Data Control Model

Legal Information

ETAC and  (the ETAC logo) are an unregistered trademarks (™) of Victor Vella for *computer software incorporating an implementation of a computer programming language*.

UML is a registered trademark (®) of the Object Management Group, Inc.

The author of this document shall not be liable for any direct or indirect consequences arising with respect to the use of all or any part of the information in this document, even if such information is inaccurate or in error. The information in this document is subject to change without notice.

Data Control Model

Victor Vella

Published by Victor Vella
(13 February 2025)

First Published: 13 February 2025 (Australian English)

Preface

I invented the data control model and corresponding data control diagrams in draft form in the year 1997 for a specialised complex multi-tasking real-time system that I was designing in a work place. However, I needed a means of representing the system visually from which I could design the software. At the time, I was unfamiliar with (or possibly never heard of) UML® diagrams (which were almost useless anyway).

My conceptual thinking about software systems consisted of instructions executing in a sequence, changing data as they execute. The execution of the instructions involved the concept of current execution control. I conceptualised the objects in the system as data with code (as in OOP). Therefore the execution control could move from one object to another, perhaps taking some data with it, then returning, perhaps with some data, to continue with the next instruction.

I quickly realised that all that I required to fully describe a software system is a sequence of data changes in that system. Therefore, I conceptually eliminated the instructions themselves (coding) and retained only the data changes of the system. But, I also retained the concept of execution control (as a control point) to indicate the order and location of the data changes. Thus, the concept of the data control model of a system was born — an entirely original work.

I developed some graphic symbols for the idea, enabling me to design the said real-time system. Years later, for publication (this one), I developed the data control model system in a more formal fashion, which I had previously used extensively, with great success, for developing the ETAC™ interpreter. I also used the data control model and diagrams for many other personal software projects to design and maintain the software.

Victor Vella

Perth, Western Australia
13 February 2025

Contents

Document Conventions

The following conventions are used in this document.

Document Conventions

Symbol	Meaning
‹*x*›	separates *x* as a unit of information from the surrounding text.
x···	middle ellipsis means zero, one, or more of the same kind as *x*.
[*x*]	means that *x* optional.
...	ellipsis represents omitted text (as usual).
text	maroon coloured italic text is a link to the text's definition.
text	underlined green text is a link into the document.

Introduction

A model is a concise representation of some desired aspects of a system or process. An appropriate correspondence exits between parts of the model and parts of the system or process for various purposes such as:

- to make inferences (especially predictions) with respect to the system or process;
- to gain a concise understanding of the nature of some aspect of the system or process; or,
- to efficiently make modifications (especially adaptations and enhancements) to the system or process.

The reason for having a model of a system or process is that it is easier and more efficient to draw conclusions from the model rather than directly from the system or process itself. A model can also be used to help in the design and implementation of a system or process. In a loose sense, a model is an analogy of some part of a system or process.

A model can be either physical or theoretical. A physical model typically resembles the actual object of the model, whereas a theoretical model typically does not. A theoretical model:

- consists of precisely defined abstract entities;
- is typically represented by symbols (graphical, written, or both); and,
- may conform to a set of definitions representing all similar models.

Mathematical models are a particular class of theoretical models. A mathematical model is specified in terms of mathematical functions and relations typically involving some form of numbers. A **data control model** (which may be shortened to **DCM**) is a theoretical model, not a mathematical model, and can be used for any system that requires a concise representation of its discrete data changes.

A *DCM* indicates some or all of the following data changes within a system.

- Where the data changes occur within the system.
- The nature of the changes.
- The sequence (temporal order) of the data changes (includes concurrent and alternative changes).
- The causes of the data changes.

A *DCM* is not intended to represent data flow as such. However, it could be used to show data flow, but it is not optimised for that purpose.

Any *DCM* is based on a single *DCM* definition (the definition itself is not a model). Although the *DCM* definition can be used to create a specific model of the discrete data changes within any specific system, the definition is optimised to create models of data changes within software systems in particular, including software systems that are distributed over a network (for example client\server systems) as well as stand-alone application programs.

Data Control Diagrams for Software Systems

A software system can essentially be described in terms of groups of data states, where only one state of each group can be current at any moment in time. The current state of a particular data group changes from one state to another at various moments in time. The changes are not random, but are related to one another in some specific way. In the final implementation of a software system, the data groups are typically represented as variables and constants. The current values of the variables for a particular data group represent the current data state of that group.

Such a system is conveniently represented by a "data control diagram" (**DCD**) which represents the *DCM* of a system in terms of graphical symbols and text. Representing all the data states of a complex software system at each moment of time is not viable. Fortunately, the data state changes that occur in a software system are localised changes, meaning that much of the data state changes occurring at a particular moment in time does not affect the other data in the system. The data control model of a software system takes advantage of this localisation while still representing the dynamic nature of the whole system. Another method of representing the data of a software system is by means of a data flow diagram. A data flow diagram, however, does not accurately represent the dynamic nature of the system — a data control diagram does.

A *DCD* of a software system can be used to:

- determine where, in the system, new modifications and enhancements are to be made;
- determine the dependencies of proposed modifications and enhancements for the system;
- determine the location of given data changes, including malfunctions (bugs), within the system;
- trial different architectures of a new system to determine which one is best before implementing the system;
- provide a quick and concise understanding of a system without having to read any source code;
- generate the source code outline for a new computer program using a software tool (if one is available);
- efficiently identify potential design flaws in the system; and,
- concisely represent the data changes and their dependencies of an existing system.

The *DCM* definition needs to be understood before a *DCD* can be understood. This document presents the *DCM* definition before describing how to represent an instance of that definition by means of the symbols in a *DCD*. Note that the *DCM* definition is not itself a model; the instances of the *DCM* definition are the models. An instance of the *DCM* definition is any system (typically represented by appropriate symbols) that conforms to the definition.

1

Data Control Model

The data control model (*DCM*) of a system is a theoretical model that represents the important data changes that the system (typically a software system) undergoes at different moments in time possibly relative to other data changes in the system. The model does not represent the actual time of the data changes but only the sequence of data changes within the system. A data control model can also be used to represent the data structures of a software system, along with the data changes within those structures during the operation of the system.

The model does not directly indicate how the information in the system is implemented. For a software system, there is no mention of functions, procedures, variables, programming languages, tasks, files, disks etc, except perhaps only for reference to the implemented system. However, the *DCM* of a particular software system may be influenced by the fact that it is a software system that the model describes.

The adaptability of a software system can be reflected in the design of the *DCM* of that system. The model then provides the criteria by which the software system can later be modified and enhanced. A *DCM* that takes into account the adaptability of a software system will lower the risk of the software later being modified in an ad-hoc fashion, and will also make it easier to locate where changes should occur in the software.

A *DCM* can also be used to describe the functioning of a system that does not have an implementation. In such a case, or in the case that the implementation is not being considered, the *DCM* is said to be "of" the system. In the case that the system does have an implementation that is being considered, the *DCM* is said to be "for" the system.

1.1 Data Control Model Definition

The data control model definition is a single definition for all *DCMs*, and consists of a number of definitions of various abstract entities that are optimally designed to be used with software systems. An actual *DCM* is anything that corresponds to instances of those abstract entities.

There are two broad categories of abstract entities in the definition — *domains* and *connectors*. A *connector* indicates a certain relation between two *domains*. In addition, there are also imaginary entities called *control points* which "move" along certain *connectors* from one *domain* to another, perhaps causing a data change in the *domains* that they move to or from. The details of the data changes in the *domains* are described in *channel records*.

The definitions of the various abstract entities of the data control model definition are presented in the following paragraphs.

1.2 Domains

A 'domain' is an entity that contains some sort of data, and may affect *control points* entering it, depending on its type. There are six types of *domains* in the data control model definition: *data domain*, *no-wait divider domain*, *wait divider domain*, *linked-wait divider domain*, *wait domain*, and *block domain*. Each instance of a *domain* in a *DCM* is uniquely identifiable.

1.2.1 Data Domain

A ‘data domain’ (sometimes referred to as just ‘domain’ when the context is understood) is a *domain* that contains data units, each having a range of possible values. The actual data units within a *data domain* may be, but need not be, specified by the *DCM* designer.

A *data domain* in the *DCM* of a system indicates a specified subset of the data in the system such that any changes that the subset undergoes, within any period of time, does not directly affect and is not directly effected by data elsewhere in the system (except for predefined sections of data within that subset which are defined especially to affect other data in the system). The data in a *data domain*, then, is not recognised outside of that *domain* (except for certain sections of the *domain*, which are termed the ‘exposed data’ of the *domain*).

A *data domain* contains units of data (‘data units’). Each *data unit* can have one value at a time of a certain number of possible values. The particular *data units* and the range of their possible values in a *data domain* are collectively called the ‘structure’ of the *domain*. For example, a *data domain* may contain a person’s name and age (the *data units* in the *domain*). Another *data domain* containing a person’s name, age, and address, for example, has a different *structure* to the first *domain*; if another *data domain* contains a person’s name and age but the range of possible names is different from the first *domain*, then this *domain* also has a different *structure* from the first *domain*.

The data in a *data domain*, throughout its life, can potentially undergo a number of sequences of changes. The set of such sequences is called the ‘sequence set’ of the *data domain*.

The *structure* and *sequence set* of a *data domain* uniquely defines that *domain*. That is to say that if two *data domains* have either different *structures* or *sequence sets* then they are different *data domains*.

The data in a *data domain* persists indefinitely by default unless otherwise indicated.

A *data domain* has the following characteristics:

- At any particular moment, each *data unit* must contain any one (and only one) of its possible values.
- The value of a *data unit* may change from one moment to another.
- The *data units* in a *data domain* are permitted to change only when a *control point* is in that *domain* unless otherwise indicated.
- A *data domain* must be associated with at least one *incoming channel* and zero or more *outgoing channels*.

A *DCM* must have at least one *data domain*.

1.2.2 No-wait Divider Domain

A ‘no-wait divider domain’ (or ‘no-wait divider’ for short) is a *domain* that allows new *control points* to be *spawned* from existing ones without those existing ones waiting for the new *control points* to return back to the *domain*. A *no-wait divider* must be associated with at least one *incoming channel* and at least one *outgoing channel*.

A *no-wait divider* operates as follows:

- For an *incoming control point*, a new *control point* is created on each *outgoing channel*. The order that the new *control points* are created is undefined.
- An *incoming control point* returns to its *source domain* at the earliest opportunity after all the new *control points* have been created (whether or not any new *control point* has returned).
- Each *outgoing control point* is destroyed after returning to the *no-wait divider*.

Note that an *incoming control point* cannot itself proceed to an *outgoing channel*, but must eventually return to its *source domain*.

1.2.3 Wait Divider Domain

A 'wait divider domain' (or 'wait divider' for short) is a *domain* that allows *control points* to be *spawned* from an existing one but waits for the new *control points* to return to the *domain* before the *parent control point* can return. A *wait divider* must be associated with at least one *incoming channel* and at least one *outgoing channel*.

A *wait divider* operates as follows:

- For an *incoming control point*, a new *control point* is created on each *outgoing channel*. The order that the new *control points* are created is undefined.
- Each *outgoing control point* goes into an *idle state* after returning to the *wait divider*.
- An *incoming control point* returns to its *source domain* after all *outgoing channels* each have an *idle control point* after returning to the *wait divider*. In this case, the *idle control points* are destroyed before the *incoming control point* returns to its *source domain*.

Note that an *incoming control point* cannot proceed to an *outgoing channel*, but must eventually return to its *source domain*.

1.2.4 Linked-wait Divider Domain

A 'linked-wait divider domain' (or 'linked-wait divider' for short) is a *domain* that allows *control points* to be *spawned* from an existing one, and is linked to one or more *wait domains*. A *linked-wait divider* must be associated with at least one *incoming channel* and at least one *outgoing channel*.

A *linked-wait divider* operates as follows:

- For an *incoming control point*, a new *control point* is created on each *outgoing channel*. The order that the new *control points* are created is undefined.
- Each *outgoing control point* goes into an *idle state* after returning to the *linked-wait divider*.
- An *incoming control point* operates as described under 1.2.5 Wait Domain.

Note that an *incoming control point* cannot proceed to an *outgoing channel*, but must eventually return to its *source domain*.

1.2.5 Wait Domain

A 'wait domain' is a *domain* that causes an *incoming control point* to wait for the new *control points spawned* by all of the associated *linked-wait dividers* to return to their respective *dividers* before the *incoming control point* returns to the *domain*.

A *wait domain* has the following characteristics:

- It must be associated with only one *incoming channel*, which can optionally have a timeout value.
- It must be associated with no more than one *outgoing channel*.
- It must be linked to at least one *linked-wait divider*.
- It can be linked only to *linked-wait dividers*.

A *wait domain* operates as follows:

- An *incoming control point* will become *idle*. If the *incoming channel* does not have a timeout value, then the *incoming control point* either continues along the *outgoing channel* (if one

exists) or returns along the *incoming channel* (if an outgoing one does not exist) when either of the following two conditions are satisfied, simultaneously for all *outgoing channels* of all the associated *linked-wait dividers*:

- a control point on an *outgoing channel* of an associated *linked-wait divider* is in the *idle state* after returning, or,
- there is no *control point* on an *outgoing channel* of an associated *linked-wait divider*.

- The returned *idle control points* on the *outgoing channels* of the associated *linked-wait dividers* are destroyed before the *incoming control point* of the *wait domain* can return.
- The *incoming control point* does not wait in the *wait domain* after returning from the *outgoing channel* (if one exists).
- The timeout value associated with the *incoming channel* (if such a value exists) of a *wait domain* is the maximum number of milliseconds that the *incoming control point* is allowed to remain *idle*. After the timeout value has elapsed, the *incoming control point* goes to the *ready state* regardless of the *states* of the *outgoing control points* of the *linked-wait dividers*. The default timeout value is infinite.

1.2.6 Block Domain

A ‘block domain’ is a *domain* that causes an *incoming control point* to be *blocked* until a *control point* from a specially indicated *channel* enters the *block domain*.

A *block domain* has the following characteristics:

- It must be associated with at least one *incoming channel*, which can optionally have a timeout value.
- It can be associated with zero or more *outgoing channels*.
- It can be associated with zero or more specially indicated *incoming channels*.

A *block domain* operates as follows:

- A *control point* from an *incoming channel* (but not an indicated *channel*) entering a *block domain* is *blocked* until another *control point* from an indicated *channel* enters that *domain*. That indicated *channel* is called a ‘u-channel’ (the ‘u’ stands for “unblock”). The *blocked control point* then becomes *ready* and continues along an *outgoing channel* if there is one; the *control point* of the *u-channel* returns to its *source domain*. There can be more than one *u-channel*.
- The continuing *control point* is *blocked* when it returns back to the *block domain* (or remains *blocked* if there is no *outgoing channel*). A second *control point* from a differently indicated *channel* entering the *block domain* causes the *blocked control point* to return to its *source domain* after becoming *ready*. That other indicated *channel* is called an ‘r-channel’ (the ‘r’ stands for “return”). The *control point* of the *r-channel* returns to its *source domain*. There can be more than one *r-channel*.
- A *control point* moving along a *u-channel* or *r-channel* sets one of two internal *signals* (a ‘u-signal’ or ‘r-signal’, respectively) that is accessed by any one of the *blocked control points*, which then acts accordingly (becomes *ready* before it unblocks or returns) after the appropriate *signal* is reset. Note that a *u-signal* or *r-signal* can be set before a *control point* to be *blocked* enters (or returns to) the *domain*; the *control point* then acts immediately based on the *signal* as soon as it enters.
- The *control point* on an *outgoing channel*, after returning, can immediately move along another (designated) *outgoing channel*, then return to move along yet another *outgoing channel*, and so on, before it finally returns to the *block domain* and gets *blocked*.
- The timeout value associated with an *incoming channel* (if such a value exists) is the maximum number of milliseconds that the *incoming control point* is allowed to be *blocked* before it continues or returns. The timeout value and its *incoming channel* are associated with a *u-signal* or *r-signal* or both. When the timeout period has elapsed, the associated

signal is set and the *incoming control point* acts appropriately after the *signal* is reset. The default timeout value is infinite.

Note that the same indicated *channel* can be both a *u-channel* and an *r-channel*. Also note that if a *block domain* has no *u-channels*, *r-channels*, or timeout values, then *incoming control points* will be permanently *blocked*.

1.3 Connectors

There are eight types of *connectors* in the *DCM* definition. A *connector* (except for a *link assignment*) can associate any two *domains* for a particular purpose depending on the *connector* type. One of the two *domains* is called a '*source domain*', and the other is called a '*destination domain*'. The *source* and *destination domains* of a *connector* must not be identical unless stated otherwise. A *link assignment* only has a *source domain*. The other end of a *link assignment* is attached to an appropriate *connector*. Each *connector* in a data control model is uniquely identifiable.

A *DCM* must have at least one *connector*.

1.3.1 Control Point Channel

A 'control point channel' (or 'channel' for short) is a type of *connector* that "connects" (associates) two *data domains* (the *source domain* and *destination domain*), allowing *control points* to "move" from the *source domain* to the *destination domain* and back. The *source* and *destination domains* are relative to the *channel*, and may be identical.

A *control point channel* may be associated with a single *control gate*, and may also allow a *control point* to repeatedly move through and return from the *channel* in succession if so indicated in both cases.

1.3.2 Static Domain Connector

A 'static domain connector' is a type of *connector* that indicates that the *destination domain* is created automatically by the *source domain* when the *source domain* is created (which must occur before any *control point* moves to the *source domain*). A *control point* in the *source domain* can access the data in the *destination domain*. If the *source domain* is deleted, then the *destination domain* is automatically deleted as well. The *source* and *destination domains* must be *data domains*, and cannot be identical.

1.3.3 Dynamic Domain Connector

A 'dynamic domain connector' is a type of *connector* that indicates that a *control point* in the *source domain* creates the *destination domain* before moving (along the *connector*) to that *domain*. The *source domain* has an automatic reference to the *destination domain* unless otherwise indicated. The *destination domain* cannot be created again unless it is first deleted. The *source* and *destination domains* must be *data domains*, and cannot be identical.

1.3.4 Domain Deletion Connector

A 'domain deletion connector' is a type of *connector* that indicates that a *control point* in the *source domain* deletes the *destination domain* after returning from that *destination domain*. Once the *destination domain* is deleted, no *control point* can move to it unless a new *destination domain* is created (via a *dynamic domain connector*). The *source* and *destination domains* must be *data domains*, and can be identical.

1.3.5 Supplied Domain Connector

A ‘supplied domain connector’ is a type of *connector* that indicates that a reference to the whole *source domain* is temporarily supplied to the *destination domain*, possibly by a *control point* in some other *data domain*. The reference allows the *destination domain* to have temporary access to the whole *source domain*. The *source* and *destination domains* must be *data domains*, and cannot be identical.

1.3.6 Reference Domain Link

A ‘reference domain link’ is a type of *connector* that indicates that there exists a reference from the data in the *source domain* to the *destination domain*; the reference is possibly created by a *control point* in some other *data domain*. Such a reference implies that a *control point* in the *source domain* has access to specific data in the *destination domain*. If the *destination domain* ceases to exist, then the reference becomes invalid. The *source* and *destination domains* must be *data domains*, and cannot be identical.

1.3.7 Divider Link

A ‘divider link’ is a type of *connector* that associates a *wait domain* and a *linked-wait divider domain*. The *wait domain* is the *source domain*, and the *linked-wait divider domain* is the *destination domain*.

1.3.8 Link Assignment

A ‘link assignment’ is a type of *connector* that indicates the creation of a *reference domain link* between two *data domains*. A *link assignment* is associated with a *source domain*, but not a *destination domain*. The *source domain* must be a *data domain*, which creates the *reference domain link* between the two other *data domains*. The destination end of a *link assignment* is associated with the *reference domain link*.

1.4 Data Control Points

A ‘data control point’ (or ‘control point’ for short) is an imaginary entity that signifies changes in the data state of different *domains* at different points in time. There may be more than one *control point* in a data control model at the same time, but any particular *control point* cannot exist in more than one *domain* at the same time. Every *control point* must originally exist in a designated *data domain*, or be created at a *divider domain*. A *control point* moves from one *domain* to another along a *route*, but once the *control point* begins moving along the *route*, it must complete that *route*, returning back to the *domain* that created it.

A *data control point* is imagined to “move” almost instantaneously from one *domain* (the *source domain*) to another (the *destination domain*) along a *passage*, possibly copying some of the data from the *source domain* to the *destination domain*. Then, at some future time, the *control point* must return back to the *source domain* along the same *passage*, possibly copying some of the data from the *destination domain* back to the *source domain*. As a *control point* moves from the *source domain* to the *destination domain*, or vice versa, it may cause a change to the data state of those *domains* after it enters or before it leaves them. A *control point* on a *passage* can only be in transit; it cannot remain on that *passage*.

Any data that is copied from the *source domain* to the *destination domain*, as a *control point* moves along a *passage*, remains persistent in the *destination domain* until just before the *control point* returns from that *destination domain*. Just before the *control point* returns, the copied data is destroyed by default, unless the *passage* is designated to not destroy it.

Movement of a Data Control Point

All of the following conditions apply to the movement of any *data control point*:

- The *control point* can move from one *domain* to another only via a *passage* connecting those two *domains*.
- The *control point* must move from its *source domain* to its *destination domain* along a *passage* before it can return from its *destination domain* back to its *source domain* along the same *passage*.
- Once the *control point* has returned from its *destination domain* to its *source domain* along a *passage*, it cannot subsequently move from its *destination domain* to its *source domain* again along the same *passage* without first moving from its *source domain* to its *destination domain* along that *passage*.
- The *control point* cannot move from its *source domain* to its *destination domain* and return, and then immediately (at the next moment) move to its *destination domain* again along the same *passage* unless that *passage* is designated for such repeated movements.

States of a Data Control Point

A *control point* can be in only one of the following states at any moment while it exists:

- **active**: the *control point* is moving along a *route*.
- **idle**: the *control point* is not *active* and not *ready* and not *blocked*.
- **ready**: the *control point* is not currently *active* but can become *active* at any moment.
- **blocked**: the *control point* is waiting (not *ready* and not *active*) for an appropriate event before it can become *ready* or *active*.

1.4.1 Spawned Control Points

All (and only) *divider domains* allow *spawned control points*. When a *control point* is *spawned*, the new *control point* is referred to as the 'child' *control point* of the spawner *control point*, which is referred to as the 'parent' *control point*. If a *parent control point* ceases to exist while its *child control point* still exists, whether the *child control point* also ceases to exist or continues to exist is undefined (the *DCM* designer needs to account for both possibilities). The *spawned control points* are necessarily *concurrent* with each other and the *parent control point*.

1.4.2 Concurrent Control Points

Two or more *control points* are said to be 'concurrent' if they exist simultaneously. *Control points* are *concurrent* in the following situation: two or more *control points* exist simultaneously on the same *path*, and\or two or more *paths* each have at least one *control point* on them. One or more groups of *control paths* can be designated to allow *concurrent control points* to move along those *paths* ('*path groups*').

Concurrent control points obey the following rules:

- The *paths* of *outgoing channels* of a *divider* automatically belong to the *path group* of the *parent control point*.
- If a *path group* has *concurrent control points* on its *paths* then no other *control point* can exist anywhere else (not even on the *paths* of other *path groups*) while any of those *concurrent control points* still exist.
- If a *control point* exists on a *path* that does not belong to any *path group*, then no other *control point* can exist anywhere else while that *control point* still exists.
- *Paths* that do not belong to any *path group* cannot have *concurrent control points* on or among them.
- No two *control points* can be *active* simultaneously.
- The positioning of *concurrent control points* relative to each other at any moment is undefined.

The last point is important. Consider any two *concurrent control points* (whether they are on the same *path* or not). Whichever *domain* one *control point* is in, the other *control point* can simultaneously be in any one of the *domains* of its *path* — which one it is in is undefined. This implies that the *DCM* designer needs to design the *DCM* to take into account all possible positions of each *concurrent control point*. Note that a *DCM* designer can use appropriate *domains*, such as *wait dividers*, *linked-wait dividers*, *wait domains*, and *block domains*, and also use *control gates* to synchronise the positioning of *concurrent control points* relative to each other.

Note that if two or more separate *paths* that cannot have simultaneous *control points* are required to have such simultaneous *control points*, then those separate *paths* need to be designed to incorporate *dividers* to make those *paths* a single *path group*.

1.5 Channel Record

A *channel record* describes the changes that a *control point* causes in its *source* and *destination domains* as it moves along the *channel* connecting those two *domains*. It also describes any data that is copied between the *domains*. There is a *channel record* for each *channel* in a *DCM*.

The following information may be present for each *channel record*.

1. An identification of the *channel* for which the entry is about. The identification is a *passage tag*.
2. The data change in the *source domain* (if any) before the *outgoing control point* moves to the *destination domain*.
3. The data that the *outgoing control point* copies from the *source domain* to the *destination domain* (if any) as it moves to the *destination domain*.
4. The data change in the *destination domain* (if any) after the *outgoing control point* moves to that *domain*.
5. The data that the *incoming control point* copies from the *destination domain* to the *source domain* (if any) as it returns to the *source domain*.
6. The data change in the *source domain* (if any) after the *incoming control point* returns to that *domain*.

2

Data Control Diagram

A *DCD* (data control diagram) is a representation of the *DCM* (data control model) of a system using graphical and textual symbols. A *DCD* has no detailed information relating to the implementation of the system but may indicate the enhancement scope of a system. That is to say, a *DCD* can indicate certain restrictions that should be adhered to when making modifications to enhance the system. However, a *channel record* may contain references to the implementation of the system.

It is important to realise that a *DCM*, and therefore a *DCD*, does not explicitly represent any hierarchical information in relation to the *data domains*.

A *DCD* consists of graphical and textual symbols representing elements of the corresponding *DCM*, and also contains graphical symbols that are particular to data control diagrams in general. Those symbols are typically interpreted with respect to the system for which the *DCD* is designed — in effect, a *DCD* symbolises the overall operation of the system.

2.1 Symbols of a Data Control Diagram

The following subparagraphs specify the graphical symbols and text used in a *DCD*. All measurements are given in "units". One unit is 12 printer points (or one pica) for a standard size *DCD*. No part of any symbol may overlap any part of another symbol unless stated otherwise. Text in angle brackets, < and >, is not a literal part of a symbol, but represents a template for actual text that may be used with a symbol. The default foreground colours are black (RGB: 0, 0, 0), and the default backgrounds are transparent.

2.1.1 Data Domains

Each of the following symbols represents a *data domain* under various circumstances.

Origin Data Domain Data Domain External Data Domain

Description

Origin Data Domain: Indicates the *data domain* containing the first *passage* of a *route*. <label> is a noun phrase description of the *domain*, written in bold type.

Data Domain: Indicates a regular *data domain*. <label> is a noun phrase description of the *domain*.

External Data Domain: Indicates a *data domain* on a different *diagram sheet* labelled <sheet ID>, but the *domain* is not itself part of the *domains* intended in the current *sheet*. <label> is the name of the *domain* on the other *sheet*.

Specifications

Width and Height:	Varies with the <label> text and user's choice.
Border Thickness	
Origin Data Domain:	0.12 units
Data Domain:	0.12 units
External Data Domain:	0.05 units

Corner Radius:	0.5 units
Border Colour Origin Data Domain: Data Domain: External Data Domain:	 Black (RGB: 0, 0, 0) Black (RGB: 0, 0, 0) Medium grey (RGB: 160, 160, 160)
Background Colour Origin Data Domain: Data Domain: External Data Domain:	 Grey (RGB: 225, 225, 225) White (RGB: 255, 255, 255) White (RGB: 255, 255, 255)
<label> Face Name:	Arial
<label> Height:	1.0 unit (character height)
<label> Weights Origin Data Domain: Data Domain: External Data Domain:	 Bold Normal Normal
<label> Position:	Centred
<sheet ID> Face Name:	Arial
<sheet ID> Height:	0.5 units (character height)
<sheet ID> Horizontal Position:	Centred
<sheet ID> Distance to Top from Symbol Centre:	–0.3 units

Domain Indicators

The following symbols indicate certain characteristics of the *data domain* containing them. They apply only to Origin Data Domain and Data Domain. The symbols are located at the bottom centre of the *domain* symbol.

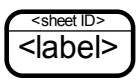
• *x* **Solo Indicator** (U+2022). Indicates that a group of *data domains* cannot have more than one *control point* in them collectively at any moment. The *x* is a number that groups the *domains*. The *x* is optional if the group consists of only one *domain*.

☼ **Cloning Indicator** (U+263C). Indicates that some of the data in the *data domain* is cloned just before a *control point* moves to the *domain*. The cloned data is written back to the original data just before the *control point* returns from the *domain*.

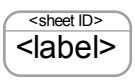
(**Ref**: 1.2.1_Data Domain)

Pseudo Data Domains

The following symbols are not of actual *data domains* as such, but represent actions similar to that of *data domains*. They represent *data domains* in a different *diagram sheet*. Pseudo data domain symbols are defined only for a *DCD*; they are not defined for a *DCM*.



Sheet Reference Data Domain



External Sheet Reference Data Domain

External Multiple Data Domain

Description

Sheet Reference Data Domain: Indicates the whole *DCD* on a different *diagram sheet* labelled <sheet ID>, but that *DCD* is itself part of the *domains* intended in the current *sheet*. <label> is optional, and is a noun phrase description of the *DCD*.

External Sheet Reference Data Domain: Indicates the whole *DCD* on a different *diagram sheet* labelled <sheet ID>, but that *DCD* is not itself part of the *domains* intended in the current *sheet*. <label> is optional, and is a noun phrase description of the *DCD*.

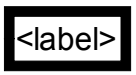
External Multiple Data Domain: Indicates more than one *domain*, but not all *domains*, on a different *diagram sheet* labelled <sheet ID>, but the *domains* themselves are not part of the *domains* intended in the current *sheet*. <label> is optional, and is a noun phrase description of the collective *domains*. The actual *domains* represented by the symbol are determined by the *passage tags* of the *passages* connected with the symbol.

Specifications

Width and Height:	Varies with the <label> text and user's choice.
Border Thickness	
Sheet Reference Data Domain:	0.12 units
External Sheet Reference Data Domain:	0.08 units
External Multiple Data Domain	
Outside:	0.08 units
Inside:	0.05 units
Gap between Inner and Outer Borders to Midlines	
External Multiple Data Domain:	0.3 units
Corner Radius	
Sheet Reference Data Domain:	0.5 units
External Sheet Reference Data Domain:	0.5 units
External Multiple Data Domain	
Outside:	0.5 units
Inside:	0.4 units
Horizontal Line Thickness:	0.05 units
Horizontal Line Distance from Top:	0.6 units
Border Colour	
Sheet Reference Data Domain:	Black (RGB: 0, 0, 0)
External Sheet Reference Data Domain:	Medium grey (RGB: 160, 160, 160)
External Multiple Data Domain:	Medium grey (RGB: 160, 160, 160)
Background Colour:	White (RGB: 255, 255, 255)
<label> Face Name:	Arial
<label> Height:	1.0 unit (character height)
<label> Position:	Centred
<sheet ID> Face Name:	Arial
<sheet ID> Height:	0.5 units (character height)
<sheet ID> Horizontal Position:	Centred
<sheet ID> Distance to Top from Symbol Top	
Sheet Reference Data Domain:	0.0 units
External Sheet Reference Data Domain:	0.0 units
<sheet ID> Distance to Top from Symbol Centre	
External Multiple Data Domain:	–0.3 units

2.1.2 Block Domain

The following symbol represents a *block domain*.



Block Domain

Description

- The *u-channel* has the letter ‹u› near the *block domain*.
- The *r-channel* has the letter ‹r› near the *block domain*.
- The same *channel* can have both ‹u› and ‹r›, in that order (ie: ‹ur›).
- An *incoming channel* can have a timeout value, represented as ‹*x* (*ms*)›, where *ms* is the maximum number of milliseconds that the *incoming control point* is allowed to be *blocked* before it either continues (*x* is ‹u›) to the next *domain* or returns (*x* is ‹r›) to its *source domain*, or both (*x* is ‹ur›). *ms* can be a question mark (?), indicating that the timeout value exists but is not specified in the *DCD*. The ‹(*ms*)› portion is optional, and its omission indicates that the timeout value is infinite. The text (if it exists) is written near the *destination domain* end of the *incoming channel* at the same position as a *passage tag* (see 2.1.15_Passage Tags and References for details).
  - Example: ‹u (250)› for an *incoming channel* means that the corresponding *u-signal* will be automatically set 250 ms after a *control point* on that *channel* enters the *domain* while it (the *control point*) still has not continued to the next *domain*, and the *u-signal* has not yet been set. The *control point* may continue before 250 ms if a *u-signal* is set by another appropriate *control point* before that time; the timeout value is then cancelled.
  - Example: ‹r (10)› for an *incoming channel* means that the corresponding *r-signal* will be automatically set 10 ms after a *control point* on that *channel* enters the *domain* while it (the *control point*) still has not returned to its *source domain*, and the *r-signal* has not yet been set. The *control point* may return before 10 ms if an *r-signal* is set by another appropriate *control point* before that time; the timeout value is then cancelled.

Specifications

Width and Height:	Varies with the <label> text and user's choice.
Border Thickness:	0.3 units
Background Colour:	Transparent
<label> Face Name:	Arial
<label> Height:	1.0 unit (character height)
<label> Position:	Centred
Face Name of Tags (u, r, ur):	Courier New
Height of Tags (u, r, ur):	0.5 units (character height)
Background Colour of Tags (u, r, ur):	White (RGB: 255, 255, 255)

(**Ref**: 1.2.6_Block Domain)

2.1.3 No-wait and Linked-wait Divider Domains

The following symbols represent both a *no-wait divider domain* and a *linked-wait divider domain* in two orientations. A *linked-wait divider domain* is identified by being associated with a *wait domain* via a *divider link*, otherwise the *no-wait divider* and *linked-wait divider* symbols themselves are identical. The diagram below represents both types of *dividers*.

Vertical *No-wait Divider* and *Linked-wait Divider*

Horizontal *No-wait Divider* and *Linked-wait Divider*

Description

Vertical No-wait Divider Domain: This orientation has all the *incoming channels* on either the left or right side and all the *outgoing channels* on the other side.

Horizontal No-wait Divider Domain: This orientation has all the *incoming channels* on either the top or bottom side and all the *outgoing channels* on the other side.

Specifications

Length:	User's choice.
Thickness:	0.4 units
Orientation:	User's choice (horizontal or vertical).

2.1.4 Wait Divider Domain

The following symbols represent a *wait divider domain* in two orientations.

Vertical *Wait Divider Domain*

Horizontal *Wait Divider Domain*

Description

Vertical Wait Divider Domain: This orientation has all the *incoming channels* on either the left or right side and all the *outgoing channels* on the other side. The two boxes are square.

Horizontal Wait Divider Domain: This orientation has all the *incoming channels* on either the top or bottom side and all the *outgoing channels* on the other side. The two boxes are square.

No *channels* are permitted to be connected to any part of either box.

Specifications

Length:	User's choice.
Thickness (central bar):	0.4 units
Box Size:	1.0 unit
Orientation:	User's choice (horizontal or vertical).

(**Ref**: 1.2.3_Wait Divider Domain)

2.1.5 Wait Domain

The following symbol represents a *wait domain*.

Wait Domain

Description

The *domain* can have only one *incoming channel* and no more than one *outgoing channel*. The *incoming channel* can be on any of the four sides, and the *outgoing channel* is optional and on another side if it exists. The shape is a square.

The *incoming channel* can have a timeout value, represented as ‹t (*ms*)›, where *ms* is the maximum number of milliseconds that the *incoming control point* is allowed to remain *idle*. *ms* can be a question mark (?), indicating that the timeout value exists but is not specified in the *DCD*. The ‹t (*ms*)› is optional, and its omission indicates that the timeout value is infinite. The timeout text (if it exists) is written near the *wait domain* end of the *channel* at the same position as a *passage tag* (see 2.1.15_Passage Tags and References for details).

Specifications

Size:	1.0 unit
Face Name of Tag (`t`):	Courier New
Height of Tag (`t`):	0.5 units (character height)
Background Colour of Tag (`t`):	White (RGB: 255, 255, 255)

(**Ref**: 1.2.5_Wait Domain)

2.1.6 Control Point Channel

The following symbols represent a *control point channel* with various indications of *incoming* and *outgoing data* (See 1.4_Data Control Points for more information).

No *incoming* or *outgoing data*.

Temporary *outgoing data* only.

Permanent *outgoing data*.

Temporary *incoming data* only.

Permanent *incoming data*.

Temporary *incoming* and *outgoing data* only.

Temporary *incoming data* only and permanent *outgoing data*.

Permanent *incoming data* and temporary *outgoing data* only.

Permanent *incoming* and *outgoing data*.

Description

A *control point channel* is connected between two *data domains*. The circle of the *channel* is always connected to the *source domain*, and the other end is always connected to the *destination domain*. An arrowhead touching the circle indicates *incoming data*, and an arrowhead on the other end indicates *outgoing data*. A hollow arrowhead indicates temporary data transfer only, and a filled arrowhead indicates permanent (with possibly some temporary) data transfer. Permanent data transfer means that the data remains persistent in the relevant *domain* even after the *control point* has returned from the *source domain*; temporary data transfer means that the data is destroyed just before the *control point* returns from the relevant *domain*. No arrowhead indicates that no data transfer occurs in the implied direction.

A *control point channel* must be presented as a right-angle path with rounded corners. The bases of the arrowheads have a concave (circle arc) shape, and the converging sides are straight lines. No part of a *channel* may overlap any other *connector* except at right angles and only on the straight parts of the *connectors*. Also, no part of a *channel* may overlap any *domain*, except as follows: both the midline of the circle circumference at the end point of the *source domain* end of a *channel* and the opposite end of the *channel* (arrowhead tip or *connector* end point) must coincide with the perimeter midline of the respective connected *domains*.

The *connector* must have a *passage tag* (see 2.1.15_Passage Tags and References for details).

Specifications

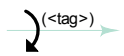
Channel Length:	Automatically determined.
Radius of Circle (to Circumference Midline):	0.25 units
Circumference Thickness:	0.12 units
Circle Interior Colour:	Transparent
Length of Arrowhead from Base Connection to Tip:	0.8 units
Height of Arrowhead:	0.633… units
Hollow Arrowhead Thickness:	0.12 units
Hollow Arrowhead Interior Colour:	Transparent
Path Thickness:	0.12 units
Path Corner Radius:	0.5 units

Repeated Step Symbol

A *control point channel* has a repeated step symbol if the *channel* is designated to allow a *control point* to repeatedly move from the *source domain* to the *destination domain* (and possibly further) before the next *data control step* of the *passage* connected to the *source domain* (See 1.4_Data Control Points for more information).

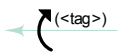
The following symbols represent a repeated step symbol on the segment of a *channel* path in the indicated direction (faint arrow line). The symbol consists of a circular arc with a triangular arrowhead at the end, and an optional tag at the top-right of the symbol. No more than one repeated step symbol is permitted on a *control point channel*.

Symbol on upward segment of *channel*.



Symbol on rightward segment of *channel*.

Symbol on downward segment of *channel*.



Symbol on leftward segment of *channel*.

Description

The symbol always points in a clockwise direction. The centre of the circular arc is pinned to the *channel* path as close as possible to the *source domain*. The <tag> represents a *passage tag*, and is situated relative to the circular arc as shown in the diagrams above. A <tag> is optional; the default is the *passage tag* of the *channel* on which the symbol exists. Note that <tag> is within parentheses, which are required.

The *passage tag* (<tag>) on a repeated step symbol indicates the last *control step* that a *control point* on the *channel* is to move to before those *control steps* (from the one associated with the symbol to the indicated last one) are repeated again. The number of times that the *control steps* may be repeated is not specified in the *DCD*; they may be repeated any number of times, including zero times. For example, if the *passage tag* of the *channel* associated with the step symbol is `3:4`, and <tag> is `3:6`, then a *control point* on the *channel* moves along the next *channel*, `3:5`, then along *channel* `3:6`, then immediately returns along *channel* `3:6` to the *source domain* of *channel* `3:4` via *channel* `3:5`. The *control point* then repeats the movements zero or more times (the number of times is unspecified) along the same *channels* before moving along *channel* `3:7` of the original *source domain*. See 2.1.15_Passage Tags and References for details.

Specifications

Symbol Length (Widest Dimension):	1.2 units
Symbol Width (Narrowest Dimension):	0.4 units

Circular Arc Thickness:	0.1 units
Circular Arc Radius:	0.65 units (calculated)
Arrowhead Length from Base to Tip:	0.25 units
Arrowhead Width:	0.5 units
<tag> Face Name:	Times New Roman
<tag> Height:	0.5 units (character height)
<tag> Position:	As shown in the diagrams above.

Control Gate Symbol

A *control point channel* has a *control gate* symbol if the *channel* is designated to allow only one *control point* on that *channel* to proceed while making all other *control points* in the *DCM* temporarily *idle*.

The following symbol represents a *control gate* symbol on a segment of a *channel* path (faint arrow line) near the *destination domain* end of the path. The symbol consists of a short line perpendicular to the direction of the path. No more than one *control gate* symbol is permitted on a *control point channel*.

Control Gate

Description

The symbol is always perpendicular to the *channel* segment direction (the diagram shows only one direct out of four directions). The centre of the symbol is pinned to the *channel* path as close as possible to the *destination domain*.

Specifications

Symbol Length:	1.0 unit
Symbol Width:	0.12 units
Distance of Symbol from Channel Arrow Base:	0.6 units (if possible)

(**Ref**: 1.3.1_Control Point Channel)

2.1.7 Static Domain Connector

The following symbol represents a *static domain connector*.

Static Domain Connector

Description

A *static domain connector* is connected between two *data domains*. The arrowhead is always connected to the *destination domain*, and the other end is always connected to the *source domain*.

A *static domain connector* must be presented as a right-angle path with rounded corners. The arrowhead is a filled triangular shape. The arrowhead tip coincides with the end point of the *connector*. No part of a *static domain connector* may overlap any other *connector* except at right angles and only on the straight parts of the *connectors*. Also, no part of a *static domain connector* may overlap any *domain*, except as follows: both the start point and the arrowhead tip of a *static domain connector* must coincide with the perimeter midline of the respective connected *domains*.

Specifications

Channel Length:	Automatically determined.
Symbol Colour:	Blue (RGB: 45, 126, 199)
Length of Arrowhead from Base Connection to Tip:	0.8 units
Height of Arrowhead:	0.633… units
Path Thickness:	0.2 units
Path Corner Radius:	0.5 units

(**Ref**: 1.3.2_Static Domain Connector)

2.1.8 Dynamic Domain Connector

The following symbols represent a *dynamic domain connector* without and with a no-reference symbol.

Dynamic Domain Connector

Dynamic Domain Connector
(No-reference)

Description

A *dynamic domain connector* is connected between two *data domains*. The arrowhead is always connected to the *destination domain*, and the other end is always connected to the *source domain*.

A *dynamic domain connector* must be presented as a right-angle path with rounded corners. The arrowhead is a filled triangular shape. The centre point of the circle and the arrowhead tip coincide with the respective end points of the *connector*. No part of a *dynamic domain connector* may overlap any other *connector* except at right angles and only on the straight parts of the *connectors*. Also, no part of a *dynamic domain connector* may overlap any *domain*, except as follows: both the midpoint of the circle and the arrowhead tip of a *dynamic domain connector* must coincide with the perimeter midline of the respective connected *domains*.

The *connector* must have a *passage tag* (see 2.1.15_Passage Tags and References for details).

Specifications

Channel Length:	Automatically determined.
Symbol Colour:	Green (RGB: 111, 184, 111)
Length of Arrowhead from Base Connection to Tip:	0.8 units
Height of Arrowhead:	0.633… units
Radius of Circle:	0.25 units
Path Thickness:	0.2 units
Path Corner Radius:	0.5 units
Size of Cross (Width and Height):	0.5 units
Thickness of Cross:	0.1 units
Distance of Cross Midline from Circle Centre:	0.8 units

No-reference symbol

The cross is a no-reference symbol indicating that the *source domain* does not have an automatic reference to the *destination domain* after the *destination domain* has been created. Without the no-reference symbol, the *source domain* will automatically have a reference to the *destination*

domain after the *destination domain* has been created. The no-reference symbol (if it exists) is near the *source domain* end of the *connector*.

(**Ref**: 1.3.3_Dynamic Domain Connector)

2.1.9 Supplied Domain Connector

The following symbol represents a *supplied domain connector*.

Supplied Domain Connector

Description

A *supplied domain connector* is connected between two *data domains*. The arrowhead is always connected to the *destination domain*, and the other end is always connected to the *source domain*.

A *supplied domain connector* must be presented as a right-angle path with rounded corners. The arrowhead tip coincides with the end point of the *connector*. No part of a *supplied domain connector* may overlap any other *connector* except at right angles and only on the straight parts of the *connectors*. Also, no part of a *supplied domain connector* may overlap any *domain*, except as follows: both the start point and the arrowhead tip of a *supplied domain connector* must coincide with the perimeter midline of the respective connected *domains*.

The *connector* may have a *passage tag reference* (see 2.1.15_Passage Tags and References for details).

Specifications

Channel Length:	Automatically determined.
Symbol Colour:	Light blue (RGB: 97, 207, 255)
Length of Arrowhead from Base Connection to Tip:	0.6 units
Height of Arrowhead (between Midlines of Converging Lines):	0.6 units
Path and Arrowhead Thickness:	0.2 units
Path Corner Radius:	0.5 units

(**Ref**: 1.3.5_Supplied Domain Connector)

2.1.10 Domain Deletion Connector

The following symbol represents a *domain deletion connector*.

Domain Deletion Connector

Description

A *domain deletion connector* is connected between two *data domains*. The cross end is always connect to the *destination domain*, and the circle end is always connected to the *source domain*.

A *domain deletion connector* must be presented as a right-angle path with rounded corners. Both the centre points of the circle and the cross coincide with the respective end points of the *connector*. No part of a *domain deletion connector* may overlap any other *connector* except at right angles and only on the straight parts of the *connectors*. Also, no part of a *domain deletion connector* may overlap any *domain*, except as follows: both the centre points of the circle and the

cross of a *domain deletion connector* must coincide with the perimeter midline of the respective connected *domains*.

The *connector* must have a *passage tag* (see 2.1.15_Passage Tags and References for details).

Specifications

Channel Length:	Automatically determined.
Symbol Colour:	Dark red (RGB: 165, 42, 42)
Size of Cross (Midline):	0.5 units
Radius of Circle:	0.25 units
Path and Cross Thickness:	0.2 units
Path Corner Radius:	0.5 units

(**Ref**: 1.3.4_Domain Deletion Connector)

2.1.11 Reference Domain Link

The following symbol represents a *reference domain link*.

Reference Domain Link

Description

A *reference domain link* is connected between two *data domains*. The arrowhead is always connected to the *destination domain*, and the other end is always connected to the *source domain*.

A *reference domain link* must be presented as a right-angle path with rounded corners. The arrowhead is a filled triangular shape. The arrowhead tip coincides with the end point of the *connector*. Two or more *reference domain links* that have the same *passage tag reference* may have overlapping start points, but otherwise, no part of a *reference domain link* may overlap any other *connector* except at right angles and only on the straight parts of the *connectors*. Also, no part of a *reference domain link* may overlap any *domain*, except as follows: the start point and the arrowhead tip of a *reference domain link* must coincide with the perimeter midline of the respective connected *domains*.

A *reference domain link* may have a *passage tag reference* (see 2.1.15_Passage Tags and References for details). However, if a *link assignment* is connected to the *reference domain link*, then the *reference domain link* cannot have a *passage tag reference* (the *passage tag reference* would be on the *link assignment*).

Specifications

Channel Length:	Automatically determined.
Symbol Colour:	Light blue (RGB: 97, 207, 255)
Length of Arrowhead from Base Connection to Tip:	0.8 units
Height of Arrowhead:	0.633… units
Path Thickness:	0.05 units
Path Corner Radius:	0.5 units

(**Ref**: 1.3.6_Reference Domain Link)

2.1.12 Divider Link

The following symbol represents a *divider link*.

Divider Link

Description

A *divider link* is connected between a *wait domain* and a *linked-wait divider domain*.

A *divider link* must be presented as a right-angle path with rounded corners. No part of a *divider link* may overlap any other *connector* except at right angles and only on the straight parts of the *connectors*. Also, no part of a *divider link* may overlap any *domain*, except as follows: both end points of a *divider link* must coincide with the perimeter midline of the respective connected *domains*.

Specifications

Channel Length:	Automatically determined.
Symbol Colour:	Dark red (RGB: 165, 42, 42)
Path Thickness:	0.05 units
Path Corner Radius:	0.5 units

(**Ref**: 1.3.7_Divider Link)

2.1.13 Link Assignment

The following symbol represents a *link assignment*.

Link Assignment

Description

A *link assignment* is connected between a *data domain* (the *source domain*) and a *reference domain link*. The large dot is on the *source domain* which sets up the *reference domain link* between its two *domains*; the small dot is on the actual *reference domain link* symbol. Note that a *link assignment* does not have a *destination domain*.

A *link assignment* must be presented as a right-angle path with rounded corners. Both centre points of each circle coincide with the respective end points of the *connector*. No part of a *link assignment* may overlap any other *connector* except at right angles and only on the straight parts of the *connectors*. Also, no part of a *link assignment* may overlap any *domain*, except as follows: the centre point of the large circle is connected to the perimeter midline of the *source domain*, while the centre point of the small circle is connected to the midline of the *reference domain link*.

The *connector* may have a *passage tag reference* (see 2.1.15_Passage Tags and References for details). Note that the *reference domain link* to which the *connector* is connected cannot have a *passage tag reference*.

Specifications

Channel Length:	Automatically determined.
Symbol Colour:	Light blue (RGB: 97, 207, 255)
Radius of Large Circle:	0.25 units
Radius of Small Circle:	0.175 units

Path Thickness:	0.05 units
Path Corner Radius:	0.5 units

(**Ref**: 1.3.8_Link Assignment)

2.1.14 Ellipsis Symbols

The following symbols represent an ellipsis symbol in two orientations. An ellipsis symbol represents omitted *domains* or *connectors*, and has the same significance as a regular ellipsis symbol (…). An ellipsis symbol is defined only for a *DCD*.

Vertical Ellipsis Symbol

Horizontal Ellipsis Symbol

Description

Vertical Ellipsis Symbol: For this orientation, the ellipsis represents omitted symbols between the symbol above the ellipsis and the symbol below the ellipsis. The two symbols must be of the same type.

Horizontal Ellipsis Symbol: For this orientation, the ellipsis represents omitted symbols between the symbol to the left of the ellipsis and the symbol to the right of the ellipsis. The two symbols must be of the same type.

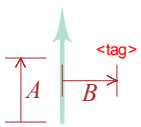
The omitted symbols are of the same type as the two reference symbols on each side of the ellipsis, and follow the same pattern as those two symbols.

Specifications

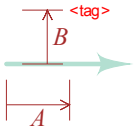
Symbol Colour:	Bright green (RGB: 0, 255, 0)
Radius of Each Circle:	0.22 units
Distance between Circle Centres:	0.8 units
Orientation:	User's choice (horizontal or vertical).

2.1.15 Passage Tags and References

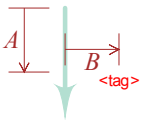
The "<tag>" part of the following diagrams represents a *passage tag* or *passage tag reference* on the segment of an appropriate *connector* path in the indicated direction (the faint arrow line). The position of a tag relative to the *connector* path is also shown (the dark red dimension lines). Note that the faint arrow lines and the dark red dimension lines are presented only for reference purposes.



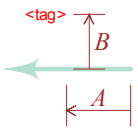
Tag on upward segment of *connector*.



Tag on rightward segment of *connector*.



Tag on downward segment of *connector*.



Tag on leftward segment of *connector*.

Description

For a *passage*, <tag> is a *passage tag*, and the colour is red (RGB: 255, 0, 0). For the other *connectors* (*reference domain link*, *link assignment*, and *supplied domain connector*), <tag> is a *passage tag reference*, and the colour is black (RGB: 0, 0, 0). Distance *A* is from the *source domain* end of the *connector* along the straight parts of the *connector* path to the edge of <tag>. Distance *B* is from the midline of the *connector* to the centre of <tag>.

NOTE: <tag> must fully cover the associated *connector* underneath it, but not cover any circles, arrowheads, crosses, etc of the *connector*. <tag> should also be as close as possible to the *source domain*, unless stated otherwise for specific *connectors*. <tag> must not cover any other *connector*.

Specifications

Distance *A*:	User's choice (within any other specified limit).
Distance *B*:	User's choice (within any other specified limit).
<tag> Colour:	Red or black (see *Description* above).
<tag> Face Name:	Times New Roman
<tag> Height:	0.5 units (character height)
<tag> Background Colour:	White (RGB: 255, 255, 255)

Passage Tag Format

The format of a *passage tag* is:

path-number[@*sheet*$_1$][(*path-group*)]:*passage-label*$_1$[[@*sheet*$_2$:*passage-label*$_2$]]

path-number is a unique positive integer, determined by the *DCD* designer, identifying a *control path*. Note that each *outgoing channel* of a *divider* necessarily belongs to a different *control path* than all the other *channels* of the *divider*.

sheet$_1$ and *sheet*$_2$ are the *diagram sheet* numbers (positive integers) on which the specified *control path* (*path-number*) exists if the *path* does not exist on the current *sheet* (in which case both *sheet*$_1$ and ‹[@*sheet*$_2$:*passage-label*$_2$]› are omitted). *sheet*$_1$ and ‹[@*sheet*$_2$:*passage-label*$_2$]› cannot both exist in the same *passage tag*. *sheet*$_2$ exists if *passage-label*$_2$ exists. ‹@*sheet*$_2$› necessarily implies ‹*path-number*@***sheet***$_2$[(*path-group*)]›.

path-group is a unique positive integer identifying the *concurrent path group* to which the *path* belongs (if any); if a *path* does not belong to a *concurrent path group*, or if there is only one such *concurrent path group* in a *DCD*, then ‹ (*path-group*) › is omitted. *path-group* exists only on the first *passage* of the *path*. Note that *path-group* exists within parentheses.

passage-label$_1$ is a unique label (with respect to the *control path* identified by *path-number*) identifying a *passage* of that *path*. See **Passage Label Format** below for the format of *passage-label*$_1$.

passage-label$_2$ is the label of a *passage* that is not on the current *diagram sheet* but is the same *passage* as the one identified by *passage-label*$_1$; effectively, *passage-label*$_1$ and *passage-label*$_2$ are different *passage* labels for the same *passage* on different *sheets*. ‹[@*sheet*$_2$:*passage-label*$_2$]› exists only on a *sheet* referenced by a *sheet reference data domain* (see **Pseudo Data Domains** under 2.1.1 Data Domains). See **Passage Label Format** below for the format of *passage-label*$_2$.

A *passage* can have more than one comma-separated *passage tag*. The first one is the proper *passage tag* for the *passage*; each other *passage tag* indicates that one of the *control steps* on another *control point route* is the same *control step* as the said *passage*.

Examples

Six examples of *passage tags* follow: `1:1`, `1:1a1`, `37(3):5`, `29:4a5b3`, `34@10:5`, `7:30[@4:57a2]`. The second last example, `34@10:5`, is a *passage tag* of a *passage* in the current *diagram sheet* that is the same *passage* with a *passage tag* `34:5` on *sheet* `10`. The last example, `7:30[@4:57a2]`, is a *passage tag*, `7:30`, of a *passage* in the current *sheet* that is the same *passage* with a *passage tag* `7:57a2` on *sheet* `4` (`@4:57a2` implies `7@4:57a2`).

Passage Label Format

A *passage-label* has the following format:

number$_1$[*letters number*$_2$]···

number$_1$ is a positive integer associated with a *passage* within the current *control path* (specified by *path-number* described above).

letters is a sequence of one or more lower-case letters (‘a’ to ‘z’) indicating an alternative *passage*.

number$_2$ is a positive integer relative to *letters*. Note that there is no space between *letters* and *number*$_2$. ‹*letters number*$_2$› may be repeated any number of times with possibly different letters and numbers.

A *passage-label* uniquely identifies every *passage* of a given *path* in the order that the *control points* move along those *passages*. *number*$_1$ is ‹1› for the first *passage* of the *path*, and ‹2› for the next *passage*, and so on. If there are alternative *passages* for a particular *control step*, then *letters* is ‹a› for one of the alternative *passages*, ‹b› for another of the alternative *passages*, ‹c› for another, and so on. (*letters* is ordered from ‹a› to ‹z›, and the next letter after ‹z› is ‹aa›, then ‹ab›, and so on to ‹az›, then the next letter after ‹az› is ‹ba›, and so on. This is standard lexicographical ordering.) For the first *passage* of each of the alternative *passages*, *number*$_2$ is ‹1›. *number*$_2$ for the next sequential *passage* after the first alternative one is ‹2› (*letters* remains the same as for the previous *passage*), and so on for subsequent *passages*. This is explained in the following examples.

Passage Tag Examples

For a simple example where a *control path* (identified by *path-number* 2 in this example) contains four *passages*, with no alternative ones, the *passage tags* on those *passages* would be: ‹2:1›, ‹2:2›, ‹2:3›, and ‹2:4›, in the order that a *control point* moves along those *passages*. The *path*, in this example, contains only one *route*. Each number after the colon is a *passage-label*. Incidentally, the colon in the *passage tag* is expressed as “col”, so ‹2:3› would be expressed as “two col three”.

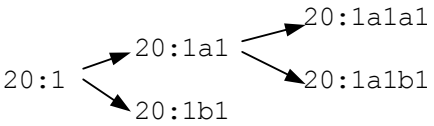
For an example where a *control path* (identified by *path-number* 3) contains four *passages*, with the third *passage* being the only alternative one, the *passage tags* on those *passages* would be: ‹3:1›, ‹3:2›, ‹3:2a1›, and ‹3:3›, in the order that a *control point* moves along those *passages*. The alternative *passage* is tagged ‹3:2a1›, and a *control point* may optionally move along that *passage*. A *control point* moving along the *path* may move along the *passages* tagged as ‹3:1›, ‹3:2›, and ‹3:3› in the given order. Alternatively, the *control point* may move along the *passages* tagged as ‹3:1›, ‹3:2›, ‹3:2a1›, and ‹3:3›. The *path*, in this example, contains only two *routes*.

For an example where a *control path* (identified by *path-number* 5) contains six *passages*, with three alternative ones at the second *control step* only, the *passage tags* on those *passages* would be: ‹5:1›, ‹5:1a1›, ‹5:1b1›, ‹5:1c1›, ‹5:2›, and ‹5:3› in the order that a *control point* moves along those *passages*. Notice that the three alternative *passages* are tagged ‹5:1a1›, ‹5:1b1›, and ‹5:1c1›. *number*$_1$ (ie: 1) of the three alternative *passage tags* is the same as the *number*$_1$ of the previous *passage tag* (5:1). Also notice that, in this example, the next *control step* after each of the alternative *steps* is the *control step* for the *passage* tagged as ‹5:2›. Therefore, a *control point* moving along the *path* may move along the *passages* tagged as ‹5:1›, ‹5:1b1›, ‹5:2›, and ‹5:3› in the given order. Alternatively, the *control point* may move along the *passages* tagged as ‹5:1›, ‹5:1c1›, ‹5:2›, and ‹5:3›. Another alternative is for the *control point* to skip the alternative *passages* altogether. The *control point* would then move along the *passages* tagged as ‹5:1›, ‹5:2›, and ‹5:3›. The *path*, in this example, contains four *routes*. Note that the *DCD*

reader must take into account all possible *routes* of a *path*. Note also that the conditions for the particular *route* that a *control point* moves along may be indicated in the *operational mode* table (see 2.4_Operational Mode Tables).

For an example where a *control path* (identified by *path-number* 6) contains five *passages*, beginning with two alternative ones at the first *control step* and continuing separately, the *passage tags* on those *passages* could be: ‹6:1a1›, ‹6:1a2›, ‹6:1b1›, ‹6:1b2›, and ‹6:1b3›. Therefore, a *control point* moving along the *path* may move along the *passages* tagged as ‹6:1a1› and ‹6:1a2› in the given order. Alternatively, the *control point* may move along the *passages* tagged as ‹6:1b1›, ‹6:1b2›, and ‹6:1b3›. The *path*, in this example, contains two *routes*.

For an example where a *control path* (identified by *path-number* 20) contains five *passages*, beginning with two alternative ones at the second *control step* and continuing separately, but with another two alternative *passages* at the second *control step* of the first alternative, the *passage tags* on those *passages* would be: ‹20:1›, ‹20:1a1›, ‹20:1a1a1›, ‹20:1a1b1›, and ‹20:1b1›. The *path*, in this example, contains five *routes*, indicated by the following *passage tags*: **(1)** 20:1; **(2)** 20:1, 20:1a1; **(3)** 20:1, 20:1a1, 20:1a1a1; **(4)** 20:1, 20:1a1, 20:1a1b1; **(5)** 20:1, 20:1b1. The *passage tags* corresponding to the *control steps* of this example are depicted in the diagram below.



Notice that the initial part of the *passage tag* of each alternative *passage* is the previous *passage tag*. For example, the initial part of the *passage tag* **20:1a1**b1 is the previous *passage tag* 20:1a1. If there were another *passage* to the right of the *passage* labelled 20:1a1b1, it would be labelled 20:1a1b2.

The scheme for *passage tags* may appear to be unnecessarily complicated, but it allows adding new alternative *passages* to a *route* without needing to alter existing *passage tags*. A simpler scheme would likely require changes to existing *passage tags* when new alternative *passages* are added to a *route*.

Note that *passage tags* must be shown in red in an actual *DCD*.

Passage Tag Reference Format

The format of a *passage tag reference* is:

{*passage-tag*}

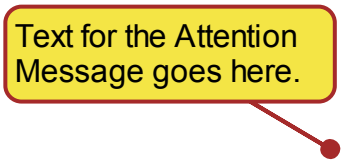
passage-tag is a *passage tag*. The braces are required.

A *passage tag reference* is used on certain *connectors* (*reference domain link*, *link assignment*, and *supplied domain connector*) to indicate that the action defined for those *connectors* occurs when the *control step* labelled by the *passage-tag* is performed. For example, if a *reference domain link* having the *passage tag reference* ‹{3:10}› is connected from *data domain A* to *data domain B*, then the actual reference from the data in *data domain A* to *data domain B* is created by *data domain A* when the *control step* indicated by *passage tag* 3:10 is performed. If, instead, a *link assignment* having the said *passage tag reference* is connected from *data domain C* to the *reference domain link*, then the reference from the data in *data domain A* to *data domain B* is created by *data domain C* (rather than *data domain A*) when the *control step* indicated by *passage tag* 3:10 is performed. In this case, the *reference domain link* itself cannot have a *passage tag reference* (to do so would cause ambiguity).

Note that *passage tag references* must be shown in black in an actual *DCD*.

2.1.16 Attention Message

The following symbol represents an attention message. An attention message is used to draw the attention of the *DCD* reader to important circumstances on the *DCD* that require immediate attention. It is not intended to present general information — general information can be presented in the description text or operational mode tables. An attention message is defined only for a *DCD*.



Attention Message

Description

An attention message is a rounded rectangle containing the message. Zero or more straight-line "pointers" can originate from the centre of the rectangle. The circle on a pointer is near the item in the *DCD* that requires attention.

Specifications

Box Width:	User's choice.
Box Height:	Varies with the text and user's choice.
Corner Radius:	0.5 units
Circle Radius:	0.25 units
Circle Position:	User's choice.
Border and Pointer Colour:	Dark red (RGB: 165, 42, 42).
Background Colour:	Yellow (RGB: 244, 229, 69)
Text Face Name:	Arial
Text Height:	1.0 unit (character height)
Text Margin from Box Midline Perimeter:	0.3 units

2.2 Description Text

A data control diagram is typically associated with description text as an attached document. The text can be for the whole diagram and can include separate sections for any of the *diagram sheets* in the diagram. The description text may include the following information.

- A description of the purpose of the *DCD*.
- Class diagrams for any of the *data domains* in the *DCD*.
- The architectural principles of the corresponding *DCM*.
- Restrictions on the modifications to the *DCD*. For example, if some parts of the *DCD* should never be modified, that information should be present. Or, if some parts are allowed to be modified only in certain ways, that information should be present.
- Indications of where and how new modifications and enhancements are to be made to the *DCD*.
- The necessary dependencies that are required for future modifications and enhancements to the *DCD*.
- The relation of the *DCD* to the implemented software. For example, the *DCD* may show only the main structures of the software.

Of course, any other useful information should be included in the description text. Note that information that does not require immediate attention should not be presented on the *DCD* itself.

2.3 Diagram Sheets

A data control diagram may be spread over one or more *diagram sheets* for convenience. A *diagram sheet* can have *domains* and *passages* that are identical to other *domains* and *passages* on other *sheets*. Identical *domains* on different *sheets* are identified by having the same *domain* labels. Identical *passages* on different *sheets* are identified by their *passage tags* and *passage tag references* (see 2.1.15_Passage Tags and References for details).

To reduce complex contents of a *diagram sheet*, a *DCD* may be split up and presented on multiple *sheets* using pseudo *data domains* which represent various parts of the *DCD* in a different *diagram sheet* (see **Pseudo Data Domains** under 2.1.1_Data Domains for details).

The following rules apply to the contents of a *diagram sheet*.

- *Connectors* cannot be continued across *sheets*.
- The same *data domain* label on multiple sheets represents the same *data domain*.
- The same *data domain* may be represented more than once on the same *sheet*. In this case, the representations have the same *domain* label but with ‹[*n**m*]› appended to the end of each label, where *n* and *m* are positive integers, and *n* is a sequential number unique among the *domain* representations, and *m* is the total number of *domain* representations. For example, if three representations of the *domain* labelled MainIFace are desired on the same *sheet*, the three representations must be labelled ‹MainIFace [1\3]›, ‹MainIFace [2\3]›, and ‹MainIFace [3\3]›.
- Each *sheet* must be labelled by a positive integer, *n*, as ‹Sheet *n*›.
- All *sheet* numbers of a *DCD* must be in sequence beginning with ‹Sheet 1›, which is the main *sheet*. The next *sheet* is ‹Sheet 2›, and so on.
- In addition to ‹Sheet *n*›, a *sheet* may have a title in parentheses. For example, ‹Sheet 1 (Main Program)›.

A *diagram sheet* could also consist of a textual reference to a *sheet* on another *DCD* in a different file. For example, the text of a *sheet* could be ‹See diagram sheet 5 in the file C:\MyFiles\DCDfiles.dcd›. A summary of the contents of the referenced *sheet* would also be included.

2.4 Operational Mode Tables

A *DCD* containing only the symbols mentioned in the paragraphs above would be insufficient to show the details of data changes occurring within the *data domains*. To express such data changes, additional information will need to be included in the *DCD*. That information is presented in one or more *operational mode* tables.

An *operational mode* table describes *operational modes* (the set of *passages* involved in named operations) and *channel records* for a *DCD*. The *DCD* designer decides the appropriate *operational modes* and corresponding *channel records*. There is one *operational mode* table for each *operational mode*.

An *operational mode* table consists mainly of one or more *channel records* that are related to the corresponding *operational mode*. An *operational mode* table must be presented in the following format.

Operational Mode *n* (*Heading*)

passage-tag, ···
  Description
  REF: *reference*
  [SRC: $details_1$]
  [OUT: $details_2$]

[DST: *details*$_3$]
[IN: *details*$_4$]
[SRC: *details*$_5$]
[NOTE: *details*$_6$]

passage-tag, ···

···
[COMMENT: *Comment*]

where

n is a unique positive integer, beginning with 1 (one), and increasing sequentially for each *operational mode*.

Heading is a short description of the *operational mode*. For example, Initialisation. Note that the *heading* is in parentheses.

passage-tag is the *passage tag* of the *passage* to which the rest of the indented data refers. The format for *passage-tag* is: *path-number*[(*path-group*)]:*passage-label*$_1$ as defined at 2.1.15_Passage Tags and References. There can be more than one comma-separated *passage-tag*. The first one is the proper *passage tag* for the *passage*; each other *passage tag* indicates that one of the *control steps* on another *control point route* is the same *control step* as for the said *passage*. For example, 1:2.

Description is a short description of the *channel record*. For example, Gets the Welcome Page data from disk.

reference is typically a function name in the implemented software that corresponds to the *channel record*. For example, GetData(). It could also be a reference to another *operational mode* that contains the *passage-tag*. For example, if the *passage-tag* is 2:6, but a different *operational mode*, say *operational mode* 3, contains a *channel record* with one of its *passage-tags* being 2:6, then *reference* would refer to that other *operational mode*. So, *reference* would be See op mode 3. The rest of the data in the *channel record* would typically not exist since it would be in the referenced *channel record* (2:6) of *operational mode* 3. *reference* should be a dash (−) if it is not needed.

details$_1$ is a short description of the important changes in the data of the *source domain* before a *control point* moves to its *destination domain*. For example, Convert position and size to screen coordinates.

details$_2$ is typically a list of the data that is transferred from the *source domain* to the *destination domain* by a *control point*. The data could be the names of actual variables in the implemented software, or it could be simple descriptions of such data. If *reference* is a function name, *details*$_2$ is typically a description of the arguments to that function. If no data is transferred to the *destination domain*, then ‹OUT: *details*$_2$› is omitted. For example, num rows, num columns.

details$_3$ is a short description of the important changes in the *destination domain* after a *control point* moves to it. For example, Reset the data to default values.

details$_4$ is typically a short description of the data that is transferred from the *destination domain* to the *source domain* by a *control point* on its return to the *source domain*. If *reference* is a function name, *details*$_4$ is typically a description of the return values of that function. If no data is transferred to the *source domain*, then ‹IN: *details*$_4$› is omitted. For example, Dialog box response.

details$_5$ is a short description of the important changes in the *source domain* after a *control point* returns from its *destination domain*. For example, Check that the response was valid.

details$_6$ is any important notes relating to the *channel record*. For example, The Welcome Page is reference number zero.

Comment is any important notes relating to the *operational mode*. For example, A request could be the first, previous, next, or last request.

The text in the IN and OUT fields can use the following expressions:

A := *expression*	Typical assignment.
$A \leftarrow B$	Value in B is transformed then copied to A.
$A \leftarrow (B, \cdots, C)$	Values in B, $\cdots$, C are transformed; the result is copied to A.
$(A_1, \cdots, A_n) \leftarrow (B_1, \cdots, B_m)$	Values in B_1, $\cdots$, B_m are transformed; the result is copied to A_1, $\cdots$, A_n.
A ? B	If A is `true` then B.
A ? B : C	If A is `true` then B else C.
A_1 ? A_2 : A_3 ? A_4 : A_5 ? $\cdots$? A_n	Equivalent to: A_1 ? A_2 : (A_3 ? A_4 : (A_5 ? $\cdots$? A_n)).

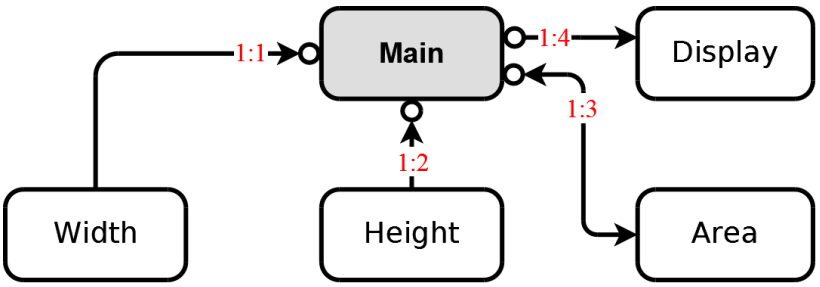
2.5 Standard Interpretation

The *DCM* definition is optimised for modelling data changes within software systems. Since a *DCD* typically represents a *DCM* of a software system, it is appropriate to have a defined standard interpretation of the elements of the *DCD* that show the relationships between the *DCM* and the software system.

The advantage of having such a standard is to enable software developers to correctly interpret the correspondence between the *DCD* and the software represented by it. Developers can then design and make appropriate modifications to the software via the *DCD*. The standard interpretation of a *DCD* is intended only for stand-alone application programs.

The main elements of a *DCD* are *domains*, *connectors*, and *data control points*. So these are the components that require a standard interpretation. In the standard interpretation, *data domains* correspond to data structures in the software being modelled, and *passages* represent function calls. *Control points* represent the execution control of the software along with the data passed to, and returned from, the function calls. The data structures are typically class instances in the object-oriented paradigm, and functions are typically function members of the classes. An optional class structure diagram can also be included in the standard interpretation. So, the rules for the standard interpretation of a *DCD* are quite simple.

The diagram below is an illustration of a simple *DCD* for a program displaying the result of multiplying the width and height of a rectangle to obtain the area for displaying. The source code (written in the ETAC™ programming language) that the *DCD* represents is also shown.



The following *operational mode* table that goes with the *DCD* above.

Operational Mode 1

1:1

Get the width.
REF: GetW()
IN: width

1:2

Get the height.
REF: GetH()
IN: height

1:3

Calculate the area.
REF: CalcA()
OUT: width, height
DST: Calculate the area: width times height.
IN: area

1:4

Display the area value.
REF: Print()
OUT: area
DST: Display the area value to the console.

The source code for the *DCD* above is shown below.

```
Height :- data: {Val :- 20;  GetH :- fnt:() {Val;};};
Width :- data: {Val :- 10;  GetW :- fnt:() {Val;};};
Area :- data: {CalcA :- fnt:(pW pH) {(pW * pH);};};
Display :- data: {Print :- fnt:(pV) {write_con num_to_str pV;};};
start_local;
   W :- Width.GetW();  H :- Height.GetH();  A :- Area.CalcA(W H);
   Display.Print(A);
end_local;
```

The *data domain* labelled Main corresponds to the code between `start_local` and `end_local`. Each of the other *data domains* correspond to the same-named data objects in the code.

The *control point* begins in Main (which corresponds with the execution control beginning at the first statement after `start_local`). The *control point* then moves along the *passage* labelled `1:1`, returning the width value from Width, and storing that value in Main. The movement of the *control point* corresponds with the function call of `GetW()` (see ‹`REF: GetW()`› under `1:1` in the *operational mode* table). The *control point* then moves along *passage* `1:2`, returning the height value from Height, also storing that value in Main (see ‹`REF: GetH()`› under `1:2` in the *operational mode* table). After returning to Main, the *control point* then moves to Area, calculates the area from the transferred width and height obtained earlier, then returns back to Main, storing the area value there (see under `1:3` in the *operational mode* table). Then, the *control point* moves along *passage* `1:4` with the value of area, where it is displayed to the console (see under `1:4` in the *operational mode* table). Finally, the *control point* returns to Main then disappears.

Of course, the program above could have been written is a simpler way as follows:

```
Width :- 10; Height :- 20; Area :- ?;

Area := (Width * Height);
write_con num_to_str Area;
```

The simpler way still conforms to the *DCD*, but not in a way that is consistent with the standard interpretation of a *DCD*. However, the code does not conform to the *operational mode* table.

The example above is quite simple, and, in practice, there would not be much merit in providing a *DCD* for it. However, consider a much more complex program, like a diagram construction or an animation program, that requires a significant modification during its life. Having a *DCD* for that program would enable a programmer to determine the effects and consequences of such a modification via the *DCD* before the actual source code is modified.

3

Data Control Diagram Examples

This chapter shows some examples of *DCDs* with explanations. For a proper understanding of the symbols in a *DCD*, the examples need to be read in the order presented.

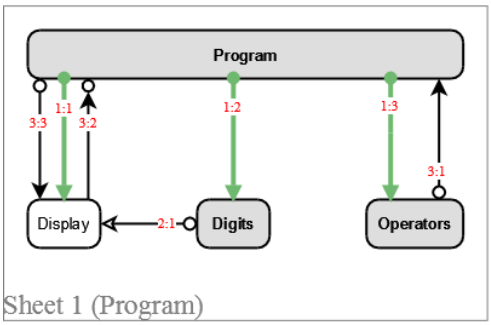
3.1 Simple Calculator

The following is a *DCD* for a simple calculator with the usual four arithmetical operations and an equal operation. The calculator program is written in the ETAC™ programming language. The features of the calculator are as follows:

- No parentheses.
- No operator precedence (all operations are left associative).
- Cumulative operations (operators evaluate previous operations).
- Standard infix evaluation.
- No current display or operation clearing.
- No back erase.
- No memory.
- Operators: '÷', '×', '−', '+', '='.
- Digits: '0' to '9', '.'.

The *DCD* consists of two *diagram sheets* containing the *DCD* symbols, three *operational mode* tables, and four class diagrams. *Operational mode* 3 is presented twice: the first presentation (alongside *sheet* 1) represents a top-level view, and the second presentation (alongside *sheet* 2) represents a lower-level view of the actual calculation process. The skeleton source code of the calculator program is also presented later, created entirely from the information in the *DCD*.

Note that the *DCD* below does not represent the graphics aspects of the calculator program but only its essential features. Representing the graphics aspects as well would be too complex for a first illustration of a *DCD*.



Operational Mode 1 (Initialise)

1:1
Create Display object.
REF: dspPrepare()

1:2
Create 10 digit buttons and a decimal point button.
REF: digPrepare().
DST: @goSetMseBtnFnt() // Set mouse button handler.
@goSetMseHitFnt() // Set button hit handler.

1:3
Create 4 operator buttons and equal button.
REF: oprPrepare()
DST: @goSetMseBtnFnt() // Set mouse button handler.
@goSetMseHitFnt() // Set button hit handler.

Operational Mode 2 (Number Pad)

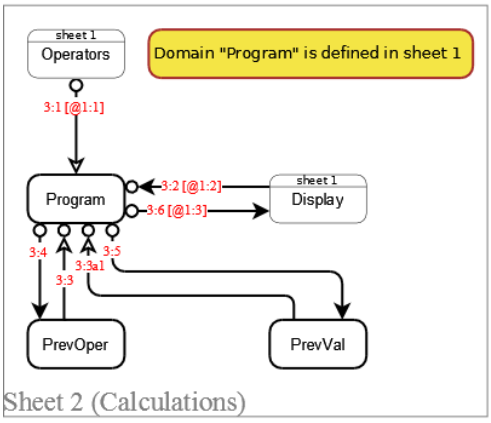
2:1
Put digit or decimal point.
REF: dspPutDigit()
OUT: digit or decimal point.

Operational Mode 3 (Operators)

3:1
Activate operation (including equal).
REF: gCalculate()
OUT: operator.
DST: Do specified calculation and store operator.
NOTE: gCalculate() will display an error message if it fails.

3:2
Get display value and store.
REF: dspGetValue()
DST: Reset display to clear for next digit.
IN: display value.

3:3
Evaluate and display result.
REF: dspSetValue()
SRC: Evaluate stored values.
OUT: result or ERROR.



Operational Mode 3 (Operators)

3:1
Activate operation (including equal).
REF: 3@1:1

3:2
Get display value and store.
REF: 3@1:2

3:3
Get previous operator.
REF: -
IN: operator.
SRC: operator != "" ? 3:3a1

3:3a1
Get previous value.
REF: -
IN: Previous value.
SRC: Use operator with displayed and previous values. Store result.

3:4
Put received operator.
REF: -
OUT: received operator.

3:5
Store value or result.
REF: -
OUT: stored value or result.

3:6
Display value.
REF: 3@1:3
OUT: stored value or result

Program
Main calculator program.

```
_lgPrevOper: str = ""
```
Stores the previous operation.

```
_lgPrevVal: dec = 0.0
```
Stores the previous value.

```
gCalculate(in pOper:str)
```
Performs calculation.
pOper => operator label.

Digits
The digit buttons maintenance module.

```
digPrepare()
```
Prepares the Digits module.

Operators
The operator buttons maintenance module.

```
oprPrepare()
```
Prepares the Operators module.

Display
The display maintenance module.

```
_ldspClearNext: bool = false
_ldspValue: dec = 0.0
_ldspStrValue: str = "0"
```

```
dspSetValue(in pValue:dec|?)
```
Sets the display value.

```
dspGetValue(): dec
```
Gets the display value.

```
dspPutDigit(in pDigit:str)
```
Appends a digit or decimal point to the end of the display value.

```
dspPrepare()
```
Prepares the Display module.

The program begins with *operational mode* `1`, the initialisation phase (see ‹`Operational Mode 1 (Initialise)`› along with the corresponding diagram on ‹`Sheet 1 (Program)`›). The *control point* for that *operational mode* first comes into existence in the Program *domain*. It then creates the Display *domain* before initialising it via *passage* `1:1`. After returning to the Program *domain*, the *control point* creates the Digits *domain* before moving to it via *passage* `1:2` to initialise the *domain*. Finally, the *control point* creates and initialises the Operators *domain* via *passage* `1:3`, then returns to Program and disappears.

Operational mode `2` describes what happens when the user clicks a digit pad or decimal point (button). The *control point* for that *operational mode* is created at the Digits *domain* before it moves to the Display *domain*, via *passage* `2:1`, taking with it data indicating the digit or decimal point that was clicked. The digit or decimal point is displayed to the user. The *control point* then returns to the Digits *domain* and disappears.

Operational mode `3` describes what happens when the user clicks an operator button. The *control point* for that *operational mode* moves from the Operators to the Program *domain*, via *passage* `3:1`, taking with it data indicating the operator that was clicked. While the *control point* is in the Program *domain*, it performs the specified calculation and stores the given operator. As part of

the calculation, the *control point* moves to the Display *domain* via *passage* 3:2. While in that *domain*, the calculator display is cleared. Upon returning to Program, the *control point* takes with it the displayed value (before it was cleared). The *control point* then moves to the Display *domain* with the final calculated value, via *passage* 3:3, after evaluating the stored values. The *control point* then eventually returns to the Operators *domain*, where it began, and disappears (gets destroyed).

For this particular *DCD*, the designer decided to show the details of the calculations on a separate *diagram sheet*, rather than on the main *sheet*. The *DCD* of ‹Sheet 2 (Calculations)› shows the details of *operational mode* 3. The first *passage*, 3:1, of ‹Operational Mode 3› (next to ‹Sheet 2›) corresponds to *passage* 3:1 in ‹Sheet 1›. This is indicated by the *passage tag* ‹3:1[@1:1]› in ‹Sheet 2›. This *passage tag* means that the current *passage tag* is 3:1 (the **3:1** of ‹**3:1**[@1:1]›), but the *passage tag* is officially presented in ‹Sheet 1› (the **@1** of ‹3:1[**@1**:1]›) also under *passage tag* 3:1 (the **3** along with the second ‹**:1**› in ‹**3**:1[@1:**1**]›) of that *sheet*. It is just a coincidence that both *passage tags* are the same (3:1). The *passage tag* 3:1 under ‹Operational Mode 3› (next to ‹Sheet 2›) also has a reference to the corresponding *passage tag* in ‹Sheet 1›. The 3@1:1 at REF: reads as: ‹3:1› (**3**@1:**1**) of ‹Sheet 1› (3**@1**:1). A similar correspondence occurs with the *passage tags* 3:2[@1:2] and 3:6[@1:3] of ‹Sheet 2›.

The *control point* on ‹Sheet 2›, for ‹Operational Mode 3›, first moves from *data domain* Operators then to Program, via *passage* 3:1. The *data domain* Program is defined in ‹Sheet 1› (as per the yellow attention message). *Data domains* Operators and Display are also defined in ‹Sheet 1›, but they are not intended as being part of ‹Sheet 2›, so the appropriate *data domain* symbol is used to indicate that fact in ‹Sheet 2›. The *control point* moving from Operators to Program corresponds with the *control point* at *passage* 3:1 of ‹Sheet 1› (as mentioned in the previous paragraph) also moving from Operators to Program. Therefore, the same processes occur with respect to the current *control point* as they do with the corresponding *control point* in ‹Sheet 1›; those processes need not be repeated here. Similarly, the *control point* then moves to Display, via *passage* 3:2, then returns, as it does for the corresponding situation with *passage* 3:2 in ‹Sheet 1›. The *control point* then moves to PrevOper via *passage* 3:3, returning the operator that was stored there. Now, at Program, if the returned operator is an empty string then the *control point* moves along *passage* 3:3a1, otherwise it skips that *passage*. This is indicated at ‹SRC: operator != "" ? 3:3a1› under *passage* 3:3 of ‹Operaational Mode 3›. The *control point* then moves along *passages* 3:3a1 (if required), 3:4, 3:5, then, finally, along 3:6 before eventually returning along *passage* 3:1 to Operators, where it began. *Passage* 3:6 of ‹Sheet 2› corresponds with *passage* 3:3 on ‹Sheet 1› as indicated by 3:6[@1:3].

The software implementation of the diagram shown in ‹Sheet 2› is shown below for reference. In this case, the *data domains* PrevOper and PrevVal in the diagram are implemented as global variables (_1gPrevOper and _1gPrevVal, respectively) in the code. The code contains commented references to the *passage tags* in ‹Sheet 2›. Note that the standard interpretation of a *DCD* does allow *data domains* to be interpreted as variables if necessary.

```
gCalculate :- fnt:(pOper[*str*]) [*3:1*]
{
   DVal :- Display.dspGetValue(); [*3:2*]

   if ( _1gPrevOper != ? ) then [*3:3*]
   {
      DVal := [*3:3a1*]
         when PrevOper is [*3:3*]
            "+" then {(_1gPrevVal + DVal);} [*3:3a1*]
            "-" then {(_1gPrevVal - DVal);} [*3:3a1*]
            "*" then {(_1gPrevVal * DVal);} [*3:3a1*]
            "/" then {(_1gPrevVal / DVal);} [*3:3a1*]
            else {DVal;} [*3:3a1*]
         endwhen;
   }
   endif;

   _1gPrevOper := pOper; [*3:4*]
   PrevVal := DVal; [*3:5*]

   _1gDisplay.dspSetValue(DVal); [*3:6*]
};
```

Reading data control diagrams may seem daunting at first, but the ideas expressed in them are actually quite simple once the intention and concepts underlying them are understood. The great advantage of having a *DCD* representing a computer program is that important overview information about the program can be gained quickly without needing to read the program source code. The consequences of various events that can occur to the program can be traced easily with a *DCD*. Also, a *DCD* can be used to set the structure of a program before implementing it.

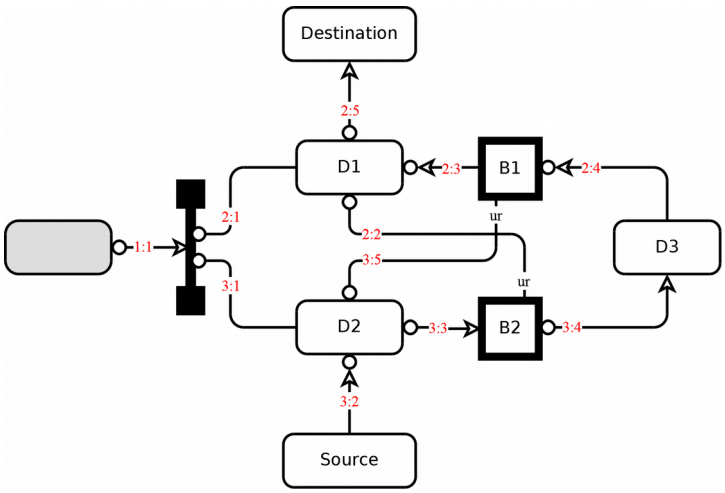
The *DCD* above also contains class diagrams for the various *data domains*. Class diagrams in a *DCD* are optional. In this case, with the class diagrams, the outline source code of the calculator program can be extracted from the *DCD*, as shown below. All that needs to be done is for the details of the source code to be filled in by the software designer.

```
start_local;
ArgStr :-; [* Assign the stack argument string. *]
_1gPrevOper :- ""; [* Previous operation. *]
_1gPrevVal :- 0.0; [* Previous value. *]

   [* Performs calculation. *]
   gCalculate :- fnt:(pOper[*str*])
   {
   };

   [* The display maintenance module. *]
   Display :- data:
   {
      _1dspClearNext :- false;
      _1dspValue :- 0.0;
      _1dspStrValue :- "0";

      [* Prepares the Display module. *]
      dspPrepare :- fnt:()
      {
      };

      [* Sets the display value. *]
      dspSetValue :- fnt:(pValue[*dec|?*])
      {
      };

      [* Gets the display value. *]
      dspGetValue :- fnt:() [* => dec *]
      {
      };

      [* Appends a digit or decimal point to the end of the display value. *]
      dspPutDigit :- fnt:(pDigit[*str*])
      {
      };
   };

   [* The digit buttons maintenance module. *]
   Digits :- data:
   {
      [* Prepares the Digits module. *]
      digPrepare :- fnt:()
      {
      };
   };

   [* The Operator buttons maintenance module. *]
   Operators :- data:
   {
      [* Prepares the Operators module. *]
      oprPrepare :- fnt:()
      {
      };
   };

   [* PROGRAM *]

end_local;
```

Essentially, the *DCD* represents the source code above, making it easy to gain an overview of the source code without needing to read it.

3.2 Reader and Writer

The following is a *DCD* fragment, that can be used in other *DCDs*, to transfer data from one *data domain*, Source, to another, Destination, in a synchronised way without data corruption. The reading and writing requests are asynchronous, but the data transfer is synchronous. The *DCD* illustrates the operations of *concurrent control points*. The *operational mode* tables for this illustration are not needed since it is an abstract example.



A *control point* begins at the shaded (unnamed) *data domain* at the left of the diagram, moving to the *wait divider domain* via *passage* `1:1`. The *wait divider spawns* two *control points*: one for the reading (*control path* `3`) and one for the writing (*control path* `2`) of the data. Meanwhile, the initial *control point* waits for the *spawned control points* to return to the *wait divider* before the initial *control point*, itself, returns to the shaded *data domain*. The order that the two *spawned control points* are created is undefined (this is where the asynchronicity occurs).

Beginning with *path* `3`, the *control point* moves to D2, along *passage* `3:1`, then to Source along *passage* `3:2`. Upon returning from Source, the *control point* returns some data with it from Source (this is the data to be eventually transferred to Destination), and passes that data to the *block domain*, B2, via *passage* `3:3`. The *control point* is then *blocked* by B2 until another *control point* moves along *passage* `2:2` later to release the *blocked control point*.

The *control point* on *path* `2` moves to D1 along *passage* `2:1`, then moves along *passage* `2:2` to B2, unblocking the previously *blocked control point* on *path* `3`. That unblocked *control point* then moves to D3 via *passage* `3:4`, taking the data from Source with it and storing that data in D3. The *control point* on *path* `2` eventually returns to D1, then moves to the *block domain*, B1, via *passage* `2:3`, and is *blocked*. The *control point* on *path* `3` now returns to B2, but is not *blocked* again because *passage* `2:2` is marked to not *block* it on return. The returning *control point* returns to D2, then moves along *passage* `3:5` to unblock the waiting *control point* on *path* `2`. The *control point* on *path* `3` can now return to the *wait divider* and eventually disappear.

That unblocked *control point* on *path* `2` now moves to D3 along *passage* `2:4`, and returns the data that was previously stored there by the *control point* that was on *path* `3`. The *control point* now returns to B1, without being *blocked*, and then to D1, passing the data that was in D3 to Destination via *passage* `2:5`. That *control point* can now return to the *wait divider* to die (RIP). Data has been transferred from Source to Destination successfully without corruption.

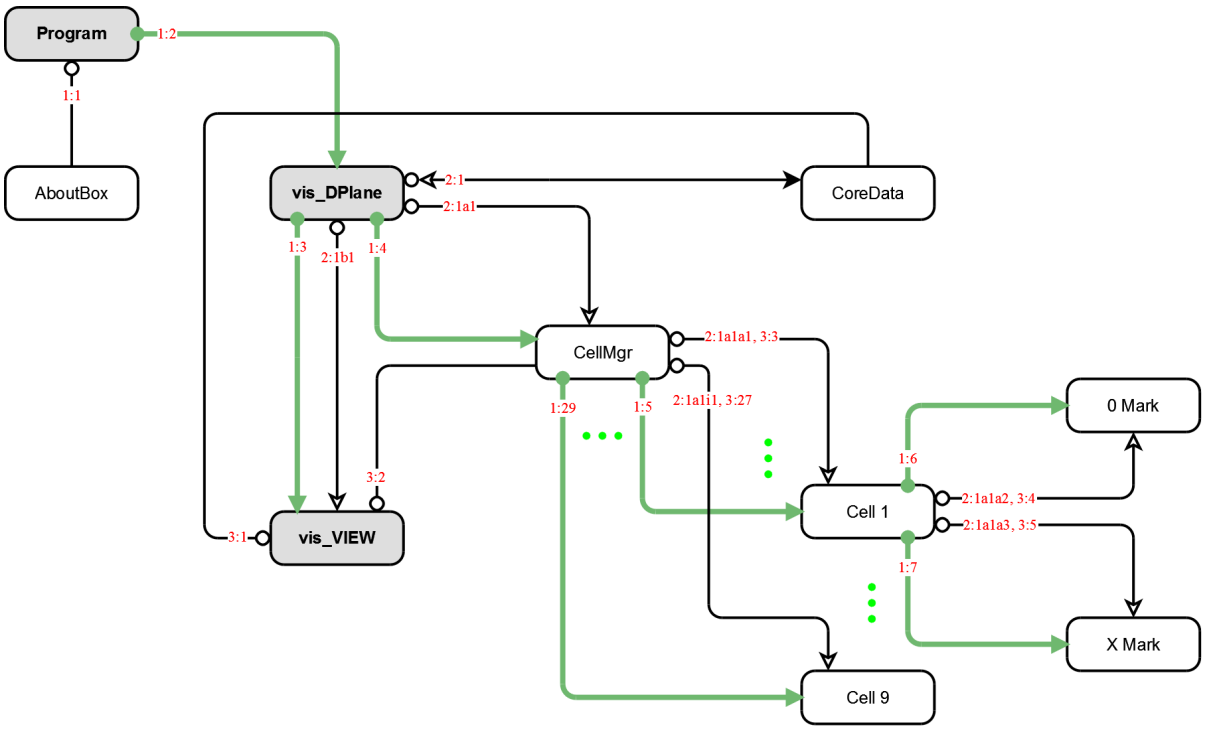
The important point here is that it does not matter whether a *control point* begins on *path* `2` or *path* `3` first. The data will still be transferred without corruption.

The *DCD* illustrated above, without the shaded *domain* and the *wait divider*, can be inserted into some other suitable *DCD* where *path* `2` and *path* `3` would each belong to different *concurrent path groups*.

3.3 Noughts and Crosses

The following is a *DCD* for the popular noughts and crosses game, also known as tic-tac-toe. In this case, the *DCD* is intended to be implemented in the ETAC programming language using VIS (Visual Interaction System). The vis_DPlane and vis_VIEW *domains* shown in the *DCD* are data objects defined in that language. However, with very little modification, the *DCD* could also be implemented in any other graphics programming language.

When implemented, the program would display space for nine symbols in a 3 × 3 grid. Each cell in the grid can contain only a nought (○) or a cross (×) or neither. The game begins with each cell being blank. Of two players, one player is allowed to put only a nought in the cells, and the other player is allowed to put only a cross in the cells. Each player has a turn in putting their mark in a cell by clicking on the cell. The first player to fill three cells in a line with their mark, either horizontally, vertically, or diagonally, wins the game. The game may not have a winner.



Operational Mode 1 (Initialisation)

1:1
Display introduction and copyright information.
REF: -

1:2
Create the main drawing plane.
REF: -

1:3
Create the main VIEW.
REF: -

1:4
Create the cell manager.
REF: -

1:5, 1:8, 1:11, ..., 1:29
Create the cells.
REF: -

1:6, 1:9, 1:12, ..., 1:30
Create the cell graphics object for 'O'.
REF: -

1:7, 1:10, 1:13, ..., 1:31
Create the cell graphics object for 'X'.
REF: -

Operational Mode 2 (Play)

2:1
Record the mark for the specified cell for the current player.
REF: -
OUT: cell number
DST: Set next player.
IN: player status (before next player set).

2:1a1
Set the appropriate mark in the specified cell.
REF: -
OUT: mark, cell number

2:1a1a1, 2:1a1b1, ..., 2:1a1i1
Show the specified information on the VIEW.
REF: -
OUT: mark

2:1a1a2, 2:1a1b2, ..., 2:1a1i2
Show or hide the graphics object.
REF: -
OUT: bool value.

2:1a1a3, 2:1a1b3, ..., 2:1a1i3
Show or hide the graphics object.
REF: -
OUT: bool value.

2:1b1
Display message on the status bar.
REF: -
OUT: text message.

Operational Mode 3 (New Game)

3:1
Initialise data for a new game.
REF: -

3:2
Clear the cells.
REF: -

3:3, 3:6, 3:9, ..., 3:27
Clear the cell.
REF: -
OUT: 0

3:4, 3:7, 3:10, ..., 3:28
Hide the graphics object.
REF: -
OUT: false.

3:5, 3:8, 3:11, ..., 3:29
Hide the graphics object.
REF: -
OUT: false.

Only the elements of the above *DCD* that have not been explained in the previous examples will be explained for this *DCD*.

In the *DCD*, some *passages* have two *passage tags*, for example, ‹2:1a1a1, 3:3›. The first *passage tag* is the proper *passage tag* for the *passage*. The second (and third, etc, if they exist) *passage tag* belongs to a different *control point route* having the same *step* as the first one. The implication here is that, if the *passage* corresponds to a function call in the implementation, then that same function will be “called” by the *control points* on each of the different *routes* during some time when those *control points* are *active*.

The *DCD* shows what appear to be green ellipses. In fact, they are ellipses. The ellipsis symbol represents omitted symbols implied between the two symbols near each end of the ellipsis. The implied symbols are omitted for convenience, and can mentally be inserted by the human *DCD* reader. In the *DCD* above, the leftmost ellipsis symbol, between *passages* 1:29 and 1:5, represents the *passages* 1:26, 1:23, 1:20, 1:17, 1:14, 1:11, and 1:8. The sequence can be determined from the *operational mode* table for ‹Operational Mode 1› under 1:5.

The first vertical ellipsis represents omitted *passages* between 2:1a1a1 and 2:1a1i1, and also between 3:3 and 3:27. ‹Operational Mode 2› specifies the implied *passage tags* of those *passages*.

The second vertical ellipsis represents omitted *data domains* between Cell 1 and Cell 9. Included with those omitted *domains* are the *passages* that are connected with them, as implied by ‹Operational Mode 2›.

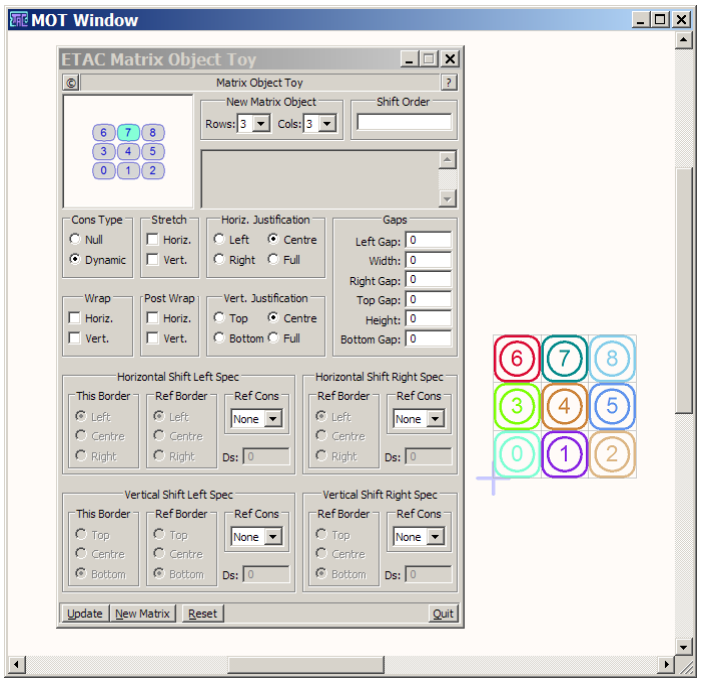
The *DCD* also shows alternative *passages* along which a *control point* can move. After moving along *passage* 2:1, the *control point* can move either along *passage* 2:1a1 or 2:1b1. If it moves along *passage* 2:1a1, it can then move along any one of the *passages* 2:1a1a1, 2:1a1b1, ···, 2:1a1i1. If it moves along *passage* 2:1a1a1, it then moves along *passage* 2:1a1a2. If it moves along *passage* 2:1a1b1 (implied in the *DCD*), it then moves along *passage* 2:1a1b2 (implied in the *DCD*)), and so on.

The *DCD* of an application program presents a quick overview of the program without needing to read the source code. It also helps to structure the source code in accordance with the *DCD*. It is important, though, to make sure that the *DCD* is always up-to-date.

3.4 Matrix Object Toy

The following *DCD* is for a matrix object toy program written in the ETAC programming language. A matrix object is a graphics component in VIS (Visual Interaction System) which is part of the language. The program allows the user to play with the various features of a matrix object.

The diagram below is how the matrix object toy program appears to the user. The matrix is the shape containing the nine coloured rounded rectangles and circles with digits in them on the “MOT Window” window. The dialog box controls the layout of the coloured rectangles.



The *DCD* for the matrix toy program is a little more complex than the *DCDs* in the previous examples. To make the *DCD* easier to read, it is split up into four *sheets*. Only the features not explained in the previous examples will be shown here.

The *DCD* features a mixture of statically and dynamically created *domains* (these correspond to the creation of class instances in the implementation). The blue thick *connectors* indicate that the *domain* at the arrowhead is statically created, typically as a subcomponent of the *source domain*. So, before the program begins, UserIFace creates uifMainUserIF automatically as a subcomponent, and CoreData also creates cdMODefData as a subcomponent. The thick green *connectors* indicate that the *domain* at the arrowhead is dynamically created on demand in the program. In this *DCD*, MatObjDef is deleted (indicated by the thick red *connector* with a cross as the arrowhead) each time before a new instance of it is created.

The *passage* labelled `2:6` (near the top-right of the diagram) has a repeated step symbol (see **Repeated Step Symbol** under 2.1.6_Control Point Channel). A *control point* on that *passage* repeatedly moves from CoreData to MainDP an unspecified number of times before moving along *passage* `2:7`.

Near the middle of the *DCD*, there is a *supplied domain connector* (see 2.1.9_Supplied Domain Connector) with the *passage tag reference* `{1:8}` (see *Passage Tag Reference Format* under 2.1.15_Passage Tags and References for details). The *passage tag reference* refers to *passage tag* `1:8`. So, after a *control point* moves along *passage* `1:8` to uifMainUserIF, a reference to the whole of cdMODefData is supplied to uifMainUserIF as indicated by the *supplied domain connector*. Normally the reference is temporary, but in this *DCD*, there is also a *reference domain link* with the same *passage tag reference* `{1:8}` (see 2.1.11_Reference Domain Link). The *reference domain link* creates a persistent reference from some data in uifMainUserIF to the whole of cdMODefData.

Sheet `2` of the *DCD* represents some details of *operational modes* `1` and `3` that are not shown in *sheet* `1` so that it would not appear too cluttered. In *sheet* `2`, *passage* `3:1` corresponds to *passage* `3:4` on *sheet* `1` (`3:1[@1:4]`). Also, in *sheet* `2`, *passage* `1:1` corresponds to *passage* `1:18` on *sheet* `1` (`1:1[@1:18]`).

Operational Mode 1 (Initialisation)

1:1
Create the main dialog box but do not display it.
REF: uifOpenMainUIF()

1:2
Create and set up the main dialog box.
REF: @NewDialog()
OUT: dialog box ID, uifResFile, DIS_MUI

1:3
Show the main dialog box.
REF: uifShowMainUIF()

1:4
Start the main dialog box. Wait until user quits.
REF: @StartDialog()
OUT: true

1:5
Initialise the core data.
REF: mifInitialise()

1:6
Get the matrix object definition data reference.
REF: cdGetMatObjDefData()
IN: cdMODefData (ref)

1:7
Set the matrix object definition data reference to the dialog box.
REF: uifSetMatObjDefData()
OUT: cdMODefData (ref)

1:8
Set the matrix object definition data reference.
REF: muiSetMatObjDefData()
OUT: cdMODefData (ref)
DST: muiMatObjDefData := cdMODefData

1:9
Create the drawing plane and an empty invisible VIEW for displaying the matrix object.
REF: cdInitialise()

1:10
Create the drawing plane.
REF: vis_DPlane()
DST: @dpSetMseBtnFnt() // Set the mouse button handler.

1:11
Create the main VIEW.
REF: vis_VIEW()
DST: @vSetCloseVIEWFnt() // Define and set the close button handler.

1:12
Create new matrix object definition data.
REF: moddNewMODefData()
OUT: default num rows, default num columns
DST: Reset the constrainer data to default.

1:13
Create the matrix object component.
REF: vis_MatObject()

1:14
Delete the matrix object definition component.
REF: @visDelete()

1:15
Create a new matrix object definition component.
REF: vis_MatObjDef()
OUT: data from cdMODefData

1:16
Initialise the constrainer selection module.
REF: uifInitConsSel()

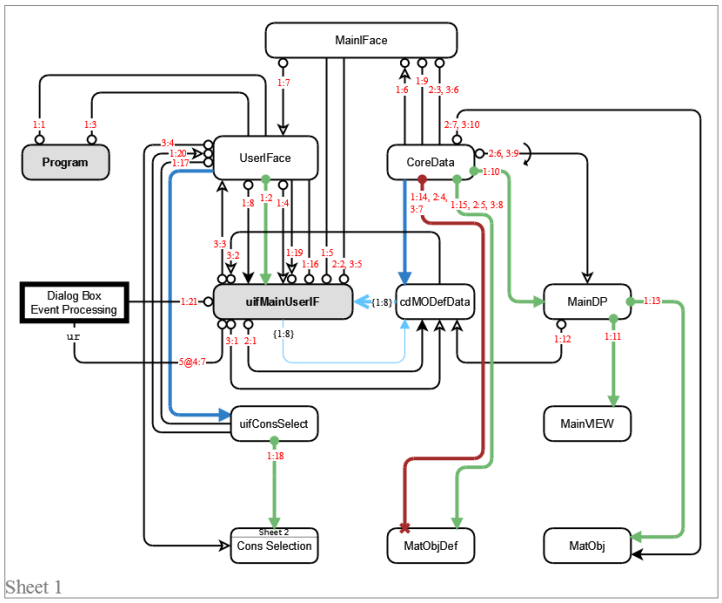
1:17
Create the drawing plane and a VIEW for displaying the constrainer selection "buttons".
REF: csInitialise()

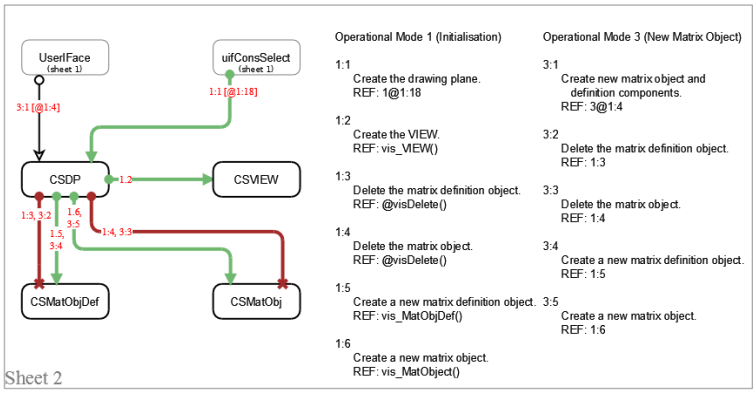
1:18
Create the drawing plane.
REF: vis_DPlane()
DST: @dpSetMseBtnFnt() // Set the mouse button handler.

1:19
Get the VIEW ID for the constrainer module.
REF: uifGetCSVIEWID()
IN: VIEW ID

1:20
Get the VIEW ID for this constrainer module.
REF: csGetVIEWID()
IN: VIEW ID

1: 21
Wait for the user to quit the program.
REF: Before return from @StartDialog()





Operational Mode 1 (Initialisation)

1:1
Create the drawing plane.
REF: 1@1:18

1:2
Create the VIEW.
REF: vis_VIEW()

1:3
Delete the matrix definition object.
REF: @visDelete()

1:4
Delete the matrix object.
REF: @visDelete()

1:5
Create a new matrix definition object.
REF: vis_MatObjDef()

1:6
Create a new matrix object.
REF: vis_MatObject()

Operational Mode 3 (New Matrix Object)

3:1
Create new matrix object and definition components.
REF: 3@1:4

3:2
Delete the matrix definition object.
REF: 1:3

3:3
Delete the matrix object.
REF: 1:4

3:4
Create a new matrix definition object.
REF: 1:5

3:5
Create a new matrix object.
REF: 1:6

Operational Mode 2 (Process)

2:1
Store the dialog box field data into the matrix object definition data.
REF: _muiStoreMatObjDefVals()
OUT: dialog box data.

2:2
Update the matrix object layout.
REF: mifUpdateLayout()

2:3
Update the matrix object layout.
REF: cdUpdateLayout()

2:4
Delete the matrix object definition component.
REF: 1:14

2:5
Create a new matrix object definition component.
REF: 1:15

2:6
Create extra images if required.
REF: _cdNewContImage()
OUT: index.

2:7
Attach the matrix object definition to the matrix object, and update the matrix object with the current object list.
REF: @matoAttachDef(), @matoGObjIDList
OUT: cdMatObjDefID, cdContGObjCIDs

Operational Mode 3 (New Matrix Object)

3:1
Create new matrix object definition data.
REF: moddNewMODefData()
OUT: num rows, num columns
DST: Reset the constrainer data to default.

3:2
Recall the matrix object definition data to the dialog box fields.
REF: _muiRecMatObjDefVals()
IN: matrix object definition data.

3:3
Create new matrix object and definition components.
REF: uifNewMatObj()
OUT: num rows, num columns.

3:4
Create new matrix object and definition components.
REF: csNewMatObj()
OUT: num rows, num columns.

3:5
Update the matrix object layout.
REF: 2:2

3:6
Update the matrix object layout.
REF: 2:3

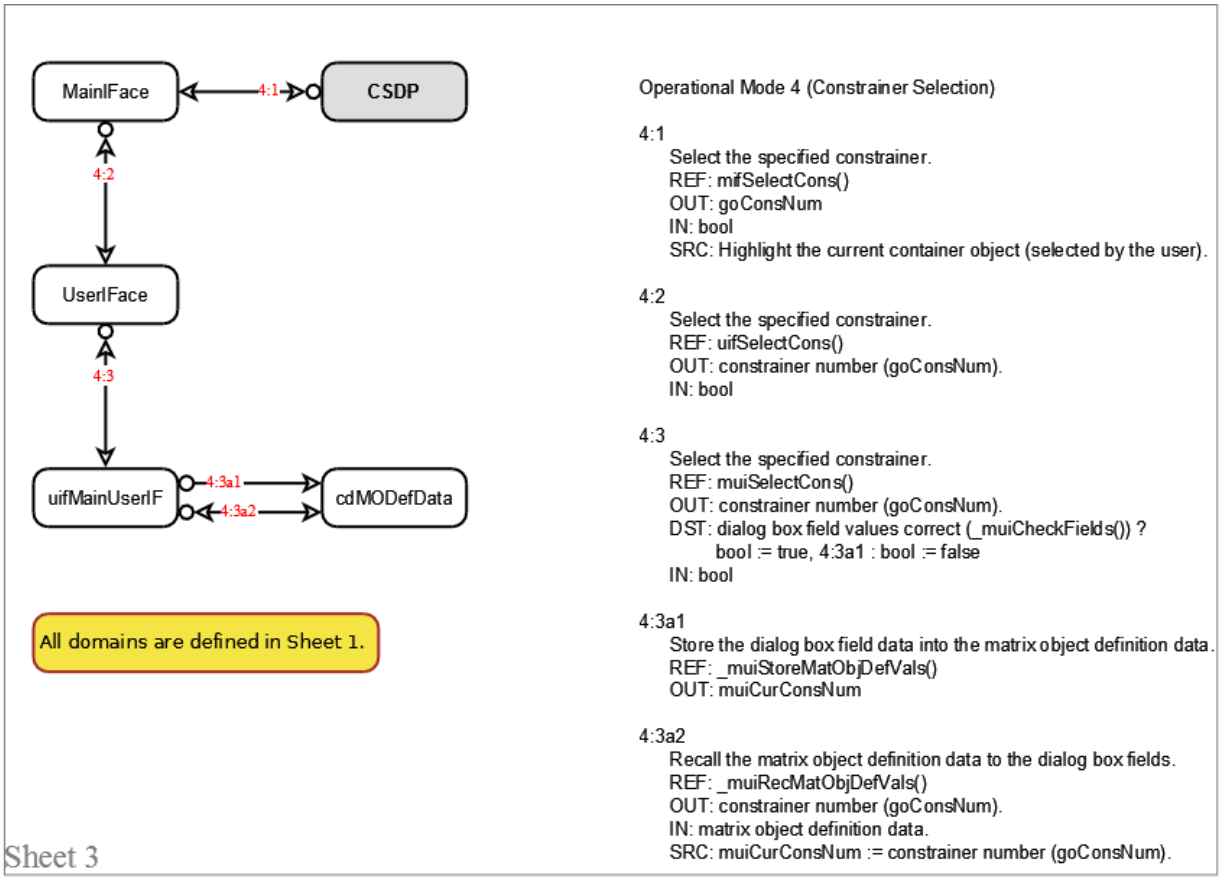
3:7
Delete the matrix object definition component.
REF: 1:14

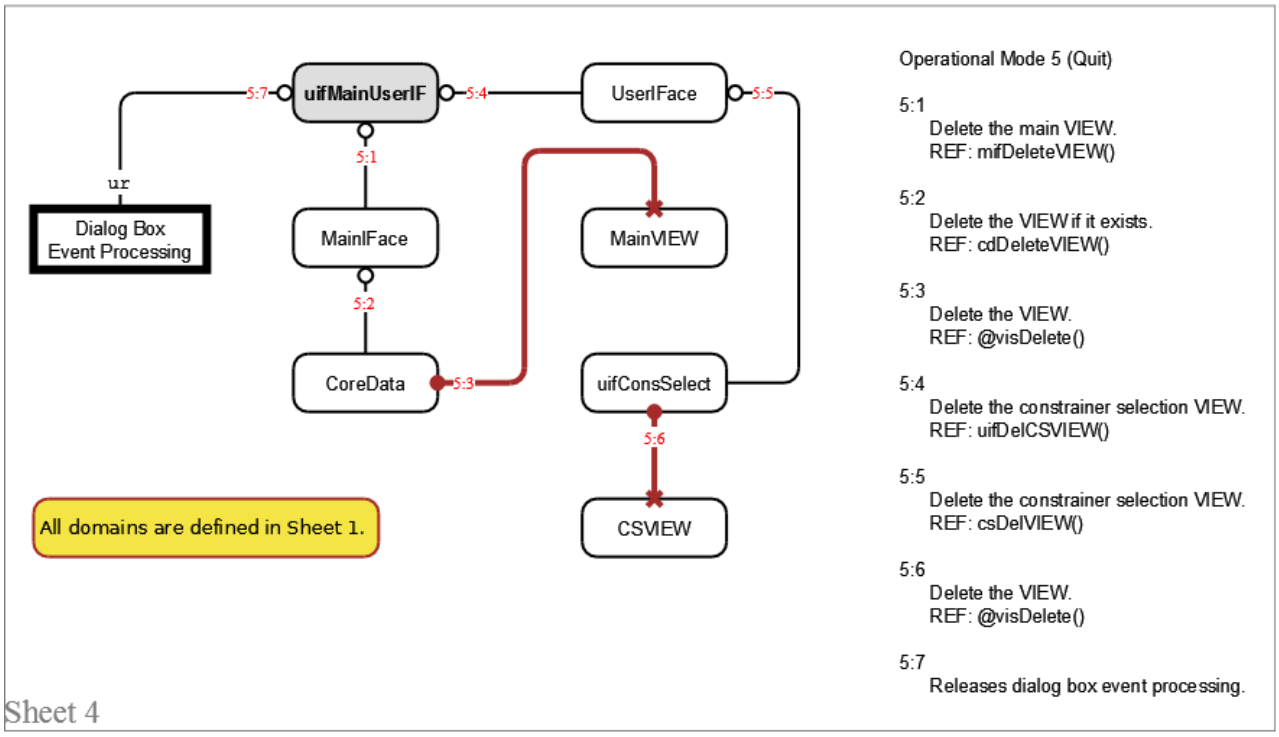
3:8
Create a new matrix object definition component.
REF: 1:15

3:9
Create extra images if required.
REF: 2:6

3:10
Attach the matrix object definition to the matrix object, and update the matrix with the current object list.
REF: 2:7

Sheets 3 and 4, below, represent *operational modes* 4 and 5, respectively, of the *DCD*. Notice that *passage* 5:7 of *sheet* 4 is referred to from *sheet* 1 by the *passage tag* 5@4:7.





The whole *DCD* above for the matrix toy program is quite detailed. The *DCD* designer decides how many *sheets* to use for a *DCD* and what to put on them. For example, the *DCD* above could have been constructed to show each *operational mode* in its own separate *sheet*. Or, even designed with more than one *sheet* for a given *operational mode*, as was done for *operational modes* 1 and 3. If a *DCD* is created in a diagrammatic software application program with layers, then each *operational mode* can be put into its own layer.

Appendix A

Software Architectures

A.1 Introduction

The architectural structure of a computer program or software system is the most important factor in determining the adaptability of that program or system for future enhancements. Most software is intended to be enhanced and modified to meet new circumstances. Attempting to modify a poorly structured piece of software usually results in ad-hoc complex structures that become difficult to enhance further. Such unnecessarily complex structures also promote unpredictability in the software design, which typically results in software malfunctions.

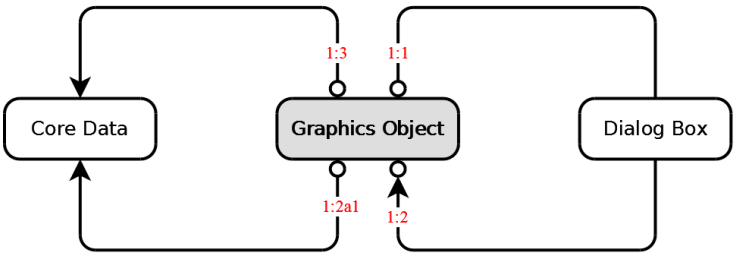
However, creating an appropriate architectural structure for a computer program or software system is not a straightforward process. The optimal structure depends on knowing how the software is going to be modified in the future. Such knowledge is not typically available, with the result that the optimal structure cannot easily be determined in advanced.

Despite not knowing the future trajectory of software enhancements, it is possible to specify an architectural structure for computer programs that will be reasonably stable for any future enhancements. Such a structure is based on the hardware bus-line architecture, and has the same advantages of being able to add new components with minimal effect on the existing structure. In this document, that architectural structure for software is called the "bus-line interface" architecture. That structure is compared with the typical structure used in software design called the "direct access" architecture, which is not so versatile but is simpler in design (at the beginning).

The *DCD* of the two architectures will be illustrated in the following sections based on the same hypothetical piece of software that modifies the text of a graphics object via a dialog box. The properties of the graphics object are also stored in a component of the software.

A.2 Direct Access Architecture

The Direct Access architecture is a typical software design architecture where various software components access data in other components directly. Designing a software system with such an architecture does not require much foresight — components can be created and accessed as the need requires.



Operational Mode 1

1:1
Show Dialog Box.

1:2
Get and Replace Text.
IN: text
SRC: Replace text.
size changed ? 1:2a1.

1:2a1
Store Size.
OUT: size

1:3
Store Text.
OUT: text

The *DCD* above uses the direct access architecture. It is simpler, but relatively less versatile. For example, the Graphics Object *domain* directly accesses the Core Data and Dialog Box *domains*. To use the Graphics Object *domain* in a different application program, the *domain* usually needs to be modified to remove the access to the other two *domains*. Also, if the Dialog

Box *domain* is changed to a different *domain*, the Graphics Object *domain* will also need to be modified to communicate with that new *domain*. With progressively more enhancements, the complexity of the architecture increases greatly, resulting in a more ad-hoc structure which becomes difficult to change. This architecture does not easily allow the swapping in and out of *domains* at run-time if the access to those *domains* is embedded in other *domains*.

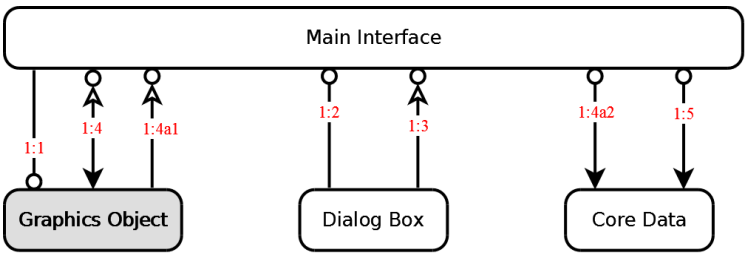
The major disadvantage of the direct access architecture is that the complexity of the architecture increases rapidly out of proportion as new modifications are made. There is also an element of arbitrariness in choosing how to access certain *domains*. For example, if a particular *domain* needs to access the data in another *domain*, it may do that directly, or it may do that through a third, or even a fourth, *domain*. As a result, there is no general scheme by which data is accessed by one *domain* from another.

Another disadvantage of the direct access architecture is that it is difficult to use a particular *domain* in another application program without needing to modify a copy of that *domain* to remove access to all the original connections to it. In other words, in a complex direct access architecture, the *data domains* cannot be easily reused in other application programs because those *domains* are specifically integrated into the original program.

The direct access architecture is suitable for simple software with only a few components, where the software is not going to be enhanced in the future. The Noughts and Crosses (tic-tac-toe) program shows an example of a direct access architecture (see 3.3_Noughts and Crosses).

A.3 Bus-line Interface Architecture

The Bus-line Interface architecture is a software design architecture consisting of a single interface component where all other major components have access to each other only via that interface component. Designing a software system with such an architecture requires some foresight for how the system may optimally evolve in the future.



Operational Mode 1

1:1
Replace Text

1:2
Show Dialog Box

1:3
Get Text
IN: text

1:4
Replace Text
OUT: text
IN: size changed (bool).
changed = true ? 1:4a1.

Operational Mode 1 (continued)

1:4a1
Get New Size
IN: size

1:4a2
Put Size
OUT: size

1:5
Put Text
OUT: text

The *DCD* above uses the bus-line interface architecture. Main Interface is the bus-line interface component. The architecture is more complicated, but also relatively more versatile. For example, the Graphics Object *domain* does not directly access any other (non-interface) *data domain*, and can therefore be used in a different application program with no (or very little) change. Also, the Dialog Box *domain* can be changed to a different *domain* without needing to

modify other *domains*; only parts of Main Interface may need to be modified. With progressively more enhancements, the complexity of the architecture does not increase much. Also, the bus-line interface architecture allows different versions of *domains* to replace existing ones at run-time, for example, when swapping one project to another, different instances of some *domains* need to be swapped for the new project. This can be done at the bus-line interface.

In the bus-line interface architecture, sub-components that are exclusive to parent components can have access to each other via the parent component. In this case, the parent component acts as a kind of mini bus-line interface for the sub-components.

A major advantage of the bus-line architecture is that, if designed with foresight, the complexity of the architecture does not increase rapidly as new modifications are made. Each major *domain* can access any other *domain* only through the bus-line interface. Consequently, there is less arbitrariness in choosing how to access other *domains*. For example, if a particular *domain* needs to access the data in another *domain*, it can only do that through one interface. As a result, it is easy to add additional *domains* to the architecture.

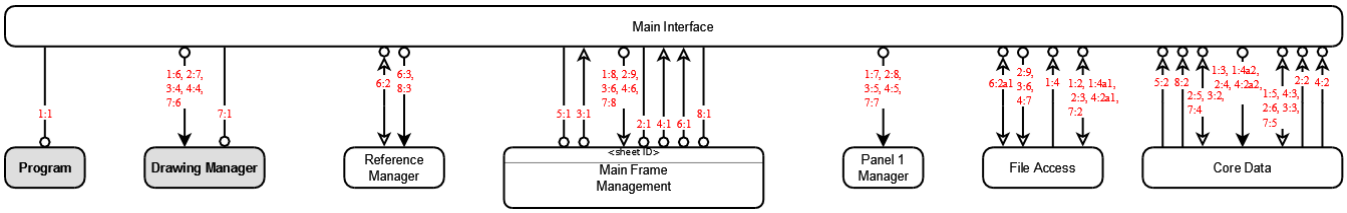
Another advantage of the bus-line interface architecture is that it allows the design of each *domain* to be independent of other *domains*. This makes it is easy to use any major *domain* in another application program without needing to modify a copy of that *domain*, except for the interface code with the bus-line interface. The consequence is that any major *domain* can easily be reused in other application programs also having the bus-line interface architecture.

The bus-line interface architecture is suitable for software with more than a few components, where the software is going to be enhanced in the future.

Examples of the Bus-line Interface Architecture

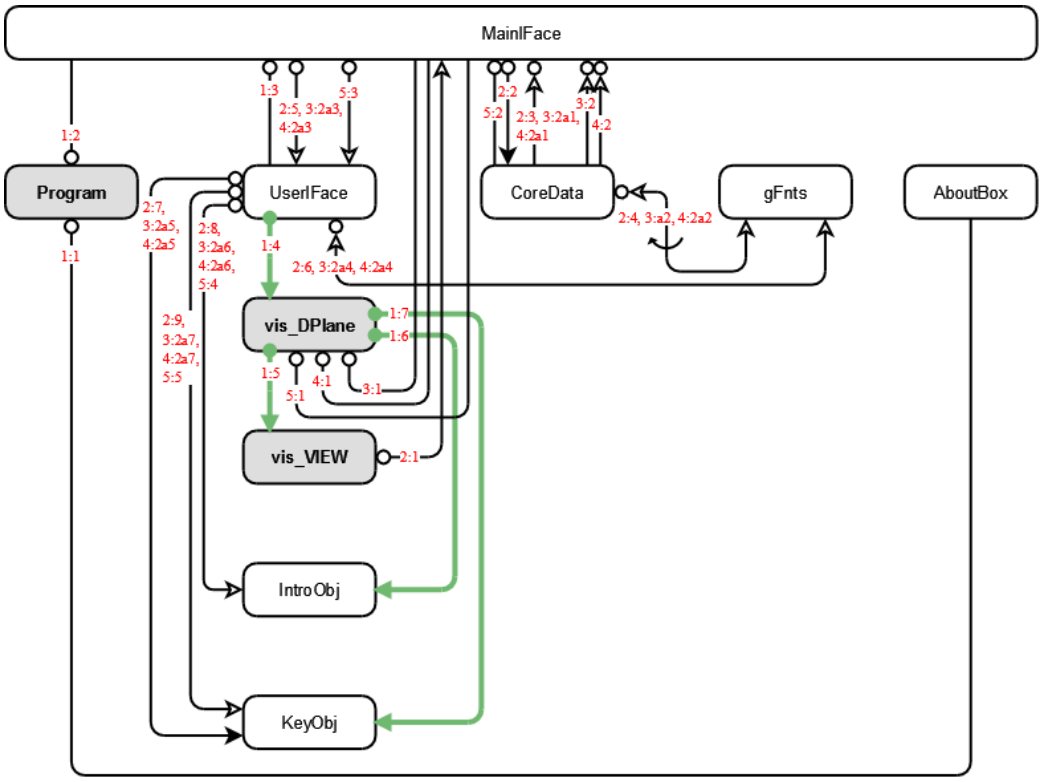
1. The following *DCD* is of a tutorial program that illustrates the VIS (Visual Interaction System) aspect of the ETAC™ programming language. Details of the eight *operational mode* tables are not shown. Also, the details of the Main Frame Management component are not shown. Main Interface is the bus-line interface.

The eight *operational mode* titles are as follows: Initialisation (1), Next Tutorial (2), Lesson Request (3), Specified Lesson (4), Run Script (5), Reference Topic (6), Welcome Page Click (7), Help (8).

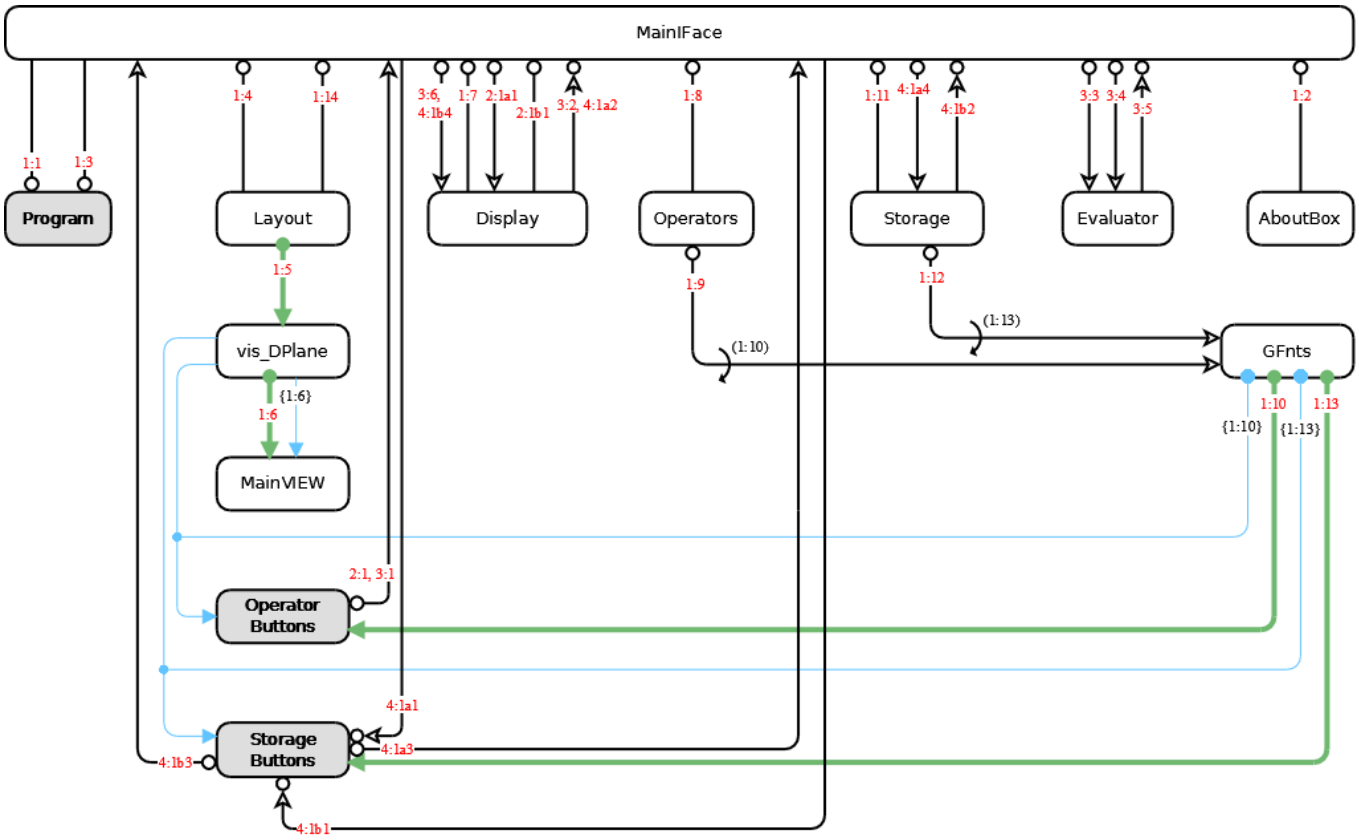


2. The following *DCD* is of a key-code toy program that presents the details of keyboard key presses and releases on a window. The program is intended to be implemented in the ETAC programming language using VIS (Visual Interaction System). The details of the five *operational mode* tables are not shown. The major components are the Program, UserIFace, and CoreData *domains*. Communication between those *domains* occurs via MainIFace, which is the bus-line interface. Notice that the *domains* vis_DPlane, vis_VIEW, IntroObj, and KeyObj are sub-*domains* created dynamically through the UserIFace *domain*. Also notice that communication between the sub-*domains* and the major *domains* occurs via the bus-line interface, not directly.

The five *operational mode* titles are as follows: Initialisation (1), Key Event (2), Previous Event (3), Next Event (4), Clear Event (5).



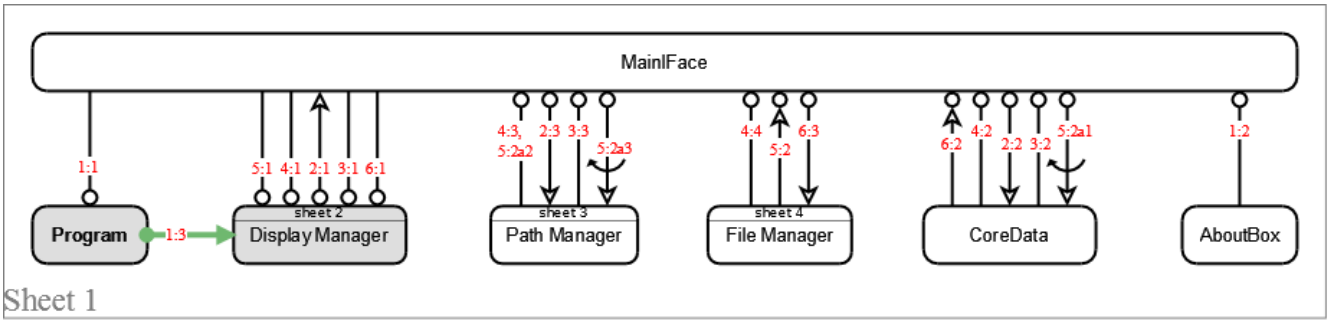
3. The following *DCD* is of a basic calculator program that has two storages and the change sign, reciprocal, and parentheses buttons. The four arithmetical operators obey the usual precedence — multiplication and division before addition and subtraction. The program is intended to be implemented in the ETAC programming language using VIS (Visual Interaction System). The three *operational mode* tables are not shown. The major *data domains* are shown in the row below the bus-line interface, MainIFace. Communication among those *domains* occurs via MainIFace only. The other *domains* are sub-*domains* created dynamically (except for GFnts). Communication between the sub-*domains* and the major *domains* occurs via the bus-line interface, not directly.

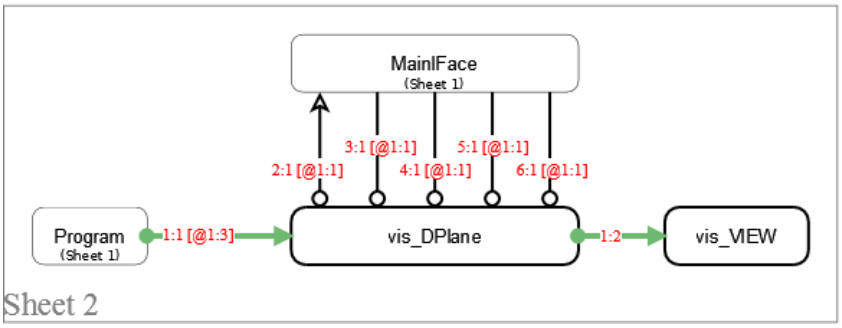


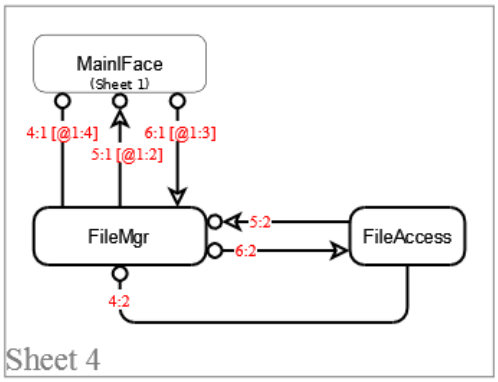
4. The following *DCD* is of a demonstration program that draws any number of polylines in a window. The drawing in each window can be saved to a file, and reloaded when desired. The program is intended to be implemented in the ETAC programming language using VIS (Visual Interaction System). The details of the six *operational mode* tables are not shown. The major *data domains* are shown in the row below the bus-line interface, MainIFace. Communication among those *domains* occurs via MainIFace only. The other *domains* are sub-*domains* created dynamically. Communication between the sub-*domains* and the major *domains* occurs via the bus-line interface, not directly.

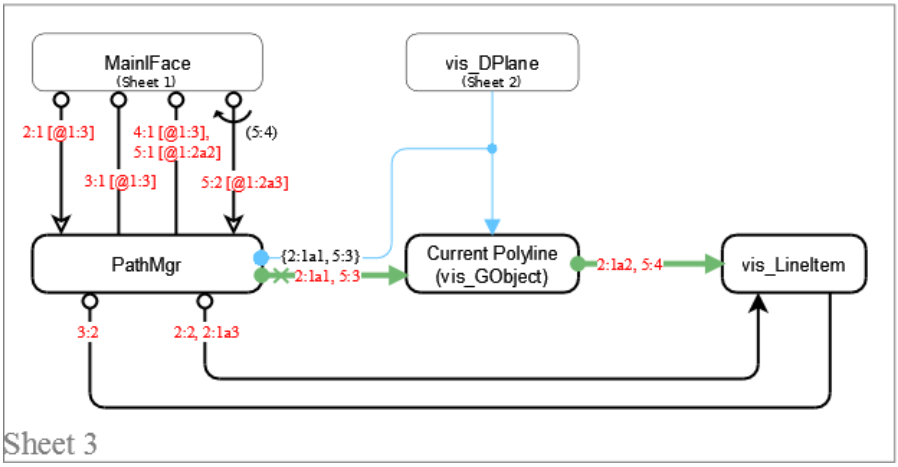
The six *operational mode* titles are as follows: Initialisation (1), Mouse Click (2), Mouse Double-click (3), New Drawing (4), Open Drawing (5), Save Drawing (6).

The three pseudo *data domains* (see **Pseudo Data Domains** under 2.1.1_Data Domains) Display Manager, Path Manager, and File Manager are represented in their own *diagram sheet* for convenience (and for illustration purposes in this case), even though they are logically part of the main *sheet* ‹Sheet 1› — all the *sheets* could have been combined into the main *sheet*.









Appendix B

Developing Programs Using DCDs

B.1 Introduction

A *DCD* (Data Control Diagram) is not only useful for showing the essential functioning of a piece of software, but is also useful in designing adaptable software.

The process of designing an application program involving a *DCD* can be divided into a number of steps. Ideally, the steps would be performed linearly without review. However, given the nature of programming languages, and other factors, the steps may need to be reviewed and modified any number of times as required.

The following five steps can be performed in designing an application program using a *DCD*.

1. **Overview of the application**. This is a written top-level overview of the application program from the user's perspective.
2. **User's specifications**. This is similar to a user's manual but with more detail, and describes the details of the application from the user's perspective. There are no programming concepts in this specification.
3. **Design architecture**. This specifies the architecture of application in terms of a *DCD*. The architecture is determined by the complexity and future adaptability of the program.
4. **Implementation**. This is the coded implementation of the application, beginning with an outline code first created from the partially designed *DCD*. The *DCD* can then be completed. The rest of the implemented code can then be created from the completed *DCD*.
5. **Testing**. This can be aided by the *DCD*.

The process of designing an application program using a *DCD* will be illustrated in this Appendix by an example. The program is a demonstration program that allows the user to draw any number of polylines in a window; the program is written in the ETAC™ programming language using VIS (Visual Interaction System).

B.2 Overview of the Application

A well designed application program needs to begin with some kind of overview of what the program is to accomplish from the user's perspective. The application name should also be decided at this stage, which, in this illustration, will be called PolylineDemo.

The following is the overview of PolylineDemo.

- The application program allows the user to draw any number of polylines in a window (called a "VIEW" in the ETAC programming language using VIS).
- Each polyline will have a default colour and width.
- There will be only one main window.
- The drawing in the window can be saved to a file and reloaded when desired.
- Any individual polyline can be moved and resized by the user but cannot be deleted.
- All the polylines in the window can be deleted at once.
- The window can be resized by the user.
- The window will have scroll bars and the user can scroll and zoom the drawing in and out.

This overview gives an idea of what the program is about.

B.3 User's Specifications

At this stage, the full details of the user's interaction with the program need to be specified. This is similar to a user's manual but with more detail. Also, the exact interface appearance of the visual elements of the program needs to be specified. The overview of the program can be used as a basis for this stage. The actual user's manual can be based on this user's specifications.

For PolylineDemo, the exact colour and thickness of the polylines needs to be specified. How the polylines are created needs to be specified. How the window is to be cleared needs to be specified. How the polylines are saved and loaded from a disk file needs to be specified. How the user is to move and resize a polyline needs to be specified. The mouse buttons that the user is required to press to scroll and zoom the image needs to be specified. And so on.

The polylines will be 0.2 units thick and black. Each polyline will be created by the user sequentially clicking the left mouse button to define the polyline vertices; the last point on the polyline will be indicated by a double-click; the next click will begin a new polyline. The window will be cleared (and all polylines deleted) by the user pressing the `Ctrl+N` keys on the keyboard. The current polylines are saved to a file by the user pressing the `Ctrl+S` keys. A polyline file is read into the application by the user pressing the `Ctrl+O` keys. Moving and resizing polylines features are built into VIS, as are scrolling and zooming of the drawing.

In general, a mock-up of the program should be created to test the specifications, even if the mock-up is just a number of graphical images.

B.4 Program Architecture

This is the stage at which the *DCD* is created. A *DCD* designer may consider a few proposals for the architecture before a final one is accepted. The *DCD* need not be completed before the implementation stage begins. For example, the *DCD* may be specified just enough for an implemented outline code to be created. The programming language used for implementation may need to be taken into account for the architecture. PolylineDemo will be implemented in the ETAC programming language using VIS (Visual Interpretation System).

For simple programs that are unlikely to change, the direct access architecture can be used. However, the bus-line interface architecture will be used for PolylineDemo. The reason for this is that the program can be used as a base for a more complex program with many more features, therefore the program needs to be adaptable.

The *DCD* for PolylineDemo needs to include the diagram itself, along with the appropriate *operational modes*, and also the class diagrams for the major class instances. The outline of the source code can then later be created from this information.

First of all, the major *data domains* need to be decided. Of course, being a bus-line interface architecture, the bus-line interface *domain*, MainIFace, needs to exist. The program must begin somewhere, so a Program *domain* needs to exist as well. Graphics application programs are typically designed with a main application window within which the graphics window exists. The main window is usually called a "frame window", and handles the menu, status bar, user panels, etc. However, the PolylineDemo program will have the graphics drawn directly to the client area of the frame window. So, there needs to be a *domain* for the main window called FrameMgr (frame manager) in PolylineDemo to handle user interactions. Each polyline consists of a number of points that need to be stored and managed to create the graphics to display. The management of the polylines is relatively independent of the frame window, so will have its own *domain*, PathMgr (polyline path manager). The polylines in a window can be written to a file, and also loaded from a file. Therefore, a *domain*, FileMgr (file manager), to manage the disk input and output needs to be created.

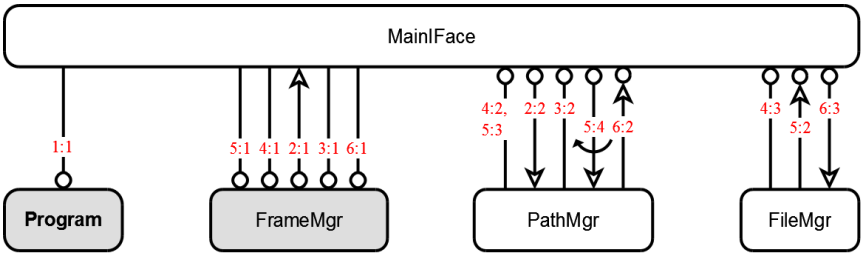
So, we have MainIFace, Program, FrameMgr, PathMgr, and FileMgr as the major *data domains*. MainIFace is responsible for the communication among the other major *data domains*,

as required for the bus-line interface architecture. Program is responsible for the initialisation of the major *domains*. FrameMgr is responsible for the user interaction with respect to the main window. PathMgr is responsible for creating and storing path information and displaying the polyline paths. FileMgr is responsible for transferring the polyline data to and from a disk file and obtaining the file path from the user.

The *operational modes* can be defined, which may require the creation of other *data domains*. For most application programs, an *operational mode* to initialise the *data domains* of the application is required. Most other *operational modes* are based on user events. There is a mouse click event, mouse double-click event, a window clearing event, a file loading event, and a file saving event. The following *operational modes* can therefore be defined for PolylineDemo:

1. Operational Mode 1 (Initialisation)
2. Operational Mode 2 (Mouse Click)
3. Operational Mode 3 (Mouse Double-click)
4. Operational Mode 4 (New Drawing)
5. Operational Mode 5 (Open Drawing)
6. Operational Mode 6 (Save Drawing)

The *passages* and *operational mode* tables can be created, at least as a first draft. The *DCD*, so far, is shown below.



The six *operational mode* tables, so far, are shown below.

Operational Mode 1 (Initialisation)

1:1
Initialise.
REF: mifInitialise()

Operational Mode 2 (Mouse Click)

2:1
Append a new point to the polyline.
REF: mifNewPoint()
OUT: point.

2:2
Append new polyline point.
REF: pmNewPoint()
OUT: point
DST: Save and display polyline.

Operational Mode 3 (Mouse Double-click)

3:1
End current polyline.
REF: mifEndPath()

3:2
End polyline.
REF: pmEndPath()

Operational Mode 4 (New Drawing)

4:1
New drawing.
REF: mifNew()

4:2, 5:3
Clear display.
REF: pmClear()

4:3
Reset file path.
REF: fmResetPath()

Operational Mode 5 (Open Drawing)

5:1
Open drawing.
REF: mifOpen()

5:2
Open file.
REF: fmOpen()
IN: file data (set of paths).

5:3
Clear display.
REF: 4:2

5:4
Create and display new path.
REF: pmNewPath()
OUT: path points.
NOTE: Repeat for each path.

Operational Mode 6 (Save Drawing)

6:1
Save drawing.
REF: mifSave()

6:2
Get path points for all paths.
REF: pmGetPaths()
IN: set of paths.

6:3
Save paths.
REF: fmSave()
OUT: set of paths.

At this point, the class diagrams of the major *domains* can also be created, as shown below.

MainIFace
Main Interface

`mifInitialise()`
Initialises the program.

`mifNew()`
Creates a new drawing.

`mifOpen()`
Opens a new drawing.

`mifSave()`
Saves the drawing.

`mifNewPoint(in pXPos:dec,in pYPos:dec)`
Appends a new point to the polyline.

`mifEndPath()`
Ends the current polyline.

PathMgr
Manages the visible path in the VIEW.

`pmNewPoint(in pXVal:dec,in pYVal:dec)`
Appends a new polyline point to the current path.

`pmEndPath()`
Ends the polyline.

`pmClear()`
Clears the display.

`pmNewPath(in pPointSeq:seq)`
Creates and displays a new path.
pPointSeq:
Sequence of path points.

`pmGetPaths(): seq`
Gets a list of the current polylines.

FileMgr
Manages file data.

`fmOpen(): seq`
Opens a file for loading.
Return:
Sequence of path points.

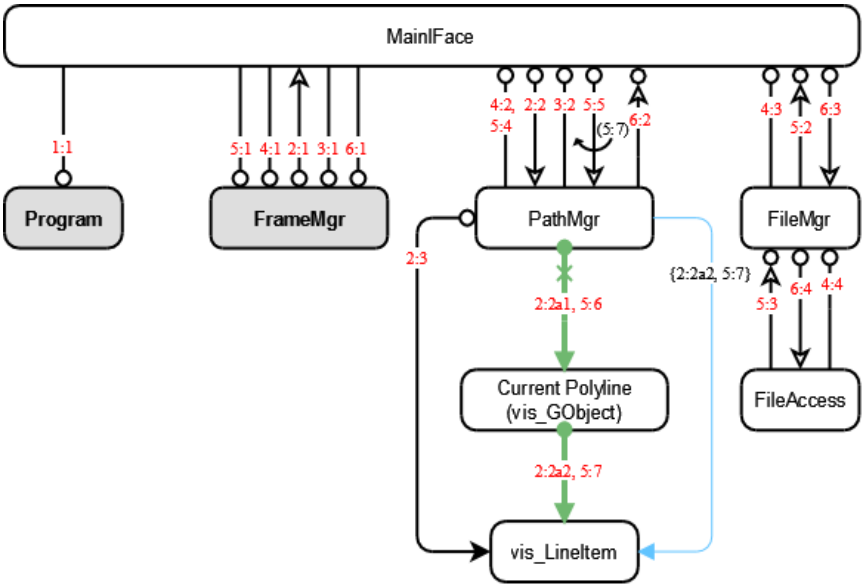
`fmSave(in pPathPoints:seq)`
Saves paths.
pPathPoints:
Sequence of path points.

`fmResetPath()`
Resets the file path.

The *DCD*, so far, can be considered complete, and can be used to implement PolylineDemo in most graphics programming languages. However, since PolylineDemo is to be implemented in the ETAC programming language, further sub-*domains* can be created specific to that language. In this illustration, the sub-*domains* belonging to PathMgr and FileMgr will be created.

In VIS of ETAC, drawings are displayed by the `vis_GObject` component, which is a data object that manages various graphics item sub-components like polylines. In VIS, a polyline is represented by the vis_LineItem component which is a sub-component of the Current Polyline component. So, these two components can be created as sub-*domains* under the PathMgr *domain*. In addition, it is convenient to have a dedicated *domain*, FileAccess, managed by FileMgr, that merely reads from and writes to data files, converting the raw data to the appropriate format for the file.

The *DCD* for the PolylineDemo program with the new *domains* is shown below.



The six *operational mode* tables are shown below. Some *passage tags* were needed to be renamed to account for the new *domains*.

Operational Mode 1 (Initialisation)

1:1
Initialise.
REF: mifInitialise()

Operational Mode 2 (Mouse Click)

2:1
Append a new point to the polyline.
REF: mifNewPoint()
OUT: point.

2:2
Append new polyline point.
REF: pmNewPoint()
OUT: point
DST: new polyline ? 2:2a1.

2:2a1, 5:6
Create new polyline graphics object and make current.
REF: vis_GObject()

2:2a2, 5:7
Create new line item.
REF: vis_LineItem()

2:3
Append new polyline point.
REF: liNewPoint()
OUT: point

Operational Mode 3 (Mouse Double-click)

3:1
End current polyline.
REF: mifEndPath()

3:2
End polyline.
REF: pmEndPath()

Operational Mode 4 (New Drawing)

4:1
New drawing.
REF: mifNew()

4:2, 5:4
Clear display.
REF: pmClear()
DST: Delete all polyline objects.

4:3
Reset file path.
REF: fmResetPath()

4:4
Reset file name.
REF: faResetFName()

Operational Mode 6 (Save Drawing)

6:1
Save drawing.
REF: mifSave()

6:2
Get path points for all paths.
REF: pmGetPaths()
IN: set of paths.

6:3
Save paths.
REF: fmSave()
OUT: set of paths.

6:4
Write data to file.
REF: faWrite()
SRC: Convert paths to raw data.
OUT: file data.

Operational Mode 5 (Open Drawing)

5:1
Open drawing.
REF: mifOpen()

5:2
Open file.
REF: fmOpen()
IN: file data (set of paths).

5:3
Get file data from file.
REF: faRead()
IN: file data or NULL
SRC: Convert data to set of paths.

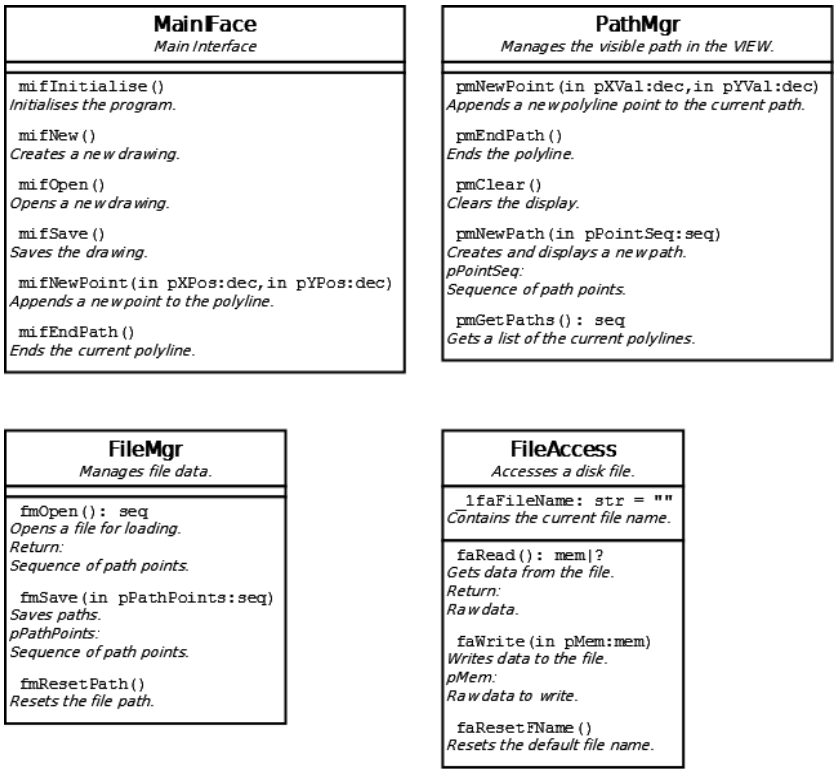
5:4
Clear display.
REF: 4:2

5:5
Create and display new path.
REF: pmNewPath()
OUT: path points.
NOTE: Repeat for each path.

5:6
Create new polyline graphics object and make current.
REF: 2:2a1

5:7
Create new line item.
REF: 2:2a2

The class diagrams are shown below.



The *DCD* above can potentially be constructed in more (or less) detail. It is up to the *DCD* designer to decide how much detail and what structure a *DCD* will have. The choice on the amount of detail and the structure of the *DCD* should depend on the nature of the program itself and its projected evolution.

After the PolylineDemo program is created, it can be modified using the *DCD*. For example, if cut, copy, and paste features are needed for the program, an appropriate *data domain* can be hooked onto the bus-line interface (MainIFace) for that purpose; the *domain* would then communicate to the other *domains* through that interface. Similarly, if a do\undo feature is required later, a suitable *domain*, hooked to the interface, can be defined for that purpose. Using the *DCD* to design the new features allows the *DCD* designer to quickly inspect the effects of those features on other *domains* and the program as a whole without needing to read complex source code where the effects of those features on different parts of the program can be overlooked.

B.5 Implementation

This is the stage at which the program source code is created. The *DCD*, along with the class diagrams, can be used to directly create a source code outline. All that needs to be done with that outline to obtain a workable program is to fill in the details.

The source code outline for the PolylineDemo program, written in the ETAC programming language, is shown below.

```
start_local;
ArgStr :-; [* Assign the stack argument string. *]

   [* Main Interface *]
   MainIFace :- data:
   {
      [* Initialises the program. *]
      mifInitialise :- fnt:()
      {
      };

      [* Creates a new drawing. *]
      mifNew :- fnt:()
      {
         [* Calls: pmClear() *] [* Calls: fmResetPath() *]
      };

      [* Opens a new drawing. *]
      mifOpen :- fnt:()
      {
         [* Calls: fmOpen() *]
      };

      [* Saves the drawing. *]
      mifSave :- fnt:()
      {
         [* Calls: pmGetPaths() *] [* Calls: fmSave() *]
      };

      [* Appends a new point to the polyline. *]
      mifNewPoint :- fnt:(pXPos[*dec*] pYPos[*dec*])
      {
         [* Calls: pmNewPoint(). *]
      };

      [* Ends the current polyline. *]
      mifEndPath :- fnt:()
      {
         [* Calls: pmEndPath() *]
      };
   };
```

```
[* Manages the main program VIEW and user events. *]
FrameMgr :- data:
{
   [* Prefix: frm *]

   [* NOTE: The following functions are called via event handler functions. *]
   [* Calls: mifNewPoint() *] [* Calls: mifEndPath() *]
   [* Calls: mifNew() *] [* Calls: mifOpen() *] [* Calls: mifSave() *]
};


[* Manages the visible path in the VIEW. *]
PathMgr :- data:
{
   [* Appends a new point to the current path. *]
   pmNewPoint :- fnt:(pXPos[*dec*] pYPos[*dec*])
   {
   };

   [* Ends the polyline. *]
   pmEndPath :- fnt:()
   {
   };

   [* Clears the display. *]
   pmClear :- fnt:()
   {
      [* Calls: pmNewPath() *]
   };

   [* Creates and displays a new path. *]
   pmNewPath :- fnt:(pPointSeq[*seq*])
   {
      [* Creates: vis_GObject component and its vis_LineItem sub-component. *]
   };

   [* Gets a list of the current polylines. *]
   pmGetPaths :- fnt:() [* => seq *]
   {
   };
};


[* Manages file data. *]
FileMgr :- data:
{
   [* Opens a file for loading. *]
   fmOpen :- fnt:() [* => seq *]
   {
      [* Calls: faRead() *]
   };

   [* Saves paths. *]
   fmSave :- fnt:(pPathPoints[*seq*])
   {
      [* Calls: faWrite() *]
   };

   [* Resets the file path. *]
   fmResetPath :- fnt:()
   {
      [* Calls: faResetFName() *]
   };
};
```

```
    [* Accesses a disk file. *]
    FileAccess :- data:
    {
       _1faFilename :- ""; [* Contains the current file name. *]

       [* Gets data from the file. *]
       faRead :- fnt:() [* => mem|? *]
       {
       };

       [* Writes data to the file. *]
       faWrite :- fnt:(pMem[*mem*])
       {
       };

       [* Resets the default file name. *]
       faResetFName :- fnt:()
       {
       };
    };

    [* PROGRAM *]

    fmInitialise();

end_local;
```

The program source code will not be developed any further in this illustration.

B.6 Testing

Designing and implementing software is a conceptually demanding process, and inevitably leads to errors. Testing a piece of software to make sure that it performs in the intended way is therefore necessary (at the time of this writing). A *DCD* representing the essence of how the software functions would make the testing process more efficient and accurate.

The various *operational modes* corresponding to a piece of software can be tested separately. Each *step* of an *operational mode* can be verified in the software. For example, *step* `6:2` in the *DCD* for PolylineDemo obtains all the polyline paths for subsequent writing to a file. Now, in the *DCD*, that *step* references the function `pmGetPaths()`, so a breakpoint can be put into the source code just after that function is called. Looking at the *DCD*, the function is called at the *step* before `6:2`, that is to say, it is called at *step* `6:1`. *Step* `6:1` references the function `mifSave()`, so the breakpoint would be placed in that function in the source code. When the breakpoint is triggered during testing, the return value of `pmGetPaths()` can be verified to make sure that all the polylines have been correctly returned by the function.

If the program does not operate in the expected way, the *DCD* is a very useful asset in locating the problem in the source code. Debugging code can easily be placed in the bus-line interface *domain* to monitor the communication among the other *domains*. The *DCD* is also used to predict the effects of any changes made in the source code.

Glossary

A

active state (refers to a *data control point*)
A *control point* is in the *active state* when it is moving along a *route*. Short term: *active*.

B

block domain
A *domain* that causes an *incoming control point* to be *blocked* until a *control point* from a specified *channel* enters the *block domain*.

See 1.2.6_Block Domain for details.

blocked state (refers to a *data control point*)
A *control point* is in the *blocked state* when it is waiting (not *ready* and not *active*) for an appropriate event before it can become *ready* or *active*. Short term: *blocked*.

C

channel record
A description of the changes that a *control point* causes in its *source* and *destination domains* as it moves along the *channel* connecting those two *domains*.

See 1.5_Channel Record for details.

child control point (in relation to a *parent control point*)
A *control point spawned* from the *parent control point*.

concurrent (refers to *data control points*)
Two or more *control points* that exist simultaneously are said to be *concurrent*.

See 1.4.2_Concurrent Control Points for details.

concurrent path group
A group of *control paths* that is designated to allow *concurrent control points* to move along the *paths* of that group. No *path* can belong to more than one such group. Short term: *path group*.

connector
A uniquely identifiable entity that associates any two *domains* (except for a *link assignment*) for a defined purpose.

See 1.3_Connectors for details.

control gate
A feature of a *control point channel* such that, when a *control point* moves through that *channel*, all other *control points* in the *DCM* become temporarily *idle* until that *control point* returns to its *source domain* (the *idle control points* then become *ready*). Short term: *gate*.

control path
The set of all possible *control point routes* that a particular *data control point* can move along. Short term: *path*.

control point channel
A type of *connector* that associates a *source domain* and *destination domain*, allowing *control points* to "move" from the *source domain* to the *destination domain* and back. Short term: *channel*.

See _ for details.

control point route
A specified sequence of *domains* for which a particular *control point* is desired and permitted to visit in the order presented in the sequence. A *passage* is specified between each pair of *domains*. The start of the *route* is called the 'origin domain'. Short term: *route*.

control point state
The state in which a *control point* is in at any one moment. The possible states are: *active*, *idle*, *ready*, and *blocked*. Short term: *state*.

D

data control point
An imaginary entity that signifies changes in the data state of different *domains* at different points in time. Short term: *control point*.

See 1.4_Data Control Points for details.

data control step
Indicates the movement of a *data control point* along a *passage*. Short term: *control step*.

data domain
An entity that contains data units, each having a range of possible values. Short term: *domain* (only when the context is understood to imply a *data domain*).

See 1.2.1_Data Domain for details.

DCD
A shorthand for Data Control Diagram.

See 2_Data Control Diagram for details.

DCM
A shorthand for Data Control Model.

See 1_Data Control Model for details.

destination domain
One of the two *domains* that a *connector* associates. The type of *connector* determines which of the two *domains* is the *destination domain*. Note that a *link assignment* does not have a *destination domain*.

diagram sheet
A *diagram sheet* contains some part of the whole *DCD*. Short term: *sheet*.

See 2.3_Diagram Sheets for details

divider domain
An entity that allows new *control points* to be spawned from existing ones. Short term: *divider*.

divider link
A *connector* that associates a *wait domain* and a *linked-wait divider domain*.

See 1.3.7_Divider Link for details.

domain
A *no-wait divider domain*, *wait divider domain*, *linked-wait divider domain*, *wait domain*, or *block domain*. Sometimes 'domain' refers to a *data domain* when the context is understood to mean a *data domain*.

See 1.2_Domains for details.

domain deletion connector
A type of *connector* that indicates that a *control point* in the *source domain* deletes the *destination domain*.

See 1.3.4_Domain Deletion Connector for details.

dynamic domain connector
A type of *connector* that indicates that a *control point* in the *source domain* creates the *destination domain*.

See 1.3.3_Dynamic Domain Connector for details.

I

idle state (refers to a *data control point*)
A *control point* is in the *idle state* when it is not *active* and not *ready* and not *blocked*. Short term: *idle*.

incoming channel (in relation to a *domain*)
A *control point channel* whose *destination domain* is the *domain* being referred to.

incoming control point
A *control point* on an *incoming channel*.

incoming data
The *destination domain* data copied by a *control point* as it returns from the *destination domain* to the *source domain*. The copied data is passed to the *source domain*. See 1.4_Data Control Points for more information.

L

link assignment
A *connector* that indicates the creation of a *reference domain link* between two *data domains*.

See 1.3.8_Link Assignment for details.

linked-wait divider domain
A *divider domain* that is linked to one or more *wait domains*. Short term: *linked-wait divider*.

See 1.2.4_Linked-wait Divider Domain for details.

N

no-wait divider domain
An entity that allows new *control points* to be spawned from existing ones without waiting for the new *control points* to return. Short term: *no-wait divider*.

See 1.2.2_No-wait Divider Domain for details.

O

operational mode
Used in a *DCD*, an *operational mode* is a set of *passages* involved in a named operation or unique number defined by the *DCD* designer. An *operational mode* is specified by a list of *passage tags* and (usually) the operation name or unique number.

outgoing channel (in relation to a *domain*)
A *control point channel* whose *source domain* is the *domain* being referred to.

outgoing control point
A *control point* on an *outgoing channel*.

outgoing data
The *source domain* data copied by a *control point* as it moves from the *source domain* to the *destination domain*. The copied data is passed to the *destination domain*. See 1.4_Data Control Points for more information.

P

parent control point (in relation to a *child control point*)
The *control point* that *spawns* a *child control point*.

passage
A *control point channel*, *dynamic domain connector*, or *domain deletion connector*.

passage tag
Used in a *DCD*, a label on a *passage* to identify that *passage*.

See 2.1.15_Passage Tags and References for details.

passage tag reference
Used in a *DCD*, a reference to a *passage tag* associated with a *reference domain link*, *link assignment*, or *supplied domain connector*.

See 2.1.15_Passage Tags and References for details.

R

ready state (refers to a *data control point*)
A *control point* is in the *ready state* when it is not currently *active* but can become *active* at any moment. Short term: *ready*.

reference domain link
A type of *connector* that indicates that there exists a reference from the data in the *source domain* to the *destination domain*.

See 1.3.6_Reference Domain Link for details.

S

source domain
One of the two *domains* that a *connector* associates. The type of *connector* determines which of the two *domains* is the *source domain*.

spawn (refers to a *data control point*)
A *data control point* that creates a new, possibly independent, *data control point*.

See 1.4.1_Spawned Control Points for details.

static domain connector
A type of *connector* that indicates that the *destination domain* is created automatically when the *source domain* is created.

See 1.3.2_Static Domain Connector for details.

supplied domain connector
A type of *connector* that indicates that a reference to the *source domain* is temporarily supplied to the *destination domain*.

See 1.3.5_Supplied Domain Connector for details.

W

wait domain
An entity that causes an *incoming control point* to wait for the new *control points* created by all of the associated *linked-wait dividers* to return.

See 1.2.5_Wait Domain for details.

wait divider domain
An entity that allows new *control points* to be *spawned* from an existing one but waits for the new *control points* to return before the *parent control point* returns.

See 1.2.3_Wait Divider Domain for details. Short term: *wait divider*.