SDP Execution Framework Design

Document Number..... .....SKA-TEL-SDP-0000015
Document Type..... .....DRE
Revision..... 02
Authors.....Andreas Wicenec, Dave Pallot, Rodrigo Tobar, Kevin Vinsen, Chen Wu, Paul Alexander, Simon Ratcliffe, Bojan Nikolic
Release 2016-07-20
Document Classification..... ..... Unrestricted
Status..... ..... Released

Lead Author	Designation	Affiliation
A Wicenec	DATA Team Lead	ICRAR
Signature & Date:		

Owned by	Designation	Affiliation
B Nikolic	SDP Project Engineer	University of Cambridge
Signature & Date:		

Approved by	Designation	Affiliation
P Alexander	SDP Project Lead	University of Cambridge
Signature & Date:		

Released by	Designation	Affiliation
Paul Alexander	SDP Project Lead	University of Cambridge
Signature & Date:		

Version	**Date of Issue**	**Prepared by**	**Comments**
01C	2016-03-24	A Wicenec	prepared for SDP delta-PDR
02	2016-07-20	A Wicenec	Prepared for SDP delta-PDR sign-off

ORGANISATION DETAILS

Name	Science Data Processor Consortium
Address	Astrophysics Cavendish Laboratory JJ Thomson Avenue Cambridge CB3 0HE
Website	http://ska-sdp.org
Email	ska-sdp-pa@mrao.cam.ac.uk

Table of Contents

1 List of Figures 6

2 List of Tables 7

3 List of Abbreviations 7

4 References 8

4.1 Applicable Documents 8

4.2 Reference Documents 8

5 Scope of the Execution Framework 9

6 Concepts and Background 11

6.1 Traditional Dataflow Designs 11

6.2 Data-driven Design 11

6.3 Graph Representation 12

6.4 Graph-based Data-Driven Architecture 12

7 Functional view 14

8 Dynamic Views 17

9 Graphs 19

9.1 Logical Graph 19

9.1.1 Construct properties 20

9.1.1.1 Control flow constructs 20

9.1.1.2 Graph Repository 20

9.2 Selecting a Template and Defining the Logical Graph 21

9.3 Translation 21

9.3.1 Basic steps 21

9.3.2 Algorithms 22

9.4 Physical Graph 23
9.5 Execution 23
10 Scalability, Scheduling and Drop Islands 24
10.1 Drop Managers 24
10.2 Drop Islands and Scalability 25
10.3 Scheduling and Load Balancing 25
10.4 Drop Islands and Scheduling 26
10.5 Node Drop Managers and Scheduling 27
10.6 Summary 27
11 Drop design 27
11.1 Drop High-Level Design 27
11.2 Drop Class Model 29
11.3 Representation of the Physical Graph 30
11.4 Input/Output 31
11.5 Drop Events 31
11.6 Drop Component Interface 32
12 State Views 32
12.1 Execution Framework State View 32
12.1.1 Execution Orchestration 32
12.2 Physical Graph State View 33
12.3 Drop State Diagram 34
12.4 Drop Events 36
13 Drop Lifecycle Management 36
14 Discussion of and Allocation of Functions to Products 37
15 Discussion of and Allocation of Requirements To Functions 38

16 Interfaces 40

18 Discussion of Element Risks 40

1 List of Figures

Figure 1: Simplified context diagram of the Execution Framework. 10

Figure 2: High-level functional breakdown of the SDP Execution Framework. 14

Figure 3: Class diagram of the Logical Graph Template system. 15

Figure 4: Physical graph deployment. 16

Figure 5: Sequence diagram of the interaction between TM, LMC and the Execution Framework. 18

Figure 6: An example of a logical graph with data constructs. 10

Figure 7: A nested-Loop (minor and major cycle) example. 20

Figure 8: Class diagram of the Drop Managers. 24

Figure 9: Block diagram of an abstract Drop. 28

Figure 10: UML class model of the Drop system employed by the Execution Framework. 30

Figure 11: State machine diagram of the Execution Framework. 33

Figure 12: The various states and transitions a physical graph can take on during its life-cycle. 34

Figure 13: The Drop state diagram is the lowest level of state machines within the Execution Framework. 35

Figure 14: The resilience states of the Drop system are captured as Phases. 37

Figure 15: Functional breakdown of the Execution Framework. 38

2 List of Tables

Table 1: Allocation of functions to products 38

Table 2: Allocation of requirements to functions. 40

3 List of Abbreviations

Abbreviation	**Expansion**
ALMA	Atacama Large Millimeter Array
API	Application Programming Interface
CSP	Central Signal Processor
DAG	Directed Acyclic Graph
DFMS	Dataflow Management System
DoP	Degree of Parallelism
DSL	Domain Specific Language
ESO	European Southern Observatory
FLOPS	Floating Point Operations per Second
HST	Hubble Space Telescope
IDM	Island Drop Manager
I/O	Input/Ouput
Ln	Level n
LOFAR	Low Frequency Aperture Array
LMC	Local Monitoring and Control
MDM	Master Drop Manager
MWA	Murchison Wide-field Array

NDM	Node Drop Manager
OID	Object Identifer
PI	Principal Investigator
PIP	Pipelines
REST	Representational State Transfer
RFI	Radio Frequency Interference
SDP	Science Data Processor
SKA1	Square Kilometre Array Phase 1
TBC	To Be Confirmed
TM	Telescope Manager
UID	Unique Identifer
UML	Unified Modelling Language
URI	Uniform Resource Identifier
VLA	Very Large Array

4 References

4.1 Applicable Documents

The following documents are applicable to the extent stated herein. In the event of conflict between the contents of the applicable documents and this document, **the applicable documents** shall take precedence.

Reference Number	**Reference**
AD01	SKA-TEL-SDP-0000013 SDP Architecture

4.2 Reference Documents

The following documents are referenced in this document. In the event of conflict between the contents of the referenced documents and this document, **this document** shall take precedence.

Reference Number	Reference
RD01	SKA-TEL-SDP-0000023 SDP Preservation Design
RD02	SKA-TEL-SDP-0000027 SDP Pipelines Design
RD03	SKA-TEL-SDP-0000026 Local Monitor and Control Design
RD04	SKA-TEL-SDP-0000018 Data Processor Platform Design
RD05	SKA-TEL-SDP-0000024 Data Challenge Supplement
RD06	Imaging SKA-scale data in three different computing environments, doi:10.1016/j.ascom.2015.10.007
RD07	SKA1 LOW SDP - CSP Interface Control Document
RD08	SKA1 MID SDP - CSP Interface Control Document
RD09	https://github.com/casacore/casacore
RD10	SKA-TEL-SDP-0000082 SDP Memo: Data-Driven Architecture Prototyping Report
RD11	Static scheduling algorithms for allocating directed task graphs to multiprocessors, doi:10.1145/344588.344618
RD12	Multilevel k-way Partitioning Scheme for Irregular Graphs, doi:10.1006/jpdc.1997.1404
RD13	A comparison of general approaches to multiprocessor scheduling, doi:10.1109/IPPS.1997.580873

5 Scope of the Execution Framework

The Execution Framework is part of the SDP Processor Software product and provides the framework and infrastructure for the various SDP Processing Capabilities[1] to be performed and executed. This includes the following broad range of capabilities:

- Near real-time calibration capability
- Fast-imaging/slow-transient capability
- Imaging capabilities
- Pulsar and Fast Transient Search capabilities
- Pulsar timing capability

[1] A SDP Processing Capability consists of connected pipelines (e.g. Receive, Imaging and Preserve) and supporting functions. See also the definition in the SDP Architecture document. In the remainder of this document we will use 'capability' for short.

While the SDP Architecture document mostly concentrates on ¯mY¨ the SDP needs to do, the Execution Framework document mostly contains the architecture of mz¯ we are proposing to tackle the SDP processing and data management challenges. As such this document is quite technical and might appear abstract in a number of aspects. The connection point to the main Architecture document are the SDP capabilities, which represent concrete instances of pipeline descriptions.

5.1 Document Overview

This document discusses first the fundamental architectural principles we have developed for the Execution Framework. Then there is a description of how the SDP capabilities will be represented and encoded in graphs and how these graphs are managed. Then there is a set of functional and dynamic views. Section 9 then describes the technical architecture of the graph framework, including a potential development environment. Section 9.3 concentrates on the translation, scheduling and deployment of a capability into an executable environment on a given platform. Section 10 explains our approach to tackle the management, scalability, load balancing and failover of the system. Section 11 finally goes down to the most basic entities of this architecture and describes their design in detail. Section 12 provides various state views for the whole Execution Framework and its parts.

The Execution Framework instantiates the interfaces to the CSP to receive the various bulk data streams (as part of explicit Receive Pipelines) and the SDP Preservation to store and maintain the final products.

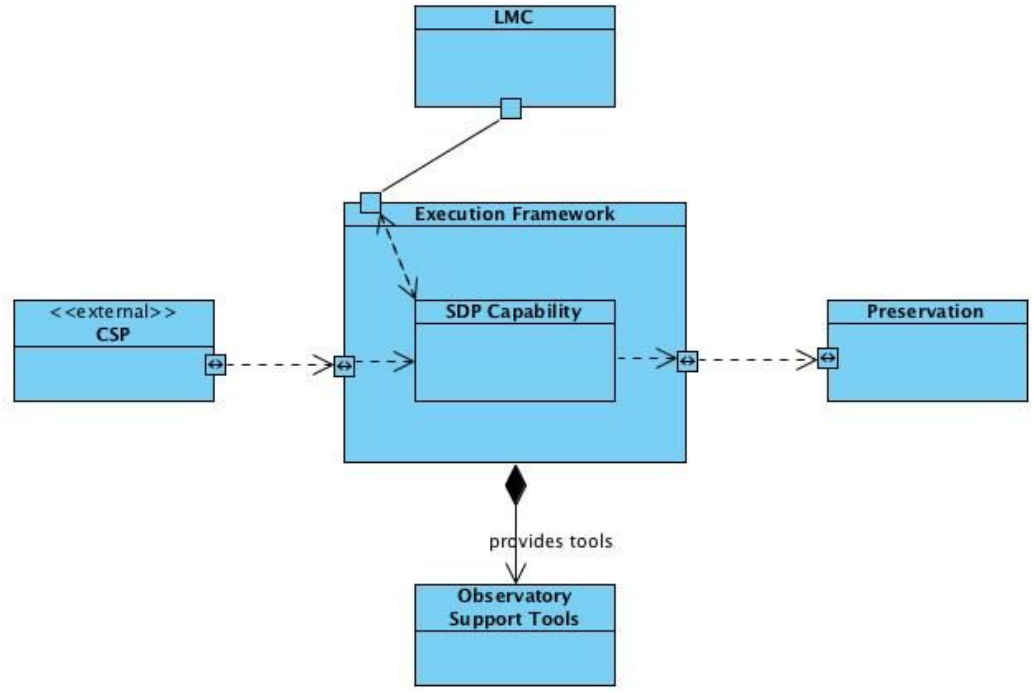

Figure 1: Lnw¢vnknfd bzx¨f°¨ dnY I¥Yw zk ¨mf . °fb©¨nzx 3¥Ywf¯z¥uÎ 6x ¢Y¥¨nb©vY¥ ¨mf nx¨f¥kYbf¦ ¨z ¨mf => * Y¥f xz¨ ¦mz¯x nx Yx± df¨Ynv nx ¨mn¦ ®nf¯Î 3z¥ ¦nw¢vnbn¨± ¨mf dnY I¥Yw Yv¦z zxv± ¦mz¯¦ Y ¦nx I vf bY¢Yanvn¨±¨¯mf¥fY¦ ¨mf . °fb©¨nzx 3¥Ywf¯z¥u ¯nv Ywz¯ ¨mf f°fb©¨nzx zk w©v¨n¢vf ¢Y¥Yw fv Yxd bzwwfx¦Yv bY¢Yanvn¨nf¦Î

Capabilities are required to run independently as well as commensally, depending on the actual observatory scheduling. In addition there might be multiple of the same capabilities running at the same time due to the subarray functionality of the SKA. Typically there will also be a whole set of variations of these broad capability categories, which will allow for instance for the processing of mosaics, drift scans and other observing modes requiring special processing of the data.
In order to address the scale of the SKA data flow and to efficiently exploit the intrinsic parallelism of radio astronomical data processing the SDP is following a data driven execution design.
This document describes the graph oriented and data driven architecture as well as the underlying data driven execution framework, including the interfaces to capability (pipeline) components and the TM/LMC Scheduling Block system.

6 Concepts and Overview of the Data-Driven Architecture

This section briefly introduces key concepts and motivations underpinning the Execution Framework design.

6.1 Traditional Dataflow Designs

A traditional dataflow computation model does not explicitly place any control or constraints on the order or timing of operations beyond what is inherent in the data dependencies among compute tasks. The removal of explicit scheduling of compute tasks in the dataflow model has opened up new (e.g. parallelism) opportunities that were previously masked by ñartificialò control flow imposed by applications or programmers. A similar example is the `make` tool, where the programmer focuses on defining each target and its dependencies. The burden of exploring parallelism to efficiently execute many individual compiling tasks in a correct order lies within the responsibility of the `make` utility.

6.2 SDP Data-driven Design

In developing the Execution Framework architecture, we have extended the “traditional” dataflow model by integrating data lifecycle management, graph execution engine, and cost-optimal resource allocation into a coherent data-driven framework. Concretely, we have made the following changes to existing dataflow models:

- Unlike traditional dataflow models that characterise data as “tokens” moving across directed edges between nodes, we instead model data as nodes, elevating them to be actors who have autonomy to manage their own lifecycles and trigger appropriate “consumer” applications based on their own internal (persistent) states. In our graph model, both application (task) and data nodes are called **,¥z¢¦ÿ**. The edges in our graphs thus represent **,¥z¢ .®fx¨¦**.
- We also introduce a small number of control flow graph nodes at the logical level such as Scatter, Gather, GroupBy, Loop, etc. These additional control nodes allow pipeline developers

[2] **NOTE:** In the following paragraphs the notion of **,¥z¢¦** is used extensively, they are formally introduced in the Drop design section below. For the understanding up to that point it is sufficient to regard them as software wrappers around both data and applications and providing the binding methods to allow the Execution Framework to function.

to systematically express complex data partitioning and event flow patterns based on various requirements and science processing goals. More importantly, we transform these control nodes into ordinary Application Drops at the physical level. Thus they are almost completely transparent to the underlying graph/dataflow execution engine, which focuses solely on exploring parallelisms orthogonal to these control nodes placed by applications. In this way, the Data-Driven framework enjoys the best from both worlds - expressivity at the application level and flexibility at the dataflow system level.

- Finally, we differentiate between two kinds of dataflow graphs - Logical Graph and Physical Graph. While the former provides a higher level of computation abstraction in a resource-independent manner, the latter represents the actual execution plan consisting of interconnected Drops mapped onto a given set of hardware resources in order to meet performance requirements at minimum cost (e.g. power consumption).

6.3 Graph Representation

Graphs are a standard structure in mathematics: they consist of vertices and edges that connect two vertices. Furthermore those edges may be directed.

For all capabilities, the architecture employs a data-driven processing model in order to achieve the required performance and exploit data parallelism. Such a data-driven approach is well represented by directed graphs, with the vertices representing processing or other actions on the data (so-called Actors) and the edges represent data dependencies (Channels). The directed graphs are acyclical to enable a deterministic partial ordering which in turn determines the execution flow of the graph and facilitates parallelisation.

In the data-driven processing system represented by a graph. Actors process data and act as sources and sinks for data. Actors may be understood as individual processing components acting on data after the required data become available. Channels encapsulate the mechanism by which data is made available between the Actors. Channels are typed and inputs and outputs of Actors must have the same types as the Channels joining them. Actors must be implemented in such a way that they do not need to know their position in the overall graph and hence the Actors always produce the same outputs given the same inputs. This is called referential transparency.

6.4 SDP Graph-based Data-Driven Architecture

Combining the data driven approach with the graph representation, the implementation of the data-driven processing architecture requires translation of these abstract concepts into a system which relates the processing graph to a specific realisation of processes coupled to data locality. Furthermore in a practical implementation it is essential to consider two further aspects:

- A real system will require some level of load balancing for an efficient implementation of the data-driven architecture. This is realised in the architecture by a hierarchical and potentially partially dynamical partitioning of the graph
- The architecture needs to be robust to partial failures of the system again requiring some dynamic partitioning of the graph.

To realise this we introduce the following functions:

- Graph preparation

- ð Graph partitioning
- ð Graph deployment and execution.

The following sections briefly define the various concepts used by the Execution Framework.

6.4.1 Logical Graph Template

A Logical Graph Template is a compact representation of the logical operations in a processing pipeline assuming no restrictions on the underlying hardware resources. Logical Graph Templates encode the sequence of the pipeline components to be executed, including the placement and parameterisation of explicit scattering and gathering components. Staff Scientists will have access to Observatory Support Tools that contain graphical user interface applications and pipeline components to help them develop Logical Graph Templates. After testing and an official release, the Logical Graph Templates will be offered to the PIs through the observation preparation tool as standard SKA processing capabilities. Each of these capabilities are also associated with a standard set of data products which, on successful completion of an execution, will be preserved in the SDP Preservation function. During SDP construction the current best practise Logical Graph Templates will be a bundled with the Observatory Support Toolset. Note that Staff Scientists do not have to understand the details contained within the Execution Framework or pipeline components themselves, this is the job of the Software Engineers developing the components.

6.4.2 Logical Graph

By combining a **=zIꞑbY∨ 4¥Y¢m Mfw¢∨Y¨f** and observation specific information contained in the Local Telescope State, the Logical Graph Generator produces a logical graph.

A Logical Graph defines the processing to be done for this processing run, including the specific drops required and the parameters and states that define this observation. The Logical Graph does not contain any hardware specific or locality information.

6.4.3 Physical Graph Template

Using profiling information of pipeline components and Data Processor hardware resources the Execution Framework then “translates” a Logical Graph into a **Hm±¦ꞑbY∨ 4¥Y¢m Mfw¢∨Y¨f**, which prescribes a manifestation of all Drops without specifying their final physical locations.

6.4.4 Physical Graph

The Logical Graph is combined with resource availability information by the Physical Graph Generator to produce a physical graph. This **Hm±¦ꞑbY∨ 4¥Y¢m** contains hardware locality information down to Drop Island level (Drops Islands are described in detail in section 10). The Physical Graph generation function takes this Logical Graph as input together with Resource Management information from LMC to generate a graph which relates specifically to realisation of the data flow and processing for the required analysis on the SDP hardware. The way in which the graph is partitioned across the available hardware is the mechanism by which load-balancing and fault-tolerance are achieved. The graph approach allows a hierarchical mode of work distribution - the relative complexity of different parts of the physical graph should be predictable to a first order approximation through scaling relations in the

parametric model and/or through benchmarking the response of the system during commissioning. The SDP will begin this work as it progresses towards construction as part of the rollout plan.

6.4.5 Master Drop Manager

The Master Drop Manager is a top level software component that takes the Physical Graph and partitions it into Drop Island sized chunks. These Physical Graph Partitions are distributed to the Island Drop Managers, that are responsible for detailed scheduling of the graph partition components.

6.4.6 Island Drop Manager

Each Drop Island has an island drop manager that is responsible for the execution of the Physical Graph Partitions that it gets assigned from the Master Drop Manager. The Island Drop Manager distributes Physical Graph Sub-partitions to individual compute nodes to be executed. The distribution of sub-partitions to nodes is a dynamic process on each of the Drop Islands until the Physical Graph Partition is completed.

6.4.7 Node drop manager

Each compute node has a node drop manager that is responsible for the deployment and monitoring of the physical graph sub-partitions that it gets assigned from the island drop manager. The node drop managers are the only drop managers in this hierarchy, which are actually managing Drops rather than just parts of the physical graph on a higher level.

7 Functional view

The high level functional breakdown of the execution framework in Figure 2 shows the graph system and the generation of the various graph flavours after the initial selection of the Logical Graph Template, as well as the execution of the final physical graph and the preservation of the products. Note that these functions will be executed on quite different time scales and some of the SDP capabilities will require a quasi static and continuous deployment, since they are executed most of the time (e.g. real-time calibration and fast imaging).

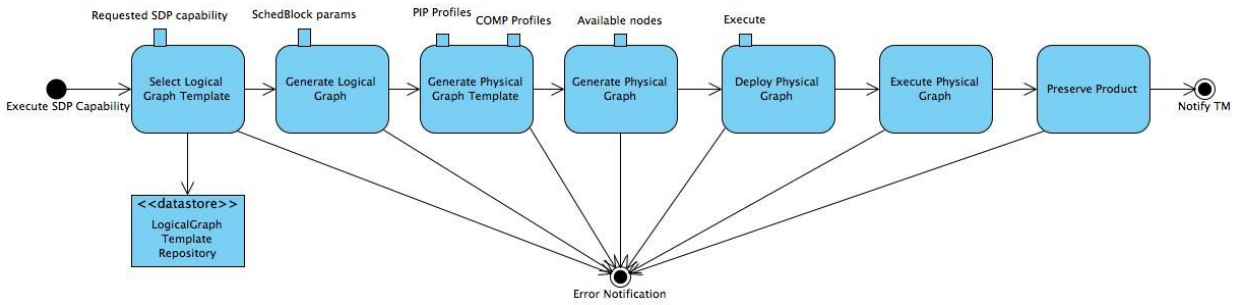

Figure 2: High-level functional breakdown of the SDP Execution Framework. At any given point in time there will be multiple instances of such graph chains running, either overlapping or in parallel or even commensal using the same data.

Below is a brief discussion of the steps involved to execute a certain logical graph template in our data-driven framework design.

- First, the **=zlnbY¥ 4¥Y¢m Mfw¢¥Y¨f** (topleft in Fig. 2) represents high-level data processing capabilities. They could be, for example, “Spectral Line Imaging” or “Pulsar Timing”.
- All logical graph templates are managed by the **=zlnbY¥ 4¥Y¢m Mfw¢¥Y¨f Kf¢z¦n¨z¥±** (bottomleft in Fig. 2). The logical graph template(s) is first selected from this repository for a specific pipeline (capability) and is then filled with scheduling block parameters, like e.g. number of channels and baselines. This generates a **=zlnbY¥ 4¥Y¢m**, expressing a pipeline with still (hardware-)resource-oblivious data graph constructs.
- Using profiling information of pipeline components and Data Processor hardware resources the Execution Framework then “translates” a Logical Graph into a **Hm±¦nbY¥ 4¥Y¢m Mfw¢¥Y¨f**, which prescribes a manifestation of all Drops without specifying their final physical locations.
- In order to meet predefined requirements such as performance, cost, etc each Drop in the Physical Graph Template is associated with an available resource unit. This information is obtained from the **Kf¦z©¥bf .¦¨nwY¨z¥**. The Resource Estimator contains up-to-date resource availability information such as compute node, storage, etc., . Once the association has been made the physical graph template is transformed into a **Hm±¦nbY¥ 4¥Y¢m** which is a set of interconnected Drops mapped onto a given set of resources.
- Before an observation starts, all the Drops are deployed onto these resources as per the location information stated in the physical graph. The deployment process is facilitated through Drop Managers, which are daemon processes managing deployed Drops on designated resources.
- Once an observation starts, Graph Execution is cascading down graph edges through Data Drops triggering Application Drops that produces the next output Data Drops. Some of the final data Drops are persistently preserved as Science Products by using an explicit persist consumer, which very likely will be specifically dedicated to a certain science data product.

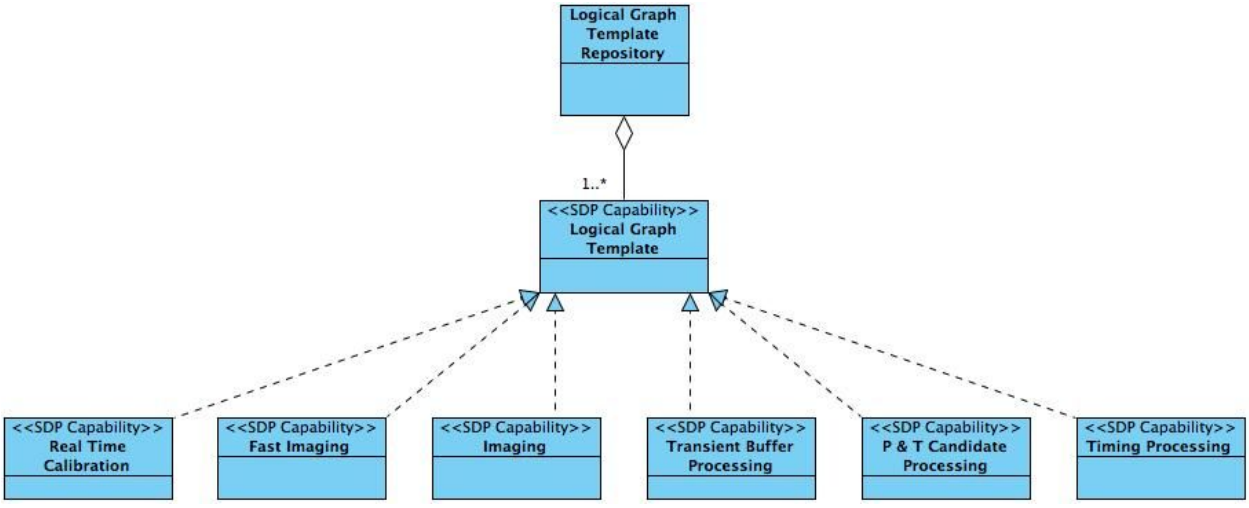

Figure 3: Class diagram of the Logical Graph Template system. The Logical Graph Template Repository is a collection of Logical Graph Templates. The official set of available Logical Graph Templates defines the processing capabilities of the SKA. The set of realisations (bottom row boxes) of Logical Graph Templates depicted here is the minimal set to be

provided by the SDP. Some of these boxes will need to be split into two or more individual realisations, e.g. the Imaging capability contains both spectral line and continuum imaging. In addition some of these realisations will require more specific modes, like drift-scan imaging and mosaicing.

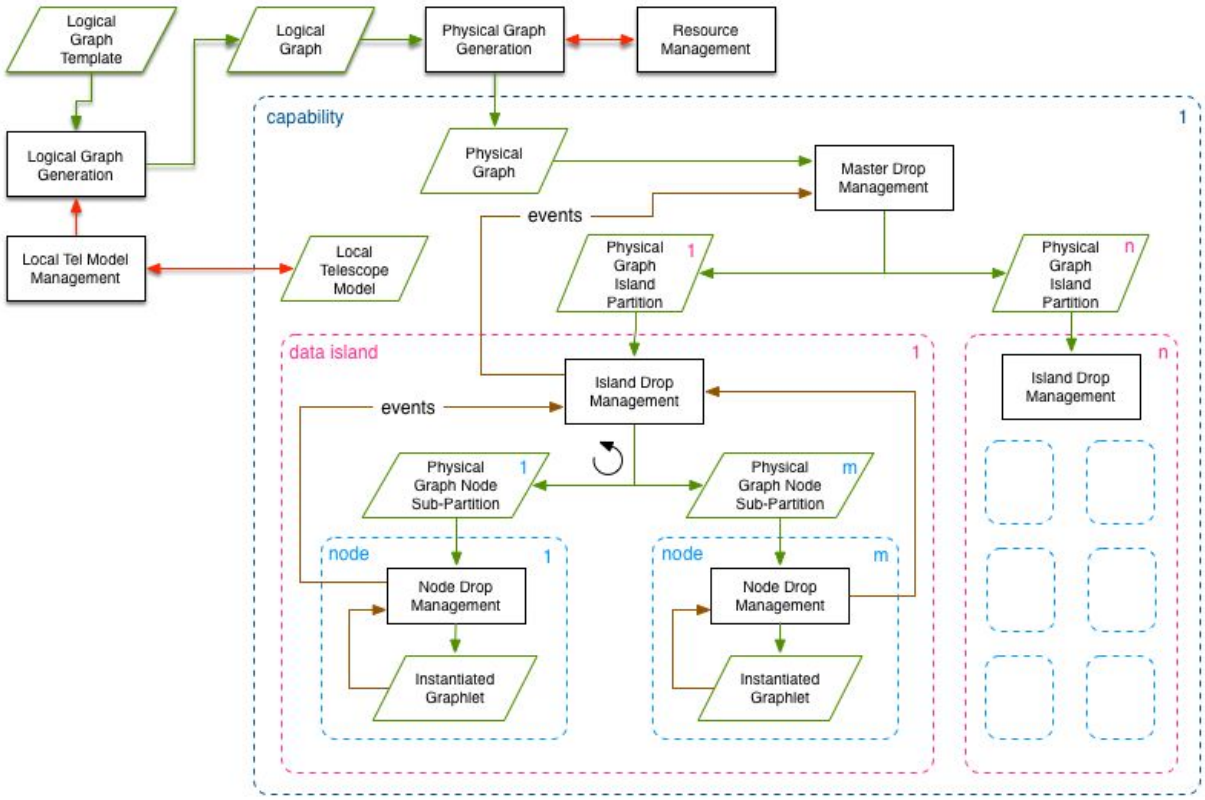

Figure 4: *Mmf ¯Y± ¨mY¨ ¨mf Hm±¦nbYv 4¥Y¢m n¦ ¦¢vn¨ af¨¯ ffx dnkkf¥fx¨ , ¥z¢ 6¦vYxd¦ n¦ kn°fd Ý z¥ ï¦¨Y¨nbîÎ . Ybm n¦vYxd nx ©¦f a± ¨mf Hm±¦nbYv 4¥Y¢m mY¦ Y ¢Y¥¨ zk ¨mY¨ l¥Y¢m Ý Y ïHm±¦nbYv 4¥Y¢m HY¥¨n¨nzxï Ý df¢vz±fd zx¨z n¨Î , ±xYwnb vzYd aYvYxbnx l n¦ dzxf ¯n¨mnx bzw¢©¨f xzdf¦ nx Y , ¥z¢ 6¦vYxd ®nY ¨mf 6¦vYxd , ¥z¢ >YxYl f¥Ë ¯mnbm d±xYwnbYw± ¦bmfd©vf¦ ¦©aÝ¢Y¥¨n¨nzx¦ åïHm±¦nbYv 4¥Y¢m ?zdf L©aÝHY¥¨n¨nzx¦ïä zk ¨mf Hm±¦nbYv 4¥Y¢m 6¦vYxd HY¥¨n¨nzx zx¨z bzw¢©¨f xzdf¦Î*

We partition the graph into ***x*** partitions which we refer to as Physical Graph Partitions - this partitioning is static and each partition will be deployed to a Drop Island. To each of these partitions we associate a group of one or more (usually multiple) compute units or nodes – we call this group a drop island. This static partitioning is done to minimise the communication between the partitions as represented in the graph. For example, in order to achieve a degree of load balancing in our static partitioning of the graph, we will utilise the splitting of our data by frequency channel as the main index. Having analysed and benchmarked the processing complexity of such a split [RD06], variation in processing load with frequency channel is predictable, hence we can change the size of the Physical Graph Island Partition to contain more or fewer frequency channels, or change the number of compute nodes in a Drop Island to be appropriate to the expected load. For each of the Physical Graph Partitions the graph is further partitioned among the compute nodes within the Drop Island to produce Physical Graph sub-partitions: this partitioning however is not static. The progress through the graph is monitored and apportioning of the graph between the nodes is triggered dynamically via

an event mechanism if either load-balancing or recovery from failure is required. To implement this repartitioning in an efficient manner one requirement is that the compute nodes within a Drop Island can gain access to all the data required for the execution of the graph which has been partitioned to that specific Drop Island.

Actors in the SDP context are realised by an abstract object referred to as a Drop: drops are discussed in more detail in the next section.

The management and scheduling of the graph is done in a hierarchical fashion. At the top level a so-called Master Drop Manager manages the partitioning of the graph between Drop Islands. Within a Drop Island an Island Drop Manager manages the scheduling and execution of the graph on the Drop Island. A further Node Drop Manager manages the execution and scheduling of the graph on a compute node. At each level Drop Managers communicate via an event mechanism to those above and below it in the hierarchy.

The graph description we have adopted is flexible, but designed for the SKA problem: the architecture is specifically designed to enable us to manage risk. For example if the scope of the SDP problem is in fact such that dynamic scheduling is not required for load balancing reasons then full static partitioning of the graph is possible and we only have a master drop manager and node drop manager. Similarly the architecture permits defining the whole available resource as a single drop island for complete dynamic scheduling if the problem and underlying hardware require and permit this to be done efficiently.

7.1 Representation of SDP Capabilities and Pipelines

In summary, within the SDP architecture, a complete capability is represented as a set of connected pipelines plus a few additional supporting functions and services (e.g. Local Sky Model and Telescope Model). Each of the pipelines is represented as a (logical) graph, where nodes on the graph are alternating Data Drops and Application Drops.

Connected pipelines are represented by explicit Drop events (see below) from one pipeline to the next. The Application Drops are pipeline components (provided by the PIP subsystem), which are pipeline algorithms wrapped into Drop instances. This mechanism makes them accessible to and manageable by the Execution Framework. The interface between the Execution Framework and the pipeline algorithms is thus captured in the Drop wrapper. In addition there is an I/O interface between pipeline components (Application Drops) and Data Drops, which allows the Execution Framework to 'hide' differences in the access to data in Drops depending on its physical location or format. One example would be to provide the same interface regardless whether the data is in memory or on some storage device. Another, more complex example would be to provide transparent access to data that is stored in a database. This concept is very similar, but more generic, to the Storage Manager approach used in Casacore [RD09].

8 Dynamic Views

The SDP and thus also the Execution Framework has to bind into and interface with the SKA observation schedule implemented and executed by TM. The Execution Frameworks interface to TM is managed and moderated by the LMC Master Controller.It will use this interface directly for schedule block information exchange, schedule block triggering and error propagation with TM. The following sequence diagrams describe the sequence of the execution of the various activities in more detail. This includes both the triggering and deployment of the near real-time ingest, calibration and fast-imaging pipelines as well as the deployment and triggering of the imaging and non-imaging pipelines.

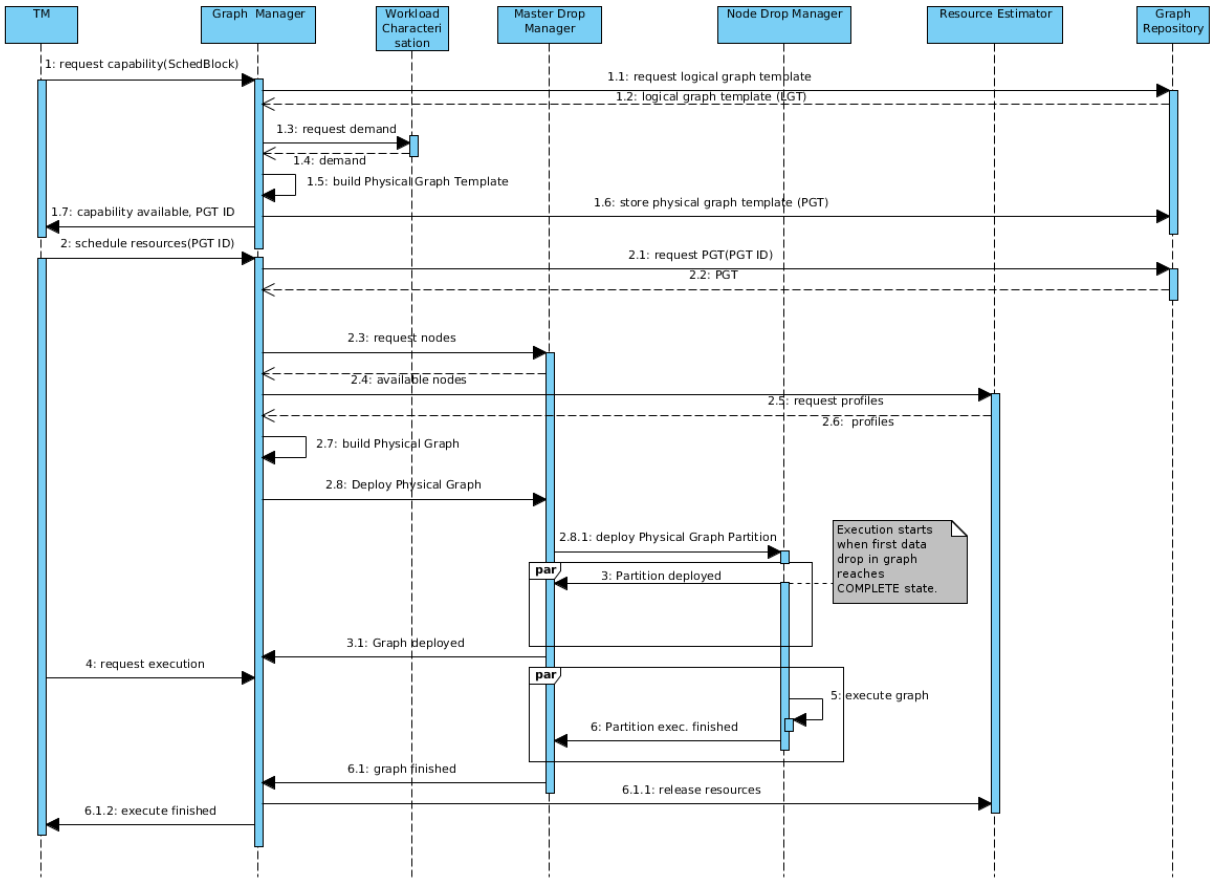

Figure 5: *Lf¤©fxbf dnYl¥Yw zk ¨mf nx¨f¥Yb¨nzx af¨¯ffx M>Ë =>* Yxd ¨mf .°fb©¨nzx 3¥Ywf¯z¥uÎ Mmf dnYl¥Yw nxb©df¦ ¨mf nxn¨nYv nx¨f¥Yb¨nzx ¨z Yb¨©Yw± a©nvd ¨mf Hm±¦nbYv 4¥Y¢m Mfw¢vY¨f ã©¢¢f¥¦fb¨nzxäË¯mnbm bYx af ¢f¥kz¥wfd mz©¥¦ z¥ f®fx dY±¦ afkz¥f ¨mf za¦f¥®Y¨nzx Yxd ¯nv af ¢Y¥¨ zk Y wndÝ¨f¥w ¦bmfd©vnxl Yb¨n®n¨±Î Mmf wnddvf ¦fb¨nzx ¯nv mY¢¢fx wnx©¨f¦ afkz¥f ¨mf za¦f¥®Y¨nzx z¥ ¢¥zbf¦¦nxl Yxd ¯nv lfxf¥Y¨f ¨mf Yb¨©Yv Hm±¦nbYv 4¥Y¢mË¯mnbm f¦¦fx¨nYw± ¥f¦f¥®f¦ ¨mf bzw¢©¨f xzdf¦Î Mmf vz¯f¥ ¢Y¥¨ zk ¨mf dnYl¥Yw n¦ ¨mf nx¨f¥Yb¨nzx Y¨ f°fb©¨nzx zk ¨mf za¦f¥®Y¨nzx z¥ ¢¥zbf¦¦nxlÎ ?z¨f ¨mY¨ ¨mf dnYl¥Yw dzf¦ xz¨ dn¦¨nxl©n¦m af¨¯ffx xfY¥ ¥fYvÝ¨nwf Yxd aY¨bm f°fb©¨nzxÎM> ¯z©vd ¨m©¦ af Yavf ¨z ¨¥nllf¥t©¦¨¨mf f°fb©¨nzx zk ¨mf ¢¥zbf¦¦nxl zk dY¨Y Yv¥fYd± f°n¦¨nxl zx ¨mf kY¦¨a©kkf¥ÎMmY¨¯z©vd nw¢v± ¨mY¨ ¨mf¥f n¦ Y Hm±¦nbYv 4¥Y¢m Y¦¦zbnY¨fd ¨z ¨mY¨ dY¨Y df¢vz±fd Yv¥fYd± zx ¨mf H¥zbf¦¦nxl HvY¨kz¥wÎ ?z¨f ¨mY¨ nk Yx f¥¥z¥*

zbb©¥¦ d©¥ıxI Yx± zk¨ımf¦¨f¢¦ df¦b¥ıafd Yaz®fı¨¯ıwaf¢Y¦¦fd¨z¨ımf=>>Y¦¨f¥*zx¨¥zwf¥Î6x¨ım¦dıYI¥Yw¯fmY®f zwı¨¨fd¨ımf Yddı¨ızxYv ,¥z¢6¦vYxd>YxYIf¥¦Ë¦ıxbf¨ımf±¯z©vd©xxfbf¦¦Y¥w±bzw¢vmbY¨f¨ımf®ıf¯Î*

9 Graph Framework

As described above, graphs are a central component of the Execution Framework and they are being used on several, logically distinct, levels. The following sections describe the anticipated development and deployment environment for the various graph flavors in more detail.

9.1 Logical Graph Development

As already described a logical graph template, as well as a logical graph, are compact representations of the logical operations in a processing pipeline neglecting the capabilities, restriction and availability of hardware resources. Constructing or writing logical graph templates will require deep domain knowledge as well as a very good understanding of the available pipeline components. In addition to 'normal' pipeline components we also introduce logical operations, which are referred to as ***bzx¦¨¥©b¨¦*** in a logical graph. The relationship between a Drop and a construct resembles the one between and object and a class in Object Oriented programming languages. In other words, most constructs are Drop templates and multiple Drops correspond to a single construct. This means in practice that for instance a scatter construct essentially collapses all the internal parallel Drop sub-graphs into one single Drop sub-graph and thus making it possible to abstract the explicit parallelism away from the developer of a logical graph.

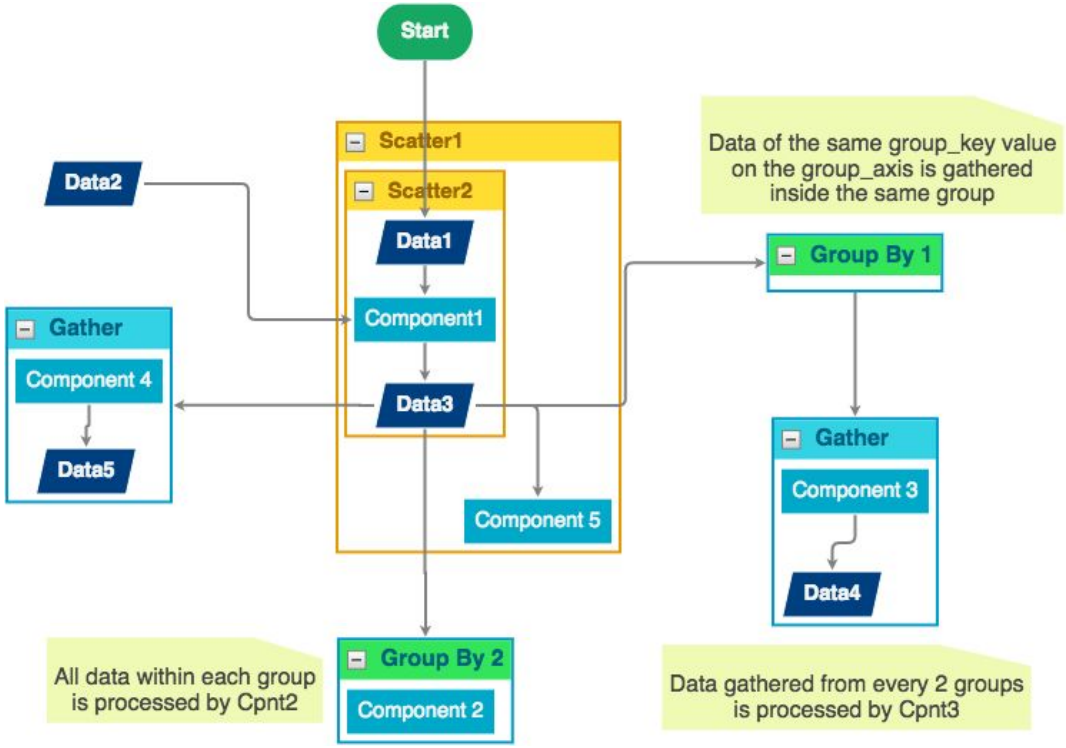

Figure 6: *=zIıbYv I¥Y¢m f°Yw¢vfÎ !x f°Yw¢vf zkYvzIıbYv I¥Y¢m¯ı¨ımdY¨Ybzx¦¨¥©b¨¦ãfÎIÎ ,Y¨Yþý ,Y¨YÄäËbzw¢zxfx¨ bzx¦¨¥©b¨¦ãıÎfÎ *zw¢zxfx¨þý *zw¢zxfx¨ÄäËYxd bzx¨¥zvkvz¯ bzx¦¨¥©b¨¦ãLbY¨¨f¥Ë 4Y¨ımf¥Ë Yxd 4¥z©¢ý)±äÎ*

9.1.1 Construct properties

Each construct has several associated properties that users have control over during the development of a logical graph. For Component and Data constructs the execution time and data volume are two very important properties. Such properties can be directly obtained from parametric models or estimated from the profiling information (e.g. pipeline component workload characterisation) and platform specification.

9.1.1.1 Control flow constructs

Control flow constructs form the "skeleton" of the logical graph, and determine the final structure of the physical graph to be generated. In the course of our prototyping [RD10] we have identified the following required flow constructs:

- Scatter to support data parallelism.
- Gather to support data barriers.
- Group By to support data resorting (e.g. corner turning in radio astronomy).
- Loop to support iterations.

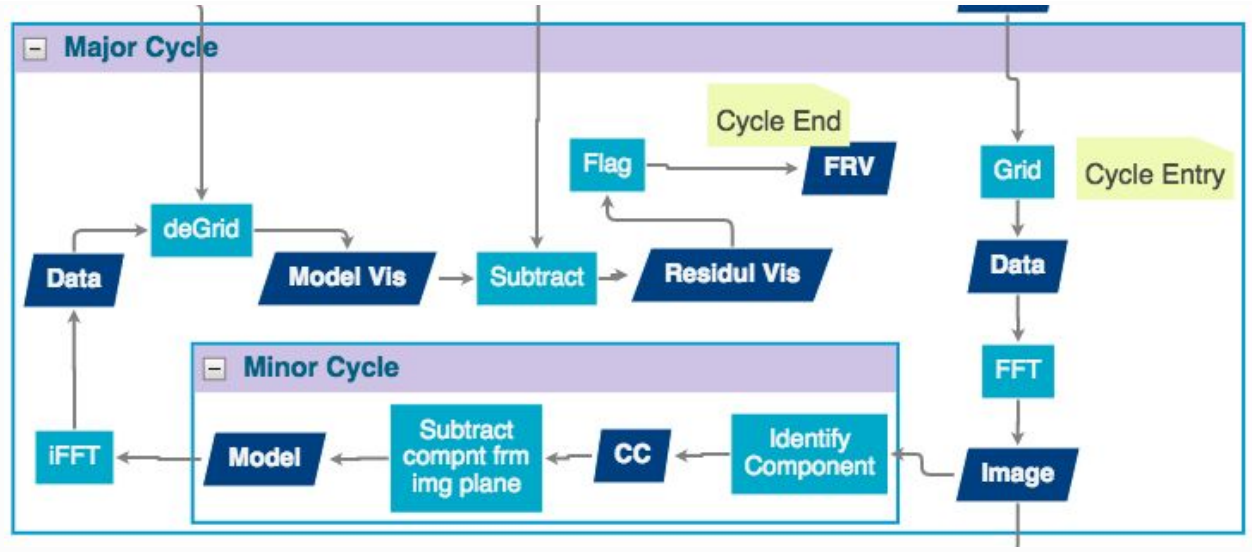

Figure 7: A nested-Loop (minor and major cycle) example of logical graph for a continuous imaging pipeline.

9.1.1.2 Graph Repository

Very much like the development of traditional data reduction software, the construction of correct and working logical graph templates is a complex and iterative process and will require dedicated science staff. The Execution Framework architecture very clearly splits the expertise required to construct the templates from the expertise to write actual pipeline components (algorithms) and the expertise to maintain and optimise the actual execution of a graph within the Execution Framework. In one of our prototypes (DFMS) we have developed a graphical web editor to construct logical graph templates[3]. This specific prototype essentially represents a graphical domain specific language (DSL),

[3] Figures 6 + 7 are screen-shots of that tool.

but similar results could be achieved by using a textual DSL. In any case the DSL code output can be seen as a representation of the logical graph template and can be stored in a repository. For practical purposes this repository could for instance be implemented as a version control system, since it has to fulfill very similar requirements:

- Check-in and check-out of graph templates
- Versioning and branching
- Release tags
- Unique identification

9.2 Selecting a Template and Defining the Logical Graph

During the creation of the detailed description of an observation staff scientists will also have to specify the processing to be carried out by the SDP. Essentially this will result in the selection of a particular Logical Graph Template from the repository. In addition to just selecting the template, the scientists can also adjust a few parameters to tune the processing to their needs. Together with a set of parameters derived from the definition of the actual observation (e.g. number of channels, antennas or stations requested,...) this will be used to translate the Logical Graph Template into a Logical Graph, which is specific for a certain observation, but still completely hardware agnostic.

9.3 Logical Graph to Physical Graph Translation

While a logical graph provides a compact way to express complex processing logic, it contains high level control flow specifications that are not directly usable by the underlying graph execution engine and Drop managers. Thus the logical graphs have to be translated into physical graphs. The translation process virtually defines all the required Drops.

9.3.1 Basic steps

The Translation involves the following steps:

- Validity checking. Checks whether the logical graph is ready to be translated. This step is similar to semantic error checking used in compilers.
- Construct unrolling. Unrolls the logical graph by (1) creating all necessary Drops (including “artifact” Drops that do not appear in the original logical graph), and (2) establishing directed edges amongst all newly generated Drops. This step produces the Physical Graph Template.
- Graph partitioning. Decomposes the Physical Graph Template into a set of logical partitions (a.k.a. Drop Islands) and generates an order of Drop execution sequence within each partition such that certain performance requirements (e.g. total completion time, total data movement, etc.) are met under given constraints (e.g. resource footprint). An important assumption is that the cost of moving data within the same partition is far less than that between two different partitions. This step produces the Physical Graph Template Partitions.
- Resource mapping. Maps each template partition onto a given set of resources in certain optimal ways (load balancing, etc.). Concretely, each Drop is assigned a physical resource id (such as IP address, hostname, etc.). This step requires near real-time resource usage information from the COMP platform or the Local Monitor & Control (LMC). It also needs Drop managers to coordinate the Drop deployment. In some cases, this mapping step is merged with the previous Graph partitioning step to directly map Drops to resources.

In the following, we use the term Scheduling to refer to the combination of both graph partitioning and resource mapping.

9.3.2 Algorithms

Scheduling an Directed Acyclic Graph (DAG) that involves graph partitioning and resource mapping is known to be an NP-hard problem. The Execution Framework will likely require several different heuristics-based algorithms to allow optimisation within different translation domains. Previous research on DAG scheduling and graph partitioning can give guidelines and implementations to perform these two steps. In the case of homogeneous resources this is fairly straight-forward. Support for heterogeneous resources can use a list scheduling algorithm. With these algorithms, the Execution Framework can address the following translation problems:

- Minimise the total cost of data movement but subject to a given degree of load balancing. In this problem, a number N of available resource units (e.g. a number of compute nodes) are given, the translation process aims to produce M Drop Islands (M <= N) from the physical graph template such that (1) the total volume of data traveling between two distinct Drop Islands is minimised, and (2) the workload variations measured in aggregated execution time (Drop property) between a pair of Drop Islands is less than a given percentage p %. To solve this problem, graph partitioning and resource mapping steps are merged into one.
- Minimise the total completion time but subject to a given degree of parallelism (DoP) (e.g. number of cores per node, although it could be driven by other factors such as memory bandwidth, etc) that each Drop Island is allowed to take advantage of. In the first version of this problem, no information regarding resources is given. The Execution Framework attempts to come up with the optimal number of Drop Islands such that (1) the total completion time of the pipeline (which depends on both execution time and the cost of data movement on the graph critical path) is minimised, and (2) the maximum degree of parallelism within each Drop Island is never greater than the given DoP. In the second version of this problem, a number of resources of identical performance capability are also given in addition to the DoP.
- Minimise the number of Drop Islands but subject to (1) a given completion time deadline, and (2) a given DoP that each Drop Island is allowed to take advantage of. In this problem, both completion time and resource footprint become the minimisation goals. The motivation of this problem is clear. In a scenario where two different schedules can complete the processing pipeline within, say, 5 minutes, the schedule that consumes less resources is preferred. Since a Drop Island is mapped onto resources, and its capacity is already constrained by a given DoP, the number of Drop Islands is proportional to the amount of resources needed. Consequently, schedules that require less number of Drop Islands are superior. Inspired by the hardware/software co-design method for embedded systems design, the Execution Framework can utilise a “look-ahead” strategy at each optimisation step to adaptively choose from two conflicting objective functions (deadline or resource) for local optimisation, which is more likely to lead to the global optimum than greedy strategies.

9.4 Physical Graph

The Translation process produces the physical graph template, which, once deployed, becomes the physical graph. The physical graph represents a collection of interconnected Drops in a distributed execution plan across multiple resource units. The nodes of a physical graph are Drops representing either data or applications in an alternating fashion. This establishes a set of reciprocal relationships between Drops:

- A data Drop is the input of an application Drop; on the other hand the application is a consumer of the data Drop.
- Likewise, a data Drop can be a streaming input of an application Drop, in which case the application is seen as a streaming consumer from the data Drop's point of view.
- Finally, a data Drop can be the output of an application Drop, in which case the application is the producer of the data Drop.

Physical graphs (or partitions thereof) are the final (and only) graph products that will be submitted to the Drop Managers. Once Drop managers accept (according to resource availability) a physical graph specification, it is their responsibility to create and deploy Drop instances on their managed resources as prescribed in the physical graph such as partitioning information (produced during the Translation) that allows different managers to distribute graph partitions (i.e. Drop Islands) across different nodes and Drop Islands by setting up proper Drop Channels. The fact that physical graphs are made of Drops means that they are describing the execution exactly. In this sense, the physical graph is the graph execution engine.
In addition to Drop managers, the Execution Framework also includes a Physical Graph Repository, which allows users to manage all currently running and past physical graphs within the system.

9.5 Execution

A physical graph has the ability to advance its own execution. This is internally established via the Drop event mechanism as follows:

- Once a data Drop moves to the 'Ready' state it will fire an event to all its consumers. Consumers (applications) will then evaluate if they can start their execution depending on their nature and configuration. A specific type of application is the Barrier Application Drop, which waits until all its inputs are in the 'Ready' state to start its execution.
- On the other hand, data Drops receive an event every time their producers finish their execution. Once all the producers of a Drop have finished, the Drop moves itself to the 'Ready' state, notifying its consumers, and so on.

Failures on application and data Drops are transmitted likewise automatically via events. Data Drops move to ERROR if any of its producers move to ERROR, and application Drops move the ERROR if a given input error threshold (defaults to 0) is passed (i.e., when more than a given percentage of inputs move to ERROR) or if their execution fails. This way whole branches of execution might fail, but after reaching a gathering point the execution might still resume if enough inputs are present.

10 Scalability, Scheduling and Drop Islands

Scalability is one of the key concerns of the SDP architecture. Obviously we will need to be able to run the full scale SKA1 use cases, but we also need to be able to run the system on significantly smaller platforms (downwards scalable), potentially even down to single compute nodes in order to maintain local testability. Managing and handling hundreds of thousands or even millions of Drops is a non-trivial challenge, although the actual number of Drops required needs proper modelling and evaluation. The hierarchy of Drop Managers we have introduced into this architecture is designed to address these challenges.

10.1 Drop Manager Design

The Drop Managers represent the main services of the Execution Framework. There are three types of Drop Managers, Node Drop Managers (NDM), Island Drop Managers (IDM) and the Master Drop Manager (see the class diagram in figure 8). Since the latter two share most of the same functionality, they are derived from a Composite Drop Manager class. The Node Drop Managers are assumed to be running on every single compute node and will thus most probably be implemented as system daemons, which will be started automatically at boot time of the compute node. The MasterDropManager (MDM) could take over the functionality of an IslandDropManager, in case there is just a single Drop Island in the deployed graph. It should be noted here that the Drop Managers are ***xz¨*** controlling the execution of a graph, they are just deploying and monitoring the execution of the graph.

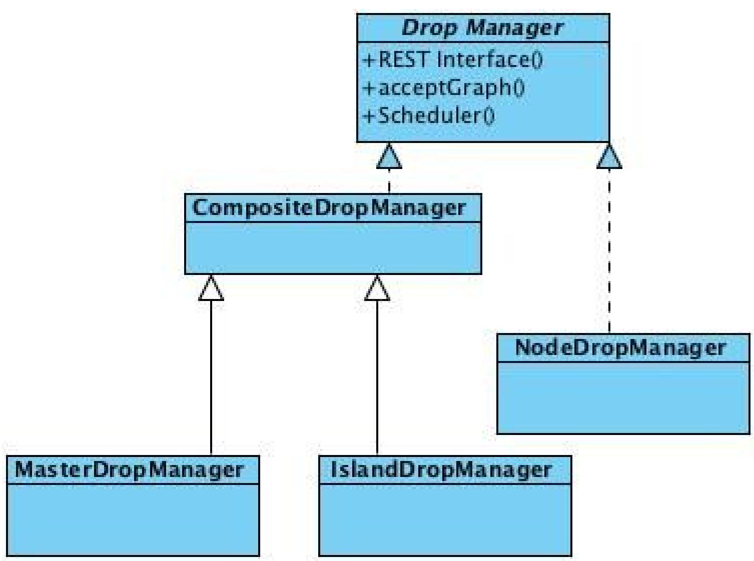

Figure 8: ***vY¦¦ dnYl¥Yw zk ¨mf ,¥z¢ >YxYlf¥¦Î Mmf Ya¦¨¥Yb¨ bvY¦¦ zx ¨mf ¨z¢n¦ uff¢nxl Yw ¨mf bzwwzx wf¨mzd¦ Yxd ¢Y¥Ywf¨f¥¦Ë ¨mf *zw¢z¦n¨f ,¥z¢ >YxYlf¥ bvY¦¦ n¦ t©¦¨ Y ¢¥n®Y¨f bvY¦¦ ¨z uff¢ ¨mf bzwwzxYvn¨nf¦ af¨¯ffx ¨mf >Y¦¨f¥ Yxd ¨mf 6¦vYxd ,¥z¢ >YxYlf¥¦Î Mmn¦ df¦nlx ¯z©vd Ywz¯ Yx fY¦± f°¨fx¦nzx nk ¯f ¯z©vd ¦ff ¨mf xffd kz¥ ±f¨ Yxz¨mf¥ vY±f¥ zk wYxYlf¥¦ kz¥ ¦bYvYanvn¨± ¥fY¦zx¦Î ,¥z¢ >YxYlf¥¦ Y¥f nw¢vfwfx¨nxl Y K.LMk©v nx¨f¥kYbf kz¥ f°¨f¥xYv bzww©xnbY¨nzxÎ Mmn¦ Yv¦z Ywz¯¦ kz¥ ¨mf nw¢vfwfx¨Y¨nzx zk Y ¯faÝaY¦fd k¥zx¨Ýfxd ¨z wzxn¨z¥ ¨mf l¥Y¢m f°fb©¨nzxÎ***

The concept of Drop Islands has been introduced to the Execution Framework design mostly in order to address scalability of the system. A Drop Island is a logical construct hosting an Island Drop Manager and thus adding a hierarchical layer between the Master Drop Manager and the Node Drop Managers. Depending on the total number of Drops in the graph, the Islands are logically generated during the generation of the Physical Graph Template in the form of Graph Partitions. The Master Drop Manager dynamically starts an Island Drop Manager on one specific node at the boundaries of the Physical Graph Partitions. The Node Drop Managers inside that partition will then be monitored by their associated Island Drop Manager. Typically, the Compute Nodes inside a Drop Island will belong to one specific Compute Island (see RD04), in such a way that the Drop Island is aligned with the underlying network topology, but the concept does not mandate a specific allocation strategy and allows to have multiple Drop Islands per Compute Island as well as span multiple Compute Islands or be independent of Compute Islands altogether. In the most extreme case there could be a single Drop Island for the whole graph execution (e.g., in the case of a Physical Graph requiring only a few nodes, and/or with a very small number of Drops); in this degenerate case, there is no Drop Island Manager at all, but the Master Drop Manager assumes that role as well. The design does also not preclude the introduction of multiple Drop Island hierarchy levels in order to scale even further.
The other reason for introducing Drop Islands stems from the need to execute multiple graphs in parallel and/or commensal on the same data. In order to be able to do this, we need to introduce some kind of scheduling and optimisation. In addition dynamic load balancing between compute nodes may also be required, because run-times of individual, parallelised pipeline components might differ significantly, depending on the actual data.

10.2 Drop Islands and Scalability

As mentioned above the design allows to dynamically generate a Drop Island, if the total number of Drops exceeds a certain threshold, or if the management of the complete graph seems too complex for a single manager. This is decided during the generation of the Physical Graph Template, by introducing Physical Graph Template Partitions. At the time of deployment, the Master Drop Manager would launch a Drop Island Manager at the boundaries of the partitions and then send the Physical Graph Partitions to those Drop Island Managers, which in turn would distribute the Drops to the assigned nodes and subsequently monitor them.

10.3 Scheduling and Load Balancing

Up to this point we have not mentioned scheduling of graph and Drop execution. As indicated above, we need to support scheduling on at least two levels, graph level scheduling, but potentially also Drop level scheduling, internal to a graph in order to allow for load balancing in cases of unbalanced run times. In particular in the (normal) case when the SDP has to execute multiple graphs in parallel, graph scheduling becomes very important. The graph level scheduling will need to interact with the global TM observation scheduler, because certain combinations of SDP capabilities might not be possible to be executed at the same time, since they might exceed the available compute resources.

But even in the 'simple' case of a single graph or a sequence of single graph executions optimisation and scheduling is essential in order to execute a given graph in an optimal way. The easiest solution to this problem is scheduling of static graph deployments. That means that the whole physical graph is deployed at once and every single Drop has been assigned a fixed compute node upfront during the generation of the physical graph. Even though this sounds straight forward, it still requires that the whole graph optimisation and node assignment is working and this is non trivial, because the optimisation itself represents already a so-called NP-hard problem (nondeterministic polynomial-time hard). If static or dynamic scheduling is added, the problem gets even more complex. Such optimisation and scheduling problems are topic to current and very active research in mathematics and computer science, but there are no directly applicable solutions, yet. Current day HPC schedulers are very much task oriented and are just starting to operate also in power and data aware modes. Since the SDP needs to maximise efficiency across the whole system in order to keep capital and operational costs down, we regard this whole topic as high risk and likely to cause problems for the project (see risk analysis).
For that reason the architecture has built in quite some flexibility, but is at the same time trying to keep the scheduling complexity down where possible. In essence we are trying to design a system, which would run potentially in a less efficient way overall, but at least it would work. The individual complexity steps can be summarised in the following way:

1. fully static single graph optimisation (essentially hand crafted deployment of one optimised graph forever).
2. fully static single graph scheduling and optimisation (deploy single graphs at a time from a sequence in an optimised way).
3. partial static single graph with dynamic load balancing inside graph partitions.
4. fully static multi graph scheduling and optimisation.
5. partial static multi graph scheduling and optimisation with dynamic load balancing inside graph partitions.
6. fully dynamic single graph scheduling and optimisation.
7. fully dynamic graph scheduling and optimisation and fully dynamic load balancing of all Drops in the graph.

In general (6) and (7) are regarded to be very unlikely to be achievable and (5) very hard to obtain better efficiency than (4) without significant additional research and prototyping effort. On the other hand (3) is the minimum the SDP has to be able to achieve to meet the requirements of commensal and simultaneous observations and (4) might be desireable.

10.4 Drop Islands and Scheduling

In order to address the challenges described above the design foresees to attach a scheduler to each single Drop Island Manager by providing a scheduler interface method in the abstract manager class. It does not restrict the type of scheduler, nor do they have to be the same across different Drop Islands. In this scenario the Master Drop Manager would send Physical Graph Template Partitions (***xz¨*** final Physical Graphs) to the Drop Island Managers and leave it up to the internal schedulers to

optimise placement and execution. The heuristics of the Drop Island internal schedulers could then allow for either a static deployment, or a fully dynamic, load balancing deployment and execution.

10.5 Node Drop Managers and Scheduling

Single compute nodes of the Processing Platform will most likely consist of potentially heterogeneous, many-core systems with one or a few accelerators. Managing execution in such an environment is quite complex in its own right and thus we also allow to attach a scheduler to each of the Node Drop Managers.

10.6 Summary

There are scheduling systems around (e.g., [RD11, RD12, RD13]), mainly in the form of research projects, which are starting to perform scheduling across clusters of compute nodes taking accelerators, task capabilities and also data locality into account. If one of them proves to be a viable product we could run it on the Drop Island Manager level and have no scheduler inside the node. If we even find such a system viable to perform and control the whole scheduling, including the graph scheduling, we would only run one on the Master Drop Manager level. In the contrary case, if we find the scheduling on either of the levels not performing reliably or at all, we can fall back to fully static graph deployments, while still doing some mid-term graph level scheduling, maybe put together a schedule for a week worth of observing time and run those fully statically underneath. This does represent bullet point (4) above and would thus fulfill the basic requirements.

11 Drop Design

At the base of the SDP Execution Framework and the data driven architecture is the design of a Drop. The block diagram in Figure 9 provides a high level view of that design. Drop Instances essentially are wrapper objects around arbitrary items. Drops in the SDP Execution Framework represent both Data and Application items (pipeline components) and they can also represent a collection of other Drops in so-called Container Drops. Specific requirements of concrete Drop classes are modelled using standard inheritance. The design of the Drop is based on the experience with the design and implementation of a whole series of previous astronomical data flow implementations, including HST, ESO, ALMA and MWA. In addition we have performed extensive prototyping (see RD05) in order to refine the model and also the architecture of the overall Execution Framework. The design itself is not related to any implementation, but can be mapped to existing commercial and non-commercial implementations (e.g., in the non-commercial NGAS archiving system, used by ESO, ALMA and the MWA project, each stored file could be viewed as a Drop). The existing astronomical data flows mentioned above do implement substantial parts of this design. The L3 and lower requirements together with the prototyping will allow the validation and refinement of the design.

The Drop design is key to understanding how the data driven architecture is deploying and executing a physical graph. It is also required to study the Drop design internals in order to understand how the interfaces between the Drop and the Drop management and the between the pipeline components Drop payload, for pipeline components, are working.

11.1 Drop High-Level Design

As already indicated above a Drop is a software wrapper around data and application 'payloads'. The wrapper allows the Execution Framework to manage and deploy Drops in a consistent and homogeneous way. By wrapping data payloads within Drops and making Data Drops nodes on the graph, it also makes data 'active' to a certain extent. This enables to represent the physical graph as a direct connection between Drops, rather than having to design some glue system around them.

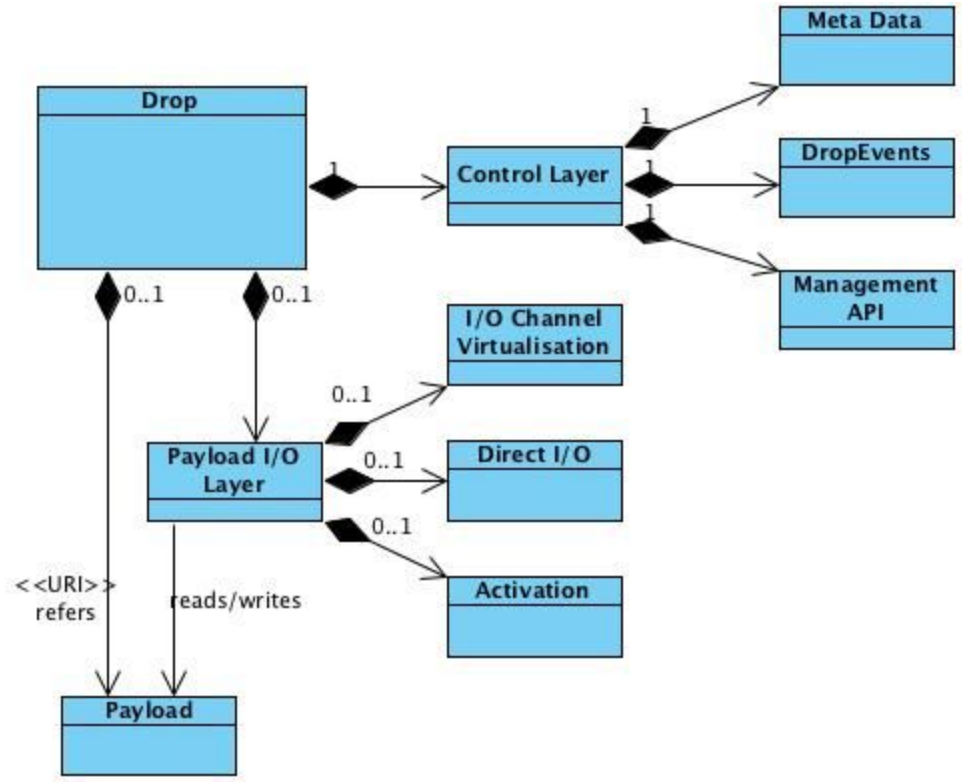

Figure 9: *)vzbu dnY I¥Yw zk Yx Ya¦¨¥Yb¨ ,¥z¢Î ! ,¥z¢ nx ¨mn¦ ®nf¯ bzx¦n¦¨¦ zk *zx¨¥zv =Y±f¥ Yxd Y HY±vzYd 6ÙA =Y±f¥Î Mmf *zx¨¥zv vY±f¥ ¢¥z®ndf¦ ¨mf nx¨f¥kYbf ¨z ¨mf ,¥z¢ >YxY I fwfx¨ 3¥Ywf¯z¥uË ¨mf HY±vzYd 6ÙA vY±f¥ ¢¥z®ndf¦ ¨mf nx¨f¥kYbf ¨z ¨mf Yb¨©Yv ¢Y±vzYdÎ 6x ¨mf bY¦f zk ,Y¨Y ,¥z¢¦ ¨mn¦ fxbY¢¦©vY¨f¦ mz¯ ¨mf dY¨Y nx ¨mf ¢Y±vzYd n¦ Yb¨©Yw± afnx I ¯¥n¨¯fx Yxd ¥fYdÎ 6x ¨mf bY¦f zk !¢¢vnbY¨nzx ,¥z¢¦ ¨mn¦ ¦¢fbnknf¦ mz¯ ¨mf Y¢¢vnbY¨nzx n¦ ¦©¢¢z¦fd ¨z af Yb¨n®Y¨fd Yxd dfÝYb¨n®Y¨fdÎ 6x ¢Y¥¨nb©vY¥ kz¥ ,Y¨Y ,¥z¢¦ ¨mf¥f wn I m¨ Yv¦z af Yx 6ÙA *mYxxfv Sn¥¨©Yn¦Y¨nzx Y®YnYavf nx z¥df¥ ¨z f°¢z¦f dY¨Y k¥zw ¨mf ,¥z¢ fn¨mf¥ ¢Y¥¨nYw± z¥ k©w± ¨m¥z©I m Y ¢f¥kz¥wYxbf z¢¨nwn¦fd nx¨f¥kYbfÎ Axf f°Yw¢vf wn I m¨ af wfwz¥± wY¢¢nxIË Yxz¨mf¥ K, >! k¥zw ¯n¨mnx Yx Ybbfvf¥Y¨z¥Î*

An abstract Drop Object consists of a control layer, a payload I/O layer, and a reference.

- The control layer will contain metadata for the drop indicating its type and also provide the functionality necessary for messaging between drops (drop events). Principle messaging is to notify linked Application Drops in the graph of the availability of data so that their actions can be triggered. The Data Drop produced by the action in turn will then trigger its linked Application Drop(s) once all the data has been written. The Management API provides methods to connect the Drops with the Drop Managers and the Drop Lifecycle Management. This includes methods like:
  - Activate, de-activate application payload.

- Setup a specific Virtualisation channel
- Set or get expiration timestamps
- Set or get checksum
- Set or get user defined meta-data
- Set or get target and current resilience.

- The Payload I/O layer provides the I/O methods to access the payload. For Data Drops access to the data can either be provided by a reference to the actual I/O methods of the underlying data object (direct I/O). Alternatively it can provide virtualised I/O methods in order to transparently hide differences in actual I/O methods, depending where the data actually resides physically.
  - The I/O Virtualisation API supports linking the drop via a specific or a set of I/O channels. Each channel will require specific implementation for each type of virtualisation required. The main idea is to allow for I/O optimisation which would be transparent to the application code. Examples of such channels include:
    - An output buffer into which an actor places output data which are then made available as the input buffer for the connected actor in the graph
    - A memory mapped file
    - Asynchronous data pre-fetching to memory, cache or accelerators.
  - For Application Drops the access to the payload needs to describe how the payload can be activated and de-activated. Since the Execution Framework needs to support a variety of ways to implement applications, this design allows to homogenize the startup and shutdown of the applications. Examples of such procedures include:
    - Docker container fetching and initialisation
    - Virtual machine startup
    - Launching a shell script

11.2 Drop Class Model

As shown in the formal UML class diagram below, we consider various different derived drop classes:

- Data Drops
  - Provide a link to a data source via a Unique Resource IdentifierI (URI)
  - Provide an I/O interface to data so that they may move data through the memory hierarchy
  - A standard use of a data drop will be to move data into memory at which point the drop moves to the data ready state
  - Messaging is used to link drops in the graph.
- Application (Pipeline Component) Drops
  - They provide a link to a processing component via a URI and provide the interface to launch this kernel on a data ready message
- Container Drop
  - Provide a higher level of abstraction for Drops belonging together in a logical sense. For instance a SDP standard science data product may consist of a set of individual data products, each of which might be an aggregation of many individual Data Drops.

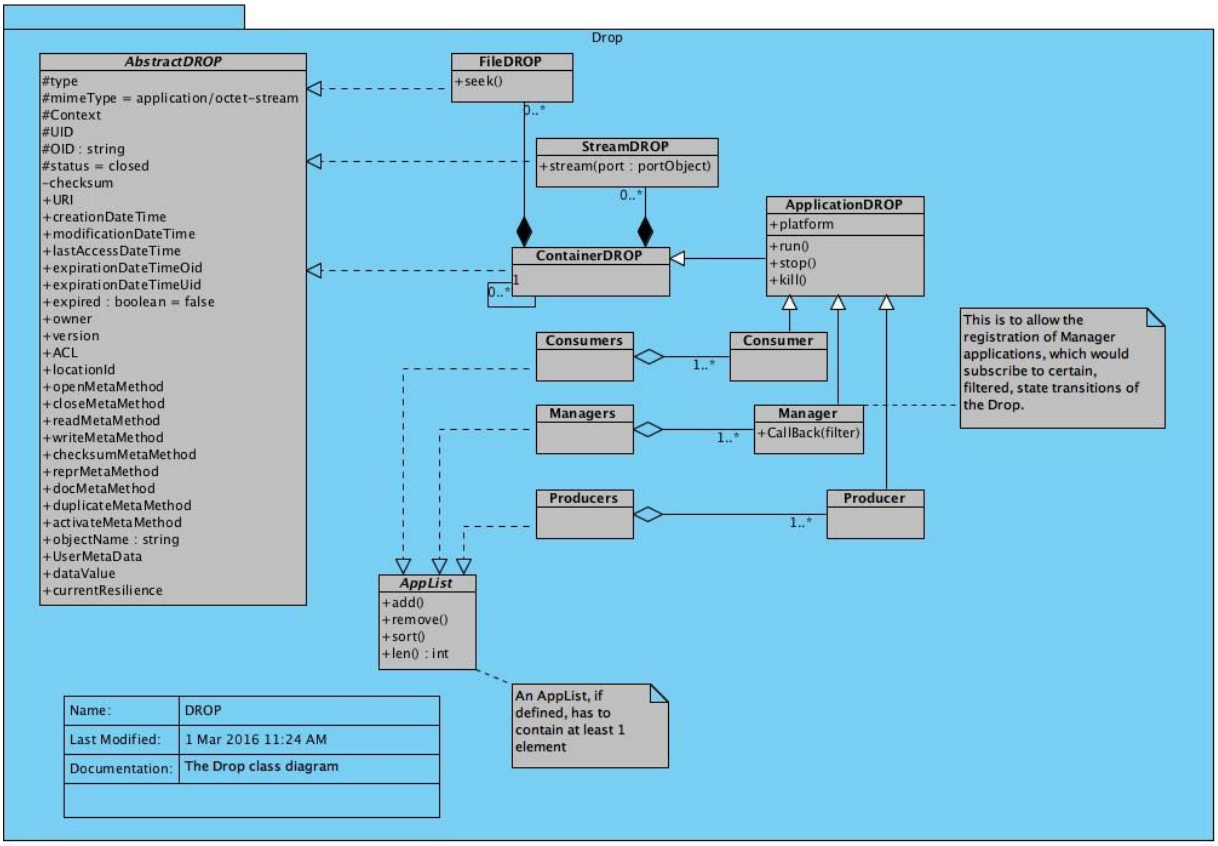

Figure 10: *N>= bvY¦¦ wzdfv zk ¨ïmf ,¥z¢ ¦±¦¨fw fw¢vz±fd a± ¨ïmf . °fb©¨ñzx 3¥Ywf¯z¥uÎ ,¥z¢¦ Y¥f f¦¦fx¨ñYw± zatfb¨ ¯¥Y¢¢f¥¦ Y¥z©xd ñ¨fw¦Ë ¯mnbm bYx af dY¨ŸË Y¢¢vnbY¨ñzx¦ z¥ z¨ïmf¥ ,¥z¢¦Î L¢fbnknb ¥f¤©ñ¥fwfx¨¦ zk bf¥¨Ÿnx ¨±¢f¦ zk ,¥z¢¦ Y¥f wzdfwfd©¦ñxl ¦¨YxdY¥dñxmf¥ñ¨Yxbfî ,¥z¢¦ Ywz¯ ¨ïmf¥fIñ¦¨¥Y¨ñzx zk H¥zd©bf¥ Yxd *zx¦©wf¥ Y¢¢vnbY¨ñzx¦ Y¦ ¯fw Y¦ >YxYIf¥ Y¢¢vnbY¨ñzx¦Î Mmf Y¨¯¥ña©¨¨f¦ Yxd df¥ñ®fd bvY¦¦f¦ñx¨ïmñ¦ dñYI¥Yw Y¥f ñxdnbY¨ñ®f Yxd f°fw¢vY¥kz¥ ¯mY¨ ñ¦ ¥fYw± ¥f¤©ñ¥fd kz¥ ¨ïmf bzw¢vf¨f . °fb©¨ñzx 3¥Ywf¯z¥uÎ*

11.3 Representation of the Physical Graph using Drops

Essentially, after deployment, the complete information of the physical graph is embedded in the Drops and the connection between Drops. During the deployment of the Physical Graph the Drops will be instantiated and at that point the producer and consumer applications for every Data Drop are known and can thus be provided to the initialiser. Once this is done the connection between the Drops is fully determined and can be seen as point-to-point connections between Drops. As can be seen in the UML diagram above, the Drop objects 'know' their producer and consumer ApplicationDrops and can send events directly to them. The Drop object allows to define complete lists of producers and consumers and thus many to one and one to many relationships can be implemented. Since pipeline components will be implemented as ApplicationDrops, that in turn allows pipeline components to be triggered by events or similar mechanisms like remote procedure calls, issued by DataDrops after certain state transitions. For instance, if an ApplicationDrop subscribes to the READY state transition of a certain DataDrop, it can start executing on the reception of that event. More explicitly, the execution framework architecture implies a completely asynchronous execution model and thus the actual event mechanism is assumed to be a callback to allow full flexibility for the implementation. The Drop design contains references to lists of consumers

and producers [Figure 6]. Each item in such a list represent application Drops. Thus, after deployment, the physical graph is implemented within the Drops and acts in a completely independent fashion, driving its own execution. In particular there is no controlling instance required during runtime, except for failover handling or when parts of the graph are dynamically scheduled. In that case the Drop Manager will react on individual failures, possibly communicating them upstream or triggering the dynamic scheduling.

The attributes listed in the Drop class diagram above and the derived classes are indicative and non-inclusive. The ContainerDrop allows for the definition of complex relations between Drops in order to form Science Data Products and Packages. The derived classes FileDrop and StreamDrop are just but two examples of explicit derived Drop classes supporting additional, specialised methods or attributes.

The detailed design of the Drop system is ongoing work. An important driver are the requirements for the pipelines [RD02] and real world examples. Pipelines and the supported observing modes they encode will evolve and the Drop design will need to be updated accordingly. Drops are connected using the producer and consumer pattern and create a graph representing an execution plan, where inputs and outputs are connected to applications, establishing the following possible relationships:

- None or many data Drop(s) can be the input of an application Drop; and the application is the consumer of the data Drop(s).
- A data Drop can be a streaming input of an application Drop in which case the application is seen as a streaming consumer from the data Drop's point of view.
- None or many Drop(s) can be the output of an application Drop, in which case the application is the producer of the data Drop(s).
- An application is never a consumer or producer of another application; conversely a data Drop never produces or consumes another data Drop.

The difference between normal inputs/consumers and their streaming counterpart is their granularity. In the normal case, inputs only notify their consumers when they have reached the READY state, after which the consumers can open the Drop and read their data. Streaming inputs on the other hand notify consumers each time data is written into them (alongside with the data itself), and thus allow for a continuous operation of applications as data gets written into their inputs. Once all the data has been written, the normal event notifying that the Drop has moved to the COMPLETED state is also fired.

11.4 Drop Input/Output

I/O can be performed on the data that is represented by a Drop by obtaining a reference to its I/O object and calling the necessary POSIX-like methods represented in [Figure 10] as open, read, write and close meta methods. In this case, the data is passing through the Drop instance. The application is free to bypass the Drop interface and perform I/O directly on the data, in which case it uses the data Drop dataURL to find out the data location. In that case it is the responsibility of the application to ensure that the I/O is occurring in the correct location and using the expected format for storage or subsequent upstream processing by other application Drops.

The Execution Framework needs to provide various commonly used I/O storage classes, including in-memory, file-base and object storage.

11.5 Drop Events

Drops that are connected by an edge in a physical graph but are deployed on separate nodes or islands from each other will have to use a remote method invocation interface to allow them to communicate with each other. It's the job of the Master Drop and Island Managers to generate and exchange stubs between Drop instances before the graph is deployed to the various Drop Islands and nodes within islands respectively. Execution within a single address space result in basic method calls between Drop instances.

The location of the drops in the system is the only criterion to decide the mechanism they will use to communicate with their peer drops, and is calculated when mapping a Physical Graph into a set of resources (see 9.3.1).

11.6 Drop Component Interface

In general, the interface between a pipeline component and the Drop framework should be implemented as a generic class, which can be inherited by application programmers in order to make the integration as easy as possible. This way any kind of application can be easily plugged into the framework.

In particular, delivering applications as Docker containers is a particularly good way of performing this integration, and thus the framework could offer built-in support for them. Docker containers have the following benefits over traditional tools management:

- Portability.
- Versioning and component reuse.
- Lightweight footprint.
- Simple maintenance.

12 State Views

While the SDP architecture is following the paradigm of stateless components, the Execution Framework, which binds the components together into a SDP capability and eventually a executable pipeline graph, is designed around a set of state machine views. This is required in order to enable the life-cycle management of the Drops, the graph partitions on the nodes and the whole graph execution. This section summarises the various state machines.

12.1 Execution Framework State View

The Execution Framework is part of the overall SDP deployment, in particular it is part of the SDP Processor Software product. In order to work properly the Execution Framework will rely on a number of services to be up and running. The required platform services are assumed to be up and running here and are covered by the description of the SDP Processor Platform.

12.1.1 Execution Orchestration

It is assumed that the core services of the Execution Framework (Master and Node Drop Managers) will be implemented as system daemons which are started at boot time of the individual compute nodes. The Master Drop Manager will likely be co-located with the Graph Manager on a high availability node external to the actual compute cluster. Our current baseline is to co-locate both the Graph Manager and the Master Drop Manager with the LMC Master Controller. Note that since the execution is completely asynchronous, the Master Drop Manager as well as the Island Drop Managers could actually fail or be restarted after the deployment phase, without the execution being affected at all. The Node Drop Managers will announce themselves on some kind of zero configuration network (like e.g. Bonjour) and can then be identified and registered by the Island and Master Drop Managers. Node Drop Managers, which can not be discovered will render those nodes unavailable for the Execution Framework, even if those nodes are up and running. In addition, we do expect the implementation of the interaction between the Master and the Node Drop Managers to allow for automatic discovery of new or recovered compute nodes. This will allow semi-dynamic removal and adding of nodes to the Execution Framework pool.

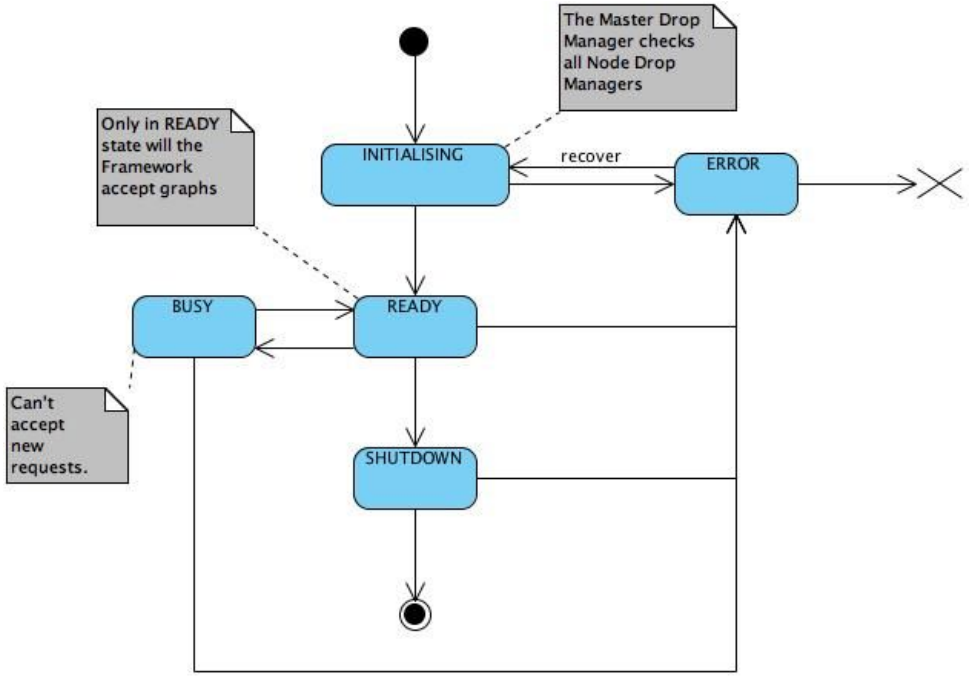

Figure 11: *L¨Y¨f wYbmnxf dnYI¥Yw zk ¨mf . °fb©¨nzx 3¥Ywf¯z¥uÎ Mmf . °fb©¨nzx 3¥Ywf¯z¥un¦ f¦¦fx¨nYw±¥f¢¥f¦fx¨fd a± ¨mf >Y¦¨f¥ ,¥z¢ >YxYI f¥å> , >å Yxd Yw nx¨f¥Yb¨nzx¦ af¨¯ ffx¨mf =>* Yxd ¨mf . °fb©¨nzx 3¥Ywf¯z¥u Y¥f mYxdvfd a± ¨mf > , >Î , ©¥nxI 6?6M6!=6L6?4 ¨mf > , > ¥f In¦¨f¥¦ Yxd bmfbu¦ Yw dn¦bz®f¥Yavf ?zdf ,¥z¢ >YxYI f¥¦Î6k Yw ¥f¦z©¥bf¦ Y¥f k©w±©¨nm¦fd ¨mf > , > ¯nw ¨¥Yx¦n¨nzx¨z)NLVÎ Tmfx¥f¤©f¦¨fd a± ¨mf =>* ¨mf > , > ¯nw ¦m©¨dz¯x Yxd df¥f In¦¨f¥ ¯n¨m Yw ¨mf ?zdf ,¥z¢ >YxYI f¥¦Î , f¢fxdnxI zx ¨mf nw¢vfwfx¨Y¨nzx ¨mf > , > ¯nw fn¨mf¥ af Y ¦±¦¨fw dYfwzx z¥Y ¢¥zbf¦¦ Yb¨n®fv± wYxYI fd a± =>*Ë ¢z¨fx¨nYw± zx ¨mf ¦Ywf bzw¢©¨f¥Î*

12.2 Physical Graph State View

The whole physical graph also follows a state machine during its life-cycle. However, error conditions of a fully data parallel system are quite hard to model in a traditional way, since there is an almost

continuous transition between successful and complete failure. This can be captured by a 'DEGRADED' state, but the transitions between SUCCESS, DEGRADED and FAILURE are then based on thresholds, heuristics and policies. Moreover, it is quite likely that the SDP will always produce somewhat DEGRADED products, be it for excessive RFI, or failed computing nodes.

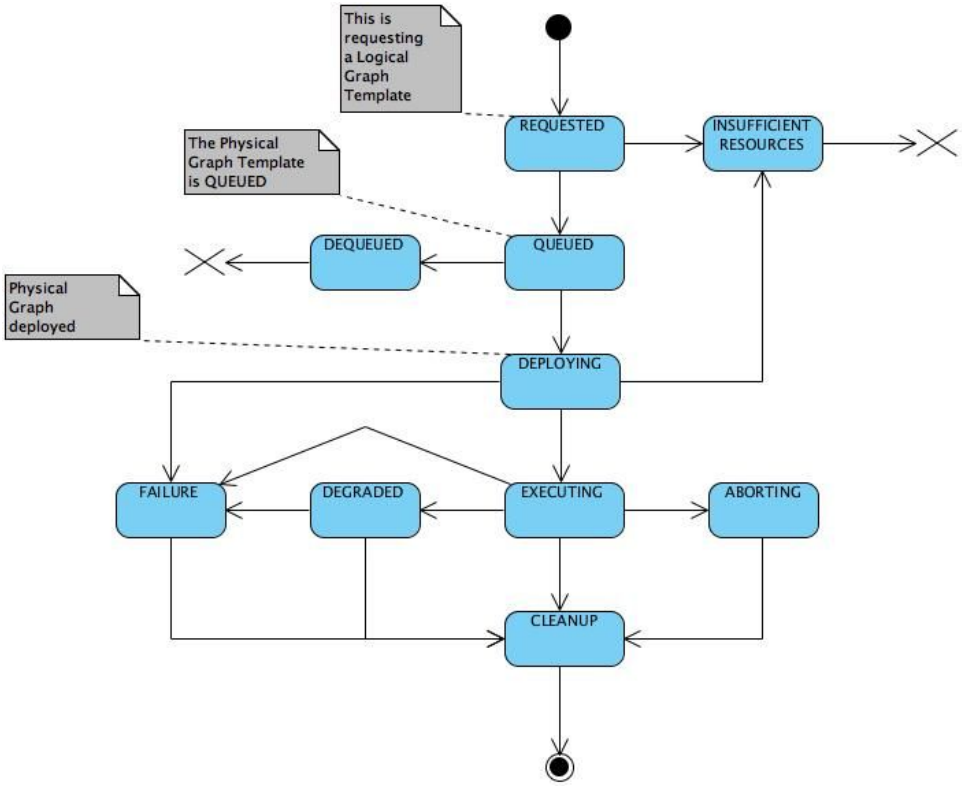

Figure 12: *Mmf ®Y¥nz©¦ ¦¨Y¨f¦ Yxd ¨¥Yx¦n¨nzx¦ Y ¢m±¦nbYv I¥Y¢m bYx ¨Yuf zx d©¥nx I n¨¦ vnkfÝb±bvfÎ Mmf ¦¨Y¨f Y ¢Y¥¨nb©vY¥ I¥Y¢m n¦ nx Yv¦z ¥f¢¥f¦fx¨¦ ¨mf ¦¨Y¨f zk ¨mf z®f¥Yw L, H bY¢Yamm¨± f°fb©¨nzxÎ Axbf ¨mf ¢m±¦nbYv I¥Y¢m mY¦ affx df¢vz±fd n¨ bYx af Yaz¥¨fd a± ¨mf => * ¨m¥z©Im ¨mf >Y¦¨f¥ ,¥z¢ >YxYI f¥Î*

12.3 Drop State Diagram

On the lowest level each Drop has to follow certain state transitions. The basic lifecycle of a Drop is simple and follows the basic principle of writing once, read many times. Additionally, it also allows for data deletion.The model [Figure 13] is based on previous work for ALMA, but has been simplified in a few areas. It covers the life-cycle of a Drop from initialisation to deletion and beyond. If the initialisation fails for some reason the Drop identifiers are void and thus this is marked in the model as a special state. The system has to support multiple versions and copies of the same Drop, thus the model shows both local (one Drop) and global (all versions and copies of a Drop) state transitions. The region highlighted with the green box marks the main cycle during which the Drop is being written. In order to support streaming operations the state can change from 'Dirty' to 'Locked', then back to 'Dirty' between each of the blocks received. Once all blocks have been received, the state will change to completed and the Drop will be set to read-only. If a Drop is detected to be corrupted (i.e., its payload checksum has changed, or its metadata has become corrupted), but can be recovered from

another copy or other means, it will not be available until it is fully recovered. If there is only one copy of a Drop, or if all copies and versions are corrupted the Drop is globally corrupted, making recovery impossible. This will then trigger deletion of all copies and versions. Drops can also be deleted explicitly, thus there is a state transition between Completed and Deleted. Since this state transition is for a single Drop (local) it is possible to undelete (recover) a deleted Drop as a side-effect. In addition to explicit deletion, Drops carry two different expiration datetime values, one indicating local expiration and the other indicating global expiration. This allows for the implementation of the data life-cycle across the whole data infrastructure and in particular across a hierarchy of storage layers including the medium term and long term persistence storage. Local expiration will trigger automatic deletion of a Drop at one specific location. Global expiration will eventually delete all versions and copies of a particular Drop. Since this is an asynchronous operation there is both a logical and a physical 'Deleted' state. This also means that a logical delete operation is very fast while the physical delete can be deferred to a garbage collection process.

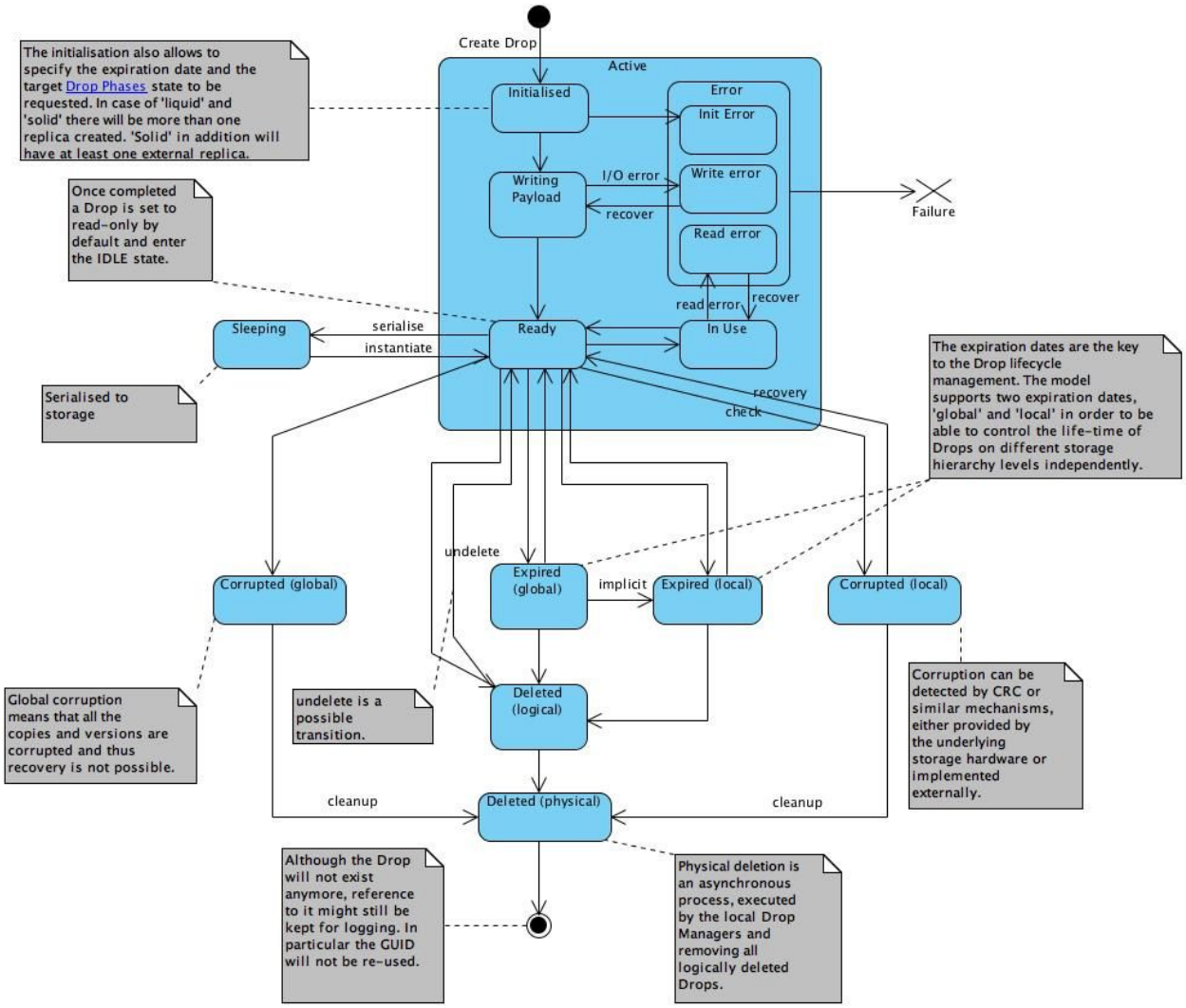

Figure 13: *Mmf ,¥z¢ ¦¨Y¨f dnY I¥Yw n¦ ¨mf vz¯f¦¨ vf®fv zk ¦¨Y¨f wYbmnxf¦ ¯n¨mnx ¨mf .°fb©¨nzx 3¥Ywf¯z¥uÎHf¥¦n¦¨fd ,¥z¢¦ bYx vn®f af±zxd ¨mf f°fb©¨nzx vnkf¨nwf zk ¨mfn¥ Y¦¦zbnY¨fd I¥Y¢m ¢¥zd©bnxI ¨mfwÎMmf bzw¢z¦n¨f Ï!b¨n®fb ¦¨Y¨f ¥f¢¥f¦fx¨¦ ¨mf ¢z¦¦navf ,¥z¢ ¦¨Y¨f¦ d©¥hxI f°fb©¨nzxÎ ! ,¥z¢ bYx fx¨f¥ ¨mf ÏLvff¢nxIb ¦¨Y¨fË nk n¨ n¦ ¦f¥nYn´fd ¨z Y ¢f¥¦n¦¨fx¨¦ ¦¨z¥YI f vY±f¥Î6¨bYx ¨mfx af¥fÝYb¨n®Y¨fd ¨z ¢Y¥¨nbn¢Y¨f nx Y I¥Y¢m f°fb©¨nzxÎ6x z¥df¥¨z bY¢¨©¥f Yxd bzx¨¥zv ¨mf ¦¨Y¨f¦ nx ¨mf vz¯f¥ ¢Y¥¨ zk ¨mf km I©¥fË ¨mf L, H Y¥bmn¨fb¨©¥f Yv¦z kfY¨©¥f¦ Y ,¥z¢ =nkfb±bvf >YxY I f¥Î?z¨f ¨mY¨¨mn¦ dnY I¥Yw bz®f¥¦ Yw ¨±¢f¦ zk ,¥z¢¦Ë nxbv©dnx I ,Y¨Y ,¥z¢¦ Yxd ! ¢¢vnbY¨nzx ,¥z¢¦Î Lnxbf ,¥z¢¦ Y¥f nww©¨YavfË¯f dz xz¨ kz¥f¦ff Y ¨¥Yx¦Yb¨nzx ¨±¢f wzdfv zk ¦¨Y¨f ¨¥Yx¦n¨nzxÎ*

12.4 Drop Events

Changes in a Drop state, and other actions performed on a Drop, will fire named events which are sent to all the interested subscribers. Systems can subscribe to particular named events, or to all events.

In particular the Node Drop Manager subscribes to all events generated by the Drops it manages. By doing so it can monitor all their activities and perform appropriate actions as required. The Node Drop Manager, or any other entity, can thus become a Graph Event Manager, in the sense that they can subscribe to all events sent by all Drops and make use of them. This functionality can be used for example to visualise the progress of the execution of a graph.

13 Drop Lifecycle Management

The SDP needs a subsystem that automates and manages the migration of Drops through the various life cycle states as well as the location of the Drops across the storage hardware, both vertical (storage hierarchy) and horizontally (Fast Buffer, Staging/Mid-term storage and Long Term Storage). The goal of Drop lifecycle management is the optimal placement of Drops in terms of cost and performance and employs a multi-tiered storage system to do so. The actual hardware landscape both within compute nodes, but also more globally in the SDP Processing Platform will likely change significantly during the lifetime of the SKA. Thus it is important to provide a flexible system, which allows adaptation to the underlying platform, while minimising performance loss. It is also important to note that the lifecycle of a Drop can span from minutes to many years and therefore has to seamlessly cover the processing as well as the preservation domain.

Drop lifecycle management has to support a number of requirements:

1. Migration of Drops from one medium to another: SKA1-SYS_REQ-2728
2. aggregation of Drops into Science Data Products and Data Packages: this is non-trivial because of concurrent workflows, SKA1-SYS_REQ-2128, SDP_REQ-252, and the need for data tracing, provenance and access control, SKA1-SYS_REQ-2821, SDP_REQ-255
3. migration between storage layers, includes SDP_REQ-263
4. replication/distribution including resilience (preciousness) support, incl. SKA1-SYS_REQ-2350, SDP_REQ-260 - 262
5. retirement of expired (temporary) Drops, incl. SDP_REQ-256

In addition to the life cycle of single Drops the architecture also introduces the notion of Drop Phases (Figure 14). The phases are related to the resilience of a Drop against loss or corruption. In particular in cases where resilience is implemented using replication and/or mirroring this adds a complete additional complexity to the lifecycle management, the Drop state diagram and in fact the definition of the abstract Drop class itself. In particular it is not enough anymore to deal with Drops as singular

objects, but there are now multiple Drops with identical payloads, potentially in quite different physical locations. As a direct consequence the Drop class introduces an OID ***Yxd*** an UID for a Drop. The OID is the same for all identical copies of a Drop, the UID is a unique identifier for every single Drop instance in the whole system. The lifecycle management system now needs to take care of all the Drops with the same OID in a coherent way. It also needs to make sure that any negative Phase change (see Figure 14) due to a failure is counteracted by producing a new copy of a Drop with a new UID. The other consequence originating from the resilience Phases is the introduction of multiple expiration timestamps into the abstract Drop class. This allows the lifecycle management system to clean up all instances of a Drop with the same OID at once, or each of them individually. This is particularly important for multi-level storage and cache systems with limited capacity. The required information for the Drop Lifecycle Management will be kept in a distributed Drop Lifecycle Database. This database will also be used as a master index for the Drop locations.

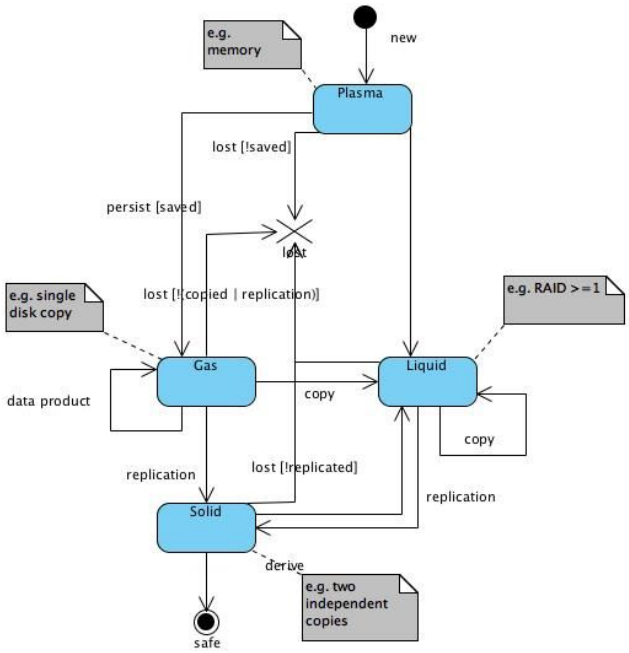

Figure 14: *Mmf ¥f¦ɯfxbf ¦¨Y¨f¦ zk¨ɯf , ¥z¢ ¦±¦¨fw Y¥f bY¢¨©¥fd Y¦ HmY¦f¦Î , ¥z¢¦ bYx bmYx l f HmY¦f¦ k¥zw HvY¦wY¨z 4Y¦Ë =ɲ¤©ɲd Yxd Lzɯdˈ df¢fxdɲx l zx¨ɯf b©¥¥fx¨ vf®fv zk ¢¥z¨fb¨ɲzx Y l Yɲx¦¨vz¦¦ Yxd bz¥¥©¢¨ɲzxÎ Mmf dfkɲxɲ¨ɲzx zk¨ɯf ¥f¤©ɲ¥fwfx¨¦ ¨z ¥fYbm Y bf¥¨Yɲx HmY¦f f¦¦fx¨ɲYw± ɲ¦ Y ¢zɯb± dfbɲ¦ɲzxË f°Yw¢vf¦ Y¥f ¢¥z®ɲdfd ɲx¨ɯf kɲl©¥fÎ*

14 Discussion of and Allocation of Functions to Products

The product tree and the functional breakdown for the Execution Framework are summarised below. The allocation of the functions to the products is quite straight-forward

- C.1.2.3.1 Graph Event, Schedule & Workflow Framework
  - C.1.2.3.1.1 Graph Event Manager
  - C.1.2.3.1.2 Graph Scheduler
  - C.1.2.3.1.3 Graph Workflow Manager

- C.1.2.3.2 Graph Management Framework
  - C.1.2.3.2.1 Logical Graph Manager
    - C.1.2.3.2.1.1 Logical Graph Repository
  - C.1.2.3.2.2 Physical Graph Manager
    - C.1.2.3.2.2.1 Physical Graph Repository
- C.1.2.3.3 Drop Implementation Framework
  - C.1.2.3.3.1 Drop Component Interface
  - C.1.2.3.3.2 Drop Channels
  - C.1.2.3.3.3 Drop Control
  - C.1.2.3.3.4 Drop IO Framework
- C.1.2.3.4 Drop Management Framework
  - C.1.2.3.4.1 Master Drop Manager
  - C.1.2.3.4.2 Node Drop Manager
  - C.1.2.3.4.3 Island Drop Manager

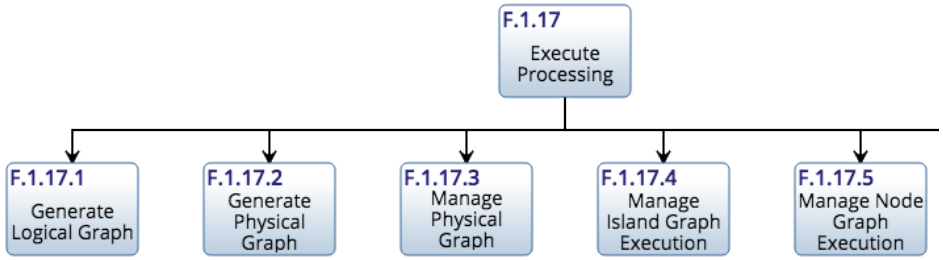

Figure 15: *3©xb¨ħzxYv a¥fYudz¯x zk ¨ṁf . °fb©¨ħzx 3¥Ywf¯z¥uÎ*

MYavf þÍ !wzbY¨ħzx zk k©xb¨ħzx¦ ¨z ¢¥zd©b¨¦

Product	Function
C.1.2.3.2.1	F.1.17.1 Generate Logical Graph
C.1.2.3.2.2	F.1.17.2 Generate Physical Graph
C.1.2.3.4 Drop Management Framework	F.1.17.3 Manage Physical Graph
C.1.2.3.4.3 Island Drop Manager	F.1.17.4 Manage Island Graph Execution
C.1.2.3.4.2 Node Drop Manager	F.1.17.5 Manage Node Graph Execution

15 Discussion of and Allocation of Requirements To Functions

The requirements for the Execution Framework are still in active development and the mapping to L1 requirements is quite tricky, because of the abstract and generic nature of the Execution Framework in particular and software in more general terms. The design of the Execution Framework is a result of the analysis of the complexity, variety, commensality and parallelism of the L1 and L0 requirements and led to a system design, which allows the implementation and execution of many different pipelines both in parallel and sequentially on a variety of different input data streams. In other words, obviously there is no L1 requirement stating 'The SDP shall implement an Execution Framework'. Thus all of the requirements directly related to the Execution Framework are derived requirements and will heavily depend on a fairly complex analysis of the SDP capabilities to be implemented. In addition the detailed performance requirements for the Execution Framework are heavily dependent on the actual implementation of the algorithms (pipeline components). Thus we are listing here a sketch of the requirements we had been working against.

15.1 Performance Requirements

P1. Handle incoming data rates of ~0.4 TByte/s/array in up to 16 (TBC) independent streams/observations [RD07 and RD08]

P2. The Execution Framework shall switch between observations within less than 30 (TBC) seconds, which satisfies SKA1-SYS_REQ-2133, mode transition. The overhead induced by the Execution Framework management and control shall be less than 5 % (TBC) of total SDP FLOPS.

P3. Drop generation and deletion time shall be less than 10 seconds/1 million Drops across the complete deployment, not including physical deletion of the payload.

P4. The Execution Framework shall be able to manage a total of not less than 10 million Drops at any given point in time, across all graphs deployed in parallel.

15.2 Functional Requirements

F1. Support batch/queue data processing

F2. Predictable processing times and latency of data product delivery for selected products.

F3. Automated data and job placement optimisation based on profiling of pipeline components and hardware parameters

F4. Fault tolerance at job level

F5. Resilience for precious data

F6. Allow for (limited) communication between compute islands

F7. Allow parallel graph executions (e.g. for subarrays).

F8. Allow multiple graphs to (re-)use the same data (data commensal scheduling).

F9. Support high level graph development independent from low level platform or algorithmic concerns.

F10. Clean and lean, well documented framework API to enable wrapping/integration of existing as well as new pipeline algorithms. In particular this means that the framework needs to support the very

loose integration of command line level algorithms as well the very tight integration of algorithms directly into the framework in order to leverage the full potential of the built-in I/O layer.

F11. Integration of Execution Framework monitoring and control with LMC and TM

F12. Integration of Preservation

F13. Integration of multiple storage and memory levels and technologies, like slow hard drives, fast hard drives, SSD, NVRAM, and Hybrid Memory Cubes and flexibility to benefit from upcoming technologies.

Table 2: *!wzbY¨ìzx zk ¥f¤©ì¥fwfx¨¦ ¨z k©xb¨ìzx¦Î*

Function	Requirements
F.1.17.1 Generate Logical Graph	P1, F9, F10
F.1.17.2 Generate Physical Graph	F3, F5, F12, F13
F.1.17.3 Manage Physical Graph	P2, P3, P4, F7, F8, F11
F.1.17.4 Manage Island Graph Execution	P2, P3, P4, F2, F7, F8
F.1.17.5 Manage Node Graph Execution	P2, P3, P4, F7, F8

16 Interfaces

The Execution Framework will enable the external SDP interface with CSP, by providing the environment to execute various receiver pipelines. The implementation of the receiver pipeline components is responsibility of the Receive Components product. The Execution Framework will also have internal interfaces with the Control SDP and Preservation products providing and consuming control and monitoring information with Control SDP and providing and consuming science data products to/from the Preservation product. The Execution Framework will provide an API for the development of the Pipeline components and then interface through that API with the components.

17 Discussion of Element Risks

Almost the complete architecture described above has been prototyped as part of the Data Flow Management System prototyping activity (DFMS, see memos [RD10] and [RD5]).

The prototyping followed two basic principles:

1. Minimal viable product
2. Constant feedback

The first means that the DFMS implemented only functions, which deemed risky or necessary to verify the architecture. The second means that very often during the implementation we found details or issues with the architecture, which needed to be addressed and then verified again.

Note that the DFMS primarily is a prototype of the Execution Framework, not a prototype of explicit pipelines (neither Receive nor others) or the SDP control layer. However, in order to verify that the Execution Framework architecture could work in a real world environment, we have also implemented actual science reduction pipelines to deal with VLA and LOFAR data. The former proved very valuable already for the CHILES project [RD06] .

In general we would thus rate the risk for the overall element architecture, including implementability, quite low. However we have not yet addressed the following high level risks to the degree to be confident about fulfilling the performance requirements:

- **Scaling to SKA1 fails:** We would judge the risk 'moderate'. The SKA1 pipelines will require the deployment of several million Drops across the whole processing platform for a full-scale imaging capability. Although we have confirmed that the overhead introduced by the framework is small enough to allow this in principle, we still need to verify scaling properties in a more systematic way. This is in the plan for the next data challenge. To mitigate this risk we have put this verification into the upcoming Data Challenge. The impact would be major and thus even if the risk is moderate, we need to understand very well where the limits are.
- **Translation from Logical to Physical Graph too complex:** We would judge the risk 'possible'. This class of problems has been identified as one of the NP-hard problems in computer science. In essence this means that the risk will essentially always exist, but in general such problems can be simplified by narrowing the parameter space to make them manageable. This, together with research in the area is our approach to mitigate the risk. As a fallback we can use ad-hoc quasi-static deployments and make them more dynamic as we go. In addition this is an extremely active field, with lots of research strains and even the algorithms we are using right now in the DFMS already look very promising, if not sufficient. This topic is part of our on-going activities. Due to the fallback options, the impact would be moderate to low.
- **I/O Interface insufficient:** We judge this risk as 'very likely' for the current DFMS implementation, and 'probable' for the first real implementation. The architecture itself is very flexible in this regard and allows to 'plug-in' a wide variety of existing or upcoming I/O frameworks. The amount of work required to identify, interface and optimise such frameworks is a separate related, but lower level risk. We are working on a number of frameworks, such as ADIOS, HDF5 and some advanced database engines. Again this requires more work and prototyping, but is planned and can be distributed. The impact is very high, since it will limit the amount of science that can be done.
- **Integration with LMC hard:** We would judge this risk 'unlikely'. In order to mitigate it even further we are looking into an integration prototype. The impact would be high.
- **Integration with future Pipeline components:** We would judge this risk 'unlikely', mostly based on our experience with interfacing existing pipeline components with the DFMS. The real verification will have to be the integration with prototypes of SDP PIP algorithms. The impact would be very high.
- **Logical Graph language syntax insufficient:** We would judge this risk as 'likely' for the DFMS prototype, but it is actually an implementation risk, not an architectural risk. The Execution Framework allows to change the implementation of how to express logical graphs in a modular way, thus we could use a more expressive syntax in the future. Our current approach is to implement more logical graph constructs as we go and find them required to express

more complex SDP capabilities. The impact would be moderate, because we can always make short-cuts.

- **Dynamic Scheduling too complex:** We would judge this risk as 'very likely', mostly because fully optimised dynamic scheduling is a really hard problem. The impact is moderate, since we can always fall-back to less optimal scheduling, although this might affect the science capabilities. On the other hand it is not really clear what level of dynamic scheduling is actually required. This topic requires additional prototyping to reduce the risk.