FS_1(fs_2)))$

33. *type: wf-AdrDen: FS_2 → BOOL*
33.0 *wf-AdrDen(mk-FS_2(2,d))* ≜
.1 *(∀a∈rng c)(is-DIR_2(d(a)) ∧ (∀a' ∈ rng(d(a)))(is-PG(d(a')))*

Abstraction from FS_2 to FS_1

34.0 *retr-FS_1(mk-FS_2(c,d))* ≜
.1 *mk-FS_1(c, [a ↦ d(a) | a ∈ rng c] ,d \ rng c)*

Here we now have assumed *Adr = Dn | Pa*, that is that:

33.0 *wf-AdrDen(mk-FS_2(c,d))* ≜
.1 *(∀a ∈ rng c)*
.2 *((is-Dn(a) ∧ is-DIR_2(d(a))).*
.3 *∧ (∀a' ∈ rng(d(a)))*
.4 *(is-Pa(a') ∧ is-PG(d(a'))))*

and:

34.2 *pre-retr-FS_1(fs_2)* ≜ *wf-AdrDen(fs_2)*

We leave to the reader the rather straightforward transcription of adequacy, semantic functions and correctness theorem.

11.2.3 Storage and Disk (Stage 3)

We are given (28.-31.). We now face the reality of storages and disks. By a storage we shall understand a memory medium access to whose information is orders of magnitude faster than to information on what we shall then call disks! As was evident from e.g. lines 23.3 and 24.2 access to pages (on disk) goes via catalogue and directories, the latter also on disk. Thus two disk accesses per page access. [In this discussion we think of the catalogue as residing in storage.] To cut down on disk accesses we therefore decide to copy into storage the directories of those files whose pages we wish to access. In the resulting model all pages will still be thought of as stored only on the disk.

Syntactic Domains

In order to advise the system of an intent to begin and end operations on pages we introduce two new commands: open and close:

35. $Cmd_3 = Cmd_0 \mid Open \mid Close$
36. $Open :: Fn$
37. $Close :: Fn$

(The informal semantics of those are basically to bring a directory copy into storage, respectively to over-write the disk copy with the storage copy, subsequently deleting this latter copy.)

Semantic Domains

Now both storage and disk have directories:

38. $FS_3 :: STG_3 \; DSK_3$
39. $STG_3 :: CTLG_2 \; (Fn \; \overrightarrow{m} \; DIR_2)$
40. $DSK_3 = DSK_2$

which together with (29.,30.,32.) completely specifies the 3rd stage (of the 2nd development level). Observe our decision to let all, so-called opened files be represented in storage by directories identified by file names whereas directories on disk are identified by directory names, such as listed in the catalogue. Now we access directories directly, by-passing the catalogue. New directory entries (23.5) etc. are only to be recorded in the opened, that is storage, directories -- hence the copying-back upon closing!

Domain Invariants

As in the previous stage we express invariance in terms of an auxiliary function and the invariance of the abstraction of this stage of concretization retrieved back to the previous stage. The auxiliary function guarantees that retrieval is meaningful.

41.0 $inv\text{-}FS_3(fs_3) \triangleq (wf\text{-}StgDskOverlap(fs_3) \wedge inv\text{-}FS_2(retr\text{-}FS_2(fs_3)))$

The *wf-StgDskOverlap* checks that only catalogued files are opened (42.1) and that identical page names of opened files of storage and disk directory copies map to identical addresses (42.3):

42.0 $wf\text{-}StgDskOverlap(mk\text{-}FS_3(mk\text{-}STG_3(c,ods),dsk)) \triangleq$
.1 $((\underline{dom}ods \subseteq \underline{dom}c)$
.2 $\wedge(\forall f \in \underline{dom}c)((ods(f) \mid \underline{dom}(dsk(c(f))) = dsk(c(f)) \mid \underline{dom}(ods(f))))$

where the last line expresses the mutual restriction of file directories in storage (to the left) and on disk (to the right) to common domains.

Abstraction: from FS_3 to FS_2:

43.0 $retr\text{-}FS_2(mk\text{-}FS_3(mk\text{-}STG_3(c,ods),dsk)) \triangleq$
.1 $mk\text{-}FS_2(c, dsk + [\ c(f) \mapsto ods(f) \mid f \in ods\ ])$

Semantic Functions

In general:

44. *type:* $Elab\text{-}Cmd_3: Cmd_3 \to (FS_3 \xrightarrow{\sim} FS_3)$

and in specific:

45. *type:* $Int\text{-}Open: Open \to (FS_3 \xrightarrow{\sim} FS_3)$
46. *type:* $Int\text{-}Close: Close \to (FS_3 \xrightarrow{\sim} FS_3)$

45.0 $Int\text{-}Open_3[mk\text{-}Open(f)](mk\text{-}FS_3(mk\text{-}STG_3(c,ds),dsk)) \triangleq$
.1 *if* $(f \in dom\ c) \wedge (f \neg\in dom\ ds)$
.2 *then* $mk\text{-}FS_3(mk\text{-}STG_3(c, ds \cup [f \mapsto dsk(c(f))]), dsk)$
.3 *else undefined*

46.0 $Int\text{-}Close_3[mk\text{-}Close(f)](mk\text{-}FS_3(mk\text{-}STG_3(c,ds),dsk)) \triangleq$
.1 *if* $(f \in dom\ c) \wedge (f \in dom\ ds)$
.2 *then* $mk\text{-}FS_3(mk\text{-}STG_3(c, ds\backslash\{f\}), dsk + [c(f) \mapsto ds(f)])$
.3 *else undefined*

47.0 $Int\text{-}Crea_3[mk\text{-}Crea(f)](mk\text{-}FS_3(mk\text{-}STG_3(c,ds),dsk)) \triangleq$
.1 *if* $f \in dom\ c$
.2 *then undefined*
.3 *else* (*let* $a \in Adr - dom\ dsk$ *in*
.4 $mk\text{-}FS_3(mk\text{-}STG_3(c \cup [f \mapsto a], ds), dsk \cup [a \mapsto [\,]])$)

Creating a file does not "automatically" open it. If it did then line (47.4) should read: $mk\text{-}FS_3(mk\text{-}STG_3(c \cup [f \mapsto [\,]]), dsk \cup [a \mapsto [\,]])$

48.0 $Int\text{-}Eras_3[mk\text{-}Crea(f)](mk\text{-}FS_3(mk\text{-}STG_3(c,ds),dsk)) \triangleq$
.1 *if* $(f \in dom\ c) \wedge (f \neg\in dom\ ds)$
.2 *then* $mk\text{-}FS_3(mk\text{-}STG_3(c\backslash\{f\}, ds), dsk \setminus \{c(f)\} \cup rng(dsk(c(f))))$
.3 *else undefined*

We are permitted only to erase closed files.

```
49.0 Int-Put_Z[mk-Put(f,g,pg)](mk-FS_Z(mk-STG_Z(c,ds),dsk)) ≜
 .1    if (f ∈ dom c) ∧ (f ∈ dom ds)
 .2       then if p ∈ dom(ds(f))
 .3               then mk-FS_Z(mk-STG_Z(c,ds),dsk+[(ds(f))(p) ↦ pg])
 .4               else (let a∈Adr - dom dsk in
 .5                     let ds' = ds+[f ↦ ds(f) ∪ [p ↦ s]],
 .6                         dsk' = dsk ∪ [a ↦ pg]] in
 .7                     mk-FS_Z(mk-STG_Z(c,ds'),dsk'))
 .8       else undefined

50.0 Val-Get_Z[mk-Get(f,p)](mk-FS_Z(mk-STG_Z(c,ds),dsk)) ≜
 .1    if (f ∈ dom c) ∧ (f ∈ dom ds) ∧ (p ∈ dom (ds(f)))
 .2       then dsk((ds(f))(p))
 .3       else undefined

51.0 Int-Del_Z[mk-Del(f,p)](mk-FS_Z(mk-STG_Z(c,ds),dsk)) ≜
 .1    if (f ∈ dom c) ∧ (f ∈ dom ds) ∧ (p ∈ dom (ds(f)))
 .2       then mk-FS_Z(mk-STG_Z(c,ds+[f↦ds(f)\{p}]),dsk\{(ds(f))(p)})
 .3       else undefined
```

Conclusion to Second-Level Definition

Only after a number of stages of development, in which an abstract, implementation unbiased architecture has been gradually biased towards a particular realization, did we introduce the (user) interface notions of open and close. We see these more as concessions to current technological constraints than as representing user-meaningful intrinsic notions. No decision has yet been made, by the models up to now, whether open and close are user- or system-specified commands. It is perfectly possible for a system to automatically issue the operations in response to some analysis of user processes.

11.3 THIRD-LEVEL (File-Handler) ARCHITECTURE (Stages 4-5-6)

The major aim of this level is to render the file handler architecture more robust to crashes. [Robustness is a measure of the reliability notion introduced in section 11.0.] Our solution to the problem of increased recoverability is to introduce two means, that is two notions, of

accessing pages (on disk). One access 'path' goes from the storage catalogue via the storage directory -- as before -- to the disk pages. Another, "redundant", access 'path' is then introduced. It goes from a disk copy of the storage catalogue via directories also residing on disk to disk pages. Thus we shall "maintain" two kinds of file subsystems, but, as a new idea, not necessarily each others images. At any one time the two file sub-systems are to be well-formed, but not necessarily retrievable to the same abstract file system. We shall generally permit for example writes, updates and deletes only to be recorded in storage directories. We shall furthermore require that updates are treated as writes. Given now a situation where the storage and the disk file sub-systems are "equivalent", a sequence of writes, updates and deletes will then render the two different -- but with the ability that should a crash occur, then the disk sub-system can be used to replace the storage sub-system. Since we do not wish to backup too far into the past we introduce an operation which applies to files and renders the copies of directory and all pages for that file in storage and on disk the same, that is "equivalent". It does so by copying the storage copies onto the disk, overriding its information.

This level will consist of three stages (4-5-6) of transformation. Stage 4 introduces all the above-hinted and user-oriented notions. Stages 5 and 6 transform the model of stage 4 into successively more implementation-oriented models.

11.3.1 Storage and Disk File Sub-Systems (Stage 4)

Our departure point is (29.-31., 35.-51).

Semantic Domains

We repeat (29.-31., 38.-40.)

52. $FS_3 :: STG_3\ DSK_2$ (38.)
53. $STG_3 :: CTLG_2\ (Fn \overset{m}{\to} DIR_2)$ (39.)
54. $CTLG_2 = Fn \overset{m}{\to} Adr$ (29.)
55. $DIR_2 = Pn \overset{m}{\to} Adr$ (31.)
56. $DSK_2 = Adr \overset{m}{\to} (DIR_2 \mid PG)$ (40.,30.)

The "only" change (to the Domain equations) is in (56.)

57. $FS_4 :: STG_3 \; DSK_4$
58. $DSK_4 :: CTLG_2 \; DSK_2$

Domain Invariant

Storage directories may denote pages not denoted by disk directories. Not all disk pages need be denoted. Deletion of pages are only recorded in storage directories, that is no deletion of disk pages take place. There are two 'consistent' file sub-system notions:

59.0 $inv\text{-}FS_4(mk\text{-}FS_4(s,d)) \triangleq$
.1 $(consistent\text{-}StgSS(s,d) \wedge consistent\text{-}DskSS(d))$

Storage sub-system 'consistency' speaks of invariance of the file-system "rooted" in storage catalogue and directories. Disk sub-system 'consistency' speaks of invariance of the file-system "rooted" in disk catalogue and disk directories. We express both in terms of the invariance of FS_2 abstractions of the file-subsystems. We can only abstract the storage sub-system if opened files are all catalogued (60.1).

60. *type:* $consistent\text{-}StgSS: STG_3 \; DSK \to BOOL$
61. *type:* $consistent\text{-}DskSS: DSK \to BOOL$

60.0 $consistent\text{-}StgSS(mk\text{-}STG_3(c,d), mk\text{-}DSK_4(,dsk)) \triangleq$
.1 $(\underline{dom}\ d \subseteq \underline{dom}\ c) \wedge inv\text{-}FS_2(mk\text{-}FS_2(c, SabsDSK_2(c,d,dsk)))$

We abstract to FS_2 file systems since all we are interested in is access paths from catalogue via directories to pages irrespective of the notions of opened/closed.

62. *type:* $SabsDSK_2: CTLG_2 \; (Fn \; \overset{m}{\to} \; DIR_2) \; DSK_2 \xrightarrow{\sim} DSK_2$
62.0 $SabsDSK_2(c,d,dsk) \triangleq$
.1 $(\underline{let}\ as_s = Saddrs(c,d,dsk)\ \underline{in}$
.2 $(dsk \mid as_s + [c(f) \mapsto d(f) \mid f \in \underline{dom}\ d\})$

63.0 $Saddrs(c,d,dsk) \triangleq$
.1 $(\underline{let}\ das = \underline{rng}\ c,$
.2 $opas = \underline{union}\{\underline{rng}\ dir \mid dir \in \underline{rng}\ d\},$
.3 $cpas = \underline{union}\{\underline{rng}(dsk(a)) \mid a \in \{c(f) \mid f \in \underline{dom} c \setminus \underline{dom} d\}\}\ \underline{in}$
.4 $das \cup opas \cup cpas)$
63. *type:* $CTLG_2 \; (Fn \; \overset{m}{\to} \; DIR_2) \; DSK_2 \xrightarrow{\sim} Addr\text{-}set$

The idea of $SabsDSK_2$ is to retrieve only those closed directories on disk which are accessible from the storage catalogue, and only those pages which are accessible via those closed directories and via the opened (storage) directories, and to extend the resulting DSK_2 object with the opened storage directories. Where these before were denoted by file names, they are now, on disk, denoted by the directory name recorded in the storage catalogue. *das* stand for directory addresses of all storage-accessible opened and closed directories, *opas* (*cpas*) for addresses of all most recently written and updated pages of all such opened (such closed) files. Since we shall extract these addresses repeatedly we have "invented" an auxiliary function, *Saddrs*. for that purpose.

64.0 $DabsDSK_2(c,dsk) \triangleq$
.1 $(\underline{let}\ as_d = Daddrs(c,dsk)\ \underline{in}$
.2 $(dsk \mid as_d))$

65.0 $Daddrs(c,dsk) \triangleq$
.1 $(\underline{let}\ das = \underline{rng}\ c\ \underline{in}$
.2 $\underline{let}\ pas = \underline{union}\{\underline{rng}(dsk(a)) \mid a \in das\}\ \underline{in}$
.3 $dasupas)$

Abstraction: from FS_4 to FS_3

Two kinds of abstractions are possible: one from the storage "rooted" file sub-system, and another based on three disk "rooted" file sub-system:

66.0 $retr\text{-}FS_3(mk\text{-}FS_4(mk\text{-}STG_3(c,d),mk\text{-}DSK_4(,dsk))) \triangleq$
.1 $mk\text{-}FS_3(mk\text{-}STG_3(c,d),dsk|Saddrs(c,d,dsk))$

67.0 $retr\text{-}FS_{3D}(mk\text{-}FS_4(,mk\text{-}DSK_4(c,dsk))) \triangleq$
.1 $mk\text{-}FS_3(mk\text{-}STG_3(c,[]),dsk|Daddr(c,dsk))$

Garbage Collection

As is evident from the informal and (subsequent) formal description of write, update and delete commands we shall witness file systems with both disk directories and pages which are denoted by no catalogue, respectively no or only such inaccessible ("dead") directories. Garbage-collection then is an operation which removes all such "dead" directories and all such pages:

68. *type:* $Garb\text{-}Coll: FS_4 \xrightarrow{\sim} FS_4$

68.0 $Garb\text{-}Coll(mk\text{-}FS_4(mk\text{-}STG_3(sc,d),mk\text{-}DSK_4(dc,dsk))) \triangleq$
.1 $(let\ as_s = Saddrs(sc,d,dsk),$
.2 $\quad as_d = Daddrs(dc,dsk)\ in$
.3 $mk\text{-}FS_4(mk\text{-}STG_3(sc,d),mk\text{-}DSK_4(dc,dsk\setminus(as_s \cup as_d))))$

Syntactic Domains -- and Informal Semantics

When a crash occurs one needs, in order to recover, to "roll-back" to a not too far distant past consistent state -- namely the disk sub-system current at the time of the crash. To avoid the "distance" between the time of the crash and the previous time when the subsystems retreived to the same abstract FS_3 system being "too big", one needs, "now and then", to bring the disk sub-system to reflect the state of the storage sub-system. For that purpose a so-called check-point command is introduced. It applies to individual, opened, file and bring the storage and disk catalogue entries for that file to both denote the same disk directory which is to be that of the opened file. A crash can be thought of as a command (issued by some Demon nice enough only to issue it properly in-between the "execution" of other commands). Its first effect is to "blank-out" all storage information. Its second effect is then to restore the storage sub-system to that of the disk sub-system.

69. $Cmd_4 = Cmd_3 \mid Check \mid Crash$
70. $Check :: Fn$
71. $Crash :: ()$

Semantic Functions

72. *type:* $Elab\text{-}Cmd_4: Cmd_4 \to (FS_4 \xrightarrow{\sim} (FS_4 \mid PG))$

73.0 $Int\text{-}Crea_4[mk\text{-}Crea(f)](mk\text{-}FS_4(mk\text{-}STG_3(sc,d),mk\text{-}DSK_4(dc,dsk))) \triangleq$
.1 $if\ f \in dom\ sc$
.2 $\quad then\ undefined$
.3 $\quad else\ (let\ a \in Adr - dom\ dsk\ in$
.4 $\quad\quad mk\text{-}FS_4(mk\text{-}STG_3(sc \cup [f \mapsto a],d),mk\text{-}DSK_4(dc,dsk \cup [\mapsto [\,]])))$

74.0 $Int\text{-}Eras[mk\text{-}Eras(f)](mk\text{-}FS_4(mk\text{-}STG_3(sc,d),dsk_4)) \triangleq$
.1 $if\ (f \in dom\ sc) \wedge (f \neg\in dom\ d)$
.2 $\quad then\ mk\text{-}FS_4(mk\text{-}STG_3(sc \setminus \{f\},d),dsk_4)$
.3 $\quad else\ undefined$

```
75.0 Int-Open4[mk-Open(f)](mk-FS4(mk-STG3(sc,d),mk-DSK4(dc,dsk))) ≜
  .1   if (f∈dom sc) ∧ (f¬∈dom d)
  .2      then mk-FS4(mk-STG3(sc,d∪[f↦dsk(sc(f))]),
  .3                  mk-DSK4(dc,dsk))
  .4      else undefined

76.0 Int-Close[mk-Close(f)](mk-FS4(mk-STG3(sc,d),mk-DSK4(dc,dsk))) ≜
  .1   if (f∈dom sc) ∧ (f∈dom d)
  .2      then (let a∈Adr - dom dsk in
  .3            mk-FS4(mk-STG3(sc+[f↦a],d\{f}),
  .4                   mk-DSK4(dc,dsk∪[a↦d(f)])))
  .5      else undefined

77.0 Int-Put4[mk-Put(f,p,pg)](mk-FS4(mk-STG3(sc,d),mk-DSK4(dc,dsk))) ≜
  .1   if (f∈dom sc) ∧ (f∈dom d)
  .2      then (let a∈Adr - dom dsk in
  .3            mk-FS4(mk-STG3(sc,d+[f↦d(f)+[p↦a]]),
  .4                   mk-DSK4(dc,dsk∪[a↦pg])))
  .5      else undefined

78.0 Int-Del4[mk-Del(f,p)](mk-FS4(mk-STG3(sc,d),dsk4)) ≜
  .1   if f∈dom sc ∧ f∈dom d ∧ p∈dom(d(f))
  .2      then mk-FS4(mk-STG3(sc,d+[f↦d(f)\{p}]),dsk4)
  .3      else undefined

79.0 Int-Check4[mk-Check(f)](mk-FS4(mk-STG3(sc,d),mk-DSK4(dc,dsk))) ≜
  .1   if (f∈dom sc) ∧ (f∈dom d)
  .2      then (let a∈Adr - dom dsk in
  .3            mk-FS4(mk-STG3(sc+[f↦a],d),
  .4                   mk-DSK4(dc+[f↦a],dsk∪[a↦d(f)])))
  .5      else undefined

80.0 Int-Crash4[mk-Crash()](mk-FS4(,mk-DSK4(dc,dsk))) ≜
  .1   mk-FS4(mk-STG3(dc,[]),mk-DSK4(dc,dsk))
```

If necessary one can garbage collect after crashes:

```
81.0 Int-Crash'4[crash](fs4) ≜
  .1   Garb-Coll(Int-Crash4[crash](fs4))
```

11.3.2 "Flat" Storage and Disk (Stage 5)

Semantic Domains

Usually there is no "cell-space" distinction in storage between the catalogue and the collection of opened directories; and similarly: there is usually no "sector-space" distinction on disk(s) between the disk catalogue, on one hand, and disk directories and pages, on the other hand -- such as seemingly implied by Domain equations (53.), respectively (58.). We assume storage to be addressed say in "chunks" comparable to catalogues and directories; and disks addressed in "chunks" comparable to catalogues, directories and pages. We let the address space, *REF*, of storage have MASTER as a distinguished element, and address space, *REF*, of disk have COPY as a distinguished element. MASTER denotes the former $CTLG_2$ component of STG_3 and COPY the former $CTLG_2$ component of DSK_4. The storage catalogue record disk and storage addresses of opened files directories, but only disk addresses of closed files directories. The disk catalogue "copy" record only disk addresses of file directories. The storage catalogue records the disk address of its counterpart, the disk catalogue.

82. $REF = \underline{\text{MASTER}} \mid Ref$
83. $ADR = \underline{\text{COPY}} \mid Adr$
84. $FS_5 :: STG_5 \; DSK_5$
85. $STG_5 = (\underline{\text{MASTER}} \; m \; SCTLG_5) \; \underline{\cup} \; (Ref \; m \; DIR_5)$
86. $DSK_5 = (\underline{\text{COPY}} \; m \; DCTLG_5) \; \underline{\cup} \; (Adr \; m \; DIR_5 \mid PG))$
87. $SCTLG_5 = (\underline{\text{MASTER}} \; m \; \underline{\text{COPY}}) \; \underline{\cup} \; (Fn \; m \; Dadr)$
88. $DCTLG_5 = (\underline{\text{CTLG}} \; m \; (\underline{\text{MASTER}} \mid \underline{\text{COPY}})) \; \underline{\cup} \; (Fn \; m \; Adr)$
89. $Dadr = (Adr \; [Ref])$

Etcetera

We leave it to the reader to complete this stages' invariant, abstraction and semantic functions.

11.3.3 Storage and Disk Space Management (Stage 6)

Semantic Domains

Usually both storage and disk(s) represent limited resources in the sense of not permitting an infinite, but only a finite, amount of "space" for

keeping catalogues, directories and pages. Our (map-based) models so far have assumed indefinitely sized such spaces. The aim of this last stage of object transformation is to introduce the notion of storage and disk space management. We assume therefore a limited amount of space both in storage and on disk(s). Instead of always being able to fetch new disk addresses (see for example (73.3), (76.2) and (77.2)) so-called free lists of allocatable sub-spaces is maintained, both for storage and for disk(s). We also assume that each such subspace is adequate and reasonably sized for both directories and pages -- these are the only quantities to be allocated and freed. The free lists are therefore modelled as sets of storage references respectively disk addresses not allocated. The quotation FREE denotes these lists:

90. $FS_6 :: STG_6\ DSK_6$
91. $STG_6 = STG_5 \underline{\cup} (\underline{\text{FREE}} \text{ m} Ref\text{-}set)$
92. $DSK_6 = DSK_5 \underline{\cup} (\underline{\text{FREE}} \text{ m} Adr\text{-}set)$

Domain Invariant

93.0 $inv\text{-}FS_6(mk\text{-}FS_6(stg,disk)) \underline{\Delta}$
.1 $(\underline{\text{FREE}} \in \underline{dom}\ stg) \wedge (stg(\underline{\text{FREE}}) \cap \underline{dom}\ stg = \{\})$
.2 $\wedge(\underline{\text{FREE}} \in \underline{dom}\ dsk) \wedge (dsk(\underline{\text{FREE}}) \cap \underline{dom}\ dsk = \{\})$
.3 $\wedge\ inv\text{-}FS_5(mk\text{-}FS_5(stg\backslash\{\underline{\text{FREE}}\}, dsk\backslash\{\underline{\text{FREE}}\}))$

Semantic Functions

One illustration is sufficient to illuminate the idea:

94.0 $Int\text{-}Crea_6[mk\text{-}Crea(f)](mk\text{-}FS_6(stg,dsk)) \underline{\Delta}$
.1 $\underline{if}\ f \in \underline{dom}(stg(\underline{\text{MASTER}}))$
.2 $\underline{then}\ \underline{undefined}$
.3 $\underline{else}\ \underline{if}\ dsk(\underline{\text{FREE}}) = \{\}$
.4 $\underline{then}\ \underline{undefined}$
.5 $\underline{else}\ (\underline{let}\ a \in dsk(\underline{\text{FREE}})\ \underline{in}$
.6 $mk\text{-}FS_6(stg+[\underline{\text{MASTER}} \mapsto stg(\underline{\text{MASTER}})+[f \mapsto (a,\underline{nil})]],$
.7 $dsk\cup[a \mapsto [\,]]+[\underline{\text{FREE}} \mapsto dsk(\underline{\text{FREE}})\backslash\{a\}]))$

Conclusion to Third-Level Definition

It turned out, perhaps somewhat surprisingly, that it was not too cumbersome to enrich our architecture to first embody the user-oriented aspects

of reliability, recoverability, checkpointing and garbage collection; and then to turn it, some would think, radically, in the direction of a rather straightforward implementation.

11.4 SUMMARY

There remains to implement the resulting architecture. But since those (many) stages are not the concern of this chapter we shall leave it out! The concern, instead, of this chapter was to advocate, and show through a reasonably realistic example, the idea, respectively feasibility, of stepwise transformation of software architectures. We remind the reader of our remarks of the first three subsections of section 11.0. The conclusions we draw from the example are the following:

(1) It is desirable to study the architecture of what one is about to implement. Once implemented the product, whatever it is, will have great impact on users and/or systems. Considering the enormity of most such impacts, the intellectual, human and economic expenditures to be distributed over the "life-time" of the product, it is quite reasonable to spend far more time on studying the architecture.

(2) We say that "we study the architecture". By that we mean that the techniques of writing down, or formulating, Domain invariants and Domain abstractions (retrievals), besides being required in correctness proofs, also play an indispensable rôle in clarifying and adjusting the architecture proposal.

(3) We can demonstrate that spending increased resources on "paperwork", that is on architecture proposals formulated as exemplified is more advantageous than binding oneself to prototype implementations being disturbed by realization aspects, most often of the kind: "How do I program my way around 'this or that' host system peculiarity which I know will not be present in the actual system". We are all too often rushing into implementation before having properly understood our objectives.

(4) It is possible, as demonstrated, to conquer complexity through decomposition. But it is a developmental decomposition represented by "approximation". First we approximate what we consider most

important. In other examples than the one shown, reliability could be considered most important, and probably should have lead to a rather different sequence of architecture transformations. What we have shown is the kind of techniques used in stepwise architecture transformation.

11.5 BIBLIOGRAPHY

The incentive to provide a formal model of specifically the file handler illustrated in this chapter came from J.R.Abrial [Abrial 80*(4)]. Abrial acknowledges C.A.R.Hoare. The system modelled is believed to be that of OS6 [Stoy 72ab]. The model in [Abrial 80*(4)] introduces almost all notions in a first stage.