



Fermi National Accelerator Laboratory

FERMILAB-Conf-92/58

Architecture Flow Diagrams under **team work**®

T. Nicinski

*Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510*

February 1992

To be presented at the *Symposium on Assessment of Quality Software Development Tools*, New Orleans, LA, May 27-29, 1992.

To be published in *IEEE Software*, May 1992.

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Architecture Flow Diagrams under *teamwork*[®] *

Tom Nicinski

Fermi National Accelerator Laboratory / P.O. Box 500 / Batavia, IL 60510

Abstract

The Teamwork CASE tool allows Data Flow Diagrams (DFDs) to be maintained for structured analysis. Fermilab has extended teamwork under UNIX™ to permit Hatley and Pirbhai Architecture Flow Diagrams (AFDs) to be associated with DFDs and subsequently maintained. This extension, called TWKAFD, allows a user to open an AFD, graphically edit it, and replace it into a TWKAFD maintained library. Other aspects of Hatley and Pirbhai's methodology are supported.

This paper presents a quick tutorial on Architecture Diagrams. It then describes the user's view of TWKAFD, the experiences incorporating it into teamwork, and the successes with using the Architecture Diagram methodology along with the shortcomings of using the teamwork/TWKAFD tool.

1 Introduction

For the requirements specification for the Digital Sky Survey (DSS) [2], Fermilab needed a methodology, for specifying a data acquisition system, to supplement *Data Flow Diagrams* (DFDs).¹ Because of hardware and performance constraints placed on the data acquisition system, an architecturally oriented view of the system was needed. This view would interact with the DFD view, each prompting refinements in the other. Hatley and Pirbhai's Architecture Diagram methodology [3] proved to be the answer.

Initially, the DSS team drew *Architecture Flow Diagrams* (AFDs) and *Architecture Interconnect Diagrams* (AIDs) using a standalone drawing package. Because of the size of the DSS project, a large number of drawings were involved. Controlling update ac-

cess to the diagrams and the the maintenance of information about which AFDs/AIDs were associated with which DFDs quickly became tedious and error prone. A tool was needed, but none was available which merged Cadre Technology, Inc. *teamwork*'s DFDs with Hatley and Pirbhai Architecture Diagrams.

The TWKAFD extension to *teamwork* was developed at Fermilab to allow users to associate Architecture Flow Diagrams with Data Flow Diagrams [6]. Architecture Interconnect Diagrams are also supported and an *Architecture Module Specification* (AMS), which describes the allocation of DFDs to AFDs, is maintained. These modeling methodologies are accessed when graphically editing a DFD with the *teamwork* CASE tool: a user can open an AFD or AID for an entire DFD or selected process within a DFD by using the TWKAFD tool.

2 Architecture Diagrams (The Methodology)

The functional requirements for a system are defined by a process model using DFDs. Processes within the DFD model are then *allocated* to physical entities within *Architecture Flow Diagrams* (AFDs). Thus, AFDs show where processes are carried out. They are composed of *modules* (locations where processes occur) and *interconnects* (data and control connections between modules). The "mapping" of DFDs to Architecture Flow Diagrams is necessary because

... for the specification of systems, we need to capture not only *what* the system requirements are, but also *how* the system will fulfill those requirements.

The means for capturing this system mechanization is the architecture model, whose principal purposes are

*Sponsored by DOE Contract DE-AC02-76CH03000.

¹Data Flow Diagrams and structured analysis are described in books by Yourdon, DeMarco [1], Hatley and Pirbhai [3], etc.

- to show the physical entities that make up the system
- to define the information flow between these physical entities
- to specify the channels on which this information flows

These purposes are fulfilled using diagrams, supported by textual specifications and a dictionary [4].

An architecture module corresponds to one or more processes (bubbles in a DFD). Architecture modules should be named appropriately, but they do not need to be named after DFDs. The allocation of a process from a DFD to an architecture module implicitly allocates the DFD's children to that module. The allocation of DFDs to AFDs is maintained in the *Architecture Module Specification* (AMS).

Within an AFD, *information flow vectors* represent the information that flows between architecture modules. They are comprised of data flows (represented by solid arrows, \longrightarrow) and control flows (represented by dashed arrows, $--\rightarrow$). Like DFD data flows and control flows, information flow vectors should be labeled with the types of information going through them. Just as modules, information flow vectors do not need to be labeled after DFD data and control flow names.

Each AFD has a corresponding *Architecture Interconnect Diagram* (AID). The AID shows the same modules as the AFD, but it focuses on the physical communications channels between those architecture modules. These *information flow channels* show physical connections and do not necessarily match the AFD's information flow vectors. But, there is a mapping between information flow channels and vectors as data and control need to flow across some medium. Different channel types are available; some are represented as

Symbol	Meaning
—	Electrical bus
+++	Mechanical link
—○—○—	Optical link

Information flow channels are labeled with the type of media used for communication (for example, an optical link can be labeled with "fiber optic cable," "infrared beam," etc.). If multiple channels exist between two modules, then they are graphically represented with repeated symbols. The characteristics of the channels, such as timing needs, interconnect bandwidths, burst rates, etc., are described textually in an *Architecture Interconnect Specification* (AIS).

AFDs and AIDs are each composed of five major units. They offer different perspectives of functionality within the system.

- *Input and Output Processing* show the data and control flows that the Control Model/Process Model will use and produce. The processing performed here is not necessarily required by the DFD model. Instead, it represents the additional processing necessary to allow architecture modules to communicate amongst each other. It may also transform data to an internally usable form.
- *User Interface Processing* shows human-related interfaces to the AFD/AID. User Interface Processing is not included as part of Input and Output Processing to emphasize some unique considerations, such as ergonomics, that affect how this processing is performed.
- *Control Model/Process Model* performs the majority of the work depicted by the AFD/AID. It contains the bulk of the requirements specified by the DFD model.
- *Maintenance, Self-test, and Redundancy Management Processing* shows the processing done to maintain the work done within Control Model/Process Model. It can include modules for self-monitoring and data collection (that will be used for system maintenance).

Any section not used within an AFD/AID can be omitted for clarity.

One important reason for the architectural exercise is that the architecture model may suggest or even necessitate the repartitioning of processing within the DFD model:

Overall, then, the transformation of the process model into an architecture model is an iterative process that resolves all the interfaces and allocates processes to architecture modules through tradeoffs and design decisions. The result is a fully integrated system specification covering both the functional requirements and the physical design [5].

2.1 An Example

The DFD of Figure 1 depicts part of what many people do every day, entertain themselves from a couch. In this case, it's either recording or playing back a movie from a VCR. This example is not meant to be

particularly complete or rigorous. It shows a DFD one level down from the operation of a complete entertainment system.

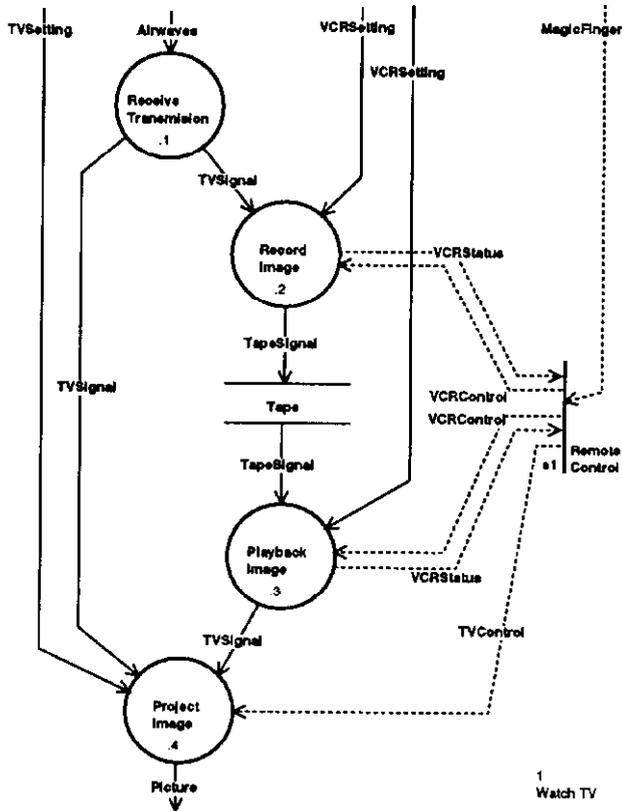


Figure 1: Example DFD 1, *Watch TV*

The AFD of Figure 2 corresponds to the DFD of Figure 1. It has only three architecture modules compared to the four process bubbles of the DFD. Although many of the names used in the AFD, especially those of data flows and control flows, are the same as those of the DFD, they need not be the same.

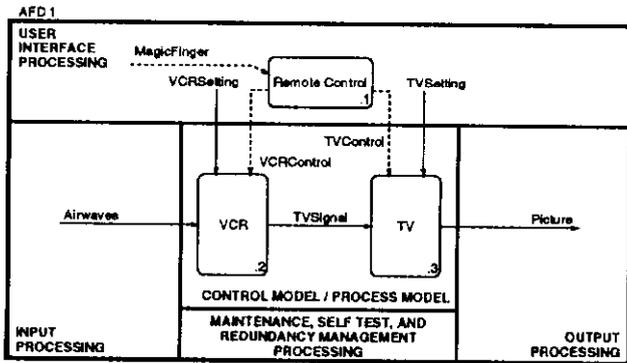


Figure 2: Example AFD 1

An *Architecture Module Specification (AMS)* lists

which DFD bubbles are allocated to the AFD modules:

- AFD 1 is allocated to DFD *Watch TV* (1).
- AFD module *Remote Control* (1.1) does not have any DFD processes allocated to it. It does correspond to the C-spec of the DFD, but the architecture model does not reflect this.
- AFD module *VCR* (1.2) has DFD processes *Receive Transmission* (1.1), *Record Image* (1.2), and *Playback Image* (1.3) allocated to it.
- AFD module *TV* (1.3) has DFD process *Project Image* (1.4) allocated to it.

The corresponding AID for the AFD of Figure 2 is shown in Figure 3.

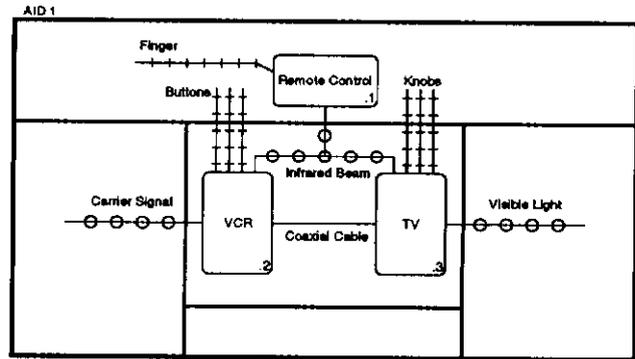


Figure 3: Example AID 1

This AID has the same architecture modules as those of the AFD. The AID shows the different media used to transport data and control information between the various architecture modules. The information flow channels in this example are not the same as the information flow vectors of the example AFD.

It may be necessary to reference the corresponding AFD while reading an AID. Information flow channels between modules can be considerably different than the information flow vectors between modules of the corresponding AFD. But, there must be at least one physical channel depicted to show the information flow of a vector or group of vectors. TWKAFD does not provide any means of correlating information flow vectors with information flow channels.

3 Restrictions on the Architecture Methodology

Although there are no hard and fast rules (at least according to Hatley and Pirbhai) as to how an AFD

hierarchy relates to its corresponding DFD hierarchy, some restrictions are placed on the Architecture Diagram modeling supported by TWKAFD:

- An AFD can have more than one DFD allocated to it, but a DFD cannot be allocated to more than one AFD. A DFD's functionality cannot be split between multiple AFD modules.
- DFDs at one level of the hierarchy can only be allocated to AFDs at the same depth within the corresponding AFD hierarchy.
- If a parent DFD is allocated to an AFD, the DFD children can only be allocated to modules of that AFD. The DFD children cannot be allocated to modules of another AFD.

These restrictions are necessitated by the TWKAFD implementation. Additionally, they resolve ambiguities in the use of the methodology, leading to a clearer interpretation of a model. Through the use of tools, refinements in methodologies can be made. For example, *teamwork* does not place any restrictions on how stores are interpreted (for example, are reads destructive?). But, *teamwork/SIM*TM [7] resolves these ambiguities with rules in order to drive a dynamic simulation of a system represented by DFDs.

4 Using TWKAFD (The Tool)

TWKAFD manages FIG format files which represent AFDs and AIDs. TWKAFD invokes the the XFIG Utility² to allow the user to graphically edit AFD and AID files. Editing is performed with simple drawing primitives rather than with AFD/AID object primitives. For example, an AID optical link (—o—o—) is drawn with a line (—) and circles (o o o) to get —o—o—.

Besides restrictions on the Architecture Diagram methodology (as discussed in Section 3), an additional rule is enforced:

- An AID cannot be opened unless the corresponding AFD file exists and has at least one DFD allocated to it.

²XFIG is the X Window SystemTM version of FIG.

TWKAFD does not perform all the work for a user, especially when bookkeeping matters are involved. It is important to remember that TWKAFD is built using *teamwork* facilities, but is *not* tightly integrated into *teamwork*. The user must consider the following situations:

- Changes made to DFDs, especially deletions and renumberings, do *not* get automatically reflected in the corresponding AFDs, AIDs, and the AMS. For example, deleting a DFD process will not remove the allocation entry from the AMS; the user must update the AMS as an explicit step.
- Changes made to an AFD are *not* automatically reflected in the corresponding AID, and vice-versa.
- TWKAFD takes *teamwork*'s approach to handling the relationships between DFDs and AFDs. This was a design decision to try and make the TWKAFD environment similar to the *teamwork* environment. Thus, the user does not need to learn two sets of behaviors. For example, the reallocation of a DFD to another AFD will *not* cause the updating of allocations of descendent DFDs to the "new" descendent AFDs. Therefore, it is possible to get the AMS "out of synch" with what may be expected by the user.

4.1 Generations of Architecture Diagrams

TWKAFD maintains generations of AFDs and AIDs in a similar concept as *teamwork*'s maintenance of DFD generations. These generations are maintained within an SCCS (Source Code Control System) library. Also maintained within the library area is the AMS, although only one version is kept. A standalone utility, *twkafd.fetch*, allows the latest generations of AFDs and AIDs to be fetched. These can then, for example, be incorporated into documents, etc.

5 The User Interface

When opening a DFD, the DFD menu bar of Figure 4 is displayed by the DFD editor. It has been extended to include a pull-down menu to allow access to AFDs and AIDs. The AFD pull-down menu works just as any other *teamwork* menu.

File Whole.DFD ... OOA	AFDs/AIDs
	Open Latest AFD
	Open Latest AID
	Allocate ...
	Reallocate ...
	Deallocate
	Renumber AFD ...
	Delete AFD ...
	Delete AID ...

Figure 4: DFD Menu Bar

Some of the menu choices from the DFD menu bar are described below to give a taste of using TWKAFD. TWKAFD attempts to determine as much information about what needs to be done without querying the user. However, this is not always possible because TWKAFD cannot be tightly coupled to the DFD editor. For example, *teamwork's* DFD editor will automatically assign DFD names (which are numbers) to newly created processes. However, it is impossible to associate a DFD with an AFD module without querying the user for an AFD module name (also a number following *teamwork's* style).

5.1 Open Latest AFD

The user can open an AFD for the entire DFD currently being edited or for one selected process bubble within that DFD. In either case, if the chosen DFD³ is not already allocated to an AFD, TWKAFD will create a new AFD; otherwise, the latest generation of the associated AFD is opened.

In the case where an AFD will be created, if no ancestor or descendent DFD of the chosen DFD is allocated to an AFD, then the user will need to enter a complete AFD name:

Create AFD

Enter the complete name of the AFD to be created.

The AFD name must have 3 levels [as process 3.2.6].

AFD name:
| |

Notice that the AFD name must have the same number of *levels* as the DFD chosen to be allocated to that

³Because a process bubble within a DFD can expand to a DFD, TWKAFD treats a process as a DFD.

AFD.

A chosen DFD can have some ancestor DFD that is allocated to an AFD. In this case, the name of the AFD to be created must reflect that ancestry by taking the beginning portion of its name from the nearest ancestor AFD:

Create AFD

Enter the complete name of the AFD to be created.

An ancestor of DFD 2.5.3.1 is allocated to AFD 6.2.
Thus, the AFD name **MUST** be prefixed by that ancestor's AFD name.

The AFD name must have 4 levels [as DFD 2.5.3.1].

AFD name:
|6.2.

The final condition is where the chosen DFD has a descendent DFD that is allocated to an AFD. In this case, the name of the AFD that is to be created will be taken from the nearest descendent AFD. For example, an AFD is to be created for DFD 6.1. If a descendent DFD, 6.1.4.7, is allocated to AFD 2.10.7.1, then the created AFD will be named 2.10.

5.2 Open Latest AID

An AID can only be opened or created when its corresponding AFD exists, both in the AMS and in the AFD/AID library. The allocation of a DFD to an AFD is not sufficient in and of itself. The AID takes on the same name as the AFD to which the chosen DFD is allocated.

When an AID is created, the latest generation of the AFD is copied as the initial generation of the AID. This is done as an AID's modules must match those of the AFD. The user is still responsible for converting the duplicated AFD to an AID:

- Convert all data and control flows to the appropriate module interconnections. These do not have to match the AFD flows in either routing or count.
- Label module interconnections with the media used for communications, rather than the content of the communications as in an AFD.

5.3 Renumber AFD ...

An existing AFD and its corresponding AID can be renamed:

Renumber AFD

Enter the complete name of the AFD to be renumbered.

Old AFD name:

The user must then enter the new AFD name:

Renumber AFD

Enter the complete name that AFD 2.4 will be renamed to. Take note that the following DFDs will have their allocation records changed:

1.3

New AFD name:

An AFD with the new name cannot already exist. The AMS will be updated: all DFDs which were re-allocated to the old AFD name will be changed to be allocated to the new AFD name. As there are *no* restrictions or checks if the number of levels in the AFD name is changed, the user must be careful with this operation.

If the new AFD name has any DFDs already allocated to it, the user is given a chance to abort the renumbering operation:

Renumber AFD

AFD 4.7.9 [the new name] already has the following DFDs allocated to it:

3.10.1 6.11.13

Click OK if you still wish to renumber AFD 2.4 to 4.7.9.

5.4 Error Messages

Whenever an error is encountered, a message is displayed on the screen. The *title* of the message indicates whether the error occurred when working with an AFD or an AID. Additionally, the title has a *condition* name which narrows the location of the error. Finally, the message text gives a fuller explanation of the problem.

In most instances, the display of a message indicates that the requested operation was not completed. TWKAFD makes every effort to return the state of the architecture model to what it was just prior to the start of the operation. In case TWKAFD is unable to "fix things up," the message text will contain a brief description of what should be done to try to resolve the problem.

6 Merging *teamwork* and AFDs

TWKAFD is implemented with the *teamwork* Extensibility Language [8] and a group of UNIX C shell scripts. The *teamwork* Extensibility Language allows the addition of pull-down menus to *teamwork* editor menu bars. The *teamwork* Extensibility Language is not intuitive at first, so a mini tutorial is presented here.

In general, menu definitions are of the form:

```
( Menu
  ( Name "string" )
  ( Variable variable_definition )
  ( MenuItem
    ( Name "string" )
    ( Variable variable_definition )
    ( Action(SysCall "interpreted_string" )
  ) ) )
```

where Menu, Name, Variable, MenuItem, Action, and SysCall are keywords. *Variable_definition*, *string*, and *interpreted_string* are user-defined. A *string* is any set of ASCII characters nested between double quotes ("). An *interpreted_string* is a string parsed for variables. Definitions are nested and scoping rules apply to variables.

It is important to understand that interpreted strings are the key to doing any work with the *teamwork* Extensibility Language. When interpreted strings are parsed, variables are referenced. The variable's return value is substituted into the interpreted string. The syntax for referencing a variable is to prefix it with a percent sign (%).

```
%variable          or  %(variable)
%variable(arg1, ...) or  %(variable(arg1, ...))
```

Variables can have arguments passed to them. They can also be enclosed within parentheses if the character following the variable is not a variable terminator.

6.1 “Subroutine” Calls

The *teamwork* Extensibility Language does not provide subroutines. Instead, a variable is referenced. In essence, most everything done within the *teamwork* Extensibility Language involves variable references, even performing conditional tests:

```
%IF(%EQ(%var1,%var2),%then_action,
      %else_action)
```

where `.IF` and `.EQ` are *teamwork* control variables. Two important control variables are

```
.SYS_CALL(command)
.RETRIEVE(file)
```

where `.SYS_CALL` passes the interpreted string, *command*, to the native shell for execution. `.RETRIEVE` can then be used to read a result from a *file*. For example:

```
%SYS_CALL(%STRING(echo 'Hello' >
%tmpfile))
%IF(%EQ(%RETRIVE(%tmpfile),
      %QUOTE(Hello)),...)
```

6.2 TWKAFD Implementation

As much of the work as possible is done with the *teamwork* Extensibility Language. The attempt is to reduce the dependence upon the native system under which TWKAFD is being used.⁴

Still, a considerable amount of work is done using C shell scripts. All scripts have a standard set of arguments (via the variable `afd_stdarg`) passed to them, mainly to indicate which *teamwork* object was selected by the user. Each script returns a status by echoing to `stdout`. Usually, an output of “Success” indicates successful completion by the script. Any other output is the actual error message to be displayed by the *teamwork* Extensibility Language code.

Below, is an example section of code. It implements the Open Latest AFD menu item:

⁴TWKAFD currently works only on SunOS™ platforms. But, because TWKAFD is X based, this is not a restriction.

```
(MenuItem
  (Name "Open Latest AFD")
  (Variable (Id afd$afd.open)          (Value "
# Initialize and determine what object we'll be
# working with (this DFD or one of its processes).
%afd$init
%afd$check.objtyp
%.IF(%EQ(%afd.objtyp,%NULL),%afd$return(
  %.STRING(%AFDBADCHOICE),
  %.STRING(An AFD cannot be opened for a
%(t.SELECTED_OBJECT_TYPE). Select a process to
open an AFD for it, or select nothing to open a AFD for DFD
%(t.OBJECT))),
  %.NULL)

# Check whether the chosen object is already allocated
# to AFD. If it is, there's no need to query the user for
# an AFD number. Otherwise, afd$afd.afdnam.get will
# query the user for an AFD name and do legality
# checking. If all is fine, afd.afdnam will have the
# AFD name.
%.SYS_CALL(%STRING(
  ams.get %afd_stdarg 'AFD' 'EXACT' > %afd_stdarg))
%.ASSIGN(%afd.afdnam,
  %.REMOVE_WS(%STRING(%afd$status)))
%.IF(%EQ(%afd.afdnam,%NULL),
  %afd$afd.afdnam.get(%QUOTE(Create AFD)),
  %.NULL)

# Open AFD. aXd.open obtains exclusive access to the
# chosen object, invokes XFIG, and updates the AMS.
%.SYS_CALL(%STRING(
  aXd.open %afd_stdarg 'AFD' '%afd.afdnam' >
  %afd_stdarg))
%afd$return.on.error(%STRING(%AFDOPENERR))
)
(Action(SysCall "%afd$afd.open"))
)
```

The `Action(SysCall "...")` line is invoked when the user selects the menu item from the pull-down menu. Few extensions to *teamwork* are so simple where `Action` can perform all the work. Instead, a variable under the `MenuItem` is defined which does the work and `Action` simply references that variable.

In the above example, *teamwork* variables, prefixed by `t.`, are used. `t.OBJECT` is the name of the *teamwork* object, such as a DFD, a DFD process bubble, a store, etc., that is selected. `t.SELECTED_OBJECT_TYPE` specifies the type of that object: process, store, data flow, etc. Additional control variables are also used: `.REMOVE_WS` removes white space, including carriage returns, from its argument while `.ASSIGN` assigns its argument's value to a variable. For `.IF` conditionals, `.NULL` as an action does nothing.

As can be seen in the example, user-defined variables are used to behave as subroutines:

<code>afd\$init</code>	Initialize variables to known values.
<code>afd\$check_objtyp</code>	Determine which object the user selected to operate on.
<code>afd\$return</code>	Return to <i>teamwork</i> with a message, aborting the current menu operation.
<code>afd\$status</code>	Retrieve the status text from the file described by the variable <code>afd_stsf</code> .
<code>afd\$afd_afdnam.get</code>	Get an AFD name from the user and check its legality.
<code>afd\$return_on_error</code>	Return to <i>teamwork</i> in case of an error (<code>afd\$status</code> does not return "Success"); the error text returned by <code>afd\$status</code> is displayed.

6.3 Impressions

Developing with the *teamwork* Extensibility Language is not efficient. The language is not very readable and the need to devise communication techniques between the *teamwork* Extensibility Language and host system scripts is an obstruction to productivity. In addition, there are no debugging facilities available!⁵ Coding was improved by using variable references to nest "subroutine" calls.

Nevertheless, the design, implementation, and testing of TWKAFD took only one person-month. This included the time spent learning the *teamwork* Extensibility Language. Considerable time could have been saved if the *teamwork* documentation [8] was clearer and extended examples (more than one line) were provided.

7 Results

It is important to classify where our successes and failures lie. The methodology of using Architecture Diagrams to supplement and enhance our use of DFDs for structured analysis has been successful and quite fruitful. On the other hand, the use of the *teamwork* CASE tool with its limited ability to be extended was not an unqualified success.

⁵The Beta release did dump somewhat useful information when it encountered any error within the *teamwork* Extensibility Language code. But, this "feature" disappeared with the standard release.

7.1 The Methodology

The Architecture Diagrams did provide us with a useful alternate view of the Digital Sky Survey data acquisition system. Our initial uses of this methodology saw software developers specifying the data acquisition system using DFDs; the hardware-oriented engineers used Architecture Diagrams. The hierarchical decomposition of the system did not necessarily go down to the same levels between DFDs and AFDs. Usually, AFDs decomposed further down than the DFDs allocated to them.

Our meetings showed that each view of the system exposed flaws in the other view. For example, the data acquisition system had hardware constraints placed on it prior to system specification. These constraints were easily incorporated into the AFDs/AIDs, but did not surface in the initial DFDs. Yet, it was important that the process structure reflect these real-world constraints. The iterative feedback between DFDs and AFDs allowed us to converge on two system views that specified the same system.

7.2 The Tools

The tools used to facilitate the use of both the DFD and AFD methodologies did not work as smoothly. They did allow multiple users to work concurrently on the same project. Both *teamwork* and TWKAFD provide exclusive access to the objects being edited. Both tools also enforce some formalism in the use of the methodologies (although users must be careful not to let the tool define the methodology). Still, problems were encountered:

- The XFIG Utility is inadequate for drawing and manipulating AFDs and AIDs. The greatest deficiency is that there are no AFD/AID constructs. As mentioned earlier, the user needs to use simple drawing primitives to build up an AFD/AID item. Moving or modifying these constructed structures is a cumbersome and sloppy process.

XFIG is also inadequate as it does not provide any means to allow the user to correlate AFD information flow vectors with AID information flow channels. This information still needs to be maintained by hand.

- The lack of automatically updating bookkeeping operations when changes in a DFD affect AFDs and the AMS or vice-versa prevented the full use of *both* tools. During initial stages of specification, many changes were necessary. These were

not done as quickly as they should have been because of the reluctance to manually update many affected structures.

Problems such as these can be remedied if *teamwork* did not limit its Extensibility Language to just providing new menu items. Instead, if it were possible to extend the *teamwork* editors, software could be developed to automate much of the TWKAFD tool. For example, when a DFD bubble (process) is deleted, the DFD editor can notify TWKAFD by calling a TWKAFD-supplied routine.

The resolution to these problems is to have the users of these tools work consistently, especially between each other.

The ability to extend a vendor-supplied tool to incorporate a new methodology is a powerful capability that can enhance a user's performance. However, in this case, there is considerable room for improvement in the *teamwork* product.

References

- [1] Tom DeMarco, *Structured Analysis and System Specification*, YOURIDN Press, 1979.
- [2] *A Digital Sky Survey of the Northern Galactic Cap*, 1991.
- [3] Derek J. Hatley and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, 1988.
- [4] *Ibid.*, p. 19.
- [5] *Ibid.*, p. 25.
- [6] Tom Nicinski and Bill Burt, *Architecture Flow Diagrams under teamwork*,[®] Fermilab PN 449, 1991.
- [7] *Teamwork/SIM[™] User's Guide*, Release 4.0, Cadre Technologies, Inc., 1990.
- [8] *Teamwork[®] User Menus User's Guide*, Release 4.0, Cadre Technologies, Inc., 1990.