Department of Computer Science and
Department of Space and Climate Physics,
University College London,
University of London.

Modelling grid architecture

Joe Lewis-Bowen

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
at the University of London.
2007

Abstract

This thesis evaluates software engineering methods, especially event modelling of distributed systems architecture, by applying them to specific data-grid projects. Other methods evaluated include requirements' analysis, formal architectural definition and discrete event simulation. A novel technique for matching architectural styles to requirements is introduced.

Data-grids are a new class of networked information systems arising from e-science, itself an emergent method for computer-based collaborative research in the physical sciences. The tools used in general grid systems, which federate distributed resources, are reviewed, showing that they do not clearly guide architecture. The data-grid projects, which join heterogeneous data stores specifically, put required qualities at risk.

Such risk of failure is mitigated in the EGSO and AstroGrid solar physics data-grid projects' designs by modelling. Design errors are trapped by rapidly encoding and evaluating informal concepts, architecture, component interaction and objects.

The success of software engineering modelling techniques depends on the models' accuracy, ability to demonstrate the required properties, and clarity (so project managers and developers can act on findings). The novel formal event modelling language chosen, FSP, meets these criteria at the diverse early lifecycle stages (unlike some techniques trialled). Models permit very early testing, finding hidden complexity, gaps in designed protocols and risks of unreliability. However, simulation is shown to be more suitable for evaluating qualities like scalability, which emerge when there are many component instances.

Design patterns (which may be reused in other data-grids to resolve commonly encountered challenges) are exposed in these models. A method for generating useful models rapidly, introducing the strength of iterative lifecycles to sequential projects, also arises. Despite reported resistance to innovation in industry, the software engineering techniques demonstrated may benefit commercial information systems too.

Contents

Chapter 1 Overview

This document describes the application of software engineering methods to 2 data-grid projects: AstroGrid and, especially, EGSO. Data-grids are open computer networks for information sharing. AstroGrid supports UK astronomy, EGSO is the European Grid of Solar Observations. After analysing the requirements of data-grids with well-recognised techniques, 3 innovative high-level design modelling practices are evaluated:

- formal architecture description (described in Section 4.2), using Architecture Common Model Environment (ACME), an architectural description language (ADL),
- process event modelling (described in Chapter 5), using the Finite State Process (FSP) language and its Labelled Transition System Analysis (LTSA) tool,
- discrete event simulation, with the SimPy Python toolkit (introduced in Section 6.2).

This work improves the implemented quality of the projects (for example, by determining a safe complete protocol for EGSO middleware; see Section 5.5), as fully described in discussion accompanying later descriptions of the methods. Though the value of the novel techniques is proven, shortcomings are noted. Broader lessons are also drawn from the model instances developed for the case studies; there is concrete evidence that reusable design solutions can solve challenges shared across the emergent data-grid domain.

Section 1.1 of this chapter discusses the goals of the research and the thesis' argument. Section 1.2 gives a working definition of data-grids and describes their proponents' aspirations. Section 1.3 introduces the domain and methods, and is therefore an overview of subsequent chapters.

1.1 Contribution, argument and audience

The research contribution is at the boundary of 3 domains: e-science (defined Section 1.2 below), software engineering and solar physics. The work in each benefits the others:

- As an emergent domain reliant on computing solutions, e-science needs disciplined software engineering (rather than chaotic reactive software development). The application of lifecycle management, requirements' analysis and modelling to solar physics e-science projects is useful and original in itself (see Chapter 2). Good software engineering analysis benefits the projects it is applied to and subsequent projects, which can reuse successful generic techniques and abstracted design patterns.
- Software engineering also benefits from this work through the application of its methods in genuine development. Case studies of established methods are useful, and the evaluation of innovative techniques is essential. A significant contribution of this research is therefore the encoding of high-level software designs in FSP (analysed by LTSA; see Chapter 5). Security and goal

requirements' analysis (Chapter 3), static architectural description (Chapter 4), and event simulation methodologies (Chapter 6) are also evaluated.

- The contribution to solar physics is less, but there is clear practical benefit to the science from well-developed data-grid systems. Once the research had contributed to the initial design, it supported review by critical experts in EGSO and AstroGrid development, validating the evolving systems. The use cases, including the detailed possible scientific data-grid investigations, also guided project managers, scientific community representatives and developers' understanding of data-grid capabilities (see Chapter 3).

The motivation for the research arose from the needs of solar physics for better exploitation of data resources via the Internet. Through rigorous requirements' analysis, it was demonstrated that astronomers need a data-grid. By reviewing grid tools and projects, it is apparent that current emergent e-science practices need software engineering; application of rapid evolutionary architectural modelling can usefully guide early design and mitigate the risk of projects being built with weak quality. As it is also recognised that engineering methodologies benefit from the published evaluation of case studies, this experience report has value beyond e-science. The argument running through this thesis therefore narrates the application of engineering to solar physics data-grids to demonstrate the value (and limits) of software modelling.

As the research includes 3 domains, it should be of interest to a variety of readers:

- Software engineers (who are familiar with the methods applied) should be able to judge the utility of modelling and analysis techniques. Of especial interest is the ability of models to capture the behavioural quality; methods to demonstrate that designed systems satisfy non-functional requirements (NFR) are not established.
- Data-grid professionals, with either scientific or computing backgrounds, should be interested for the reported experiences and development guidance. Beyond the domain review, they can cross check their project requirements against the abstract data-grid requirements analysed here, follow the modelling methodology arrived at through experience, and reuse the emergent design patterns noted.
- Astronomers who analyse observations on computer networks may also find this work of interest, as it uses concrete scientific examples to show how e-science can be done with data-grid systems.

1.2 The data-grid vision

The grid concept arises from computational science, making the analogy to electricity supply – the distributed delivery of power from several providers to many users, who are not aware of the means of supply. Grid technology connects heterogeneous computing resources, enabling new ways of working for distributed users. Users should be able to collaborate in dynamic virtual organisations, securely sharing just the necessary agreed assets [44]. They

should also experience transparent service, so it does not matter where their data is held or how their applications complete requested actions. The grid vision differs from the World Wide Web (WWW), whose users can only request published assets, and business enterprises, with inflexible application integration across sites.

Grids were initially conceived as providing high performance processing by facilitating remote access to diverse super-computing resources. Data-grids (including EGSO and AstroGrid) are a specialised class, where stored resources are more important. They arose from the physical sciences, which benefit from more efficient information exchange than the WWW can deliver. They emphasise broad access to complex data resources, streamlining searches, queries and analysis (their characteristic requirements are described in Chapter 3).

The vision for grid capability goes further. There is evidence that the rate of exponential growth in available bandwidth and storage capacity is faster than Moore's Law [100], so it will make economic sense to increase productivity by integrating more resources on networks (instead of investing in better localised computing capabilities). The easier transformation of low-level data to higher-level information through analysis and annotation in data-grids supports knowledge and developed nations' economies at the highest level. Automating the organisation of information assets is an idea that predates computing [16], and the revolutionary impact of inventive information infrastructure that were intended to meet physical sciences' needs has a precedent in the WWW [60]. For the scientists, data-grids may form part of the computerised ways of working that are transforming the scientific method; proof by experimental validation of theoretical models with a combination of *in silico* experiments and data mining is now possible.

However, data-grids must be capable of going beyond functional delivery to uphold good quality of service; global infrastructure relies on collaborative support through distributed responsibility. Additionally, the freedom that the grid grants users will lead to failure through voluntary withdrawal if resources are untrustworthy (not respecting intellectual property) [79]. Grid technology therefore needs software engineering, to evolve beyond existing technology and meet emergent needs in the novel domain.

1.3 Topics presented

This document includes the 5 elements of scientific investigation:

- The *hypothesis* that architectural modelling adds value to data-grids has already been introduced in Section 1.1.
- This is demonstrated to be a novel and valuable topic to investigate by the *domain review* of data-grids and their application to solar physics, described in Section 1.3.1. and given in Chapter 2 and Chapter 3
- Overviews of the 3 *methodologies* used for the investigation are given in Section 1.3.2.
- The experimental *findings* are given with details of the methodologies' application in the core of this document: Chapter 4, Chapter 5 and Chapter 6.

- A *critique* of the results is given in Chapter 7, describing their limits and general implications. Chapter 8 summarises the thesis' findings and considers future research.

14 appendices, with details of the domain analysis and models, support these elements.

1.3.1 Domains: software engineering, solar physics

Software engineering for data-grids

Software engineering defines methods for developers, managers and other system stakeholders that ensure consistent project quality. Chapter 2 describes how lifecycle engineering mitigates the risk of software crisis in general, and then examines architectural styles that are relevant to the grid domain. In this way, the chapter introduces modern software engineering methods to the data-grid developers who need them, unambiguously defines the architectural styles referred to subsequently, and provides an overview of grid tools' capabilities.

The software engineering domain review starts with software lifecycles, as they define a framework on which other engineering methods can be placed. The V-diagram is a powerful representation of the sequence of development activities (shown in Figure 1), which associates successively earlier and later stages (about software coding at the centre) at increasingly greater abstraction (away from concrete implementation details). The reader may note that this structure also applies to this document; from the abstract discussion of software engineering for grids, it focuses on increasingly detailed design modelling methods for the specific data-grid projects, then observations pull back out to general implications of the research.

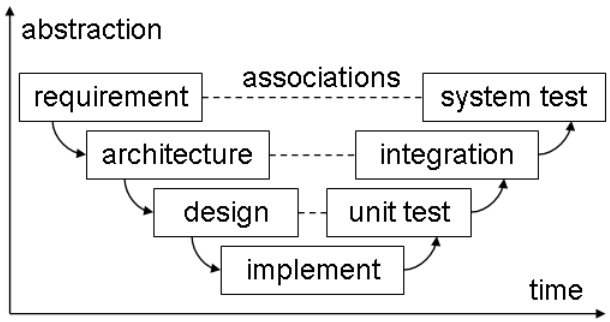

Figure 1: The V-diagram indicates how the abstraction of early design steps correspond to later tests; detail focuses in the middle of the lifecycle [99].

Solar physics data-grid requirements

The domain review of e-science is completed by Chapter 3, which notes a broad variety of scientific data-grid projects' architectural styles. It then describes the data-grid needs of solar physics by narrating the analysis conducted of the domain's requirements. The understanding of the EGSO and AstroGrid projects' needs that emerged directed the subsequent modelling investigation. The research was therefore embedded in other project activities (shown on Figure 2, on which components of this research, numbered by section, are marked with grey rounded rectangles, connected to clear rectangles for italic EGSO and AstroGrid deliverables,

associated with citation references); the dependencies between the research contributions and the projects' artefacts make them mutually supportive.

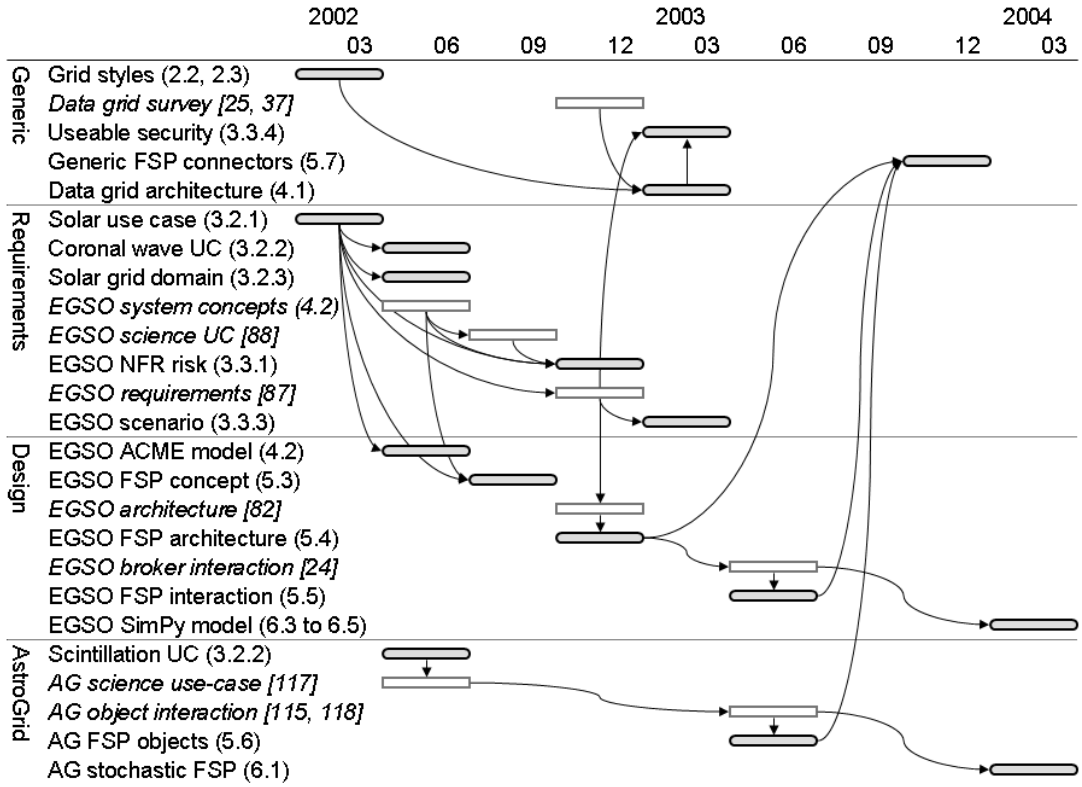

Figure 2: Gantt chart of EGSO and AstroGrid project activities associated with the research contributions (in grey), illustrating dependencies.

1.3.2 Methodology of the 3 modelling schemas

Software architecture

The EGSO and AstroGrid requirements' analysis research component (presented in Chapter 3) also serves as a record of basic case studies for the techniques, including: various levels of use case analysis, goal decomposition, domain modelling, NFR analysis and security modelling (these representations' details being given in Appendix A through to Appendix D). In contrast, the method for fitting data-grid requirements to architectural styles (described in Section 4.1, detailed in Appendix E) is an original requirements engineering method; it is a lightweight technique that bridges data-grid needs and design.

Chapter 4 also describes the experience of capturing EGSO's preliminary sketched architecture in ACME, working toward ADL specification and analysis of the system (the model is given with maintenance documentation in Appendix F). This work identified gaps in the informal architectural concepts, and made functional partitions and interface requirements explicit. It identified 4 fundamental data-grid component types, which could be adapted to different architectural roles, by capturing the basic communication patterns of their connectors. However, difficulties were encountered in communicating findings to EGSO project managers, so further investigation of this method (for example, by experiments with more detailed designs of the component types) was not carried out.

Event modelling

The event modelling described in Chapter 5 represents the largest contribution that the research makes in assessing software engineering methodologies. Models are specified in the declarative FSP language, which captures processes' state transition events (an original tutorial for FSP is given in Appendix G). Models were created for 5 data-grid design levels: concept, architecture, interface, object and (outside specific projects' lifecycle) generic patterns (described in Section 5.3 to Section 5.7, detailed in Appendix H to Appendix L). All were assessed with the LTSA tool, successfully demonstrating the utility of this technology for high-level design. Additionally, as reported, this work was found to be demonstrable, and therefore positively affected the quality on the EGSO and AstroGrid projects; it proved to all project stakeholders that the designs supported desirable behaviour such as reliability, and sometimes indicated slight refinements to those designs. Lessons learnt (drawn together in Chapter 7) can be applied to other applications of modelling and future data-grids.

Simulation

Event modelling through formal specification overlaps with discrete event simulation. FSP language extensions exist that include timing and probability properties; Chapter 6 reports on limited experience in applying this innovation by modelling AstroGrid messaging (details of the developed models are in Appendix M). The results of SimPy models of EGSO broker design are also described (in Section 6.3 to Section 6.5, with the models' details and experimental results being given in Appendix N); broader lessons concerning data-grid scalability emerge from these.

The reported experience demonstrates the different strengths of the evaluated procedural modelling methods (discussed in Chapter 7); they capture emergent system qualities and permit experimental proof against hypotheses (even without empirical performance data). Chapter 7 also describes observed reusable data-grid design patterns, a general model development lifecycle, and the techniques' value in commercial software development. Chapter 8 summarises the thesis and describes further potential research.

Key points

At the end of each chapter, the key points arising are listed. For this introduction:

- This thesis is primarily a report on the application of software engineering methods (especially modelling, but also requirements' analysis and lifecycle management) to e-science data-grids (specifically EGSO and AstroGrid). This benefits software engineering as well as e-science (and, to a lesser extent, solar physics).
- By supporting access to scientific tools and observations, data-grid networks enable collaborative analysis and rapid dissemination of information about data, and therefore accelerate the growth of knowledge.

Chapter 2 Software engineering for data-grids

This chapter first describes what software engineering can achieve (Section 2.1). From generally recognised software lifecycle practices, discussion moves onto the importance of modelling and architectural abstraction (Section 2.2). Data-grids tools are then introduced (Section 2.3; specific projects' requirements are described in Chapter 3), concentrating on the architectural styles they follow. Their capabilities demonstrate why data-grids are a software engineering challenge.

The chapter therefore summarises software engineering best practice (focussing on architectural design) and its application to data-grids. Scientific readers who have come to data-grid development without extensive software engineering knowledge may benefit most from reading it. However, it is also a critique of engineering practices and data-grid tools; conclusions about the value of the techniques are presented alongside the evidence that supports them. Overall, the chapter sets up the backdrop against which the modelling research makes a valuable contribution.

2.1 The purpose of software engineering

Software crisis

A software system's lifecycle runs from its conception, through production, into deployment, use and subsequent maintenance or evolution. It is widely recognised that failure rates of major software systems are too high; systems are frequently delivered late, over budget, with bugs and shortcomings that make them unsuitable for their intended use. This is termed 'software crisis', first applied to bespoke systems [31], but still relevant in modern software development using reliable components [22,14]. Note, 'components', is a term borrowed from electronic circuits, which is taken to include low-level hardware and compilers up to specialist functional libraries and data management protocol application programmer interfaces (APIs) in computer systems.

Developers have noticed that coding effort only mitigates the software crisis up to a point, beyond which it becomes too costly to investigate unwanted behaviour and improve functionality (without introducing more faults). It is unplanned bug fixes and *ad hoc* interfaces between components that make systems unmaintainable and brittle (where brittleness means that catastrophic failure is caused by small errors). Good engineering can overcome these problems. Especially, effort invested early in the lifecycle can help systems succeed when deployed on a large scale or for use over a long period of time, as data-grids expected to be.

The Waterfall Process

Engineering's Waterfall Process manages projects' lifecycles to avoid software crisis [92]. It formally divides the phases of system development into sequential steps, encompassing at the minimum: requirements' analysis, design, implementation and testing. It emphasises gates between these phases, typically implemented by organisations as processes for documenting and reviewing progress. Note that such gates only meet business project

management needs [109], so they may not give technical support to later phases. Guarding transition between steps facilitates early detection of faults and divergence from project goals. In this way, the path taken through the lifecycle can be imagined as an increase of function and quality over time, until the target corner of a lifecycle cube is met (Figure 3) [34].

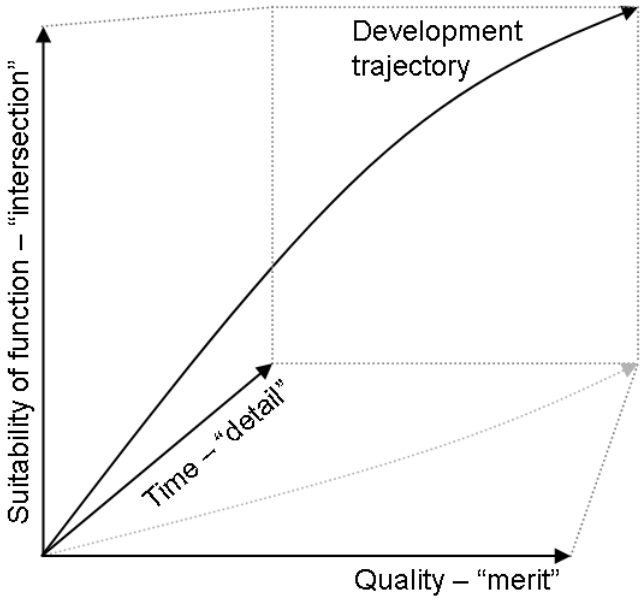

Figure 3: Development lifecycle cube; project quality and functional intersection with requirements improve early, whilst detail is low, so suitable levels are reached within time.

However, in practice projects following a Waterfall Process can still be delayed and unusable after delivery (falling on the suitability axis and ending later on the time axis of the lifecycle curve of Figure 3). This is typically because the early steps are difficult to complete. Requirements are often poorly defined when projects begin, as the customers may not appreciate what technology can provide or understand future users' needs. As the project progresses, they may wish to refine their requirements, but the Waterfall Process makes it awkward and expensive to modify work in progress. (In fact, as originally specified, the waterfall acknowledged the possibility of significant iterations, accepting that these are hard to plan for.)

The Spiral Model

Recognition of the Waterfall's shortcomings lead to the development of iterative development, exemplified by the Spiral Model [11]. This explicitly compresses and repeats lifecycle steps, so that requirements' analysis can be repeated after the development of early versions of the software. At the early iterations, partial implementations (prototypes with incomplete functionality or mock-up systems) are developed with minimal effort, so there can be large changes to functionality after testing. In later iterations, more coding and testing effort can raise quality (for example, improving performance with advanced coding techniques); requirements' refinement should then focus on behavioural targets rather than functional changes (following the curve shown in Figure 3).

The Spiral Model also has weaknesses though. The quality improvement deferred to later iterations may never be completed when the customer accepts the prototype system as functionally complete. The delivered system is therefore low on the quality axis of the lifecycle cube. Lax development practices, which permit inefficient or unmaintainable code developed at early iterations to be reused alongside functionality requested later, also lead to a brittle system with complex and chaotic structure. Developers often exacerbate this by not completing documentation or supporting test effort, believing this is the intent of compressed lifecycle iterations. In fact, the Spiral Model strongly emphasises the importance of documentation at reviews within iterations (so that project management milestones and decision gates can be synchronised with implementation progress checkpoints and reviews).

Modern rapid techniques

Modern lifecycles, notably Extreme Programming [7], attempt to compromise developers' desire for lightweight process overheads and the customers' needs for quality and early flexibility. For example, the test-first method is recommended; the tests are guaranteed to fail before functionality is implemented, but the process helps engineers develop 'just enough'. Though such methods may help systems rise sufficiently high on the quality and functional axes within good time in the lifecycle cube, they depend on integrated commitment to novel methods by the customer and developers to deliver successfully without the review process' artefacts that ensure traceability.

It is also recognised that individual projects' lifecycles must fit broader streams of development within organisations. The term Architectural Business Cycle has been coined for engineering synergy between projects to maximise the possibility of component reuse [6]. Such strategic long term planning maximises the return on effort invested in developing high quality where it is of most benefit, and gives projects a head start in the lifecycle cube. This is analogous to using well-engineered parts in an industrial production line.

Applicability to data-grids

Academic data-grid projects, including EGSO, use a changing body of distributed developers who must coordinate their own efforts. The users are scientists and administrators with focussed technical skills; their evolving needs in this novel domain can be phrased as narrow solutions, expressed with reference to their familiar ways of working. Both developer and user interests are represented by managers removed from day to day activity by other commitments. A high standard of software lifecycle management is therefore desirable; it should avoid wasted development effort and prevent project failure. (In this context, the investigation presented in this thesis guides projects' development trajectory, whilst being validated itself by other lifecycle artefacts.)

Though data-grids are therefore exposed to the same risks as large commercial software projects, they can still leverage an academic Architectural Business Cycle. Despite supporting diverse scientific domains, projects are not isolated; different subjects are likely to share common infrastructure. By investing effort in developing high quality reusable components, and exploiting others' published resources, the whole emergent enterprise stands

to gain. (It is in this context that the identification of design patterns for the emergent domain has value.)

2.2 Software engineering architectural styles

The value of architecture

The architecture of software captures high-level system design; it is used early in the development lifecycle, and to reverse engineer established systems [94]. Systems' architecture is independent of technology, but traceable to implementation instances. Its abstract view of essential features typically just represents system components and their connections. When a system is developed to a unified overall vision, defining key connections early, there is lower risk of poor quality emerging at system integration (when it may be impossible to reengineer components' coupling). Making architecture explicit permits evaluation and refinement before complex implementation design detail is specified.

The architectural view permits identification of styles common to diverse systems, capturing knowledge that would otherwise remain as unrelated experiences. It also encourages traceability between customers' requirements and the developers' activities, allowing responsibility to be divided between architectural elements. Further, architecture should be more than an abstract sketch of an envisioned system. As in construction, the architect should go beyond just meeting users' needs to deliver a solution that has the beauty of an integrated and economic design, functional elegance being achieved by consistently following sound principles to deliver homogenous quality. However, there are few examples of admired software solutions; most projects are judged successful if they merely escape the software crisis.

Beyond the straightforward association of architectural components to required functionality (defined simply, what is output for given input), so-called NFR are harder to trace. NFR include qualitative behavioural properties – for example, systems should be: optimal, scalable, robust, usable, maintainable, and secure [98]. Though such a definition of NFR is criticised as being a meaningless grouping, alternative classifications – to development versus operational requirements [13] or execution, non-runtime and business requirements [6] – are not widely recognised. There is agreement that NFR are not met by individual components, but by their integrated properties, and that NFR are traded-off against each other [73,66]. For example, better security implies poorer usability, and better performance is balanced against tighter component coupling and thus worse maintainability. Possible systems can therefore be placed in a design space that is more complex than the lifecycle cube, where NFR are at opposing ends of the same axes.

Architectural models

Software models, like those of other engineering disciplines, evaluate planned products to reduce the risk of them failing to satisfy users' needs. The earlier in the lifecycle that problems are identified, the easier they are to resolve, reducing the cost of errors. Models are especially valuable in innovative domains, like data-grids, which lack established engineering solutions or even clear understanding of user needs.

Models' abstract representations simplify reality's complexity, permitting systematic evaluation of the essential properties captured in the chosen view. They include purely cognitive descriptions and incomplete constructions. 5 classes of software engineering models, referred to in further discussion, are identified here:

1. *informal* descriptions, including sketched system diagrams,
2. *formal* representations in mathematical grammars that support analytic proof,
3. designs with *formal diagrams*, exemplified by object-oriented descriptions,
4. system *prototypes* – incomplete implementations of the real system,
5. *simulations,* which implement a model of the designed system and its environment.

Of these, this research captures and analyses architecture with models of class 2 (Sections 4.2 and 5.1), relating them to models of class 1, 3 and 4. The simulation of data-grid designs, in class 5, is described (in Section 6.2). Whilst architecture must capture the overarching principles guiding system development, the models' accuracy and value depends on their fidelity to requirements and traceability to detailed design. They must also be communicable to other stakeholders if their conclusions are not to be lost. The investigation's data-grid models are assessed against these criteria in Section 7.1.

Architectural styles

Diverse systems that face similar challenges share architectural features [1,94]. Common relations at the resolution of component interfaces are similar to lower level design patterns (typically described in an object-oriented way) [49]. Though both are abstractly represented, they are only valuable if they can be applied to concrete applications. Styles and patterns are then recognised as communicable directions for repeatable success.

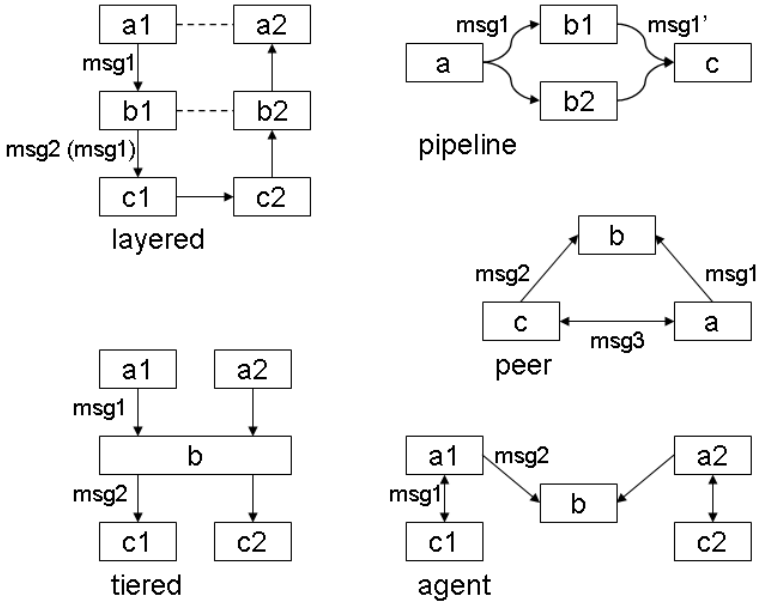

Figure 4: Five distributed system styles, typified by their components' communication relationships; all apply to data-grids.

5 examples of styles, derived from distributed information systems architectures, are listed below (with examples) and represented in Figure 4. Whilst data-grids are still an emergent domain, distributed systems' styles can guide for projects like EGSO. Their suitability is rigorously evaluated in Section 4.1; here their key features and component communication styles (which provide criteria for unambiguous classification) are given.

- *Layered architecture*

A system may be simplified by dividing it into layers with interfaces. Each layer has unique responsibilities, and distributed instances have a direct virtual communication path. In this way, programs at one layer can ignore issues handled in other layers, simply relying on their service. At the highest layer, the application may use an API without coupling to its implementation, whilst at the lowest layer the physical operation may be implemented mechanically, ignoring the variety of use and design subtleties at higher levels.

The layers translate the logical content of data and control messages (information and commands) to diverse representations. Enterprise databases (integrating the heterogeneous schemas of distributed repositories) and high level programming languages (supported by compilers and virtual machines) are examples of layered architectures.

- *n-tier architecture*

Business logic (functionality associated with a user's needs) may be separated from process logic (technical solutions for classes of application) using tiers. This architecture allows transparency and flexibility between the front end user driven behaviour and back end system administration [35]. Transparency means that different, distributed resources may be used homogeneously, allowing the redundancy and growth that supports reliability and scalability. Flexibility means that components may be changed without affecting the rest of the system, or easily composed in novel ways. Functionality is provided by components via platform independent interfaces. The middleware that enables tier abstraction typically provides minimal basic services via core component interfaces. Specific systems may implement and reuse components within this framework to build their functionality.

The interaction about a tier is independent but connected, so that messages used by the application have a many-to-many relationship with messages using back end resources. The Common Object Resource Broker Architecture (CORBA) and Java 2 Enterprise Edition provide component-based middleware for diverse distributed systems in conceptual tiers. Generic interfaces define web services that are hosted on application servers such as IBM's Websphere [146].

- *Peer-to-peer architecture*

Peer-to-peer nodes have symmetric relationships, for example functioning both as client and server when creating and performing service requests [80]. In a peer-to-peer network, a large number of nodes may communicate and share resources without knowing details of the whole network or dependence on central points of control.

Communication sessions in peer-to-peer networks are typically a triangular sequence of advertisement, requests for service (forwarded until a match is made), then direct interaction between the advertiser and the requester. Internet Protocol (IP) networks have peer-to-peer

characteristics, though file-sharing services such as Gnutella are the paradigm example of this architecture. JXTA is a flexible middleware for peer-to-peer resource sharing.

- *Data-flow pipeline architecture*

Processing components may be organised in sequence, so that the output of one forms the input of the next. Branching is possible, allowing concurrent progress, but requires later synchronisation if paths rejoin. Different scheduling strategies may be used to suit the functional requirements, and may require some intelligence to make the best use of resources.

The messages between components may belong to one job, but have different content as each component transforms its input according to its function. A pipeline of processes or filters (such as in a Unix shell script) is an example of data-flow architecture, and many parallel computing tasks (such as finite element simulation) run in a data-flow sequence.

- *Blackboard agent-based architecture*

Complicated tasks can be tackled by dividing work amongst software agents that use a shared 'blackboard' data area. This architecture may solve a problem by applying a variety of analytic or heuristic methods to one data set, or pool information about the content or relations of distributed data sets. Agents may run on distributed resources, so this architecture is also a concurrent solution.

Agents avoid passing messages between each other by only accessing the blackboard. Their messages are of a similar type with different content, and the blackboard is a central critical resource. Artificial intelligence and data mining applications, including WWW catalogues generated by Internet bots, use this architecture for information processing.

2.3 Style imposed by grid tools

The architectural styles described in Section 2.2 are established for distributed systems. Systems that federate computing resources in grids can be built with emerging infrastructural tools. This section describes how the most widely used of these work, assessing their fit to distributed system architectural styles. Where they do not rigorously follow proven styles, applications implemented using them are at risk; where good engineering principles are encouraged, their use should consistently uphold quality. This review also shows how general grid solutions do not solve data-grids' specific needs (discussed for project instances in Chapter 3), and therefore why further architectural analysis is valuable.

Globus

The Globus project intends to identify key grid services and define the protocols at their interfaces [40]. It identifies and includes the required grid infrastructure features:

- Discovering, sharing and maintaining information on distributed resources.
- Controlling the use of known resources by brokering service requests.
- Mapping services to heterogeneous platforms (taking advantage of resource specialities rather than reducing all to basic common services).
- Support quality of service, for example with data replication and task migration.

- Providing authorisation proxy services (allowing single user sign-on) for resource security.
- Enabling scalability (both for usage growth and functional extension) by a wholly distributed architecture.

Globus follow the pattern of the Transaction Control Protocol (TCP) by defining the narrowest possible protocol to serve the maximum variety of higher-level applications, using the broadest mixture of lower level networks and devices (the 'hourglass' stack). The team has transferred the Internet's application of the OSI communication stack to the grid by adapting and adding layers. Applying this layered architectural model provides clear partitioning of functionality and simplifies application implementation [44].

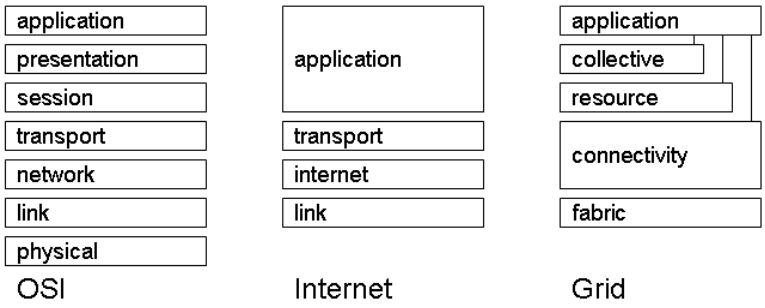

Figure 5: Globus layered model of grid components matched against an interpretation of the Internet stack and the OSI model [44].

As shown in Figure 5, the layering is not strict, and the interpreted Internet stack is not exactly matched to the original OSI model. Though the layers communicate with each other in sequence, the application layer may link directly to three lower layers. From the top layer (immediately below the application) down they are:

- *Collective*, providing grid infrastructure information such as resource catalogues, certificate authorities, replication status and distributed scheduling.
- *Resource*, providing access to the grid resources (and monitoring their performance).
- *Connectivity*, (overlapping with TCP/IP in the transport and network layers) providing grid message passing (including authentication).
- *Fabric* is the collective of other underlying layers (that may include link and physical layers) and the fundamental resources instances (with specific operating systems and control interfaces).

As part of the protocol definition, APIs are defined. Globus do not expect their protocols to be used by grid application developers directly. Instead components that provide grid services by wrapping the APIs should be provided. Globus provide open source libraries for such components (in Globus Toolkit 2), whilst encouraging development of alternative implementations. 4 specific families of grid services are defined, illustrating the capabilities that tool developers should provide and application writers can expect:

- *Grid Security Infrastructure* provides certificate based delegated user authorisation and access restriction policy enforcement. It incorporates X.509 certificates, Kerberos tickets, and the Secure Socket Layer protocol.
- *Grid Resource Application Management* (GRAM) provides remote access to heterogeneous platforms for applications based on a Resource Specification Language [28].
- *Grid FTP* provides high performance data management, based on logical addresses mapped to physical locations.
- *Metacomputing Directory Service* provides resource discovery and live status information in distributed catalogue. (MDS is composed of the Grid Resource Information Service and collective Grid Index Information Service. It uses the Open Lightweight Directory Access Protocol.)

Components further simplify application development by taking responsibility for providing generic services, allowing transparent use of grid resources. For example, the resource location, platform gateway, and network error recovery methods may all be hidden from the application.

In general, a toolkit of standard components may also encourage developers to apply design patterns that suit the grid environment well. Applications may then be built rapidly and economically using the components; this software engineering goal has been carried forward from middleware, object-oriented and previously module production. Normally components are defined by their interface and the services they provide. However, to be applied successfully their behaviour and the characteristics of their connectors must also be captured when designing composite systems (to ensure both functional and NFR will be met). This is a challenge to all component-based systems that is being addressed by software engineering techniques such as architecture description languages (described in section 4.2).

The Globus project has generated a broad suite of grid tools in their APIs, implementations and components. Their success at providing a narrow, open protocol is proven by the broad variety of applications now using Globus and the variety of fabric layer systems incorporated. However, the initial goals of clearly defined layers and the value of components hiding grid interfaces from the application developer are at risk due to the complexity and coupling of the parts of Globus. Closer binding of the toolkit to layered and component based architecture models, possibly formally expressed, should help Globus users to deploy reliable, well-suited systems.

The Globus project also defines the Open Grid Systems Architecture (OGSA) standard, which provides a mechanism for distributing grid components as web services [41,105]. It builds on the concept of virtual organisations, providing the business-to-business interface. By incorporating the Web Service Definition Language (WSDL) and providing service ports via protocols such as Simple Object Access Protocol (SOAP), Globus services should be able to wrap and interface with existing and emerging shared application methods (from Dynamically Linked Libraries to Enterprise Java Beans). The Globus services maintain a 'soft state', helping autonomous activation, aiding quality of service. Globus Toolkit 3, which implements OGSA, therefore merges with the Web Services solutions described below.

Grid-enabled Condor

Condor manages the exploitation of idle workstations' unused computing capacity (and its functionality can be wrapped as OGSA services [21]). Its central server can monitor the availability of several hundred computers on a local network. It then distributes clients' tasks submitted to its queue to any free resources, creating a high throughput distributed computing cluster simply. Note that the clients are responsible for managing any parallelisation. This means that Condor suits embarrassingly parallel problems, which have many instances of the same process with different parameters, rather than tightly coupled activities with data staging and process synchronisation issues.

Scaling Condor up to a true computational grid would require integration of distributed computing clusters, removing the central point of failure. Complex jobs on dedicated high performance computers should be able to interoperate with cheaper resources managed by Condor. Such resource diversity would mean heterogeneous protocols must transparently interoperate. As the network connecting resources is more unreliable than a LAN, automatic resource discovery and fault recovery must also be provided. Additionally, resource security means that local restricted access policies must be upheld, whilst users' authorisation must be globally accepted.

The Globus team, in making Condor-G, has met these requirements. They leave the original Condor local system management unchanged [48] so that Globus recognises whole clusters as single nodes in the greater distributed system. In this architecture, different resolutions of resource information and task distribution services are provided by Condor and Globus' GRAM component. Condor-G applications therefore validate the Globus model by their simple construction from grid components with both local and wide area levels of resource distribution, though 2 different resource management solutions are used.

Legion

The Legion project goals are to provide a simple, flexible and scalable grid middleware that allows site autonomy using meta-objects and reliability using object recovery [56]. Its infrastructure aggregates high performance distributed heterogeneous resources, but unlike Globus, its architecture is fundamentally object-oriented [51]. It represents grid elements as objects to encapsulate their state and meet specific grid challenges:

- The signatures of objects' methods are decoupled from their implementation, allowing core interface to be reused through inherited class definitions and polymorphic object instances.
- Simple object references can help information discovery and be reused in a dynamic environment.
- Objects may represent mobile agents operating without central control.
- A general mechanism for managing the inevitable errors of the unreliable network can be built by catching thrown exceptions.

Meta-object class interfaces control the resource objects, presenting a generic facade that enables platform transparency and other benefits. For example, to enforce security the

'may I' meta-object method is invoked for an object class before any instances may be created or used. Strong and flexible security is therefore simply achieved without centralised control. Objects on diverse platforms interact via a shared Legion interface definition language (IDL).

Legion defines the core middleware objects required to support distributed operations, including: hosts, vaults (persistent stores for recoverable object states), implementations (that wrap underlying legacy executables) and binding agents. Binding agent objects enable resource use, and illustrate how Legion provides transparency. An application object calls the service it requires using a context name (knowing nothing of the object that will eventually run the required operation). The binding agent associates the service context name with a location object identifier (still just an abstract tag), which is then associated to a location object address using the binding agent's internal dynamic tables of active resource objects. The binding agent then invokes the resource indirectly using its class interface, allowing local management of service instances.

Beyond defining an open standard and reference implementation for a grid infrastructure, Legion's object-oriented architecture achieves specific goals:

- Scalability and portability are ensured by the distributed modular design and general IDL.
- Site security and resource management autonomy is provided by meta-objects.
- Reliability is provided by object recovery from vaults.

Despite achieving its goals, Legion has not matched Globus' uptake. This may be because proven object-oriented distributed middleware solutions – specifically CORBA and the Java family – are now tackling the grid domain too.

CORBA adaptation to grids

CORBA, defined by the Object Modelling Group (OMG), is a widely used middleware standard for distributed systems [35]. Its object-oriented principles comply with Unified Modelling Language methodology (UML), also defined by the OMG. Its object-oriented architecture is also aligned with the higher layers of the OSI stack; activation and marshalling provide session and presentation layer operations between distributed applications. By resolving the needs of distributed systems, CORBA provides a solution for some of the requirements for a grid system.

A CORBA application invokes operations without knowing whether they are locally hosted or implemented on a remote server. An Object Request Broker (ORB) communicates between the application's host and the remote server, providing location transparency in the same way as Legion. Object class interfaces are specified in a standard IDL, and object instances' members are marshalled to enable interoperability. Operations may be invoked on inactive objects, which the server activates from storage, to enable local performance management by dynamically freeing unused resources. Server objects may also be transparently migrated, being activated on a standby machine when an error is trapped, for example. In this case the application is not aware of its task's redirection, and high reliability is presented.

The described mechanisms for access, location, platform, activation, migration and failure transparencies within CORBA seem well suited for supporting grid systems. CORBA's penetration means many existing heterogeneous distributed systems and software could be rapidly enabled for the grid, entering with high qualities of service.

However, CORBA only indirectly supports other basic grid capabilities. For example, security is typically implemented at the servers, which demand application authorisation. In contrast, delegated certification provides single sign on for authorisation transparency across arbitrary servers in grid middleware. Also, though CORBA domains may communicate with each other via the Internet Inter-ORB Protocol (sharing object addresses and interfaces), grid middleware's decentralised resource discovery is not a basic architectural feature of CORBA.

CORBA has been used to directly implement grid projects [93,108]. A distributed engineering simulation model implemented under CORBA has also been wrapped by Globus interfaces and successfully scaled up to multiple sites [71]. In this project CORBA was treated as an application layer protocol, hiding its distributed service capabilities. That means the transparency offered by CORBA was lost when equivalent Globus service location information was exposed to the application. Therefore, though it was demonstrated that CORBA could work with the Globus grid middleware, there is still scope for adapting CORBA as a grid middleware itself.

Web Services and the Service Oriented Architecture

The Service Oriented Architecture (SOA), implemented by Web Services standards, has emerged as a more popular model for application integration than CORBA [4]. It treats all entities on the network as services, including resources, applications and aggregations of other services. Service interfaces are described and communicate in platform-independent ways (typically with WSDL interface specification and SOAP messaging for Web Services. WSDL and SOAP use Extended Mark-up Language (XML) self-describing structure). SOA complies with peer architecture, as defined above, permitting services to be dynamically discovered (in Web Services, typically via a UDDI registry).

SOA is popular in industry both for legacy application integration (wrapped in Web Service interfaces) and for component oriented product line development (often using Java, with its strong distributed system, Internet and XML library support). Its emphasis on inter-domain operation and dynamic discovery also makes it more suitable for grid applications than CORBA. EGSO and AstroGrid are both examples of data-grids that use Web Services.

However, basic SOAP interaction assumes stateless interaction, which is unsuitable for coordinating complex grid tasks. Transaction and conversation standards are emerging for business needs, whilst OGSA makes distributed state explicit to work around this limitation. In general, SOA is an emergent domain with unstable Web Services standards, putting long-term maintainability at risk. All major enterprise system vendors are promoting standards for complex capabilities (over the established 3 core standards: WSDL interface presentation, SOAP atomic message exchange and UDDI discovery); their competing and interdependent solutions run against the simple narrow protocol principle of Globus. Therefore, though SOA is more suitable

for grid infrastructure than CORBA, caution is needed in Web Services implementation beyond the basic standards.

Merging Unicore and Web Services

Unicore is another dedicated grid middleware standard, less widely used than OGSA and Legion. Its compatibility to OGSA and the Globus toolkit has been demonstrated [95]. It is also compatible with Web Services, supporting SOAP client interaction. However, this work noted that the provision for service aggregation, dynamic data mapping, assertion and security key authentication in WSDL made the equivalent grid capabilities unnecessary. As Unicore restricts service data addressing, prevents asynchronous subscription to pushed data, and limits security binding, it is less suitable than OGSA for engineering grid capabilities with the flexibility and interoperability of Web Services.

Java application networks

Sun Microsystems' platform independent Java seems well suited for implementing grid applications. Its interpreted code does not need an additional IDL for platform transparency. The constrained relation between applets (for client functionality) and servlets (additional web server functionality) alone does not guarantee sufficient security for flexible authorisation policies. However, grid applications may be built with beans (service component containers); Enterprise Java Beans (EJB) implements a tiered architecture. Java's scalability is supported by packages' globally unique names, and its Remote Method Invocation (RMI) protocol supports distributed systems (protocol bridges have also been implemented [128])

Sun also provide JINI for resource federation in flexible distributed systems [130]. It supports migration transparency by moving Java objects (rather than binding to dynamic service locations, as CORBA does). Network platform independence is achieved by hiding object interaction behind its interface; the interaction may be implemented in CORBA, SOAP or Java RMI. JINI has scope for distributed control, resource reservation, intelligent scheduling, transactions and authorisation by security proxies. However, the protocol does rely on the availability of Java Virtual Machines on heterogeneous platforms (unlike Web Services, which may be implemented in any language). Java itself also needs high performance capabilities to become suitable for the computationally hard applications that Globus and Legion are used for [64].

Sun have also defined the JXTA protocol for transparent peer-to-peer interaction across traditional domain boundaries (via relay peers). Discovery is enabled by resource indexes held at centralised rendezvous peers, whilst the protocols accommodate indexed resources disappearing. Though the exemplar application given is of personal computers accessing a remote printer, JXTA could be used to administer dynamic resource access in grids' virtual organisations; this solution was considered for EGSO.

In conclusion, the available grid tools do follow software engineering principles up to a point. Globus applies a weak layered architecture, whilst also defining a service-oriented tier. Condor's parallel architecture management can be exposed as a grid resource, though it is not

a high-performance solution. Legion uses strong object-oriented principles to implement a transparent tiered solution, but the stronger, established CORBA can also meet grid requirements. Though they are not intended for grid problems, grid-scale applications may also be built with Web Services and Java technologies (through EJB tiers, JINI distributed management or with JXTA peers).

Chapter 2 key points

- To avoid software crisis, projects of the scale and originality of data-grid should follow modern techniques, such as iterative development with customer involvement, to guarantee development quality.
- Good choice of a system's architectural style, specified early in the project lifecycle, reduces the risk of costly failure to meet NFR.
- The review of 5 distributed systems architectures, with current grid tools' engineering principles, guides the fitting of design styles to data-grid projects (described in Section 4.1, after the review of their requirements in Chapter 3).

Following these conclusions about grid projects' need for software engineering, the research described in subsequent chapters supports the specific EGSO and AstroGrid data-grid projects. The application of the innovative architectural description and event modelling techniques drives down the risk of behavioural quality failures with iterative modelling. Analysis validated designs early in the projects' lifecycles, sufficiently rapidly and clearly to positively influence implementation.

Chapter 3 Solar physics data-grid requirements

This chapter describes how the data-grid vision (of Section 1.2) is applied in concrete projects. If the domain review of Chapter 2 was of particular interest to e-scientists who were unfamiliar with software engineering and grid tools, then this chapter is suited for software engineers who are interested in e-science applications. The review focuses on solar physics to illustrate the broader data-grid domain. It may be read as a narrative, flowing from a broad examination of e-science projects to increasingly detailed analysis of solar physicists data-grid needs.

Section 3.1 – a domain review of e-science projects across the sciences – shows that data-grid architectural engineering is immature; some relevant styles are overlooked, whilst the informal application of others limits their use. The concluding Section 3.4 describes requirements these data-grids share. Section 3.2 describes the general use case analysis of solar physics data-grids, in preparation for the modelling investigation. Section 3.3 details the techniques that were used in analysing EGSO's specific requirements. The tour of solar physics data-grid needs, through diverse views, introduces the challenges for the subsequent modelling investigation, but also serves as a record of requirements' analysis techniques' case studies. Section 3.4 discusses the spectrum of their application to EGSO and AstroGrid, assessing the documented requirements' relative value.

3.1 Data-grids by example

When the data-grid modelling research started, there were only informal taxonomies of grids. The seminal textbook identified 4 classes of grids [43]: high performance computing, data intensive, instrumentation oriented, and collaborative virtual environments. Later, computational, data and service grids were further decomposed around resource management strategies [65] (then updated as the use of service grids increased [107]). Another 3-way classification scheme (looking toward future global capabilities) distinguished information, resource and service grids [91] (where the resources accessed by end-users are either information content exemplified by web-pages, computing and storage resources, or service functionality typically described by WSDL). A survey of projects at the emergence of OGSA notes the convergence agent technology with grid and service oriented architecture [30]. Grids have also been formally distinguished from other distributed systems by their abstract resource representation and discovery [102]. Though supported by association to Legion and Globus, this definition makes grids appear very similar to service oriented architecture, including web services.

Before these later publications, existing projects, which seemed to share requirements with envisioned solar physics data-grids at the level of their abstract application functionality, were reviewed. The EGSO team subsequently carried out a domain review specifically for the project [25]. That led to joint publication of generic data-grids' typifying requirements [37], with the novel method for fitting architectural styles (described in Section 4.1). The projects investigated are grouped below by their demonstrated architectural styles.

Layered grid projects

Layered architectures are fundamental to distributed information systems, and therefore all data-grids. Communication protocols rely on the OSI stack, and information presentation relies on database and file system abstraction with data storage applications. However, this style is rarely explicitly exposed in data-grid projects.

The European Data Grid is being constructed to analyse the data that is to be generated by the Large Hadron Collider at CERN [85] (as well as supporting biological and earth observation grid applications). It is already in use, distributing simulated results of planned high-energy physics experiments. Its middleware is described as layered [18], and it uses the layered grid protocol model of Globus. The 4 service layers – low-level physical fabric, core middleware, data-grid services and domain specific high-level applications – are also used for project management, bounding the scope of 10 work packages.

Specific aspects of the Data Grid project's very high volume data management design take the layered analogy further. Raw experimental data is divided and summarised as it passed from CERN's central data store to regional centres and then on to investigating institutions' servers [59]. Each geographic layer also has compute resources attached to the stores, and the registries of metadata about the data fragments that manage replication have a layered hierarchical organisation [23].

However, the 4 service layers and the data levels (of deployed storage resources and designed metadata management) are closer to tiered architecture. The higher-level abstract service capabilities rely on decoupled hidden infrastructural functions. Also, the low-level data resources are not literally available at scientists' distributed access points; replication provides location transparency by separating logical data references from the stores. As layered architecture is defined (in Section 2.2), the complete content of high-level information should be reproduced at the lower levels (albeit in different encoding structures).

The Data Grid project's architecture is necessary to divide development responsibility and achieve reasonable throughput of the multi-petabyte data load (a petabyte is 2^{40} bytes, over a million MB). It can therefore be seen more as a business model for project management than a software engineering solution. Though it does also function as a high-level software architecture, it is not a rigorously developed engineering technical model. Therefore, it should not be applied to smaller-scale projects.

n-tier grid projects

Most data-grid architectures (including all those mentioned here against other styles) explicitly or implicitly use a tiered architecture. Data-grids need the transparency that the style enables to allow client interoperability across heterogeneous platforms and data models, at distributed locations and on diverse networks.

The power of layered architectures is demonstrated by bioinformatics, which has rich and complex information resources (from chemical descriptions to experimental results), and a large suite of analysis tools [125]. However, the full decoupling of applications from back-end resources is not necessarily achieved, as scientists use scripting languages (notably, Python [103]) to inflexibly compose workflows and implement data analysis code. Such apparently low

quality software engineering is justified if this is just enough rapid development to quickly solve an immediate problem. The scientists' programming skills, which average business system users lack, then mean the scripts are their application interface, which can hide high performance legacy code at the back-end.

The tier style is also under stress in the US DOE Science Grid SciDAC architecture [61]. This places a 'problem solving environment' over the applications layer, which itself has a layered structure – technology specific programming interfaces (including CORBA, Jini and Globus Message Passing Interface) share grid-enabled libraries. Then, below the true middle-tier of common grid services and uniform resource interfaces it separates distributed resources services (including scheduling and network management), from underlying communications services. The proliferation of layers, their internal complexity and the overlapping scope of their components' roles weaken the architecture's ability to simplify integration.

Peer-to-peer grid projects

Grid computing apparently has similar scope to peer-to-peer networks, both enabling discovery and exploitation of heterogeneous resources on an Internet scale. However, there were initially few examples of projects adopting the peer-to-peer architecture. 2 examples were developed into full projects:

- A component oriented model (with Web Service event interaction, applied to chemical engineering and orbital physics analysis) supports peer-to-peer architecture [50]. This work draws attention to the distinction between networks that share information in files versus those that share computing resources.
- Commodity computing systems, which exploit a great number of resources economically, go beyond the scale constraints of centralised Condor by using a peer-to-peer architecture [47]. The SETI programme demonstrates the power of commodity computing, exploiting 1200 years of CPU time daily [33]. Though SETI is an inflexible application with centralised architecture, flexible solutions are emerging [3].

Peer networks are especially suited to decentralised data-grids, which have similar requirements to file-sharing networks. As those that take responsibility for publishing information may be unwilling to also administer the infrastructure, users must be able to access resources without relying on central services. The EGSO architecture uses a hybrid tiered and peer-to-peer architecture to lower the barrier of entry to providers [82].

Some emerging peer-to-peer networks guarantee publisher anonymity, and this should be useful in specific data-grid domains. The CLEF project makes anonymous medical records more widely available for clinical studies, using services to access back-end databases in a tiered architecture (to permit user management) [90].

Data-flow pipeline grid projects

The pipeline architecture is primarily associated to grid technology through computationally demanding applications [78]. It is used to increase the efficiency of data distribution in data-grid applications that support access to large volumes of data. The parallel

channels of the GridFTP standard (implemented in Globus and applied in Earth System Grid [42]) provide only limited support for grid enabling databases [111]. However, emerging standards and toolkits for grid database access (OGSA DAI [5] and Spitfire [8]) emphasise the service-oriented convenience of tier architecture.

Grid applications that combine data access and analysis can express pipeline architecture. It is feasible to run physical science experiments on computer models, and these are more powerful when deployed on grids. Records of previously observed physical properties and models must support such experiments, and rich visualisation helps to understand their results. Whilst a service-oriented middle tier helps applications coordinate access to these diverse back-end functions, the workflows that enable efficient use of the 3 types of resource are pipelines. This hybrid architecture is seen in engineering and chemistry grid applications [20] (in which large Condor clusters can be used for analysis [21]).

Blackboard agent-based grid projects

Agent technology seems well suited to data-grid technology, as it provides an architecture for parallel independent information extraction from distributed resources. For example, agents could automate the population of the catalogues that EGSO requires to direct users' searches, linking the data of observations to listed events and other objects of interest. Structures for organising information already exist in the domain, as the SolarSoft tree of instruments [10] and observation array dimensions (notably: time, solar coordinates and wavelength – the axes of the middle level in Figure 10, described later). Agents could coordinate their activities by covering different parts of these structures. There is also scope for agents in automated operation of the image processing functionality implemented in EGSO [113].

Agent technology is a part of the myGrid project's architecture [54], applying the observed convergence of grid and web services with agents [72]. The bioinformatics domain to which it applies includes semi-structured text records of experimental findings, in which natural language processing agents can identify key terms. This permits objects of interest (such as proteins, genes and metabolic models) to be linked and catalogued to support data-grid users. Even in this application, though, agents only form part of the architecture; myGrid follows a tiered architecture in its application of web services to support core infrastructure functions.

This survey of the state of software engineering application in data-grids contributed to the technology survey for EGSO and, in turn, publication of generic data-grid requirements [37]. Though that work tabulated projects (and the grid tools described in Section 2.3) against architectural styles, such a summary hides the weaknesses in the styles' applications noted above. In summary:

- Layers are not employed as a pure software engineering solution to organise Data Grid services and replicas, so they lack independent representation.
- Tiered architecture's capacity to hide complex heterogeneity is weakened by application scripts in bioinformatics and infrastructural complexity in SciDAC.

- The full scope of peer decentralisation is not yet met by data-grids (in prototype demonstrators, computing task distribution and EGSO's hybrid middle-tier).
- Though data-grids employ pipeline architecture for clearly distinct functional and data-flow solutions, the style is more important to high performance computing.
- Agents are not yet as widely used in data-grid applications as they could be, as they must rely on underlying infrastructure that is still in development.

3.2 Solar physics data-grid use cases

Solar physics should benefit from e-science methods (as introduced in Section 1.2). A data-grid would enable easier access to the diverse heterogeneous data resources with catalogues and transparent analysis. It would support science by enabling collaboration and facilitating the discovery of any serendipitous observations that support hypotheses.

Before the EGSO and AstroGrid projects started, the data-grid needs of the scientific community were analysed through use case analysis. Input to this study included: recent findings described in text books and collections [81,67,112] and presented at the UK Solar Physics conference (from which the second of the specific scientific use cases described below was derived), and conversations with scientists at the UCL Mullard Space Science Laboratory (MSSL) about their ways of working.

Generating requirements for novel systems through use cases is part of the unified process associated with UML [12], and advocated by other object-oriented methods [76,84]. The output should therefore be readily understood by software engineers, and be traceable to later software design, ensuring development intersects the end users' needs well. The process should not prejudice the design to an object-oriented solution; the high level design stage following use case specification would be abstract (not specifying classes and methods directly).

3.2.1 Adapting Internet resources

It immediately became apparent that solar physics and astronomy data-grid needs could only be understood with reference to existing ways of working with Internet resources. Scientists were already using distributed resources: accessing and correlating diverse archives of observations to support their investigations, and cooperating by sharing derived information and analysis tools.

Specific examples of widely accessed archives are SDAC for solar physics [143] and CDS for astronomy [121]. Data from instruments on spacecrafts – notably Yohkoh [149], SOHO [142], TRACE [148]and RHESSI [139] in solar physics – are also naturally shared via the Internet; they are electronically received, and missions are founded on international collaboration. Scientists using such data still value terrestrial observatories, and those that provide their data online are well used; Big Bear [120], Kitt Peak [131], Learmonth (hosted at another archive, [134]) and Meudon [132] are amongst those that provide solar observations. The Web is also a good forum for publishing event lists and other information derived from

observations; lists of solar flares are compiled from the GOES satellites [127], and coronal mass ejections (CME) observed by the SOHO LASCO instrument are listed, though less systematically [141].

Other current examples of scientific Internet exploitation have partial overlap with data-grid needs. SolarSoft [10] and Starlink [144] are collaboratively developed toolkits for solar and stellar analysis respectively (SolarSoft also packages metadata to support interpretation of observations, anticipating EGSO capability). As users typically use the software packages at their local institutions (rather than remotely executing functionality), only the communal development and coordinated distribution of up to date software versions address challenges for ways of working with data-grids. Networked instrumentation, such as GONG [126] for helioseismology and Merlin (and the Very Large Baseline Array) [133] for radio astronomy, can also be classed as a special kind of grid (though instrumentation grids emphasis real-time access to experiments and integration with analysis [62,110], not priorities in these cases).

10 use cases, shown on Figure 6 (and described completely in Appendix A), were abstracted from the current solar-physics activities that are relevant to data-grid use. Central to these is researchers' use of on-line solar observations, integrated with use of other types of data. Data-grid capabilities must build on this.

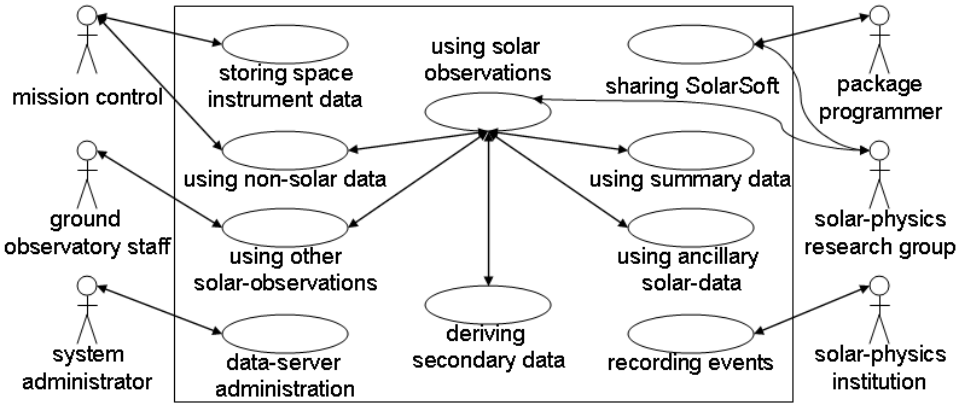

Figure 6: Current solar physics Internet use cases.

The 6 data-grid use cases built on current on-line solar data use (also in Appendix A) are shown in Figure 7. Package programmers take on additional responsibility for generating information about the data records, as they automate the identification of events and features that are currently manually catalogued (they should have the necessary scientific understanding, as these actors are in fact typically a subset of the solar researchers). Administrators can take additional actions to increase the availability of their data (for example, observatory administrators may expose interfaces to collections of non-standard, poorly catalogued observations, whilst network administrators can permit the distribution of useful data via caching servers and the execution of scientific users' tasks that are part of a distributed workflow). But, in line with the vision for data-grids (described in Section 1.2), the key use case involves the scientists, who gain better access to the data and remote analysis capabilities (for example, making use of remote pre-analysis capabilities to download only the derived data required, or exploiting a one-stop-shop interface that maps data requests onto diverse back-end

storage solutions and access protocols). Satisfaction of these use cases together would improve the research process by avoiding wasteful and duplicated search, download and analysis effort.

Solar physicists, including future EGSO stakeholders, reviewed the use cases. In this way, their expectations of data-grid capabilities were clarified, allowing the users and developers to refine their goals. The scope of future project activity was also clarified by documenting more speculative information network applications alongside the use cases. It has been noted that projects can fail when requirements are revised late in the project (Section 2.1), as a result of users' or developers' incomplete understanding of a novel problem domain. A general conclusion drawn from this instance of use case analysis would be that it drives down this risk.

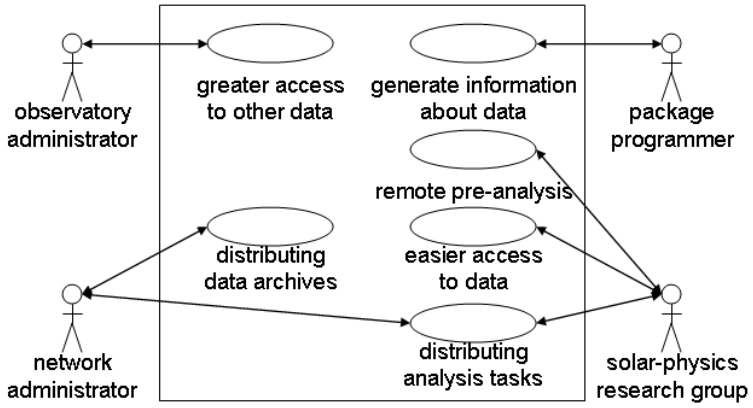

Figure 7: Anticipated future solar data-grid use cases (in addition to current uses).

3.2.2 Specific scientific use cases

2 concrete scientific use cases were supplied to the EGSO and AstroGrid projects as specific instances of the abstract data-grid requirements described above (both being instances of easier access to data, in basic and advanced ways). The actor in both cases is the scientific researcher. The first, studying coronal waves, was intended to be simple enough for a research student to carry out using EGSO. The second uses a specialist subject at the boundary of solar physics and Astronomy; if AstroGrid were capable of this, it would demonstrate its flexibility and suitability for both domains.

Coronal waves

Coronal waves are rare large-scale disturbances high in the solar atmosphere that have only recently been observed [58,104]. They are captured in continuous observation of the corona from spacecrafts' instruments: the extreme ultra-violet images of TRACE and SOHO EIT, and the Yohkoh SXT soft X-ray observations. They appear as changes in brightness, clearly noticed in difference images (highlighting the changes between two images obtained at different times), which also show nearby loops of plasma swaying. The waves propagate rapidly away from flare sites, and are thought to be impulsive waves caused by coronal field lines reconfiguring. As magnetic field reconfiguration can open field lines, coronal waves may also be

associated with the release of prominences and therefore CME. They may also be related to the faster Moreton waves, which move away from flare sites lower in the solar atmosphere (the chromosphere, visible from the ground). Figure 8 shows a hypothetical series of events.

If coronal wave events are identified, a data-grid should make it easy to find other observations at the same time, so that correlation with Moreton waves or CME can be demonstrated. Analysis procedures would streamline the generation of difference images, and image recognition techniques could even automatically locate features in these images, in order to identify unknown observations of coronal waves in 'sit and stare' observation data. If the data-grid were also integrated with electrohydrodynamic coronal modelling, theories that explain their occurrence could be more easily evaluated (this would be an advanced capability, beyond the initial plans for EGSO).

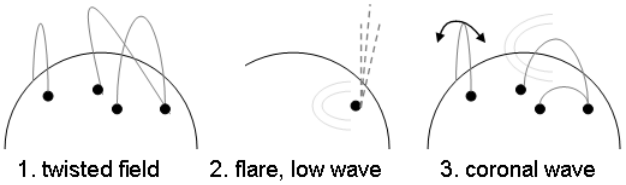

Figure 8: Hypothetical sequence of events leading to coronal waves – twisted field lines reconnect, causing a flare and Moreton waves, then coronal shock waves as field-lines reconfigure, shaking other coronal flux loops.

Magnetic clouds' radio scintillation

Scintillation is a familiar phenomenon, causing stars to twinkle as their light is bent by changes in the air's density. Ground-based radio telescopes observing distant objects, such as galaxies, detect scintillation caused by magnetic clouds moving away from the sun. These are denser than the continuous fast solar wind (which forms the heliosphere, with its own spiral structure); they arise from CME events (rather than ubiquitous open field lines). Figure 9 illustrates these circumstances.

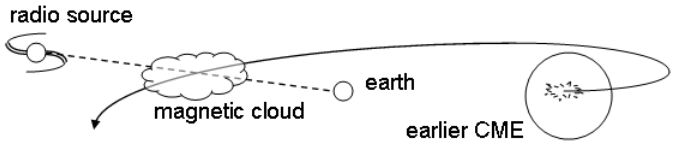

Figure 9: Circumstances required for magnetic cloud observation by radio scintillation – CME moves away from the sun (along the spiral magnetic field of the heliosphere), passing between the earth and a distant radio source (for example, a galaxy).

Magnetic clouds have been systematically observed by coordinating radio telescope observations with space weather monitors [36]. Analysing these data to produce maps and associating them with coronograph images gives new insight into the evolution of CMEs into very large clouds. Radio scintillation observations could also be correlated with *in-situ* particle detection and geomagnetic effects when the clouds pass the earth.

A data-grid that integrates solar and astronomical data could help scientists find lucky observations of radio scintillation. It should also help them associate planned scintillation observations with the earlier coronal observations. CME catalogues would be especially useful, and these would themselves be improved by magnetic categorisation of events. By using the data-grid for scintillation studies, scientists could therefore integrate their models of the sun, heliosphere and solar-terrestrial interaction.

Both these scientific use cases had only limited effect on the projects. Though the EGSO and AstroGrid scientific use case documentation did not directly include either (short-list of use cases chosen for detailed requirements specification in EGSO and demonstration iterations in AstroGrid being specified by experienced scientists), project scientists and software engineers in discussion recognised their validity. A version of coronal wave investigation was in both (emphasising their discovery in AstroGrid [117], and correlating them to CME and flares' radio classification in EGSO [88]). The AstroGrid registry team also adopted the association of CME to active regions (part of the scintillation study) as a technical use case to drive demonstrator development. These scientific use cases can therefore be taken as examples of deployed data-grid activity – concrete instances of the generic 'better data access' use case. They also represent instances of the query resolution functionality described in a generic way for the abstract models of data-grid designs in following chapters. It does not matter that these scientific use cases were not taken up by the projects, as their scientific content does not affect the technical specification or derived designs that are evaluated in the research.

3.2.3 Domain model

Following standard object-oriented development process, a domain model can be constructed after use case analysis (before more detailed object specification). This provides a reference dictionary of terms to be consistently used across the project. A model of 70 terms (given in Appendix C) was successfully generated for the solar physics data-grid domain, though it was not reviewed to the same level of the use cases. It remains incomplete, especially as it does not completely specify relations, and is therefore not presented diagrammatically. The definition of scientific information and the entities it gives are superseded by the solar metadata document developed later for EGSO [89] (notably in the 'observation metadata' group). It was also made redundant by the EGSO architecture [82] (discussed in Section 5.4), whilst its more detailed scope makes the entities incompatible with AstroGrid's sketched 9 element domain model [69].

However, a significant characteristic of the solar physics data-grid information domain is brought out by the domain analysis. Equivalent data content – the stored bytes and transmitted bits handled by the data-grid infrastructure – has 3 different layers of meaning, as illustrated on Figure 10.

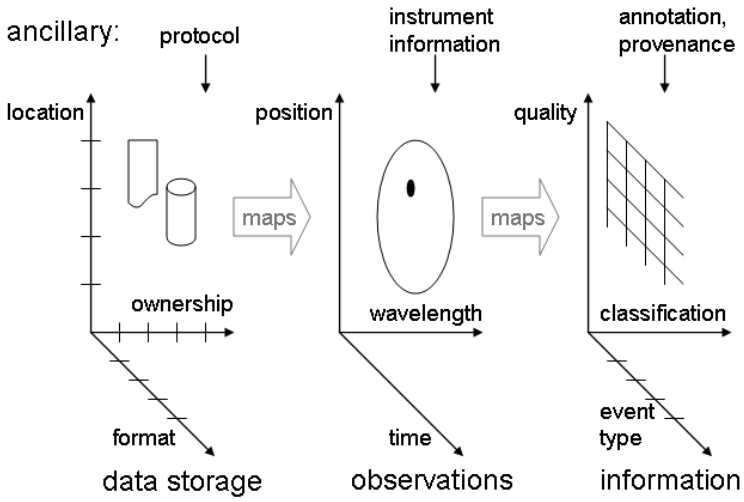

Figure 10: Levels of abstraction in the solar data-grid domain model – stored data maps to scientific observational data, and in turn value-added information about the scientific observations.

The data has semantic qualities at each of the three layers that permit its use:

- The raw observational solar data is in the middle, and each item has scientific meaning because of its location on several dimensions: observed features on the sun have coordinates in physical space, or typically reduced to the two-dimensional coordinates on an image, as well as the time interval and the wavelength in which they appear (wavelength corresponding to the physical dimension of temperature).
- The lower 'storage' data level has semantic significance to administrators and users beyond merely physically holding the scientific data. Its own dimensionality is given by the geographic location of the store (more significantly conceived as its position within the IP network which determines the transmission routes), the data ownership profile and the data format (all discrete dimensions of finite series of possible values).
- The higher information layer is that built by the scientists to add value in the data-grid (as defined by the e-science vision of knowledge arising over information and data layers), though it relies on broad solid foundations provided by the lower layers' content. Its dimensions describe the scientific observations in domain specific classification schemes for diverse event types, and by records of practical derived properties such as quality.

Each layer has ancillary data, which is required to interpret the core data in addition to the dimensional information that locates the core data elements. For the middle-layer observational data, it is the special ancillary information specified in early EGSO architecture, which includes instrument calibration data. The equivalent at the lower level is the protocol definition, and at the higher, additional information (such as scientists' annotations on observations and details of the information's provenance).

The properties of these three layers do not describe the full domain model of solar physics data-grid systems; functional and external entities are missing. However, clearly

distinguishing the layers allows architecture and further design to be discussed at the appropriate level of abstraction. Confusion of these different levels of information types and their roles had led to difficulty in technical and scientific stakeholders reaching agreement early in the EGSO lifecycle.

3.3 EGSO requirements

EGSO requirements arose after the 4 steps in the investigation carried out to support this research described above (the domain review summarised in Section 3.1, the use case analysis described in Section 3.2.1, the scientific exemplar specification of Section 3.2.2 and the domain modelling of Section 3.2.3). Though none of this abstract analysis was used to specify EGSO directly, it was presented and used in the numerous discussions of EGSO's direction. The studies helped the scientific stakeholders understand that data-grid infrastructure promised more than mere wider distribution of existing client-server data access and analysis methods (which would remain inflexible with their tight coupling).

The specific influences of tasks within this investigation on project artefacts are indicated in Figure 2 (the 4 tasks mentioned being those up to June 2002, excluding the ACME model design described in 4.2). Tasks in the subsequent investigation of data-grid requirements that were influenced by EGSO requirements are also indicated on Figure 2 (as other tasks in the requirements swim-lane as well as the generic useable security). These are described in subsequent sections: 3.3.1 for NFR analysis, 3.3.2 for goal decomposition, 3.3.3 for scenario specification and 3.3.4 for security analysis.

The EGSO team's scientists initially specified their requirements as scientific use cases [88] and envisioned applications [96]. These were worked into the prioritised technical requirements [87] that were needed to drive development. These are summarised as:

- Existing and future on-line data and information resources should be available via EGSO; users should be able to browse advertised archives and access data from different sources at the same time for comparison (for example, building image stacks [27]).
- EGSO should enable a unified gateway to these diverse resources, hiding heterogeneous data formats and access protocols, and exposing the analysis tools that are required to interpret the data.
- To provide analysis functionality, the EGSO infrastructure could include computing resources for data transformation (though high performance applications are not planned).
- Administration interfaces should support the monitoring and management of resources and the metadata that guides task resolution. Users and resources should be authenticated to enable flexible local authorisation.
- EGSO could be interoperable with other solar physics data-grids: AstroGrid, VSO [57,147] and COSPAR [122].

Once EGSO requirements were specified, the investigation moved from a general view of solar physics data-grid requirements to a concrete technical review of the project. Specific

areas of requirements were analysed, further evaluating the methodologies and supporting later modelling. NFR analysis, goal decomposition, scenario definition and security analysis are described below. This work is outside the EGSO software development critical path (so that observations remain independent of the instance system's solution).

3.3.1 Non-functional requirements' analysis

Section 2.2 notes how NFR are resolved by the whole architecture, not specific functional components. However, the initial EGSO conception (described in Section 4.2) only covered functionality, specifying how components like search servers and catalogue data types would resolve users' requests. As the NFR were not made explicit, though high-level design was to be determined from such conceptualisation, the system was at great risk. To illustrate what was missing, 12 textbook NFR categories were applied to EGSO and presented to stakeholders (at a meeting in November 2002, before the EGSO technical requirements were finalised [87]):

- *Modify* – extend by publishing new functionality on unchanged infrastructure. EGSO should permit future analysis services with novel interface types.
- *Support* – enabling access to existing resources. EGSO should expose gateways to specialist (high performance) computational analysis.
- *Port* – piecemeal process migration to new platforms whilst maintaining service.
- *Flexible* – access to diverse resources. EGSO should provide a common interface to legacy interfaces (without provider modification).
- *Integrate* – ability to incorporate novel data formats. EGSO should recognise diverse data models (and not demand modification of the source data).
- *Interoperate* – extend system functionality by interfaces to other data-grids (via portal or shared infrastructure).
- *Perform* – permit administrative optimisation during operation. EGSO may use catalogue or data caching, and rerouting around bottlenecks.
- *Scale* – infrastructure growth to incorporate arbitrary increase in network size. EGSO's infrastructural resources and observed latency should grow less than linearly in proportion to the volume of users and provider data and services.
- *Reliable* – all requests are actioned eventually. If EGSO resources are overloaded, users' tasks should be queued and not lost.
- *Robust* – the state of information that is important to the system and its users is not lost after failure. Users' tasks can be restarted from their last known state (especially if connections to parts of EGSO network are temporarily lost).
- *Integrity* – data is protected from erroneous and malicious modification. The data resources hosted on EGSO and the system's metadata must be protected.
- *Secure* – certified access policies are upheld, abuses are detected.

Stakeholders accepted the importance of some of these features, though flexibility, performance and security requirements had limits. Access to diverse data sets and legacy analysis, as well as the ability to incorporate future functionality, was required, but extension to computational modelling and reuse of the infrastructure in other domains were lower priorities.

Likewise, though some quality management capability, to optimise and scale-up the network, was needed, users would tolerate latency and EGSO would never support millions of users. Also, security policy support was necessary, but not as important as it would be in commercial applications.

Textbook NFR topics that did not apply to EGSO included:

- *Usability* – scientific users are experts in programming interfaces, so complex and graphically poor application and administration interfaces are acceptable.
- *Environmental constraints* – solar physics is a non-critical, open, collaborative domain, so the operating constraints of industrial or medical systems do not apply. For example, not required are: '5 nines' availability (for which annual outage is measured in minutes), financial transaction reliability and legally binding audit trails.
- *Maintenance* – stakeholders did not require debugging, repair and testing tools, or interfaces to report on load, critical events or other activity. Maintenance expectations may be low because of generally poor standards across scientific software, or awareness that the project's innovative position made it just a stepping-stone to future capabilities.

Despite having been tackled head on by project stakeholders, NFR remained under-specified in EGSO. In a novel domain, with unknown platform capabilities and so on, there cannot be concrete quantification of qualities like reliability or performance. Systems must be built and used to understand what degree of quality service could be achieved or would be acceptable.

3.3.2 Goal hierarchy

A goal-tree is a natural way of analysing requirements, as general high-level intentions are decomposed into detailed concrete requirements. For example, a goal for a distributed data-grid infrastructure may be partly met by transparent scalability, which in turn depends on dynamic reconfiguration and automated resource discovery. The goal-tree therefore defines a hierarchy that allows requirement 'leaves' to be grouped on a more abstract goal 'branch'. The mechanism is formalised in the KAOS method [29], which permits alternatives to be expressed and analysed to avoid conflict. Intuitive, informal methods are still useful for systems like EGSO, which do not demand the strength or overhead of formal methods (typically used when developing critical systems). Goal analysis drives down the risk of a project proceeding with inconsistent or incomplete requirements; conflicts are made explicit as alternatives, and underspecified gaps are revealed.

A goal tree was generated whilst the scope of EGSO was still being debated to bridge the solar physics use case analysis to architectural analysis (described in Chapter 4). Nathan Ching and Clare Gryce later produced another directly from the agreed EGSO requirements. Both are given in Appendix B; the tree produced for this investigation has just 47 goals, compared to EGSO's 169. This partly reflects the evolution toward concrete requirements, as some of the earlier goals carry over to the EGSO tree. However, the EGSO goal analysis was done for the explicit purpose of identifying gaps and inconsistent priority in existing

requirements. It has unbalanced depth and breadth across different parts of the tree because it does not attempt to define a taxonomy of the problem space.

It was surprising that in both cases EGSO stakeholders did not take up the goal trees, despite their intuitive representation. Instead, developers and managers worked from the traditional linear technical specification documents, perhaps because of their familiarity. These documents did evolve, and goals were discussed in demonstration driven development cycles, but the goal trees themselves were not used.

3.3.3 Generating scenarios

Scenarios are defined as concrete narratives of system use [19], in contrast to the formal classes of use captured in use cases. As they describe system behaviour from the users' perspective, they can capture both the breadth of functionality and the non-functional properties required in operation and maintenance.

47 scenarios (given in Appendix D) are derived from the EGSO requirements, grouped by 3 user action classes: consumer (scientist), provider administrator (acting for the information publisher) and hidden middle-tier (infrastructure administrator operations). These specifically helped development and testing of the event models of project's architecture (which was stabilised at the same time; see Section 5.4). As the scenarios were derived from earlier use cases (with their EGSO stakeholders' feedback and other ideas from other discussions), they also independently validate the completeness of the requirements; the project manager and requirements' authors reviewed them.

8 core scenarios were abstracted and specified in the EGSO architecture's terms; others merely refined the behaviour of these or captured lower priority requirements. For illustration, 4 of these are given below (summarised from [82]; these are referred to in Section 5.4, which describes the EGSO architecture):

- A consumer constructs (and saves) a new search for data resources. The query is resolved after the broker addressed by the user forwards it to another broker with the matching catalogue record.
- A consumer reuses a query for data. A broker prompts a provider that it chooses to pass results to the consumer, who visualises the result.
- A provider notifies a broker that it hosts a data set. The broker acknowledges, and subsequent consumer queries exploit the newly catalogued metadata.
- A broker redirects a consumer request to another provider that hosts the same data type, once it detects a failure for unavailability from the first choice.

As the scenarios are derived from the requirements but can be specified in architectural terms, they provide an independent bridge between requirements and design, reducing the risk of misunderstanding between the customer and developer. They may also be reused as system test descriptions (though the demonstration driven development EGSO adopted later made these unnecessary). This reinforces the relationship between software lifecycle stages, whereby earlier design stages are associated with later testing (the V-diagram, Figure 1).

3.3.4 Usable security

EGSO benefited from being used by Ivan Flechais and Angela Sasse as a case study for usable security research [39], building on UML customisation for security analysis [63]. The representation of EGSO's security assets shown in Figure 11 was produced through collaboration. At this time, the project's architecture was being defined while the security requirements were still being refined. The assets could therefore be divided to the three-tiers of consumer, broker and provider sub-systems (the tiers are described fully in Section 5.4). Information asset 'classes' were recorded with crudely quantified security properties; vulnerability is determined by availability and integrity, and protected according to actors' responsibility. Attack risk is the product of the probability and cost of an asset being compromised.

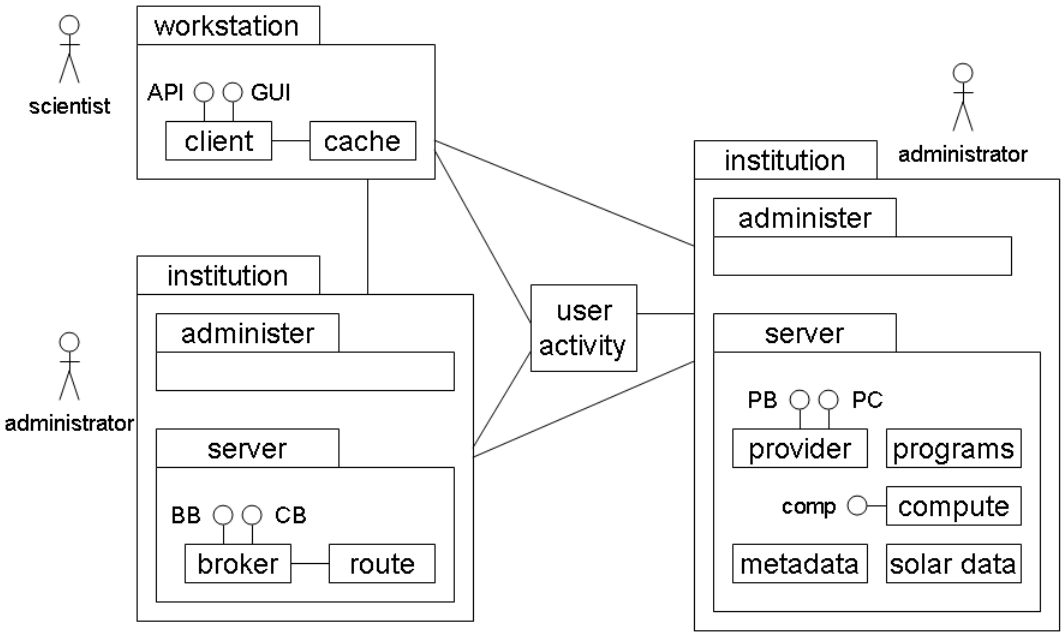

Figure 11: EGSO security assets, identified for analysis by risk and impact of security breach and responsibility.

The investigation demonstrated that EGSO's design was at risk, given the requirement for protection of user confidence and especially provider data asset integrity. Though there was no obvious dependence of critical assets to insecure interfaces, the division of roles left it unclear whether broker administrators would accept responsibility for protecting all providers. If providers lost confidence in EGSO's integrity, the infrastructure would have no value to users. In practice, this was not seen as a great risk, as key providers were expected to be the hosts of the broker nodes too. Such user interest, responsibility and trust issues would impact other data-grids were EGSO's architectural style to be more widely applied.

3.4 Broader data-grid requirements

3.4.1 Generic data-grid requirements

Following the data-grid project review (Section 3.1), then EGSO's technology review [25] and requirements' specification, Clare Gryce identified 83 requirements shared by data-grids [37] (listed in Appendix E). They form 18 high-level requirements, described below, in 3 groups: characteristic overall properties, specific capabilities, and other system properties (the NFR). These may be read as a checklist of other projects' completeness of requirements, and as unambiguous classification criteria to define data-grids. They are used to analyse architectural styles' suitability for data-grids (described in Section 4.1).

Characteristic overall properties:

1. *Data-resources.* Data-grid systems incorporate data resources distributed across organisational access boundaries into one logical resource.
2. *Data access.* Data-grids allow users to discover and access resources in a transparent way, so that back end access or format heterogeneity is hidden.

Functional capability requirements:

3. *Data management.* Data-grids include voluminous heterogeneous data, inflated by managed replicas. Variety comes from different standards and schemas.
4. *Metadata.* Data-grids use legacy domain metadata standards, especially where catalogues use semantic information about data. Such metadata may be automatically generated and distributed.
5. *Querying.* Users access data produced by complex workflow analysis or data-mining discovery, possibly as off-line batch tasks. Simple data query access by matching attribute values is complex if resources are changing or need joining.
6. *Processing.* The infrastructure processes data on distributed computing resources like compute-grids. Users' custom analysis code may be distributed, or data products may be generated in pipelines.
7. *Data transfer.* Data-grids manage very high volumes of data, sometimes copying entire data sets or storing instruments' continuous data streams.
8. *User interface.* Data-grids' user interfaces support flexible query specification and management functionality, partly hiding the infrastructure's mechanisms whilst enabling task interaction and collaborative working.
9. *Applications.* Data-grids typically incorporate existing tools, applying them to richer data resources. These include including component toolkits and API for data analysis and visualisation.
10. *Monitoring.* Data-grids record static and dynamic information about their infrastructure, enabling administrators to manage the higher-level capabilities (for example: tracing errors and optimising resources' availability).
11. *Management.* Multiple concurrent user tasks are distributed across resources, avoiding bottlenecks whilst permitting prioritisation and interaction.
12. *Interoperation*. Data-grid projects interoperate with each other, sharing resources or metadata, or accessing functionality via portals.

Other capabilities (NFR):

13. *Security.* Authentication allows flexible user role authorisation policies and reciprocal service trust. Requirements for lightweight processes, mobile users, confidentiality and data integrity make security challenging.
14. *Scalability*. Anticipating load (in terms of data volume or resource capacity) is difficult, so data-grids must be easily scaled up by at least an order of magnitude.
15. *Performance*. Processing capacity and users' perceived latency (in query execution or service discovery) define data-grid performance.
16. *Reliability*. The reliability of data-grids' core service (for example, security mechanisms and job recovery after failure) is not as great as established distributed systems.
17. *Maintenance*. Data-grids interface components (to users and resources) are typically portable. New functionality can be easily added, for example, by advertising new services.
18. *Integration*. Data-grids are constructed with component technology to facilitate the integration of diverse, evolving and legacy elements.

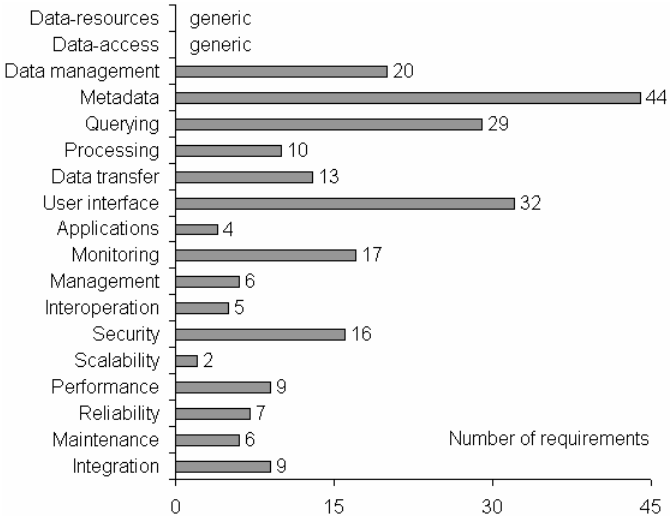

Figure 12: EGSO requirements scored against data-grid functionality (1 point per requirement that describes the capability).

EGSO's requirements can be mapped to the 18 generic data-grid properties; Figure 12 shows the number of requirements that fall into each topic. Requirements that do not fit any of the generic data-grid requirements represented domain specific functionality; they do not indicate the generic list is incomplete, nor that EGSO muddles in non-data-grid goals.

Figure 12 shows that EGSO stresses data access facilities (resource access, data management, metadata and interfaces). Requirements to ensure the future success of the infrastructure (including monitoring, integration and security) are also specified. However, the project does not stress: high volume data transfer, distributed resource scheduling, fault tolerance or scalability; EGSO users' immediate needs are for any access to the existing resources.

The lack of resource management requirements seems a more serious oversight – there are no reasons for EGSO to not support administration. Documented requirements that describe functionality miss the expectations of project managers and intentions of software engineers though, who agree that software engineering best practices should be followed to deliver high quality.

3.4.2 Comparing use cases

It is interesting to evaluate the requirements' analysis carried out in data-grid projects, as they illustrate how software engineering methods meet a demanding task. The initially vague understanding of how needs may be met by new technology was refined until technical design could implement complex distributed systems.

The use case studies carried out for the EGSO and AstroGrid projects described in this chapter each have distinct scope. This is clear when placing them on 2 axes, distinguishing user-oriented scientific from development-oriented technical requirements, and abstract visions from concrete specifications. 8 artefacts, including use cases and other narrative requirements documents, are placed on Figure 13 (these include research contributions presented here, indicated by section number, and project documents, indicated by citation number). Their placing is relative; absolute scoring is unfeasible, but individual documents are easily judged more scientific or more concrete than others.

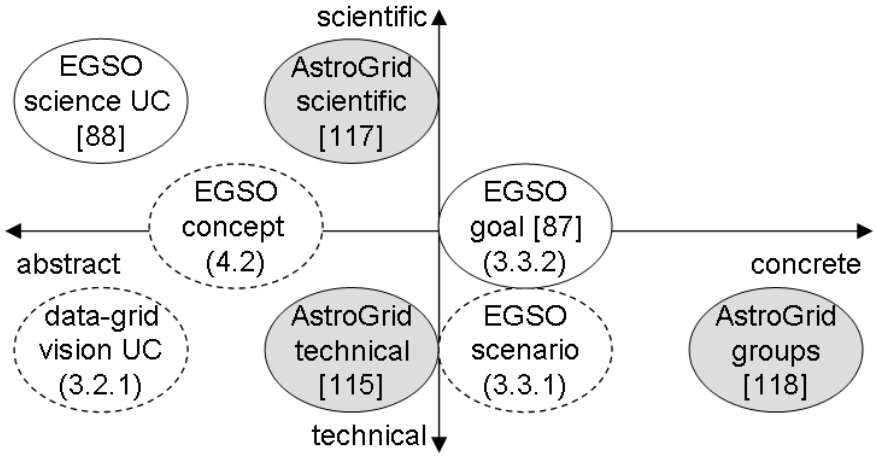

Figure 13: The diverse use cases written for EGSO and AstroGrid are all different, being abstract or concrete and scientific or technical. (Non-deliverable use cases are dashed).

This figure may be read from left to right; later documents are more concrete, as both projects progressed top-down from overall vision to detailed use cases. The contributions arising from this research (the data-grid vision, including domain model and goal tree, and the EGSO scenarios) are more technical, as software engineering is grounded in the idealised wishes of the user community. The top right corner of the diagram is noticeably empty; the concrete scientific specification of problems is the users' real work to be supported once the system is deployed.

The EGSO project's informal concept, which mixed required scientific processes and technical architectural components, may be seen as unnecessary; the delivered scientific use cases and goal decomposed technical requirements alone permit evolution of concrete specification. This document predated the others, and functioned as a conceptual bridge for stakeholders (especially the project manager). However, its poor separation of concerns was a cause of frustration in reaching agreement and evolving the document. In contrast, the AstroGrid technical use cases (that were accumulated with the scientific use cases) could directly specify the responsibilities of the development groups in coordinated iterative development (to deliver increasingly sophisticated demonstrations).

Chapter 3 key points

- Diverse data-grids share typifying requirements, but suitable architectural styles are poorly applied, putting current projects' long-term maintainability and behavioural quality at risk.
- Use case analysis for solar physics demonstrates that data-grid capabilities must be built on existing Internet uses.
- Advanced requirements' analysis – of NFR, security, hierarchic goals and usage scenarios – benefited EGSO by: mitigating the risk of software crisis through clarification of acceptable and safe behaviour, improving complete and consistent problem space definition and shared understanding, and defining criteria for verification of subsequent design and implementation.
- Abstract requirements, oriented toward scientific users, must be refined to concrete technical specification (clearly bounding the scope of each level) for system development.

Chapter 4 Architecture models

2 methods are used to identify suitable architectural designs for data-grids in general and EGSO specifically.

- The review of data-grid projects and their technology in Chapter 3 and Chapter 2 revealed 5 classes of suitable architectural styles. A novel method to find which style is most suitable for the requirements of Section 3.4.1 is described in Section 4.1. (A report on the method has also been published [37].)
- Preliminary informal designs for EGSO are encoded in ACME, to permit formal analysis of architectural qualities. The value of this technique is shown by the discussion of its application in Section 4.2.

4.1 Fitting architectural styles to data-grid requirements

The requirement-style matrix scoring method

Chapter 2 described how architectural styles for distributed systems are associated with data-grids. The informal impression that their qualities would support the general data-grid requirements described in Chapter 3 is quantified by the following novel method. The 5 architectural styles are scored against the 83 data-grid requirements to generate a matrix (given in Appendix E). The scoring scheme uses just 4 values:

- 2 indicates a style whose explicit purpose was the satisfaction of the given requirement. There are several requirements that data-grids share with other distributed systems, so the established styles directly resolve some requirements.
- 1 indicates styles that help to satisfy the given requirement. It may be given if technology associated with the style had historically exhibited the required behaviour, or if primary features of the architecture may be adapted to satisfy the requirement.
- 0: if the architecture has no obvious impact on the requirement, or has balancing positive and negative effects, it is given a neutral score. Many data-grid requirements were neutral for several styles, being below architectural resolution; the requirement would only be implemented within a component or by a subsystem.
- -1: styles that undermine a requirement are given a negative score. The goal of the architecture may contradict the requirement, or mechanisms typically implementing the style may have a negative impact on the required behaviour.

Style suitability scoring is therefore an intuitive judgement, making this method subjective and not necessarily reproducible (though scores are recorded with reasons). However, it is equivalent to industrial best practice, whereby experienced software developers judge whether requirements are met by reusable components (including common algorithms, function libraries, sub-systems and design patterns). This method rapidly covers a large design space (the matrix took just a few hours to complete); it fits diverse solutions to many

requirements on the assumption that styles achieve what they are intended to. A more thorough, tractable method would formally associate requirements to experimentally proven design properties; this would be prohibitively laborious.

Summed matrix rows indicate requirements' overall sensitivity to architectural choice (whether a good or bad architecture will support or undermine it). The total of absolute values is used so that negative scores also increase the magnitude of this measure. The average architectural sensitivity of requirements groups is then calculated to gain an overview of the matrix. (Taking the average prevents architectural choice appearing more important for groups with more requirements). Architectural sensitivity is assessed as 'low', 'medium' or 'high' by dividing the ranked requirement groups into 3 equally sized sets (the upper and lower boundaries of medium architectural sensitivity (greater than 2, less than 3.5) are therefore arbitrary).

In summarising the style suitability for the group by column, the strongest score is represented as a symbol ('++', '+' or '-'; the 'strongest' score being the furthest from 0, so a requirements group with a single requirement scoring of 2 would be marked '++', whereas a group with several scoring just 1 would be marked '+'). Finally, the column sums along the bottom of the matrix give an overall impression of an architectural style's suitability for data-grids. The calculation of these measurements is shown in a fragment of the matrix in Table 1, and the summary itself is presented in Table 2.

Requirement:	Layered	n-tier	Peer-to-peer	Dataflow	Agent	Absolute total	
16.1		1	2		-1	4	
16.2		1	2			3	
16.3		1		1		2	
Summary		+	++	+	-		Average sensitivity:
Total	0	3	4	1	-1		(4+3+1+1)÷3=3

Table 1: Fragment of the requirements architecture matrix, showing calculation of row and column summary values.

	Requirement:	Layered	n-tier	Peer-to-peer	Dataflow	Agent	Sensitivity:	
1	Data resources	++	++	+		-	6.0	High
2	Access to resources		++	++		+	5.0	High
3	Data management	+	+				0.3	Low
4	Metadata	+	+	+		++	2.7	Medium
5	Data querying	+	++		+	+	1.7	Low
6	Data processing	+	+	++	++	+	3.8	High

7	Data transfer	+			++		2.0	Low
8	User interface	+	++	++	++	+	1.9	Low
9	Applications tools	+	++	-	+		2.0	Low
10	System information		+	-		+	2.5	Medium
11	Resource management		++	++	++	+	3.2	Medium
12	Interoperable	++	++		+		5.0	High
13	Secure	++	++	+	-	+	1.8	Low
14	Load capacity	-	-	++	+		2.6	Medium
15	Performance		+	-	++	-	2.3	Medium
16	Fault tolerance		+	++	+	-	3.0	Medium
17	Extensible	++	++	+			3.5	High
18	Integrable	+	++	+			3.5	High
	Suitability:	27	63	41	24	16		

Table 2: Summary of the requirements architecture matrix, showing requirement groups' sensitivity and the styles' suitability.

Data-grid requirements architecture matrix observations

A low score for the overall style suitability in the bottom row of Table 2 either indicates that the style cannot meet the requirements or actually hinders their fulfilment. However, as they are all positive, current data-grid projects have avoided wholly unsuitable architecture in the five styles they exhibit.

The most widely used n-tier style scores highest, followed by peer-to-peer, supporting the convergence of data-grids and peer networks noted in the community [26,45,68]. The method also identifies specific requirements met by a peer-to-peer solution, such as access. This method would achieve this if applied to other systems' architectural styles beyond data-grid too. The high scores of pipeline and layered architectures arise from their distributed computing and communication capabilities, including filter transformation and data management, which suit data-grids. However, they only offer a partial solution to the domain's overall problem space. Likewise, agent technology only meets narrow functionality within data-grid operation.

Low scores for requirement groups' architectural sensitivity in the right hand column of Table 2 indicate architecture choice does not much influence whether a requirement can be met. However, the scores for the characteristic data-grid requirements, data resource access, demonstrate high sensitivity to architectural style. This indicates that the choice of architecture determines whether the top-level system goals, which define overall data-grid behaviour, are met. In general, the essential summary requirements of systems would be identified by this method (if they did not score highly, they would be missing the 'whole picture').

Other data-grid requirements sensitive to architectural choice concern flexibility (interoperable, extensible and integrable systems) and data processing (another key data-grid function). Most other non-functional (load capacity, performance and fault tolerance) and data-grid management (metadata, system information and resource management) requirements show medium architectural sensitivity. Security and straightforward functional requirements

(data management, querying and transfer, and user interface and application tools) have low demonstrated sensitivity; they can more easily be encapsulated to fit any style. The distribution of architectural sensitivity across requirements reinforces the importance of sound architecture for data-grids.

The matrix method fits between the simple association of requirements to system components (as used in commercial development to trace requirements to implementation elements) and design space analysis (notably, 'House of Quality') [66]. The 5 styles, as candidate designs, would define 'vectors' using the axes of quality, if analysis were to specify the system property space. However, abstracting the 83 stated requirements to property scales would either hide the detail that characterises the domain, or be unfeasibly complex. Applying this novel method to the data-grid case study demonstrates that scoring each style against every requirement is simple and illuminating.

4.2 Encoding EGSO architecture in ACME

Preliminary EGSO architecture

EGSO's behaviour was initially described by the system scope in the funding proposal and by box and line sketches. These informal diagrams were intended to map to candidate architecture, but proved too vague to associate with typical software components (for example: sub-systems, libraries, server daemons, file stores, or database management systems). 2 are reproduced below with their narratives.

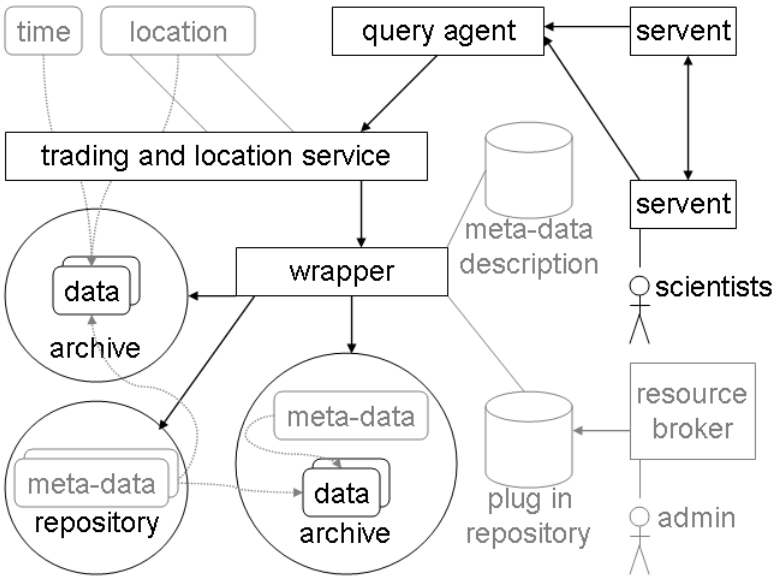

Figure 14: decentralised candidate EGSO architecture, supporting heterogeneous resources and shared user result (with peer-to-peer components).

Figure 14 shows a decentralised EGSO architecture with local data-product sharing. Archives of scientific observations and their metadata descriptions are the data resources. A wrapper layer converts the resources' heterogeneous formats to a common representation for search and comparison. The wrapper uses conversion plug-ins and metadata ontology – collected in stores administered by data providers (an ontology is defined as a repository of the

terms that add semantic value to data, recorded with complex relations going beyond the 'is a type of' relations in taxonomy trees). A distributed trading service directs user queries to suitable data-resources using metadata about available observations' timings and targets. Users' clients sustain distributed sessions of high latency over unreliable connections by delegating query management to agents. They also act as servers for each other, sharing results in a peer-to-peer style (the word 'servent' is used after Napster nodes [80]).

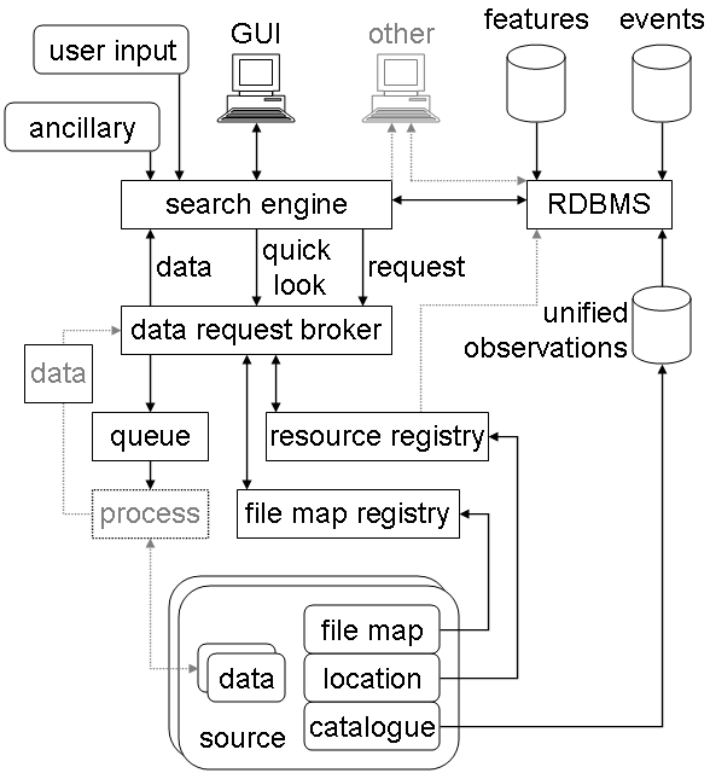

Figure 15: hierarchical candidate EGSO architecture, distinguishing metadata function and supporting both quick and processed queries (to middle and back-end tiers).

Figure 15 shows a finer grained hierarchical division of components' functional responsibility. 6 types of metadata describe the resources: 3 at the data sources ('file map', 'location' and 'catalogue'), and 3 managed by a catalogue database (marked RDBMS; they are: 'features', 'events' and 'unified observations'). Users' input queries and ancillary analysis data are also recognised resources. Quick searches (that are resolved by the 'data request broker' using registries of storage resources and data file organisation for the resources) are distinguished from those that require processing (which are queued before processing the source data, generating temporary data results to be managed by the broker). The tiered infrastructure of search engine, broker and registries therefore separate user applications from data sources. Additionally, interaction with the back-end data source resources is saved in quick searches that are resolved at the middle-tier.

Note that these early conceptual diagrams capture different design styles informally. The first is a flexible distributed solution blended from n-tier and peer architecture, though it ignores the complexity of existing solar physics resources. The second goes further in

describing data resources' organisation, though its implicit centralisation risks scalability. At this stage of system development, a choice between competing styles would best be made through analysis of architectural models. EGSO therefore offers the opportunity to evaluate the strength of formal architectural specification methodology in a genuine use case early in its lifecycle.

The merits of ACME

ACME [52] is an interchange language for ADLs (reviewed in [77]), which are in the formal category of models introduced in Section 2.2 'architectural models'. Though it does not specify its own formal calculus, it is suitable for modelling data-grid architecture as it has these qualities:

- It is *practical*, being relatively mature and having a prototype tool for graphical architectural composition and validation.
- Its *generic* syntax guarantees accurate reproduction to other representations, and permits designers to clearly evaluate just their properties of interest. For example, Wright [2] (another mature ADL with rich connector syntax) could be used to represent the behaviour of EGSO query tasks' distributed state.
- It supports the *abstraction* necessary for identifying design patterns. Templates can be reused through casting to different component and connector types, and elements can be composed hierarchically for grouping with shared interfaces.

However, it was found that generated code's syntax did not match that presented in the literature (when modelling a sub-set of the informal EGSO components with the ACME tool). It was also hard to manage a realistic number of elements or take advantage of the features of the language that supported abstraction. Subsequently, the tool was just used to validate model code.

Building EGSO architecture with patterns

The static formal ACME model of the envisioned EGSO architecture (given in Appendix F) was implemented by defining components, determined by the boxes of the architectural sketches and their accompanying narratives, summarised above. Both sketches were encoded together in a common ACME model to maximise coverage of the envisioned functionality. The sketches were also not encoded independently as they shared many key components, for example: the scientists' user interface, a trading broker to match requests to data resources, and a descriptive catalogue of resources.

In total, 11 of the decentralised architecture's 14 elements map to 13 of the hierarchical architecture's 16 elements (which has greater detail of metadata types). Appendix Section F.2 tabulates both of the sketched models' components, associating them and noting any differences in their intended coverage.

Note that just 9 of the common elements are represented in ACME. Administrative and management components are excluded, as they are not special features of EGSO or essential to the narrative of data-grid operation. Likewise, the distinction between the diverse types of metadata described in the hierarchical model is not captured, as their difference is only significant to the algorithms that match requests to resources. Both groups of excluded

elements therefore have no impact on behavioural properties that an architectural model should capture. The modelling method therefore permits economies to be made by just specifying components that are essential to the view of interest.

The 3 unique elements of the decentralised architecture concern shared working and the filter and plug-in mechanisms (which provide transparent access to heterogeneous resources) whilst the 3 unique elements of the hierarchic sketch represent the ancillary information and task queuing and processing that are necessary to support data analysis processes on a grid scale. As these areas are key ingredients of data-grid capabilities, all 6 are encoded in the ACME model. The association of the 9 common elements and 6 unique sketched elements to the ACME model's components is also given in the table of Appendix Section F.2.

In total, though, there are 22 components in the specified model of EGSO architecture (given in Appendix F and shown on Figure 16), as 7 were added beyond those sketched. These additions are necessary to manage the responses that completed activities (which the narratives only described being initiated). This gap in the early informal architectural sketches would have been bridged later in the project, but the oversight is identified earlier with formal specification, demonstrating some benefit to early rigour.

More significantly, it was found that substantial abstraction to reusable element types is possible in the ACME specification of EGSO's architecture instance. Following the narratives' descriptions of how the elements interacted to do the work of a data-grid, it is apparent that all components use a combination of just 3 types of connector to communicate: a single message requiring action, a dialogue in which exchange may be necessary to resolve an operation, and a stream of a significant quantity of data. These are formally encoded as the following types:

- *'submit'* connectors have no session – atomic messages are sent in one direction,
- *'request'* connectors represent an interactive session with a dialogue of requests and corresponding responses (these always terminate at 'data' servers, described below),
- *'write'* connectors represent voluminous data transfers, streaming content (but not necessarily control messages) in one direction.

It is also apparent that there are just 4 types of component, adapted to use in different ways across the architecture. The component types are defined by rules for the connector types they may use. Such constraints on formally defined connectors and components in an architectural specification provide a solid foundation for analysis of behavioural properties and subsequent validation of detailed design and implementation. The 4 classes of component are:

- *'origin'*, the source of 'submit' connectors (which may initiate 'request' connectors),
- *'filter',* the sink of 'submit' connectors and the source of either another 'submit' or a 'stream' connector (it may also initiate 'request' connectors),
- *'consumer',* the sink of a 'stream' connector (and may also be the source of a 'stream' or 'request' connector),

- and *'data'* components that are defined by being the target of 'write' and 'request' connectors (they do not initiate connections).

These 3 connector and 4 component types are defined as templates in the ACME specification; as generic component and connectors they could be rearranged to build alternative EGSO architectural solutions and other data-grid designs. The 22 components of the EGSO architecture are defined as instances of the component templates, and connected as shown in Figure 16. (The ACME tool did not generate this diagram; the shapes are customised to highlight their types in a non-standard way.) ACME's hierarchical construction can also allow the components to be divided into the 3 tiers of classical distributed system style described in Section 2.2 (servant application tier and archive back-end about a middle grid tier).

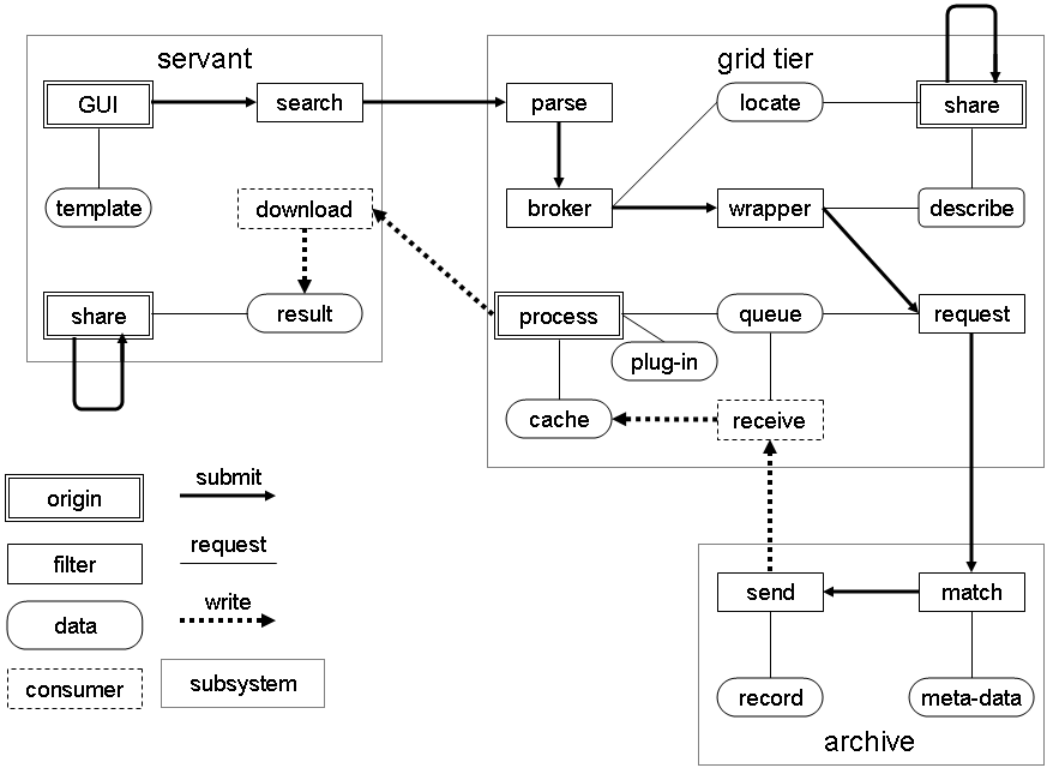

Figure 16: EGSO's envisioned architecture represented in ACME for formal analysis.

Lessons learnt from the EGSO ACME model

The static formal description of the candidate data-grid architecture, which captures the functional elements indicated by sketched boxes on the sketches, is simplified by understanding the components' types of relationship (arising from the actions that connect them, described in the sketches' accompanying narratives). The clear model derived should provide a stronger foundation for rigorous formal analysis of data-grid architecture, as identified properties would apply to all instances components and connectors of the same type. The limited set of element types and their constrained relations also reduces the problem space, making complete formal analysis more feasible.

The ACME model of EGSO specifically demonstrates how the overall architecture conforms to the n-tier style, as well as indicating potential data-grid design patterns. However, such patterns should not be recognised for reuse in other projects until they have proven useful in genuine deployed systems. While that has not yet been done, they do still indicate that economies could be made in development effort by implementing generic functionality and customising it for EGSO components. In general, ACME enables clear abstraction for economic expression of reusable architectural elements through its hierarchic and template techniques.

Unfortunately, EGSO stakeholders struggled to see either how the ACME model related to their vision of the architecture or how the model would be traced to the next design stage. This may be because the novel, formally supported, software engineering representation of the sketched system decomposition was too far beyond their experience and understanding of requirements' analysis and software design. Those committed to the waterfall lifecycle for clear development and project management may also have been unwilling to consider interface design, implicit in connector type definition, whilst requirements' analysis was still in progress. The model was advanced in 2 ways to recover material benefit for the project and attempt to demonstrate the method's value:

- A UML class diagram was developed for the generic 'data' component to demonstrate how to move onto the design stage. It had 9 classes for authorised heterogeneous connection management (though operations, attributes and message sequences were not defined). EGSO managers felt such detailed design effort was premature.
- A simpler ACME model that only encoded the architecture's tiers was implemented with preliminary state models (which could have been implemented with Wright). The value of this implementation effort (given the complexity of ADLs and their poor tool support, in contrast with the methods presented in Chapter 5 and Chapter 6) was dubious.

Following the poor reception of the ACME model, it proved easier to capture and analyse the architecture in event models, discussed in Chapter 5. In general, slow uptake by stakeholders uninitiated in formal declarative methods may be expected of ADL architectural modelling beyond this case study. See further conclusions in Section 7.1, evaluating ACME against the criteria for successful modelling defined in Section 2.2 (alongside the event modelling and simulation techniques of the following chapters).

Chapter 4 key points

- The choice of data-grid architecture can be guided by aligning requirements to candidate distributed systems' styles. This choice is especially important for NFR.
- ADL specification of data-grid architecture promises clear abstraction of essential design components, but its representation is hard to communicate.

Chapter 5 Event models

This chapter reports on the experience of modelling data-grid architecture with the FSP language and LTSA tool (also published [38]), chosen for the reasons given in Section 5.1. LTSA models software systems by representing and analysing the combined state-space of semi-independent processes in the system, as described in Section 5.2 (a complete introduction to LTSA is given in [75]).

Models were developed at 5 different design stages – 3 within the lifecycle of the EGSO project, 1 in AstroGrid, and 1 for design pattern abstraction. Their code is given with maintenance documentation in Appendix H to Appendix L. Sections 5.3 to 5.7 describe the models through:

- the project state and modelling goals,
- the implementation method and effort, how the model was validated, and noted language limitations and other observations (which apply in general, beyond the data-grid case studies),
- details of how the model was communicated to colleagues, and its specific impact on the EGSO or AstroGrid projects.

5.1 Choosing LTSA

The role of models in systems analysis was introduced in Section 2.2. For thorough software engineering analysis, design representation should meet the following criteria:

1. It must support formal reasoning, representing the candidate system in a way that supports deduction of its behaviour. Conclusions about the design would then have the full force of a mathematical proof.
2. It must represent dynamic properties (as well as static function), as data-grids' NFR are essential and sensitive to early design choice. Measurable quality should emerge from the composed model as it does in reality.
3. It must interwork with a precise diagrammatic design representation schema, such as UML, which will guide implementation better than an abstract model language. The model can then be proved to faithfully represent the real system.
4. It must be a living product with current tool support. Published methods risk being overtaken by more popular tools (an observed shortcoming of ADLs). It is difficult to generate and analyse instances of models on current platforms without live support.
5. Its successful application to genuine case studies must be demonstrated. Published methods are also at risk of being untested concepts, which cannot feasibly represent or analyse complex real-world systems.
6. It must be clear and presentable. Faults found in design models should lead to changes that other project stakeholders must accept. Models must be comprehensible by those outside their development; derived summary views or animated demonstrations build others' confidence in the method's conclusions.

FSP models and the LTSA tool have these required properties:

1. FSP is based on pi-calculus, a formal grammar with sequence information. The proof that a model is safe ensures undesirable behaviour is not possible.
2. LTSA captures the dynamic behaviour of a system, not just the static relationships of its elements. Live properties emerge from system components' programmed event sequences.
3. LTSA was designed to interwork with the ADL Darwin [74], which represents the connected components of software systems. Darwin is analogous to UML class diagrams, as it provides a graphical view of design elements' relations and properties (especially their interfaces) over formal documentation of the interfaces to their functionality. It is also possible to directly map UML message sequence chart view to LTSA [106].
4. LTSA is being actively developed. Version 2 and its revisions use the Java Virtual Machine, ensuring broad platform support.
5. FSP is used to analyse the Java process model and thread library [75].
6. FSP is clear enough for teaching, whilst LTSA provides simple views of complex models' alphabets and state transitions. LTSA can graphically present combined state charts and animate behaviour (with plug-ins), though these become unclear for complex systems (which have more possible events).

Listed below are other formal representation schemas, which may be used to capture and analyse the dynamic behaviour of data-grid designs. The reasons why they were not applied are given.

- Prolog is the most widely recognised declarative programming language, used for deductive reasoning about a set of propositions [137]. However, there is no widely established method for translating design documents to this general-purpose language (as there is for UML, for example, in the unified process). Also, Prolog has no language primitives for temporal sequence to support analysis of dynamic behaviour.
- PVS is an expressive formal reasoning tool, developed more recently than Prolog [138]. However, it lacks an established body of literature by and for PVS users; no cases of its use in software analysis could be found.
- Abstract formal grammars, such as Z, are established as software system specification languages [97]. However, they are perceived as difficult representations to work with, so data-grid stakeholders would be expected to find them at least as hard as ACME to comprehend. There is little tool support to generate descriptions in these languages, and they lack standard diagrammatic views to help understanding.
- There are other event modelling languages like FSP, including the widely known CSP [124]. However, as LTSA is used in teaching and its developers are available for consultation, efficient FSP model development is ensured in this research. Other languages would be investigated further were FSP insufficient.

5.2 LTSA features

LTSA analyses systems described as partially independent concurrent processes, whether operating system threads or complete sub-systems on a network. It combines processes around shared events, which may represent messages between entities. LTSA analyses their overlapping state space to identify whether specified error states are reached, and whether progress is possible; 'deadlock' (where no further events are possible) and 'livelock' (unbreakable circular event paths) are identified as faults.

The models that LTSA analyses are formally specified in FSP. The tutorial in Appendix G (which illustrates the method described in Section 7.3) introduces the language through a developed example. Essential features of the FSP language are noted below:

- Hierarchical processes can be defined; concurrent processes are defined by a composition operation, whilst sub-processes can simplify the description of the state model. Multiple instances of the same process type may also be programmed, an essential language feature for assessing whether competition for shared resources causes problems.
- Processes can hold state variables, which may annotate events. This is useful when information is passed between processes as the argument to a synchronisation event.
- Alternative event sequence paths can be programmed. These can be prefixed by conditional or precondition descriptions, typically tests of variables. Without conditions, LTSA chooses alternative possibilities fairly and randomly (guaranteeing all are executed when the process repeats).
- Extensions support code generation from message sequence charts (including negative scenarios), and quantified stochastic annotation (to generate simulation statistics).

Though LTSA may be thought of as best applied to low-level analysis of concurrency risks (for example, in operating systems or embedded real-time systems), it can successfully capture high level and abstract data-grid design properties. The following sections, of 5 different design stages, demonstrate this.

5.3 EGSO concept

Project state, modelling goal

Section 4.2 introduced the EGSO community's vision of a data-grid implemented by distributed components; user queries would be resolved against middle-tier catalogues, and data-products may be shared in a peer-to-peer network. However, it was unclear whether distributed independent entities would be able to work together as envisioned. Notably, representatives for scientific users, with little software engineering experience, saw no benefit in the extra components between user and provider. So 4 models were implemented to demonstrate that the sketched systems could do essential data-grid tasks and increase stakeholder confidence in the preliminary architecture.

Implementation, validation, limitations

Model development took 5 working days, following textbook examples and reusing exercise models. Each of the 4 models listed below tackles a specific challenge of the unfamiliar design for EGSO:

- A 'layer' model demonstrates that a common type of service state can be used by different layers in query resolution, from the user portal via metadata management to the data store. A search could fail at different layers, isolating data providers from bad queries and infrastructure faults. Appendix H.1 describes the model's implementation, using binary states for the layers.
- A 'queue' model demonstrates that multiple clients can concurrently submit queries to a shared queue, whilst tasks are fairly scheduled to a back end service provider. A middle tier broker therefore manages a shared resource. Appendix H.2 presents the model with a detailed description of the indexed queue slots' states and looping queue management processes.
- A 'secure' model demonstrates how requests through a service portal can be guarded by a third party's check of clients' identities. Each client's access status is held and administered by the independent authority. The description of the model in appendix H.3 highlights the shared events.
- A 'tier' model demonstrates how static and dynamic metadata records of data providers' resources and status enable voluntary location (and migration) transparency. The client can specify a preferred provider, but is routed to another with the same content if the first choice was unavailable. It is apparent from the model that transparency is not symmetrical; providers must maintain the consumers' identities associated with queries. Appendix H.4 describes the model's simplifying abstractions.

At this stage LTSA could not represent all the behavioural quality of data-grid architecture though:

- The models only weakly represent the performance of scheduling and security concerns. Stochastic LTSA or another simulation language that supports continuously variable annotation is better suited to evaluate algorithm performance (see Chapter 6). Specialist annotated object analysis (as described in Section 3.3.4) provides greater confidence that a distributed design upholds security constraints.
- These models do not attempt to express the provider interface and data format heterogeneity that must be accommodated by middle tier management entities. Evolutionary prototypes based on established design patterns that use the real provider interfaces with candidate translation mechanisms seem a better way to tackle this design challenge.

Referring back to the model types identified in Section 2.2, its clear that the abstraction slice through the full space of system design taken by formal models (a dynamic event-oriented view for FSP) exclude some early design concerns.

Observations, communication, model impact

Despite the methodological shortcomings noted, that these models could be implemented supports the EGSO architectural vision. Tracing event sequences to reach target states (such as a client receiving a resolved query) tests and demonstrates the informal system architecture. As models with multiple instances of entities do not reach deadlocks, there is no logical restriction to the scalability and reliability of these basic designs. The success of model evaluation for the immature EGSO design demonstrates LTSA's suitability for early architecture evaluation in general.

Sceptical stakeholders understood that the architecture met their requirements on seeing how the models reached their goals. Project managers and engineers recognised the models captured views of the conceptual system represented by the informal diagrams. The models mitigate the risk of design failure; had serious flaws been identified, forcing redesign, minimal effort would have been wasted.

5.4 EGSO architecture

Project state, modelling goal

Later in the EGSO lifecycle, an architectural design was specified, using UML component diagrams [82]. 12 sub-systems and 21 components were defined in 3 architectural roles: consumer, broker and provider. The 3 roles implement the tiered architectural style; they correspond with the servant, grid tier and archive subsystems of the ACME model, shown in Figure 16. The design elements' static dependencies and interfaces were specified (though communication methods were not yet defined). Stakeholders needed to be confident that this architecture was complete.

The purpose of the 3 roles is best illustrated by their interaction. Each provider sub-system advertises the data (or analysis) resources that it provides on the network by contacting a broker. The brokers' records of resources are shared, so they can resolve consumers' searches on behalf of scientific users. An essential feature of EGSO, saving traffic to data providers, is implemented by consumers' queries being resolved at 2 levels. The brokers' catalogue is used if the user just needs a summary of existing data, or the broker can forward queries for detailed data content to the provider. With multiple instances of brokers creating the infrastructure, forwarding messages between them, and the triangular relations of messages between provider, broker and consumer, EGSO follows the peer-to-peer style (as defined in Section 2.2). The remapping of consumer queries against a general data model to resources specific interfaces also implements the n-tier style.

As previously described (Section 3.3.3), scenarios were derived from the EGSO requirements to guide development of a dynamic model of the architecture. Of the 8 scenarios that captured EGSO's required core activities, refined from 47 possible scenarios capturing functional and quality behaviour (given in Appendix D), 3 represented system behaviour that could not be modelled satisfactorily (optimisation, security and protocol flexibility; see Section

5.3). Another used an analysis service that could only be modelled in the same way as the data location scenario.

Implementation, validation, limitations

The 4 remaining scenarios (those listed in Section 3.3.3) represented: transparent data location, query resolution by distributed metadata, dynamic resource growth, and query rerouting on provider failure. A single model with concurrent instances of the architectural elements was developed to implement these, creating formal events for the informal descriptions of activities in the scenarios.

Over 6 working days a naive collection of 23 processes derived directly from the architecture were refined to a model with just 10 types of process (including 2 that do not contain their own events, just aggregating others to support the composition of a combined state-space with multiple component instances). Appendix I.1 describes their implementation, with detail of the broker process' logic to resolve queries against a dynamic catalogue. The final model's 32 types of event are associated in a many to many relation with architectural components (Appendix Section I.2 lists the associations). Most events represent interaction between pairs of components. Overlapping sets in the Venn diagram in Figure 17 show how the 8 processes (that are defined by their events) share these events. Sub-systems in the architecture that represented internal mechanisms (hidden by dependent components with interfaces) were not modelled to reduce complexity; they could not affect the safe concurrent progress of the system.

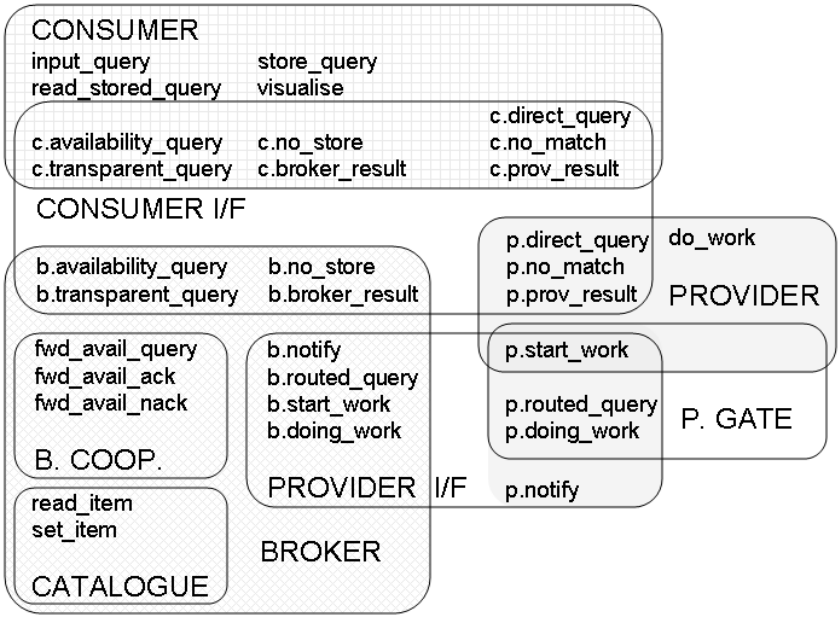

Figure 17: Venn diagram indicating how 8 processes share the EGSO architecture model's 32 events.

A model with 2 consumers, 2 brokers, and 2 providers sharing 3 data sets was animated to demonstrate the concurrent progress of the 4 core scenarios. The model deployment configuration that was tested, with the brokers' deliberately incomplete initial metadata, is shown in Figure 18.

With multiple instances of every role shown in Figure 18, LTSA could not analyze the combined state space of 2^{78} transitions. To test for safety, the model parameters were modified so that there were duplicate instances of just one role at a time (though 2 brokers were always necessary to represent query forwarding). Despite testing roles independently, it could be shown that communication semaphores would be needed to prevent broker deadlock.

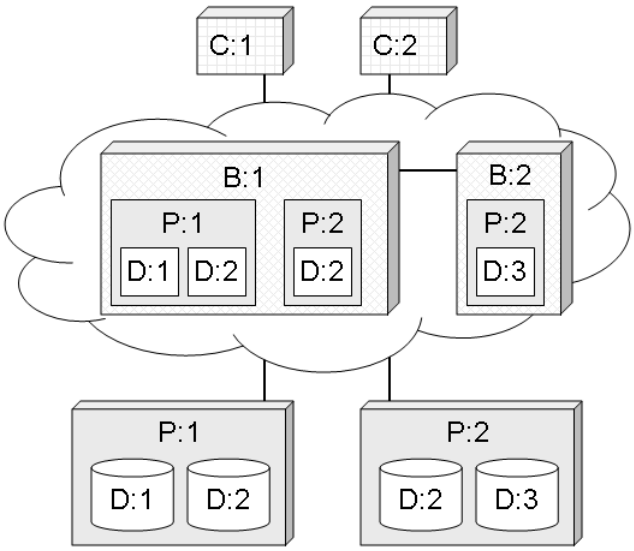

Figure 18: The instance of EGSO architecture deployment modelled.

Observations, communication, model impact

Model constructions used at the previous stage were adapted for this stage. This accelerated development and suggests that a suite of data-grid design patterns could be abstracted (and associated with well known patterns; see Section 7.2).

The differences between the architecture components and model events, and their complex relationship, emphasise the difference between complementary static and dynamic abstract views of a system. The dynamic model hides different functional components that share an interface, whilst the static architecture's component relationships are under-specified at this stage. Still, all 32 events were associated with architecture components, indicating the architecture is behaviourally complete.

By animating the core scenarios, it is clear the implemented architecture hides the complexity of dynamic resource discovery from consumers. As the tests were successful with multiple instances of each role and when resources were unavailable, the architecture was shown to be free from single points of failure, dynamically scalable and robust. In these operations the represented data-grid tasks made concurrent progress without interfering with each other's states.

By basing the model on both scenarios and the architectural components, it functions as a bridge between the scientific users' requirements and the engineers' design. The dynamic model and its test scenarios were documented with the static architecture [82], and all project stakeholders accepted its demonstration that the design would behave well. The model at this stage therefore strengthened the static architecture.

5.5 EGSO interface

Project state, modelling goal

To refine the EGSO architecture, the consumer, broker and provider roles were designed next, along with a common scientific data model and interaction sub-system. Broker interaction is essential for the reliable operation of the system, so this was designed first [24]. Message sequence charts for roles' interaction encompassed much of the system architecture whilst having little domain specific content.

In addition to basic connection and disconnection interactions, the designed messages for each pair of relations in the EGSO roles are:

- *Consumer to broker:* pose a query (to be resolved asynchronously), request the result, check the status of a previously submitted query, and stop a query.
- *Broker to provider:* check an advertised resource is available ('is alive'), forward a received consumer query to a provider, get statistics of a resource's use and – in the *provider to broker* direction – advertise new data records' availability.
- *Broker to broker:* check connection to a neighbour (and therefore connection to the rest of the network) is alive, forward infrastructure housekeeping data (users' query session, providers' and brokers' statistics), and either send unresolved queries or known provider record updates.

This design was evaluated through modelling to determine whether it was complete and sound. Faults found in the design at this stage, whilst other components that depended on the broker were still being designed, would be much easier to fix than later in development.

Implementation, validation, limitations

Messages in the broker design's UML message sequence charts could be directly translated to model events. Hidden events were then added for application functions such as a user creating a query or a database resolving it. The LTSA extension for drawing message sequence charts was not used [106], as it was found to generate FSP code that was hard to manually modify.

3 models were developed over 9 working days. Initially, single instances of the consumer, provider and broker roles were encoded as processes (with a slave broker used as the target of all broker to broker interaction); their 77 event types represented all the designed messages. Appendix J.1 describes the implementation, detailing the terse, systematic naming convention used.

This was refined to a model that captured the concurrent interaction of multiple role instances and symmetrical interaction between broker instances. A semaphore for broker communication was implemented to ensure safety; this had to be claimed by a broker when it initiated requests as well by processes making requests of the broker. To reduce complexity, entity connection and disconnection events were excluded. The behaviour expressed was still richer than the models of previous stages, with 76 event types once conditional paths on query state were implemented. Appendix J.2 describes how process states are used to represent

concurrent messaging and task progress, and why synonyms are needed for opposing ends of processes' safe communications.

A third model was implemented to evaluate query forwarding in an alternative design was still being considered; rather than metadata updates being forwarded between brokers (so that each would be aware of the whole network from local complete metadata), unresolved queries would be forwarded until the network of broker peers collectively fail to make a match against their distributed metadata. (This is the way that searches typically work in peer-to-peer networks; if a node cannot resolve a request using its local resources, it passes it on. If the result must be definitive, as for EGSO queries, a request must keep circulating until it succeeds or all nodes have failed to resolve it). 16 event types, just representing safe broker interaction, were sufficient to model the contrasting properties of the alternative design. Appendix J.3 describes how the model's semaphores make peer interaction safe.

FSP does not precisely represent the message sequence charts; synchronous events can indicate a message exchange, but not direction (from the source process to the sink). Some of the hidden events, implemented for the process acting as a client, were therefore necessary to represent the message's origin. Correspondingly, the other events in the model that were not synchronised between processes, representing the hidden work of a process acting as a server, were necessary to capture concurrent asynchronous progress.

Observations, communication, model impact

The direct association between the designed messages and model events ensures the models' validity. When models were refined, diverging from the message sequence charts, it remained clear which messages were excluded, which decomposed, and which should be added to the design.

Models were tested by animating the message sequence charts, as done for the scenarios in the previous stage. Asynchronous concurrent progress was demonstrated, and events for errors were introduced when paths could lead to undocumented, unexpected states.

LTSA safety checks for the second model proved that the design implemented a reliable service that could not block due to process instance conflict or circular dependencies. FSP progress criteria were used to show that repeating cyclic paths must eventually resolve consumer queries.

The third model showed that a safe solution to reliable query resolution against distributed metadata was more complex than the design had described. Even with a simple ring topology and query parameterisation with the forwarding node, extra query history was required (this could not be easily represented in FSP, and would need cache management to be designed in the real solution).

Therefore, modelling completed at this stage validated the interfaces and evaluated alternative designs. As these designs are domain independent, demonstrating behaviour common to many data-grids, they indicate design patterns: the second model represents a generic resource metadata management solution, and the third a peer-to-peer service discovery network.

The model argued for minor modifications to the EGSO design. 8 undocumented messages had been added, including a necessary indication that a message sequence was complete and alternative responses for error cases. The model also demonstrated the importance of guarding communication between entities that act both as client and server, at risk of deadlock or requests loss. The second model from this stage could have been maintained in parallel with interface development (were distributed teams in closer contact), so that any further design changes could also be validated against the design's original goals.

5.6 AstroGrid objects

Project state, modelling goal

The AstroGrid project had begun detailed design whilst EGSO was at the interface design stage described above. Their object models included descriptions and message sequence charts for classes interacting via public methods that were complex and subject to change. However, by discussing the distributed interaction and exploring design risks with the project's software engineers, the message sequence illustrated in Figure 19 was identified.

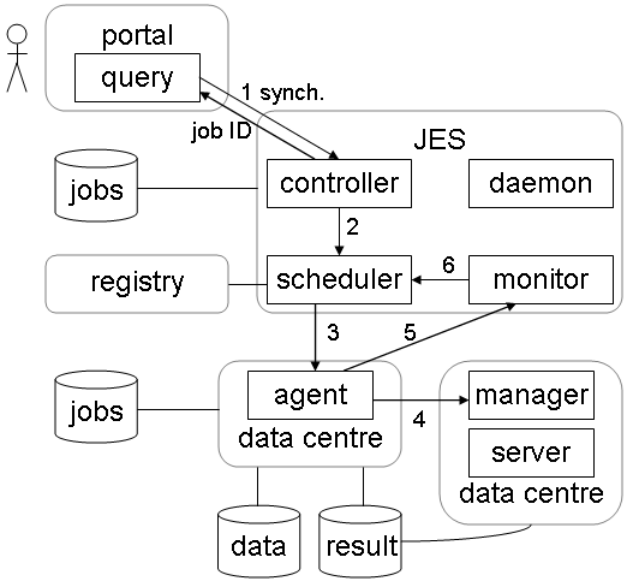

Figure 19: AstroGrid distributed job control messages are passed from the user portal to the data centre, via job-control and gateway tiers.

This sequence of events shown in Figure 19, which can be encoded in a UML message sequence chart, completely captures normal operation of AstroGrid; it spans all the sub-systems, and shows how queries are resolved and analyzed data products delivered – the essential data-grid tasks. Initial models evaluated whether the designed message sequence was complete. Engineers later wanted to ensure that the fundamentally unreliable asynchronous communications model would satisfactorily complete jobs, by identifying inconsistencies and recovering tasks within jobs that had been lost. Messages from the daemon to the JES job management systems objects and job databases, not shown on the figure, should accomplish this.

Implementation, validation

Just 2 working days were spent developing 2 models based directly on the message sequence chart. As at the previous stage, process synchronisation events captured documented messages, whilst added hidden events represented other activity.

The first model implemented the complete message set in 39 events, but only represented a single instance of each of the 9 interacting entities (with an additional process for the job state shared by 3 of the objects). The model given and described in Appendix K.1 is similar to that in Appendix J.1.

The second model captured risky concurrent dependency of the scheduler on both monitor and controller. This represented the 3 processes with additional job state and job factory processes (which hid the data centre and user portal functions) that shared 12 types of events. The model's factory for iterative job generation, inspired by the design pattern [49], is described in Appendix K.2.

Observations, communication, model impact

As for EGSO interface models, the direct translation of messages to events ensures model validity. The first model animated the essential message sequence chart, demonstrating that the completeness of the message set (no more messages were needed to invoke or censure required process activity). The second model did reveal a possible deadlock in the circular dependency of 3 job control processes when there was at least as many active jobs as entities.

Discussion with the engineers clarified that deadlock should not be a risk to their asynchronous messaging, as the job control objects in question do not establish reliable connections; their 'fire and forget' interfaces should be able to ignore messages that would block progress. This behaviour is actually demonstrated in the first model with simpler logic that represents a non-deterministic choice to avoid deadlock.

The refined AstroGrid design includes a daemon that detects inconsistent state and repeats lost messages, though this was not modelled to demonstrate the logical reliability of the design. The engineers were keen to know the expected job recovery time based on factors including the probability of failure and the recovery daemon's schedule. As poor performance is seen as a greater risk than component interaction failure, the stochastic models described in Chapter 6 are more suitable for further investigation.

5.7 Abstract data-grid design patterns

Modelling goals arising from general observations

This step was done once EGSO and AstroGrid development was in progress. So, unlike previous steps, these models were not intended to serve as a technical review evaluating specific software designs. Instead they captured design patterns discussed for both AstroGrid and EGSO, specifically regarding communication strategies and dependency on unknown resources. These patterns would be expected to arise in other data-grid projects (and any

systems with symmetric cooperating components), as they represent essential interaction for distributed dynamic resources. An EGSO model safeguards communication with semaphores, whilst the last AstroGrid model fails to synchronize the distributed state of tasks across components. AstroGrid's design side-steps synchronisation failure by accommodating lost messages, whilst EGSO must implement blocking communication with its critical brokers. It is therefore clear that reliable progress in decentralised, scalable data-grid architecture is sensitive to connector design. However, FSP only represents connection-oriented communication with synchronised event in the models so far.

Some other formal modelling languages (such as the ADLs of Section 4.2) have rich semantics for connector typing, which may capture different data-grid component connection strategies. However, despite the simplicity of FSP and the experience reported here, it may be used to represent connector logic directly. Reference processes that represent different connection types have been developed and used between arbitrary components in system models.

The capture of the design patterns, independent of specific data-grid projects, prompts the reuse of quality engineering strategies. By dynamically modelling the patterns, their value as successful solutions to data-grid challenges should be clearer. This activity takes the abstraction of architectural design to a higher level, beyond technology independence, to project independence.

Implementation, validation, limitations

As in previous models, multiple indexed process instances were implemented, having first defined the events for a non-concurrent system. Less than 2 days of effort were spent on this task.

Unlike previous models, hypothetical grid task state transitions were in separate processes from the communication events. This strategy is analogous to the separation of concerns into communication layers, the established software architecture pattern exemplified by the OSI stack (described in text books [15], and recognised as an architectural pattern [94]). In this way, common connection oriented or connectionless communication patterns could be plugged in between grid service tiers in different model implementations.

In the first model, the grid servers are arranged in the triangular dependency that maximises risk, following the way that EGSO brokers forward metadata or queries and AstroGrid controls jobs. Appendix L.1 describes how 3 service processes share numbered instances of a generic connector process and claim a task process. Complexity arises from the synonym mappings of services' shared connector events and additional task sequence states, but the final model is terse enough for customisation to be clear when it is reused.

In the second model, client and server roles are simply used as the source and sink of messages; this is sufficient to test both ends of a protocol interface. Appendix L.2 describes how the message loss in the stateless connector is modelled.

The scenarios used to demonstrate successful model implementation at this stage could not be derived from other project artefacts. Therefore the models were only tested with

hypothetical data-grid tasks, representing user service requests to unknown resources being resolved.

Note that FSP represents the real network characteristics of connectionless protocols poorly. It was necessary to add some events to demonstrate lost messages; however, on real networks, messages may be lost without the implementation of any special functionality.

Also, separating events' communication and application layers to different processes greatly increases the combined state-space that LTSA has to analyse. With 6 tasks on 6 nodes, LTSA generated 2^{104} states, compared with 2^{46} for the equivalent final AstroGrid model from the previous stage. Were a system to be built with multiple generic patterns, LTSA would probably not be able to analyse it.

The combinatorial explosion of state-space complexity is an inevitable consequence of event-oriented process analysis. Different tools employ diverse pruning strategies to simplify combined state spaces (cutting off areas that are equivalent). LTSA identifies hidden 'tau' state transitions that are not shared by processes (clearly identified by the naming convention used for events in the models of Section 5.5, discussed in Appendix J), but it may be that the strategies used by the other tools mentioned in Section 5.1 would perform better. However, careful construction and evaluation of the models can help. As noted in Section 5.4, the number of process instances can be modified to assess the consequences of their concurrency independently. The models of Sections 5.5 and 5.6 were refined to concentrate on just those events with complex interaction. Beyond tau events, linear sequences of events representing communication between processes that did not have side effects on the behaviour of other event-paths for the involved processes could be ignored. The risky aspects of designs in concurrent operation of multiple instances of their components could be demonstrated without the safe paths that inflated the state space. These experiences inform recommendations for a general method for implementing useful models, discussed further in Section 7.3 and demonstrated in the Appendix G tutorial.

Observations, communication, model impact

Others have noted the reuse of published design pattern in AstroGrid [116]. Several steps would be necessary before the data-grid design patterns could achieve the same recognition. They would have to be expressed in a more accessible way, such as UML class charts and message sequence diagrams. It would also be necessary for them to have been proven in deployed and maintained production systems for other engineers to gain confidence in their strength.

However, the models do demonstrate that, in principle, reusable data-grid patterns can be expressed in FSP. These could form templates in a toolkit that supports data-grid design modelling, helping engineers to avoid some of the common pitfalls of decentralised architecture.

Chapter 5 key points

- FSP, by formally representing shared events of concurrent processes, can be used to describe high-level data-grid designs.
- LTSA analysis of the FSP data-grid models was able to validate static designs by proving they could implement the required dynamic behaviour; they could be demonstrated to other stakeholders to raise their confidence.
- Model analysis revealed design oversights, specifically for the case studies: the complexity of peers in symmetric roles as either client or server and missing messages in planned interaction sequences.
- Generic data-grid components' interaction can be captured in FSP, indicating how future system may be reliably designed from reusable parts.

Chapter 6 Simulation

Discrete event simulation (as an instance of the simulation view introduced in Section 2.2) is a recognised method for modelling computer systems. Results from such simulation can only estimate behaviour, not inductively prove qualities (which the formal modelling methods described in Chapter 4 and Chapter 5 can). However, it was desirable to evaluate this complementary method on behalf of EGSO and AstroGrid's stakeholders once formal techniques for dynamically modelling grid-architecture had been investigated and their limits found.

Functionality was introduced to LTSA version 2.3 to analyse stochastic FSP – a language extension in which events are annotated with probabilities and timing. It was therefore natural to move from the models described in Chapter 5 to stochastic FSP models. An account of modelling the AstroGrid design problem from Section 5.6 is given in Section 6.1.

The stochastic FSP experimental method was contrasted with a more traditional simulation language that follows procedural and object-oriented programming paradigms. SimPy, a toolkit for the Python scripting language, was used. It was chosen over other languages for the reasons given in Section 6.2.

SimPy models were used to further investigate the EGSO broker design choice described in Section 5.5, still undecided after early iterations of implementation. Section 6.3 describes how useful experimental results were derived from the models, even though no realistic performance indicators were available. Section 6.4 examines more complex broker network topology and Section 6.5 a refined messaging model.

As the designs of both EGSO and AstroGrid are evaluated in this (and the previous) chapter, these 2 different design solutions in the same problem domain are compared in Section 6.6.

6.1 Stochastic FSP models of AstroGrid

Experimental application of stochastic FSP to the AstroGrid design problem failed to generate a useful model. This negative result was aggravated as it was found only after detailed problem analysis. The following description therefore just gives a brief outline of the procedure followed for implementing the model. A list of the problems found, which support the disappointing conclusions, is then given.

The goal of experimental design for simulation (introduced as a modelling method in Section 2.2) is to uncover non-trivial relations between design properties and behaviour. The simple textbook example of a simulation to find the expected service time at a resource shared by several clients is unlikely to capture behaviour that cannot be logically demonstrated. The elapsed response time is determined by the expected number of jobs in the resource's queue and its response time for each job; the number of submitted jobs is itself determined by the number of clients and their submission interval. Therefore, it is not necessary to use simulation for such a simple system.

However, this example was modelled in learning stochastic FSP, given in Appendix M, and it demonstrates the apparent economy of the language. The simple queue model applies to data-grids as each service node or tier is equivalent to the shared resource (even if they are clients or portals to other sub-systems). The busiest or slowest node would be the bottleneck, which simulation could demonstrate.

A stochastic FSP model of the AstroGrid job scheduling sub-system was implemented to satisfy the engineers' curiosity about the impact of the recovery daemon on performance (noted in Section 5.6). Clocks were added to a simplified model of the sub-system, which took some time to debug. Both the regular FSP code and the version with simulation measurement are given in Appendix M (where the descriptions of the presented models show how clocks are introduced to time event transitions).

The model evolved to capture the unreliable communication link between the job-scheduler and service providers (representing data resources or analysis computing hosts). Probabilities could be added to these events to simulate message lost, hiding the possible causes of loss on a real data-grid network. These are the final models of Appendix M (which, as described there, encode the message-loss chance with the clocks for the jobs' event transitions and the daemon along with the experimental measure).

Together, these iterations represented 7 days implementation effort without achieving a model that would pass more than preliminary testing. Compared with previous FSP modelling experience, this did not represent rapid design evaluation that returned value in demonstration to other stakeholders. As development had moved on significantly in over a month of elapsed time, stochastic FSP modelling of AstroGrid's job scheduling mechanism was therefore suspended.

Many problems were encountered in this activity, going beyond the previously reported programming and debugging difficulties of FSP modelling. The compiler regularly reported errors that could not be traced; these were assumed to be generated when LTSA could find event sequences that took no simulation time (though these should be acceptable if the events are not in a measure). When FSP had variables defined within an optional event list, which it seems could not be interpreted, LTSA 2.3 itself failed (generating Java runtime exceptions rather than catching a FSP programming error, such as 'out of scope'). These weaknesses of the tool (relative to previously reported LTSA experience) suggest stochastic FSP cannot currently be used to drive down risk in the development lifecycle of real-world projects.

The language and tool constraints also forced several compromises on the intentions of modelling the essential behaviour of data-grids. A model that was simple enough to be analysed by LTSA captured so little design detail that many architectural components were only implied by the events they triggered; their functionality, which would influence simulated behaviour, was not represented. The tool's fixed in-build continuous distribution functions (including even and Gaussian and distributions) did not easily allow customised experiments (for example, of step and linear increases in load over a simulation). The difficulty in modelling common structural programming constructions also limited modelling capability. For example, the process that represented the job recovery daemon had an unrealistic omniscient view of task status; to capture the event interfaces of the tasks' processing components, tasks were represented by

processes that held state, so the task data used for recovery effectively had global scope. One unrealistic effect of this was that job recovery could be initiated before communication events were completed (in reality, implementation would almost certainly store the job data after its dispatch). In the final model of unreliable communication, the represented lifecycle of AstroGrid user tasks have also been broken into separate state transition models for the job manager and service provider. With such a split, the process event model no longer captures the design pattern of distributed state transitions that characterises data-grid operation; the method is failing to represent the essential features that should be modelled.

For these reasons it was concluded that stochastic FSP is not suitable for evaluating data-grid designs. The tool support is not mature enough for the necessary rapid model development. Unlike previous models in LTSA, it is also unclear how analysis is achieved when demonstrating stochastic FSP (even though the graphical representation of simulation statistics is good). Better interactive animation of timed and random events and a clearer diagrammatic paradigm (beyond annotated state graphs) are needed. These would be necessary to convince a modeller's FSP-illiterate peers that simulation test results derive from a realistic representation of the system.

Though traditional simulation languages, discussed in the next section, do not typically have such strong tool support 'out of the box', engineers find their results easier to accept. This could be because these languages' representations of software elements are perceived to be closer to real code. Using complete and mature programming languages for simulation also avoids many of the development difficulties noted with LTSA.

It was noted above that simulation is not necessary where behavioural properties can be determined (rather than emerging from complex unpredictable combinations of effects). This is true in the case of the AstroGrid job scheduler and task recover daemon, and their effect on unreliable communication. To simplify reasoning, the reasonable assumption can be made that time lost due to message failure is significantly greater than network transmission time, database lookup time and the management processes execution time. The average job service time will therefore increase (over normal service time with reliable communication) by the recovery daemon check interval multiplied by the probability of error. Having determined this without simulation, the best design recommendation for AstroGrid engineers concerns operational functionality; they should provide monitors to measure the error frequency and an administration interface to adjust the daemon check interval. This will permit system administrators to tune the job time lost due to message recovery.

6.2 Choosing SimPy

Discrete event simulation tools have evolved over a long period and exist in many forms. Unlike system prototypes, simulation models represent systems' environment as well as their behaviour, enabling experiments that would otherwise be difficult and costly to reproduce. Also, unlike the other 3 model types identified in Section 2.2, simulations can be developed to be as rich as necessary, becoming as complex and functionally complete as the delivered systems. In choosing a tool, familiarity with the Python language and interpreter weighed in

SimPy's favour. Anyone with such familiarity with a programming language should be able to rapidly develop models for a quick response in technical design review using simulation tools for that language. It also had several specific advantages that other simulation frameworks lacked.

- SimPy is a relatively new SourceForge project [140]. This alone implies it will be easier to use on current platforms than historic simulation languages, such as Simula [119], GPSS [129] and Simscript [114].
- It is also easy to find demonstrations of SimPy being used for complex systems (following links from the project homepage). These go far beyond textbook simulation test cases or teaching exercises to show that the toolkit is valid and comprehensive. Other novel simulation toolkits, such as Yaddes [83], Swarm [145] and Spades [136], have not been proven to the same degree.
- It is also not intended for specialist use to one area. Along with some of those already listed above, toolkits like Atlas, Eagle, PDES and SMPL can be discounted on this criterion.
- Most importantly, SimPy is lightweight, as Python is an interpreted language that does not need compilation. Python's native support for advanced data structures, notably dictionaries, and programming techniques, such as lazy typing, accelerate development. This is in contrast with C and C++ simulation toolkits (such as Parsec [135] and COST [123]). It may even be more lightweight than Java, and therefore preferable to specialist Java techniques and toolkits (such as Silk) [53].

SimPy may be criticised for using a scripted language for simulation, a computationally intensive application. In practice, though, the time spent implementing a model of an early design is likely to be much greater than that spent running simulation experiments. Also, no significant execution time penalty was found for anything except benchmark high-performance problems if byte-code optimisation is invoked.

6.3 A SimPy model of EGSO's broker design

Implementation

The goal of simulating the EGSO broker interaction was the evaluation of alternative designs for network communication. The team's software engineers had not decided whether it was better for broker peers to forward their metadata to each other, or forward unresolved queries. In the former case, any broker could immediately direct any consumer query to a suitable provider; in the latter, consumer queries would be passed on until they could be satisfied.

The superior design should meet operational requirements for performance (minimising response time), scalability and reliability (minimising failure). It is assumed the last of these would be satisfied by there being a peer network of brokers; no broker should fail a query unilaterally, so they collectively make every effort to satisfy consumer requests. Performance and scalability are interrelated (when responsiveness falls with growth); they are also at risk in poor design and therefore worth modelling. However, it would be a challenge to simulate the

behaviour of a network that had not been deployed – which therefore had no concrete performance data.

The solution therefore examines relative message volume for the alternative designs. The time taken to pass messages over the EGSO network (built on an international scale over the Internet) was assumed to be much greater than broker metadata lookup or other sub-system function execution times. This meant it would only be necessary to analyse the number of messages passed in a simulated network. In initial modelling the simulation clock would only be necessary to order messages' sequence, not to represent true time.

Poor network performance would be indicated by more broker-to-broker messages relative to the volume of user traffic (between brokers and either consumers or providers). Performance falling sharply as the network grew would represent poor scalability. Specifically, a scalable solution should provide a less than order N relation between network size and performance.

The simulation model was written and exercised with just 1 day's effort, running experiments with a range of parameters to demonstrate the design relations described below. (The first proof-of-concept development iteration, with inflexible behaviour parameters, was designed and implemented in 3 hours.) It was developed in the simplest possible way, by implementing a global representation of the metadata, which broker instances had partial visibility of for resolving queries. Note that significantly more time was spent finding an abstract system view appropriate for design evaluation with modelling – this will be avoided when reapplying this method in the future.

The simulation code is given in Appendix N, with an explanation of how it works (subsequent SimPy models are also listed and described there). The documentation accompanying the code of the first model, in Appendix N.1, describes how the event simulation method toolkit is used through class inheritance. Appendix N.2 describes how the parameters make the refined model flexible enough to conduct the experiments described below. Details of the behaviours encoded for the alternative broker message forwarding designs are also given.

Validation

Initial runs displayed every event to verify the desired sequence was occurring. Subsequent experiments ran for 10000 'user' (Consumer and Provider) queries, and just the total Broker message count was presented.

It was assumed in coding the simulation that the size of the metadata and variability in the timing of user queries did not affect the performance measurement; this was confirmed by a dedicated model, which generated the results presented in Figure 20 (all experimental data is given in Appendix N). To simplify coding effort, it was also assumed that the topology of the network did not significantly affect performance; this is examined in more detail in the next section.

Note that there is a slight rise or fall (depending on the algorithm) in network traffic in the first few data points as metadata increases. This represents the initial lack of user queries (being forwarded or not) at the start of the experimental run when providers are filling up the initial metadata.

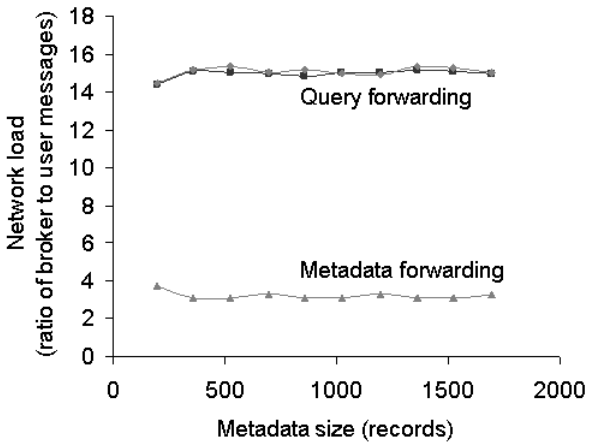

Figure 20: The ratio of broker-broker messages to user-broker messages is consistent as broker metadata on provider data resources grows, for both algorithms. All experiments used a 20 broker ring network with a provider update rate of 0.2 with respect to the consumer query rate.

Experiments

The experiments demonstrated that the ratio of broker to user traffic increases in proportion to the number of brokers, predicting a reduced performance for a larger network. This is the expected result: consider that if metadata on provider content is never forwarded, the expected time for a forwarded query to be resolved is half way round the network. If the broker network doubles in size, each user query would be expected to require forwarding twice as much.

The linear growth of the inter-broker traffic was observed for both designs, as shown in Figure 21, demonstrating that neither design is truly scalable. Ultimately this is because these data-grid designs intend to propagate every client message to every metadata repository. It seems that without centralizing metadata, in a shared database or at a blackboard server for example, there is no architectural solution that avoids at least one message per broker. Yet centralised solutions are not desirable for data-grids as they introduce possible single points of failure that impact reliability.

It is a concern that such growth in network traffic volume would prevent the network doing real work, if service time and message queuing meant all network resources became fully allocated to inter-broker messages. As there is currently no information about service time, the number of concurrent connection threads per broker or whether application level message queues are planned, the network size at which this critical limit to scalability is reached could not be simulated. (A high estimated order of magnitude for EGSO's use is for 10000 queries and updates per day, translating to user tasks being 10 seconds apart. Intuitively, the distributed resources should have no trouble processing such throughput; the extra load of inter-broker messages does not increase the number of messages that each node is required to action.)

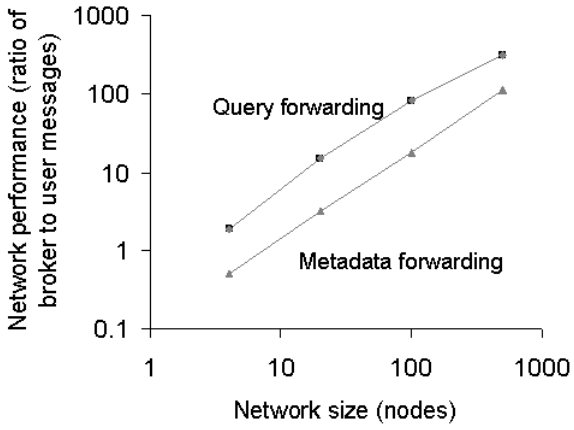

Figure 21: Linear rise in the ratio of broker-broker messages to user-broker messages with respect to network size (the number of brokers connected in a ring). All experiments use a provider update rate of 0.2 with respect to the consumer query rate. Note, broker message volume is deterministic when provider updates are forwarded, so a single simulation run is sufficient.

The simulation model also demonstrates that as the ratio of provider updates to consumer queries increases, the design that propagates provider updates becomes less efficient – because it starts to generate more inter-broker traffic than there are forwarded queries. This relation is demonstrated in Figure 22. Note that the ratio at which neither design is superior is slightly less than 1 query to 1 update; queries stop when they are resolved, whereas updates must reach every node.

Given the simulation has demonstrated this relation, the choice of design is confirmed to be dependent on the expected ratio of queries to updates. Brokers in the early system deployment should log the observed client usage to optimise the design in later versions.

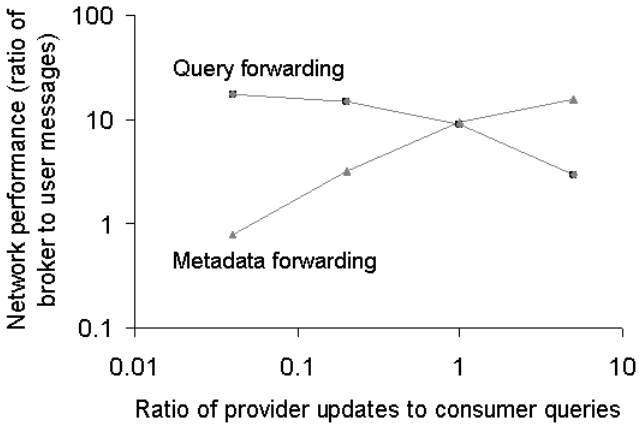

Figure 22: Converse linear rise and fall of broker-broker messages to user-broker messages for provider update forwarding algorithm versus consumer query forwarding with respect to provider update rate (with respect to the consumer query rate). All experiments use a 20 broker ring networks.

6.4 Broker network topology

The broker networks used in the simulations described so far had two non-symmetric connections per broker. This represents the EGSO design criteria; when a broker joins the network, it connects to two peers (and reconnects to another if one is lost). The number of peers that become connected to an individual broker is not designed, and is not planned to be administered. As simulated, each broker's second connection was to the first broker connected to its first neighbour, a regular arrangement represented in Figure 23.

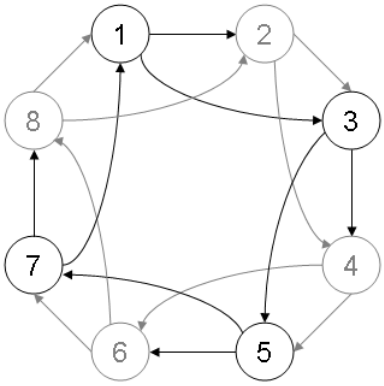

Figure 23: An illustration of connections between brokers in preliminary simulations. Note that the links are not bi-directional.

The simulated networks that produced the results in the previous section were therefore unnaturally regular and maximised the minimum number of links between an arbitrary pair of nodes. The query-forwarding algorithm may naively be expected to improve in a more random network, as it may be satisfied at a broker distant from its point of submission more quickly. The simulation code was adapted to generate more random network connections. The network was constructed by making one ring of connections – the necessary longest route around a fully connected network – then choosing a random distance along this ring to the second (different) neighbour. Appendix N.3 describes (alongside the code) how modelled networks were connected and then analysed using the Python language.

Networks generated in this way did decrease the average number of hops between brokers; Figure 24 shows this measure of connected proximity tends to an order log-N relation to the number of nodes, demonstrating the true scalability of peer networks. Note, this results goes against the 'small worlds' theory [101] which holds that a little randomness reduces the maximum number of hops between nodes more than great randomness (which is also used to argue that you are only 7 handshakes away from anyone in the world, and is applied to the Internet). This unexpected result may be due to this network only having 2 one-way connections per node, or being relatively small.

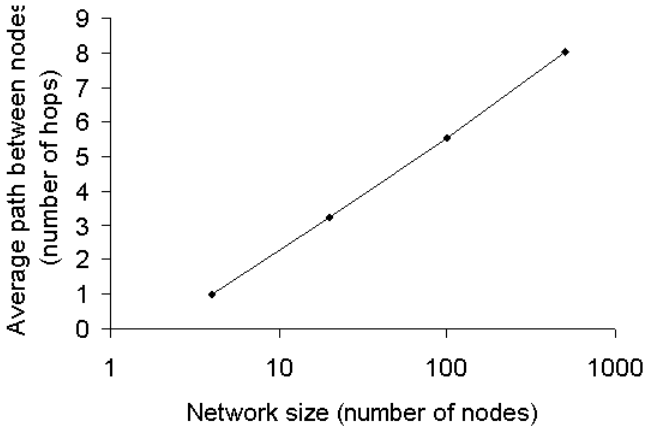

Figure 24: The minimum average number of hops between arbitrary pairs, for the most random broker network connection, grows in proportion to the logarithm of network size.

However, the simulation demonstrated that greater connectivity did not significantly increase the performance of the query propagation algorithm (shown in Figure 25). This is because a quicker query resolution only reduces propagation from the node that had the matching metadata; other nodes have no visibility that a match has been made and continue to pass on the query.

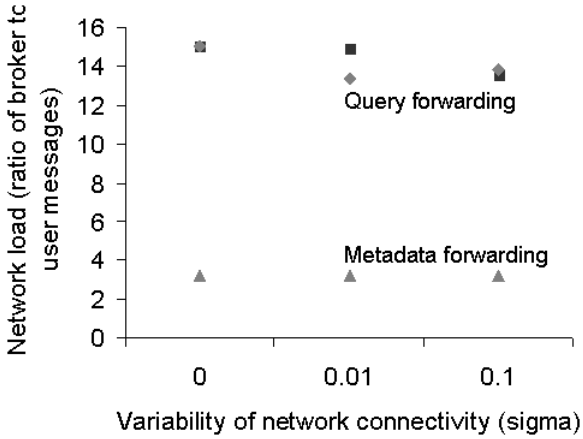

Figure 25: There is only a slight decrease in the ratio of broker-broker to user-broker messages for the consumer query forwarding algorithm as the randomness of network connection between brokers grows. All experiments use a 20 broker network with a provider update rate of 0.2 with respect to the consumer query rate.

6.5 Refining broker simulation

When the findings of the simulation described in Section 6.3 were presented to EGSO software engineers, they wondered whether an algorithm could be found that was more scalable. Another simulation model was rapidly implemented to test a hypothetical design. Could stop messages from the node that had quickly resolved a query in a well-connected network to other brokers save traffic to increase performance? (Such optimisation would have

no effect on a provider update forwarding design in any case, as every provider message must always reach all brokers.) Appendix N.4 presents the model's code, with descriptions of its parameters, of the data structures used to log consumer queries' states, and of how consumer and message processes interact with the broker network.

However, in experiments, where the stop messages themselves also count as inter-broker messages, the ratio of broker-to-broker messages to user to broker messages actually rises still further in this design. Though tests demonstrated that the stop messages prevented resolved queries reaching some broker nodes, the average number of messages was proved to be greater than it would be if the query reached all nodes.

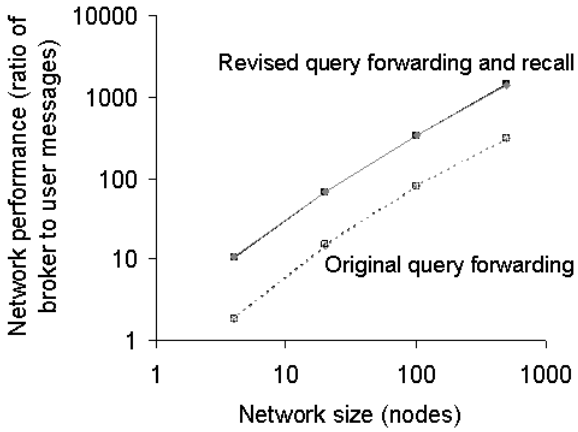

Figure 26: Simulation results showing significantly more broker-broker messages in proportion to user-broker messages, growing in a linear way as network size increases. Only consumer query submission (not provider content update) messages were simulated on a randomly (fully) connected broker network.

This result (shown in Figure 26) should be expected from design analysis. Consider that if a single broker can resolve any given query, the expected number of brokers visited before the query is resolved is half the network size. In a well-connected network, half the brokers will be reached only at just less than the average number of hops between any two nodes. The stop messages would therefore have to propagate over more connections when retracing the steps of the query than the query itself would have continued to cover had it not been stopped. This is demonstrated in Figure 27, which shows backtracked stop messages and the saved forwarded queries (numbers for broker nodes indicate the number of hops the query has travelled). The query from the grey node 0 is matched at the other grey node 3 hops away. There must then be at least as many recall messages as original query-forward messages, indicated by the shorter arrows, whereas few dashed query-forwards to the remainder of nodes 4 hops from the source can be saved.

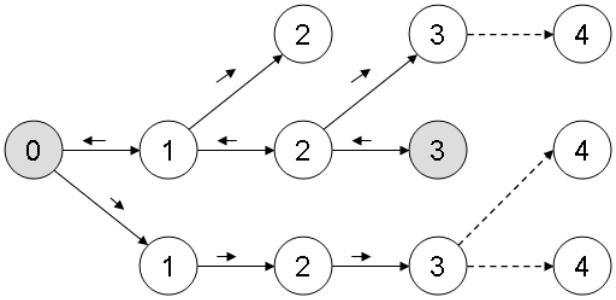

Figure 27: Demonstration of the principle that there are more messages between brokers to stop forwarding (shorter arrows) after half the nodes are reached than forwarding messages saved in a well connected network (dashed arrows).

Note that in this simulation the clock comes closer to representing real time. The broker stop message service rate was configured to be 16 times the query rate, so brokers 16 hops away could be expected to be stopped before the query was forwarded once. Though any figure for the rate of message transfer for priority stop messages in relation to regular message transfer must be a guess, this parameter choice does not affect the finding that stop messages do not improve scalable performance.

In conclusion, simulation of EGSO broker networks demonstrated performance properties, validating hypotheses derived intuitively from the design. It has therefore been shown that simulation can evaluate the designs of complex system before the main cost of implementation is born. The EGSO broker models clearly demonstrate behaviour that the software engineers had been unsure of. (To find a truly scalable design, peer-to-peer network designs could be investigated further, perhaps implementing a forwarding hierarchy or a horizon for propagation. However, such solutions are not really necessary for the EGSO broker network, which may only be expected to have of the order of 50 nodes.)

6.6 AstroGrid and EGSO compared

Chapter 3 showed how AstroGrid and EGSO requirements complemented each other. Both having been modelled through simulation and event analysis, their apparently divergent architectural and detailed design can now be compared. On closer examination, from the perspective of dynamic modelling, their solutions are strikingly similar; common design patterns for shared problems emerge.

6.6.1 Static architecture comparison

The preliminary static architecture of EGSO was stated in the architecture document [82]. As detailed design proceeded, the 3 roles of consumer, provider and broker were faithfully followed, though some details of sub-systems were modified (in separately documented designs) and the interaction subsystems emerged as a stand-alone component.

The preliminary AstroGrid design was more fluid [115], and it was not until design was divided amongst development teams that a stable overall architecture emerged [118]. Its static components may be grouped into user-facing, resource-facing and infrastructure roles to be compared with EGSO's 3 roles – consumer, provider and broker:

1. AstroGrid's portal, community services and myspace shared repository seem user-facing.
2. Its data access and other exposed analysis tools encapsulate the resources.
3. Its registry, workflow and scheduling components, messaging infrastructure and authentication service may be classed as infrastructure.

However, there are many questionable simplifications in any possible comparison between components, for example:

- The AstroGrid myspace services are expected to be deployed at both user and resource servers (with myspace location being an infrastructure function). This makes it like both the consumer repository and provider data manager of EGSO.
- The EGSO consumer's general responsibility is exactly that of the AstroGrid portal, yet the portal is imagined as being deployed on infrastructure servers, making it more like the EGSO broker.
- The AstroGrid registry is the essential and clearly delineated component for resource location, yet in EGSO both providers and brokers manage metadata about resources. It may seem that either EGSO's engineers have missed an opportunity to group functionality or that AstroGrid's have forced risky centralisation in a distributed domain.

6.6.2 Dynamic behaviour comparison

In contrast to such problematic comparison of the static design views, the dynamic view of the sequence of events that both networks implement demonstrates how similar their solutions are. 8 different shared functions are identified below.

Reliable asynchronous messaging

Both provide a reliable asynchronous messaging service, exemplified by the EGSO interaction subsystem [24]. The AstroGrid user message queue component provides only part of that function; the modelled task request rescheduling on recovery is another aspect.

Distributed task state transition

The state transitions of each task requested by network users occur on different entities. As already noted, distributed state models are characteristic of grid systems in general [7]. The models imply that AstroGrid has captured this more explicitly, as the job entry subsystem's daemon maintains distributed state consistency. However, those models are for a later design stage. It may conversely be noted that EGSO's broker interaction model had better specification of distributed provider updates earlier.

Metadata propagation

Resource metadata propagation strategy alone represents a characteristic data-grid design challenge that EGSO and AstroGrid have both been forced to tackle. An early EGSO use case concerned network failure on accidental bad metadata propagation, raised again in

security analysis. AstroGrid planned tiers of registry reliability, with core servers standing between users' locally held favourite resource references and data providers' live catalogues. The simulation models show the EGSO solution is not perfect, even though it seems better defined than AstroGrid's. Ultimately, large-scale data-grids may only succeed if they move closer to peer-to-peer network architecture.

Reliability from unreliable networks

Completing user tasks reliably on a network with regular expected element outage is another project-independent data-grid problem related to distributed state transition. The EGSO user query logging in the broker network was intended to support reliability, but was initially undefined whilst core functionality was developed. In contrast, AstroGrid workflow decomposition, task management and job dispatch and recovery were well defined and evaluated in the models. Ultimately this problem is not data-grid specific; there is a long history of design strategies for reliability in computer systems that neither project has exhausted.

Supporting cooperative working

Both projects implement solutions to help users share results, in line with the virtual organisation vision [8]. AstroGrid again captures this more explicitly with the myspace family of services, whilst EGSO only has the principle of lowering the barrier for users becoming providers. However, on closer inspection, EGSO supports very egalitarian peer networking at the broker level and in the refined consumer design [70]. It seems data-grid engineers should bear in mind that the concept of virtual organisations does not just apply to users but also to resource providers, including the infrastructure service providers.

Transparent resource access

The primary function of data-grids may be providing reliable homogenous access to heterogeneous unreliable resources. The many types of distributed system transparency implied are well met by all of the EGSO roles working together, but especially the broker network, which guarantees consistent service. In contrast, the homogenous interface and resource description are divided and limited to the portal and registry components of AstroGrid. The AstroGrid design and planned deployed workload for transforming resource metadata to a common schema is also lighter. Though other AstroGrid services are planned to be implemented homogenously across the deployed network, this may make it less flexible in future development. In general though, when tracing activity across designed components, both systems do emerge as successfully providing the essential data-grid function of transparent access.

Components acting as both clients and servers

Both EGSO and AstroGrid implement infrastructure components that act simultaneously as servers and clients, a clear feature in the LTSA models where concurrent communication was a commonly identified risk. The messaging strategies of reliable asynchronous messaging support this design characteristic, but the feature may also represent an emergent data-grid

design pattern closely related to middleware tiered architecture. A component in a data-grid system can be identified as a middle-tier entity if it receives requests from users or peers and responses to requests from resources, as well as sending on requests and metadata updates itself. Once identified, engineers know that established data-grid solutions to the problems of concurrent communication, metadata validity and so on can be applied. Section 7.2 describes the metadata relay aspect in greater detail.

Only medium scale infrastructure

The middle-tier's role is made more difficult in data-grids by quality of service requirements, notably for reliability and scaleable performance. Both EGSO and AstroGrid are committed to providing some strength in planning a distributed infrastructure deployment with inter-domain communication. However, neither provides a true Internet scale solution, as demonstrated for EGSO through simulation. Given the scientific applications typical of current data-grids, there must be further iterations of project design, modelling and deployment in larger domains before such requirements are fully met.

By modelling both systems, their behavioural commonality is clear. It has been demonstrated that fair design evaluation must use such a dynamic view, going beyond comparison of static architectural descriptions.

A crude generalisation would note that EGSO's broader resource base has generated a more flexible solution, whilst AstroGrid's bottom up approach should deliver higher quality components. In conclusion, AstroGrid and EGSO have taken different paths to implement the same functionality for shared requirements. This is an ideal situation for experimental data-grids, as they therefore provide different software engineering solutions that are both well designed to a novel domain. Users will ultimately gain the benefit of the best solution through evolutionary evaluation of competing designs' fitness.

Chapter 6 key points

- The stochastic language extensions to FSP have great scope to economically link event modelling and discrete simulation, but proved difficult to use.
- Conversely, the SimPy simulation extension to the Python language proved easy to use, and developers appreciated the value of procedural EGSO models.
- Data-grid simulation could demonstrate how designs affect the relations of behavioural qualities (notably network size to performance) before statistics were gathered from real implementations.
- Reasoning about general network designs derives hypotheses that are supported by simulation; unexpected properties (notably, the small world effect of networks with few random connections) do not always emerge.
- Despite apparent static design divergence, EGSO and AstroGrid implement similar behaviour (which is apparent in the models); both represent fit early solutions to the evolving data-grid challenge.

Chapter 7 Further observations

The previous chapters discussed and drew conclusions about the modelling methods as their applications were described. This chapter raises points that go further in assessing the research. They are grouped in 4 topics: observations about the modelling technique, design patterns for the data-grid domain in general, an emergent modelling methodology and further conclusions about the research's domain contributions.

7.1 Modelling

Observations presented in this section go beyond the evaluation of the models' value given after the descriptions of their use (in Chapter 4 for ACME, Chapter 5 for FSP, and Chapter 6 for SimPy). The discussion topics answer the following 5 questions:

- Should more rigorous formal analysis of data-grids be applied?
- Why have dynamic models rather than static models been used for the majority of the analysis?
- Has the essential architecture of data-grids been captured by the models?
- What are the different benefits of process models versus discrete event simulations?
- Has modelling been valuable to data-grid systems' development overall?

Stronger formalism

The presented research, evaluating data-grid designs, centres on FSP analysis of planned systems' behavioural properties (see Chapter 5). Though the FSP language enables formal analysis, its application in this case is not completely systematic. For example, different methods to map design elements to FSP processes have been used, and concurrent progress checks are not implemented and validated (Appendix G.4 demonstrates such testing). The method of model validation most commonly used involves manually tracing event transitions to test whether the intended functional scenarios are reproduced by the cooperating components. The FSP models are therefore being used as prototypes, enabling very early system testing of the systems (see the next topic "static versus dynamic models" concerning testing). The strength of support for designs is therefore only inductive, not deductive, just like the simulation enabled by SimPy (described in Chapter 6).

There is therefore scope for more systematic analysis using formal event transition analysis. As the encoded FSP models were at the limits of the LTSA tool's ability to compose and analyse, other tools may be investigated (for example, those cited in Section 5.1). As well as analysis of combined state space, deductive reasoning may be used to determine system behaviour from design. Analysis could step through rules taking the general form *"if the system design has this property, then it has that behavioural quality"*. However, it may be difficult to find the right level of detail that is general enough to capture implementation independent design whilst containing enough detail to analyse. Candidate abstract and concrete rules would be: *"if the design is decoupled, it is flexible"* and *"if the mediator pattern is used, resource behaviour*

can be modified without requiring its peers to change". Validating such rules would also be problematic.

However, whether more rigorous event modelling methodology is followed, stronger analysis tools are used, or system properties are deduced from atomic rules, there must still be a compromise between the analysis effort and tangible benefit. It has been shown that stakeholders understood and responded to the findings of the event modelling as it was done. Also, models could be generated quickly enough to pass on the benefits of their findings at each stage. In contrast, the formal architectural specification (Section 4.2) proved more inaccessible (than the animated FSP models), and stochastic FSP modelling (Section 6.1) was relatively laborious (compared to discrete event simulation with SimPy). The effort invested in data-grid modelling with FSP is therefore appropriate. Other innovative domains where very high qualities of performance are not necessary (see Section 3.3.1 on EGSO's NFR) should also benefit from a similar low level of systematic analysis.

Static versus dynamic models

By using FSP and SimPy, the research focuses on dynamic rather than static models of proposed data-grid systems. LTSA analyses the FSP models of event sequences to demonstrate safe progress, whilst SimPy simulates cooperating process instances to reveal emergent behaviour. Dynamic and static abstractions are introduced in the review of software engineering (Section 2.2), and examples of static descriptions of EGSO and AstroGrid are cited (both used UML component and class diagrams; see Sections 5.4, 5.5 and 6.6.1). The dynamic descriptions of the conceived systems are strongly supported by the static models; for example, the FSP architecture model is associated with sketched component diagrams (Section 5.4 and Appendix I.2). However, static descriptions alone could not always determine the analysed dynamic models' implementation; scenarios and message sequence diagrams were more useful in several cases (as discussed in Sections 5.4, 5.5 and 5.6)

Therefore, dynamic modelling can complement static representation, but there is also evidence that it should be used. Stakeholders of the data-grid projects need behavioural understanding of their innovative systems at least as much as clear vision of the systems' structures; this indicates why they appreciate dynamic modelling. Both those representing the customers and the lead developers in the EGSO and AstroGrid projects discussed how the system would work as they explored requirements and design, as apparent in:

1. the narratives accompanying sketched EGSO architectures (Section 4.1),
2. AstroGrid sub-system interaction (Section 5.6),
3. use cases (see Section 3.2 and 3.4.2),
4. message sequences (especially in interface design, Section 5.5).

The informal expressed descriptions of 1 and 2 and standard methods of 3 and 4 represent the behaviour of the envisioned systems. They contrast with the static structures, which were also used, for example in:

- the decomposition of required capabilities in the sketched architectures,
- the UML architecture's component dependencies,
- class diagrams with method signatures.

Therefore stakeholders need the stories of operational sequence to support the static data and interface specification. The dynamic view helps understanding and the sharing of ideas about innovative systems' needs and composition.

Additionally, playing out scenarios through animation of a dynamic model of the planned system is analogous to prototype testing very early in the software lifecycle (demonstrated especially in Section 5.4, testing the EGSO architecture model with scenarios). When there is a chance of early introduction of faults to a project (as is the case for a new technology), any techniques that debug those faults greatly reduce the cost of the changes and mitigate the risk of overall project failure (as discussed in Section 2.1). The dynamic model tests may also be reused as the basis for later testing (following the alignment of initial design to final testing at the same level of abstraction in the V-diagram, Figure 1). Dynamic modelling may therefore even aid early agreement of system acceptance test criteria, clarifying development goals and customer expectations.

Capturing architecture

The evidence that dynamic models encode enduring system properties (which, as noted in the "static versus dynamic models" topic, correlate early design to system tests) indicates that they are capturing overall architecture. Further examination of the models from the architect's perspective makes it clear that essential system properties are actually captured throughout the research.

The ACME model (Section 4.2) and second FSP model (Section 5.4) explicitly attempted to capture EGSO architecture. The first encodes generally defined connectors and component types (determined by rules for their connections), indicating typifying and constraining elements of generic data-grids. The second animated a static representation of the specific chosen EGSO architecture, successfully supporting the validation and verification of end-to-end operations across the planned distributed system.

Whilst not intended to be architectural, the other FSP models still capture essential overall features of data-grids (especially clear in the abstraction of the connector types from the instance models, described in Section 5.7). Architectural relations arise from the regular combination of the generally applicable design patterns (the components of high-quality architecture, discussed in Section 7.2). The completeness of designed interaction and risks of circular interaction (verified and exposed for both EGSO and AstroGrid in Sections 5.5 and 5.6) are shown through FSP modelling; these are emergent architectural properties of the whole system, which cannot be demonstrated (or repaired if faulty) within the developed components.

The simulation of network scalability (especially that described in Section 6.4), though applied to EGSO broker interaction design, has general architectural consequences for data-grids too. Whilst the strength of peer-to-peer networks is demonstrated (in the efficiency of their random connections), the limit of their suitability for controlled data-grid catalogue management is also highlighted (as messaging volume grows linearly with network size when definitive metadata must be used). Simulation is therefore shown to be a suitable experimental technique for evaluating grid architecture, even without real-world data to calibrate the relations it reveals.

However, the focus of this research on dynamic modelling does not imply that this should be the primary expression of architecture. Multiple views of the data-grid domain and its candidate systems together increase the likelihood of successful design. Above the mere satisfaction of requirements, the highest criteria for judging architecture are: elegance, through balanced and efficient composition of simple and consistent elements, with broad user appeal through flexibility, minimising the effort required to provide the desired operations, whilst reliably supporting desirable behavioural properties. Data-grid modelling indicates that both static and dynamic systematic models are capable of representing such aspects of architectural elegance, beyond their complementary value to reliable design (described in the "static versus dynamic models" topic).

Formalism versus simulation

Having used both formal event transition specification and experimental discrete event simulation to analyse the same data-grid topics, these alternative behavioural modelling techniques can be compared. An apparently trivial observation, that the declarative FSP models are terser than the procedural SimPy programs, highlights the difference in effort and therefore productivity between the techniques. With the maturity of the Python language and its established reputation for rapid development (see Section 6.2), SimPy development should be expected to be easier than FSP modelling. Simply measuring productivity by the number of lines of code written per day, SimPy modelling is over 3 times more productive (see Figure 28). Noting that FSP is more compact and cautioning that other developers in other simulation languages may have different experience moderates the significance of this anecdotal observation, though.

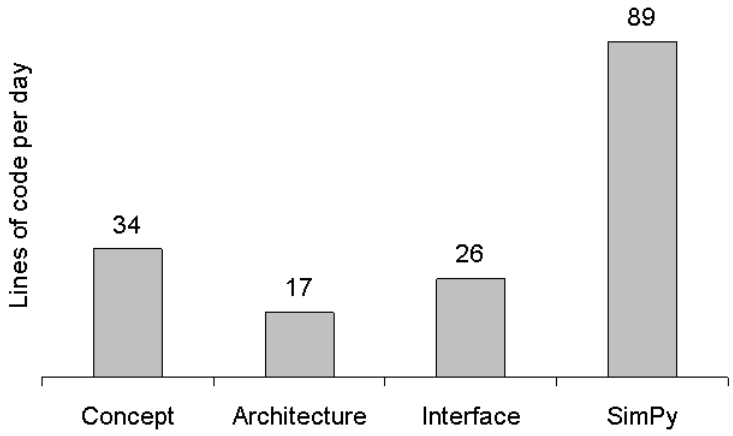

Figure 28: The average number of lines of code written per day (in modelling EGSO designs) indicates the greater productivity of modelling in SimPy over FSP.

Whilst both techniques worked (as discussed in the "observations, communication, model impact" parts of Sections 5.3 to 5.7 and in Section 6.5 especially) they also had shortcomings. All event analysis tools have trouble modelling large numbers of elements, as combinatorial explosion creates enormous state-spaces. Also, the formal methods were not as well recognised by the stakeholders as the simulation. The software engineers, and even

managers with development backgrounds, perhaps recognised running simulation programs to be stronger prototypes for the real system than models animated in the specialist LTSA tool. Simulation, though, only provides inductive proof of good design. The experimental results only support hypothesised relations of monitored statistics; they cannot prove the system avoids errors. Therefore, simulation does not provide the option for the systematic deterministic analysis (described in the "stronger formalism" topic). However, it does encourage the same level of confidence in the design as that gained later in the lifecycle through system testing, which also validates assertions about the composed system's qualities.

Ultimately the modelling techniques take different abstraction views, so the choice of technique should be determined by the modelling goal. The simplified cross-section of a designed system that FSP takes is focused on the safe interaction of semi-independent elements, whilst SimPy provides insight into the behaviour of the overall deployed system's state. FSP modelling should be chosen to prove interaction protocol safety (it finds missing messages, as noted in Section 5.5 especially), whilst SimPy suits demonstration of quality across wide ranges of system configurations (for example, different experiments on broker configurations consistently used network sizes of between 4 and 500 nodes in Sections 6.3, 6.4 and 6.5). Thus, FSP demonstrated the risk of message interference in the designed broker interaction protocol, whilst SimPy illustrated the tipping point in optimising message propagation strategies.

Assessing models' value

The value of the software engineering models are assessed by 3 criteria (as indicated by Section 2.2):

- their capability to derive valuable analytic results,
- their accuracy with respect to the known state of the real system,
- their comprehensibility (and other 'soft' issues: their impact on others' actions and their utility in project management).

The architectural analysis presented in Chapter 4, using ACME formal specification and the novel technique to fit requirements to styles, had clear shortcomings. The ACME model was incomprehensible to other project stakeholders, so lessons derived from it could not be used. In contrast, the architectural fitting was understood, but its subjectivity makes its analytic capability weak. Both schemes' had acceptable accuracy: the ACME model's components being traceable to suggested high level functions, and the style fit having the direct link to requirements.

The LTSA tool was demonstrated to yield useful analytic results at all stages of the designs' evolution. Its FSP code was linked to whatever had been specified for the planned system by diverse techniques – focussing on specific components' relations initially, then encoding scenarios for the overall architecture, and later directly representing the message sequences described in component and object interface specifications. The modelling was understood; each stage had demonstrable impact as results were accepted and presented as justification for design decisions or affected project development. It has therefore been demonstrated to satisfy the 3 criteria for successful modelling.

In this work, the orthodox discrete event simulation environment implemented by SimPy proved to have greater scope than Stochastic FSP for capturing and analysing complex system design (SimPy was successfully used to evaluate alternative designs). No findings of emergent properties that could not be predicted by intelligent reasoning about the designs were found, but it did confirm designers' hypotheses (and may have found surprising relations between designed components' properties had it been used as extensively as LTSA). Similarly, though the simulation results were not as widely presented as the FSP models (not being submitted in official project documentation) its results were easily understood. Additionally, as the simulations' implementations were directly derived from candidate designs, SimPy modelling also meets all 3 criteria. Both event transition analysis and discrete event simulation are therefore judged to successfully model data-grids.

7.2 Data-grid patterns

General data-grid patterns would be a valuable output of the presented research, beyond validation and advancement of software engineering methods (and for the EGSO and AstroGrid projects, beyond the delivery of useful applications to scientists). These abstract design templates could be reused in other projects, reliably recreating successful solutions. A pattern for metadata relay is described in detail below; its essential elements and their use in object-oriented programming terms are documented, applying textbook standards for valuable rigorous accuracy [49]. Further general data-grid patterns are also suggested by model analysis, including the EGSO hybrid architectural style and design components that go beyond data-grids; these are described in less detail.

Note that none of these patterns should be thought of as proven whilst data-grids are still a new domain. User needs are still evolving as they gain their first experience of grid capabilities beyond Internet access. Also, deployed data-grids have only just proven their functional capability. Their persistent quality and the value of the patterns they employ can only be judged as they are stressed through growth and maintenance.

Metadata relay

Data-grids require decentralised distributed metadata stores in their infrastructure to be globally distributed, and supported by independent administrators. This implies that when metadata is updated at one point, other parts of the network must automatically become aware of the change. As the decentralised architecture has no privileged point of view, updates may propagate from different parts of the network, leaving different nodes with inconsistent metadata and none knowing the definitive truth. The AstroGrid's registries (in the project's tiered architecture) and the brokers of EGSO (in peer-to-peer relations) are both instances of this situation. This pattern was captured in both FSP and SimPy models for EGSO (Sections 5.5 and 6.3).

The metadata relay pattern is composed of: the aggregated records (their actual content is unimportant), a communication sink that receives updates, a communication source that can forward the update to known peers, references to those peers, and a filter process for

received updates. The filter is the only part with non-trivial operations; it decides whether each received update should be saved as a new (or updated) record (and then forwarded to peers) or ignored. To do this, the records must be uniquely identifiable and consecutively versioned right across the network. (This would more easily be implemented with a primary key derived from record properties and a time of the original update than with special identity and version fields.) Note that it is not necessary for the sink to distinguish whether it receives the original update (generated by an administrator or automated provider process) or an update forwarded from a peer. The relation between these components is shown as a UML class diagram in Figure 29.

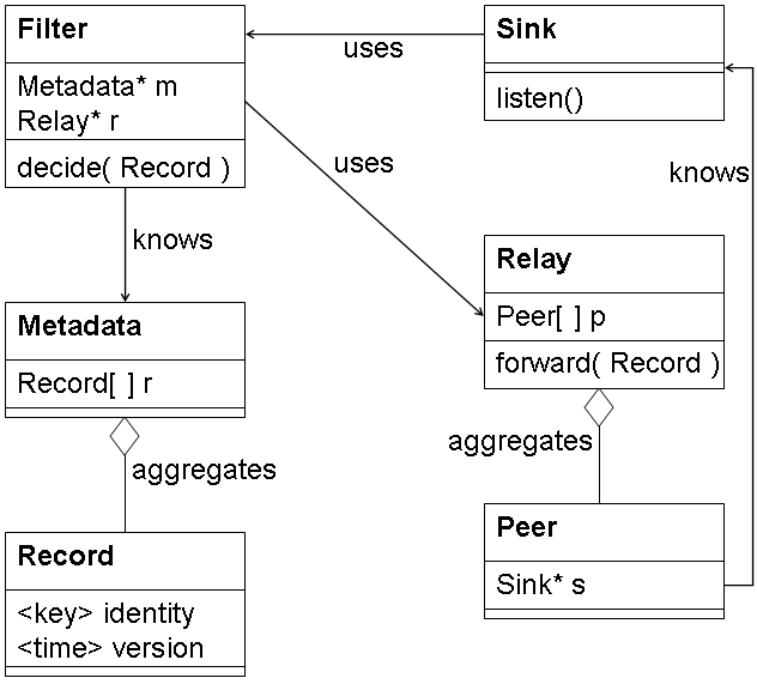

Figure 29: UML class diagram showing the components of the metadata relay data-grid pattern.

Instances of all the metadata relay components exist at each registry or broker in the data-grid. The references that each node has to its peers form the graph that defines the network. Decentralised data-grids should not impose a hierarchy of nodes, which would form a tree, so there should be circular paths through the graph; the filter stops updates continuing to cycle, and prevents obsolete updates (arriving via a slower route) overwriting more accurate later versions of records already received. Minimal hierarchical structure could be implemented (as in the relations of AstroGrid registries) by making some infrastructural metadata repositories 'leaves' of the network graph, which cannot forward the updates they receive. When deployed to such nodes, instances of the peer references would not be needed (the relay object would have an empty list of peers, or it may not exist itself).

Other patterns

Versions of the metadata relay pattern were implemented in both FSP and simulation models, and had recognised application in AstroGrid as well as EGSO. The following patterns are less widely noted, but could be defined to the same standard with further analysis.

- *Filter.* A middle-tier filter helps to make users' access to diverse data sources transparent, as it transforms application formatting to back-end instruction syntax. To do this in a flexible data-grid, a filter component relies on its own repository of formats and schemas. For example, to resolve a user's search, sent from a GUI, the EGSO brokers would use their universal catalogue to map terms and units to the fields and values of a provider's database, then translate the results back to the standard XML VOTable format for presentation [89,32]. This pattern was first captured in the ACME model (Section 4.2).
- *Distributed tasks.* Like computational grids, data-grids must allow jobs to be resolved by distributed resources, going beyond the client-server architectural style. EGSO is required to dispatch user queries to multiple providers (when necessary), whilst AstroGrid must manage workflow synchronisation of parallel resources' results. Job scheduling designs exist in parallel computing, but the special distributed state of data-grid jobs (where central control is not necessary) is demonstrated in the FSP models (Section 5.7).
- *Peer-tier style.* EGSO deliberately merges the peer-to-peer and n-tier styles of distributed systems (initially captured in ACME, Section 4.2, clearly expressed in FSP, Section 5.4, and apparent in the broker connections modelled in Section 6.4). An essential component of this broader architecture is the metadata relay design pattern already noted. Note that agent (peers) and service oriented (tiers) architectural styles are also converging in broader Internet-scale problem domains [72].

Speculative data-grid patterns

Though it is stated that NFR must be resolved in a project's high-level design, data-grid components that support infrastructural quality are conceivable. Simulation demonstrates that there are no economies of scale in the growth of networks using the metadata relay pattern (Section 6.3). Two possibilities could help the performance, though:

- In the case of the EGSO, performance depends on the ratio of user to provider messages (and the choice of broker messaging design). A broker configuration switch could change the forwarding behaviour, optimising the network for the observed usage profile. Implementing both strategies would not be difficult, as brokers resolve queries and communicate with peers in both cases.
- Further analysis (not required for the relatively small EGSO network) could reveal a solution that has much greater scalability. Following peer-to-peer designs, this may involve a 'time to live' forwarding parameter that sets a propagation horizon. Such a solution just refines the behaviour of the filter in the metadata relay.

Scalability can therefore be improved by a small change to one component in the EGSO design. A change to improve this NFR could easily be deployed across the infrastructure, if the pattern were implemented in a decoupled way.

In general, the software engineering rule of thumb, which states that NFR are resolved holistically at the architectural level, can be circumvented in decentralised data-grid systems with specific design strategies. Patterns could be implemented to meet other NFR, and these would then also become decoupled components that controlled quality.

For security, authentication design patterns are established (and now standardised in Web Services and OGSA), and analysis of data-grid resources demonstrates how distributed constraints must be upheld at each connected sub-system in the instance of EGSO (Section 3.3.4). By decoupling security from domain specific functions, standard solutions could be implemented and reused in diverse distributed resource sharing networks.

Users' experienced reliability in using distributed resources is another quality that can be separated from functional capabilities in a well-designed system. For example, if the providers of EGSO can take responsibility for accurately recording observed performance, consumers and the brokers that act for them can manage job dispatch and recovery autonomously. Centralised management of the network's topology is not necessary to allow users to avoid unreliable resources.

7.3 Modelling methodology

A systematic, reliable modelling process emerged from the reported experience of developing FSP models (Sections 5.3 to 5.7). The method is applied in the FSP tutorial (Appendix G). A complete iteration of the model lifecycle takes a short time within one of the major stages of the project, for example in a few days before an interface design review. The method ensures that the models produced faithfully represent what is known of the real system, and rapidly deliver valuable conclusions that can be understood by key stakeholders (who need not know the language).

There are 5 steps in the process:

1. *Requirements' analysis*: identify the purpose of the model and the events in it.
2. *Sequential implementation*: compose processes that represent single instances of the components and tasks.
3. *Concurrent implementation*: enable multiple concurrent component instances by indexing the processes and events.
4. *Testing*: analyze the composition, debug and refine the model.
5. *Operation*: demonstrate the model system and modify the real system's design.

Though suggestive of a waterfall lifecycle, these steps need not be followed sequentially; analysis or demonstration may be done directly after either implementation step. The process is also iterative; refined models or feedback from demonstration may demand re-evaluation of requirements or alternative implementations. Additionally, familiarity with the waterfall process may imply that step 4 would be a trivial integration of interfaces and step 5 a cosmetic milestone. However, the identification of faults and modification models at stage 4 requires considerable effort (highlighted in the tutorial appendix), whilst step 5 is essential for the intelligent application of conclusions derived from the modelling effort. Rapid model

development can therefore introduce the benefit of risk reduction through iterative design refinement even to projects that are constrained to follow a waterfall model.

In the case of EGSO, it was necessary to follow a waterfall sequence of dependent steps [9]. Early in the project, iterative development opportunities were not taken, so little coding was done to reduce risk (though development on the isolated functionality of the feature recognition component could begin). In this context, the rapidly developed models that demonstrated how alternative designs would work were very useful. Later (once initial milestones had passed), cyclic development was taken up; increasingly sophisticated capabilities were implemented in a series of demonstrators (following the successful use case driven development strategy of AstroGrid). At this stage, the models could be evaluated alongside real prototypes. However, they had less impact once it was recognised that the planned quality of service that EGSO would deliver was not as high as originally envisioned. Whilst the simulation described in Section 6.3 evaluated hundreds of brokers, the prototype broker stood alone and did not interact with peer instances at all. Whilst peer interaction was planned as functionality to be stabilised in later iterations, it became clear that EGSO would not meet the originally conceived data-grid vision that would permit decentralised maintenance of high quality of service.

7.4 Further domain contributions

Section 1.1 stated that this research was at the boundary of 3 domains (e-science, software engineering and solar physics), claiming the work contributed to each. That the investigation of software engineering has been of value to e-science has now been demonstrated, as EGSO and AstroGrid directly benefited from the requirements' analysis described in Sections 3.2 and 3.3 and the modelling contribution already noted. The domain is also advanced by the critique of existing methods given in Section 3.1 (which may guide the quality of the projects' follow on activity through better application of architectural styles), and the fitting of generic data-grid requirements to clearly defined architectural styles, described in Section 4.1.

The value that the study of e-science returns to software engineering is as a case study for modelling methodologies. ACME and FSP (described in Section 4.2 and Chapter 5 respectively) are innovative representations, which need applications to demonstrate their value. It has been noted that the value of ACME was limited in rapid architectural evaluation due to difficulty in communicating the benefit of formal, static descriptions. The success of FSP is demonstrated by the impact on the projects described in the 'observations, communication, model impact' sections of Chapter 5. Also, stochastic FSP is highly novel, so its evaluation especially benefits its maturing LTSA support; the practical problems noted in Section 6.1 do not alter its value as an economic, formal expression of the type of model successfully applied in the remainder of Chapter 6 with SimPy.

The software engineering domain is also supported by critical application of more mainstream methods, including requirements' analysis, to the e-science projects. The use case analysis (from generic high-level activities envisioned on a solar physics data-grid to specific

scientific use cases and technical scenarios, described in Sections 3.2 and 3.3.3) successfully captured the domain's needs, and usefully directed subsequent development (as indicated by the use case relations of Figure 13 in Section 3.4.2). More innovative requirements' analysis techniques were successfully applied too. Section 3.3.2 contrasted the investigation's goal decomposition with other researchers', and Section 3.3.4 described the supported specialist security modelling. These are therefore demonstrated to be practical, robust engineering methods, even though their impact on the EGSO project was less than that of the use cases.

The EGSO project, as a case study for software lifecycle management, gives further support for mainstream software engineering. The arguments for iterative development (including the judgements expressed in the review of lifecycle methods in Section 2.1) were initially overlooked, as the project proposal imposed a traditional waterfall project plan; this choice arose from the project mangers' experience with scientific instrument programmes for spacecraft, for which the careful planning of sequential development is essential. Section 7.3 shows how modelling systematically guided iterative design evolution within this constraint. The FSP models of Chapter 5 successfully gave rapid test results, trapping faults and raising confidence in designs. Modelling iterations therefore helped to move the project along, especially at the architectural development stage (described in Section 5.4), where the tested FSP model of the architecture was accepted in the funding review. This was despite preceding difficulties in agreeing on requirements, as predicted for the novel data-grid domain and aggravated by unclear separation of concerns in the diverse requirements (noted in Section 3.4.2).

The domain model (another case study of mainstream software engineering methodology, described in Section 3.2.3) is a concrete example of a medium-scale topic-space, successfully derived from early solar physics data-grid requirements' analysis. As noted, it had negligible impact on the EGSO project, but an equivalent document [89] (derived by others from the specific EGSO scientific user requirements [88]) was useful. These models support standards, notably the VOTable schema, and further research of ontologies for data-grids (or other information networks with semantic query capabilities).

The contribution of the research to its third domain, solar physics, is less. The general domain use cases (of Section 3.2.1) did guide scientists' expectations of data-grid capabilities (they were presented at MSSL beyond the EGSO team). The scientific use cases (of 3.2.2) offer specific examples to follow in solar physics research. Of course, solar physics ultimately also benefits through delivery of data-grids that have benefited from extensive, rigorous software engineering analysis; successful modelling indicates the delivered systems will support the original use cases.

When the investigation started, architectural styles were weakly applied in grid toolkits and data-grid projects (as reviewed in Sections 2.3 and 3.1 respectively). This emergent domain in general is strengthened; from the fit of generic data-grid requirements to clearly defined architectural styles (Section 4.1), styles' suitability in concrete designs are evaluated through comparative dynamic models (especially in Section 5.4 for the formal analysis of EGSO's tiered architecture and Section 6.3 for the experimental simulation of the peer-to-peer aspects of EGSO). These examples of analysis, applied early in the lifecycle, mitigated the risk of failure

and saved the cost of potential downstream errors. A systematic method for rapid, valuable high level modelling emerges (Section 7.3), and reusable design patterns are suggested (Section 7.2).

By equipping data-grid engineers with tools to analyse their complex distributed systems and abstract reusable components, this work improves the quality of data-grids and their e-science capabilities. It should also be applicable beyond academic projects; Appendix O notes practices experienced in commercial software engineering, and considers the role of dynamic data-grid models in the business context. The reported evidence suggests that it may be hard to incorporate the researched modelling into commercial production, though developers could use it to improve the quality of the delivered code.

Chapter 7 key points

- The simulation and event modelling techniques applied in the research have different advantages, respectively demonstrating emergent qualities and protocol safety.
- Design patterns for data-grids can be identified and should be studied as the domain matures. Refining the decoupled metadata relay pattern design (apparent in EGSO and AstroGrid) could enable better scalability.
- A systematic rapid modelling process is identified that, beyond the normal benefits of modelling early in a system's development lifecycle, can bring the advantage of iterative development's risk reduction to waterfall projects.

Chapter 8 Summary and direction

In Section 8.1 of this final chapter, the work presented is summarised, drawing special attention to the points that are apparent in hindsight. Section 8.2 looks forward to further study following from the research, broadening out from the case study projects to general data-grids and beyond.

8.1 Summary

Thesis argument

The argument of this thesis, presented in Chapter 1, is that advances in software engineering and e-science, in application to specific solar physics data-grid projects, are mutually supportive. Engineering techniques need validation by case studies, whilst data-grids present an original challenge – to provide the infrastructure for a new class of scientific method – which requires software engineering to succeed.

Domain review

Chapter 2 introduced software development lifecycle thinking, and then focussed on architectural styles. It also highlighted ambiguity in grid tools' frameworks. It should be noted that later, both EGSO and AstroGrid delivered incremental capabilities through a series of demonstrations, reinforcing the validity of iterative development.

Chapter 3 gave evidence of distributed system styles' application in a broad range of data-grids. It also described the data-grid requirements of solar physics, with detail of the diverse analyses carried out for the EGSO project. Decentralised support of NFR qualities by peer-to-peer architectures makes them suitable for data-grid needs. However, the immediate challenge for first generation projects like EGSO turned out to be just delivering functionality; a classic 3-tier architecture, implemented by Web Services, provides sufficient quality, despite the limited scalability and reliability imposed by central management points.

Models in 3 acts

Section 4.1 presented a novel, lightweight technique for systematically choosing an architectural style to use for data-grids based on their generic requirements. The technique would be strengthened by concrete evidence of styles delivering their advertised benefits in data-grid operation. However, there has still not yet been widespread uptake of completed data-grid projects; none have supplanted established Internet-based ways of working. In Section 4.2, the difficulty noted in communicating the implemented static architectural specification, which identified abstract interaction patterns, implies that future use of ADL specifications (in this domain) would be limited. There is still a role for such work in the design of reusable components that need high reliability, for example, Java RMI interaction skeletons [55].

The FSP models of dynamic events (analysed with LTSA, presented in Chapter 5) successfully captured and validated data-grid designs at 5 stages:

- aspects of behaviour in informal data-grid concepts (Section 5.3),

- architecturally specified components (Section 5.4),
- interaction between subsystem interfaces (Section 5.5),
- object interaction in more detailed design (Section 5.6),
- models of abstract design patterns (Section 5.7).

Models had traceability to requirements and diverse static specifications, through scenarios and message sequence charts. They demonstrated their value early in projects' lifecycles by identifying shortcomings in the specific designs studied, notably by highlighting missing messages and the risk in symmetric dependency. The models could also be explained to other project stakeholders, influencing development. Though generic components were identified and captured, the nature of the methodology limited the scope of their reuse in model specification.

Though model development using the stochastic extensions to FSP was found to be difficult (primarily because of their immaturity), data-grid simulation was successfully demonstrated in Chapter 6 through SimPy. Rapidly developed models of many operational sub-systems exhibited predicted relations in their behavioural qualities, even without calibration by genuine experimentally observed network properties.

Findings

Chapter 7 drew out the implications of the reported experience in modelling data-grids. There is clear benefit in constructing and analysing dynamic models of data-grids, and therefore innovative challenging information systems in general, as validation of early design mitigates the risk of carrying forward flaws that are difficult to resolve later. Event models and simulations have different strengths, through their complete formal analysis and support for experimentation respectively.

Though the primary purpose of the research has been the validation of software engineering methods, an original modelling method was derived and emergent generic data-grid patterns were observed. The rapid iterative method described in Section 7.3 is an instance of the modern lifecycles described in Section 2.1 that systematically guides useful modelling. The patterns described in Section 7.2 would need further study to verify that they are regularly used successfully, but could accelerate the design of high-quality future data-grids. In both cases, it is clear how the application of software engineering generates useful new knowledge, driving the evolution of the domain along with its target – in this case, data-grids.

8.2 Direction

Further EGSO and AstroGrid modelling

EGSO and AstroGrid have been modelled during their evolving designs, but models continue to be valuable later in the lifecycle. A live model is updated to reflect changes in design made when implemented code diverges to work around unanticipated real-world complexity. Retesting the modified model then validates the changes (before the implementation is complete and ready for testing itself). If the EGSO component interaction messages changed, the model of Section 5.5 could be easily updated; even if the broker role changed in EGSO's

simple peer-tier topology by introducing the hierarchy of AstroGrid's registries, it would be straightforward to refine the network simulation of Section 6.4. In general, the methods used would therefore be suitable for live modelling.

When it came to projects' component integration and deployment, the engineering value of the earlier models could be evaluated further by examining their similarity to the real systems. The scenarios of Section 5.4 would be reused as system tests to verify reliable concurrent task progress through distributed query-state updates. Performance testing as the live networks grow would support the scalable performance modelled in Section 6.3. Such studies would confirm that the models capture designed solutions to NFR, but require users who put real demand on the system in scientific investigations.

Improving data-grids

Beyond the EGSO and AstroGrid projects, other data-grid software engineers could apply modelling techniques like those evaluated in this research. Simulation has been used to evaluate specific algorithms and compute-grid topologies [86], but dynamic models of high level architecture or reusable data-grid components are not published or widely used.

Models could also be modified for operation, encoded in the maintenance sub-systems of data-grids and tracked against reality. Reflective systems are able to dynamically modify their configured properties to react to changes in their environment or requirements. Maintaining an internal representation of the real network allows automated controls to evaluate the impact of possible changes before they are made. Using the dynamic models presented in this investigation would allow them to forecast effects even on the emergent properties in non-deterministic complex networks.

For example, data-grids may relax the criteria for making matches to meet user queries with incomplete local results when the global network becomes unreliable, or reconfigure their components' relations to optimise resource use by new composite analysis services. It would therefore be feasible to design reusable components (which could be adapted to diverse data-grid projects) that uphold NFR, such as performance and reliability. This possibility goes some way to reversing the general limitation to distributed systems' maintenance noted in 2.2 (where it was stated that NFR are generally met by the overall architecture, not its components, making repair of quality shortfall later in the software lifecycle very hard).

It was demonstrated that the best design for query resolution against middle-tier catalogues by EGSO brokers depended on the ratio of user queries to provider catalogue updates in Section 6.3. It was noted that the decision could be made by observed circumstances after deployment, and this is a clear application for a reflective management component. However, in this case, it need not maintain an entire model of the system, just the decision rule derived from it. This is a case where the configuration change actually represents a change in the architectural style applied, between peer-to-peer query forwarding and a more traditional middle-tier strategy for consistent metadata provisioning.

To evaluate the other candidate data-grid architectural styles (which were noted to have not been widely applied in Section 3.1), more fundamentally diverse data-grid designs than EGSO and AstroGrid would need to be evaluated. The styles' value could be analysed with

modelling techniques like those presented here, though. Simulation would be well suited to evaluating the support of NFR by decentralised nodes that selfishly only support as much service to others as they receive themselves in a pure peer-to-peer architecture (which avoids the centralised management role taken by the EGSO broker). Likewise, event modelling could evaluate an agent-based architecture, testing whether the protocols for interaction via a shared central blackboard were safe and efficient.

Broader application

The noted convergence of data-grids with other Internet scale distributed systems (peer-to-peer networks, web services and semantic web) is evident in EGSO and AstroGrid. EGSO used a hybrid architectural style, and both implemented interfaces and output to open service standards (WSDL and the VOTable taxonomy). Lessons learnt from these projects, about the application of design patterns and suitability of early modelling, may therefore be applied beyond astronomy data-grids.

Commercial organisations are aware of the importance of integrating their information assets [17], especially when they are distributed across diverse legacy systems. The technologies they use have striking similarity to the data-grid requirements described in Section 3.4.1:

- data warehousing reconciles heterogeneous records on a large-scale,
- enterprise application integration builds a system of systems by exposing distributed capabilities, which typically include information analysis,
- enterprise search engines use indexes of distributed data resources, just like the EGSO catalogue or AstroGrid registry,
- business process workflow is managed through concurrent progress on distributed tasks' states.

As noted in Appendix O, there may be resistance to the uptake of innovative software engineering modelling methods by businesses. Industry also typically imposes more stringent NFR, for: security against criminal activity and costly errors, usability by non-technical actors, reliability of services commercially supplied according to service level agreements, and performance and manageability in the control of economically critical resources. Despite this, the data-grid modelling techniques presented may be adapted to support information engineering in the commercial domain. They may be refined to specifically address the high levels of commercial requirements. Innovation requires some risk; with focussed trials guided by clear business cases, information system modelling could have an important role.

Final key points

- The work has demonstrated the value of software engineering's dynamic modelling techniques by evaluating high-level designs with LTSA, and contrasting their FSP descriptions with static architectural models and discrete event simulation.
- As well as supporting the development of the concrete EGSO and AstroGrid projects, small evolutionary improvements to software engineering methods arise. The technique for fitting architectural styles to requirements, observed emergent data-grid patterns and the rapid iterative model development lifecycle are novel contributions.
- Dynamic models have a promising role in enabling the maximum exploitation of networked information resources.

Bibliography

Books and articles (from journals, conference proceedings and collections) are listed in the alphabetical order of the first given author (surname first). WWW page citations are then listed in alphabetical order of page title.

1. Abowd G, Allen R, and Garlan D. 'Using Styles to Understand Descriptions of Software Architecture', ACM Symposium on the Foundations of Software Engineering (SIGSOFT) 1993
2. Allen Robert and Garlan David. 'A Formal Basis for Architectural Connection', ACM Transactions on Software Engineering and Methodology 6-3 1997
3. Anderson David. 'BOINC: A System for Public-Resource Computing and Storage', 5th IEEE/ACM International Workshop on Grid Computing 2004
4. Arsanjani Ali, Hailpern Brent, Martin Joanne, and Tarr Peri. 'Web Services: Promises and Compromises', IBM Research Report RC22494 (W0206-107) 2002
5. Atkinson M, Baxter R, and Hong N C. 'Grid Data Access and Integration in OGSA', OGSA-DAI GridServe project, EPCC-GDS-WP2-ARCH v1.2 2002
6. Bass Len, Clements Paul, and Kazman Rick. 'Software Architecture in Practice', Addison Wesley 1998
7. Beck Kent. 'Extreme Programming Explained; Embracing Change', Addison Wesley 1999
8. Bell W H, Bosio D, Hoschek W, Kunszt P, McCance G, and Silander M. 'Project Spitfire - Towards Grid Web Service Databases', Global Grid Forum 5 2002
9. Bentley Bob. 'European Grid of Solar Observations', Proposal IST-2001-32409 2001
10. Bentley Bob and Freeland Sam. 'SolarSoft - An Analysis Environment for Solar Physics', A Crossroad for European Solar and Heliospheric Physics, ESA Publication 1998
11. Boehm Barry W. 'A Spiral Model of Software Development and Enhancement', IEEE Computer 21-5 1988
12. Booch Gardy, Jacobson Ivar, and Rumbaugh James. 'Unified Modeling Language User Guide', Addison Wesley 1999
13. Bosch Jan. 'Design and Use of Software Architectures', 2000
14. Brooks F P. 'No Silver Bullet; Essence and Accidents of Software Engineering', IEEE Computer 1987
15. Buchanan William. 'Mastering Networks', Macmillan 1999
16. Bush Vannevar. 'As We May Think', The Atlantic Monthly 176-1 1945
17. Butler Group. 'Exploiting Corporate Information Assets', Technology Management and Strategy Report 2004
18. Cancio German, Fischer Steve, Folkes Tim, Giacomini Francesco, Hoschek Wolfgang, Kelsey Dave, and Tierney Brian. 'The DataGrid Architecture', https://edms.cern.ch/cedar/plsql/doc.info?document_id=333671 2002

19. Carroll John M. 'Scenario Based Design: Envisioning Work and Technology in Systems Development', John Wiley 1995
20. Casanova H, Dongarra J, Johnson C, and Miller M. 'Application-Specific Tools'.In 'The Grid: Blueprint for a New Computing Infrastructure', Foster I, Morgan Kaufmann 1998.
21. Chapman Clovis, Wilson Paul, Emmerich Wolfgang, Tanenbaum Todd, Farrellee Matt, and Livny Miron. 'Condor services for the Global Grid: An investigation into the development of Condor Grid services with OGSA', UK e-Science All Hands 2004 2004
22. Charette R N. 'Why Software Fails', IEEE Spectrum 42-9 2005
23. Chervenak A, Foster I, Kesselman C, Salisbury C, and Tuecke S. 'The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets', 2000
24. Ciminiera Luigi, Sanna Andrea, Zunino Claudio, and Piccinelli Giacomo. 'EGSO Interaction Mechanisms Document v1.0', 2003
25. Ciminiera Luigi, Sanna Andrea, Zunino Claudio, Scholl Isabelle, Linsolas Romain, Tant Raphael, and Piccinelli Giacomo. 'EGSO Survey of Middleware for Federation, Cataloguing, Catalogue Search and Visualization Techniques', EGSO-WP1-D3-20021106 2003
26. Crowcroft Jon, Pratt Ian, and Twigg Andrew. 'Peer-to-peer Systems and the Grid'.In 'The Grid 2e: Blueprint for a New Computing Infrastructure and 2nd Edition', Foster Ian and Kesselman Carl, Morgan Kaufmann 2003.
27. Csillaghy A, Zarro D M, and Freeland S L. 'Steps Towards a Virtual Solar Observatory', IEEE Signal Processing Magazine 18-2 2001
28. Czajkowski Karl, Foster Ian, Karonis Nick, Kesselman Carl, Martin Stuart, Smith Warren, and Teucke Steven. 'A Resource Management Architecture for Metacomputing Systems', IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing 1998
29. Dardenne Anne, van Lamsweerde Axel, and Fickas Stephen. 'Goal-Directed Requirements Acquisition', Science of Computer Programming 20-1-2 1993
30. De Roure D, Baker M A, Jennings N R, and Shadbolt N R. 'The Evolution of the Grid', Correspondence, Department of Electronics and Computer Science, University of Southampton 2002
31. Dijkstra E W. 'The Humble Programmer', ACM Turing Lecture EWD340 1972
32. Durand Daniel, Fernique Pierre, Hanisch Robert, Mann Bob, McGlynn Tom, Ochsenbein Francois, Szalay Alex, Wicenec Andreas, and Williams Roy. 'VOTable: A Proposed XML Format for Astronomical Tables', http://cdsweb.u-strasbg.fr/doc/VOTable/ 2002
33. Durkin Tom. 'SETI Researchs Sift Interstellar Static for Signs of Life', Xilinx Xcell Journal 48-2004
34. Edinburgh Parallel Computing Centre course notes. 'Advanced Programming, Practical Software Engineering for Computational Scientists', 2002
35. Emmerich Wolfgang. 'Engineering Distributed Objects', John Wiley 2000

36. Fallows R A, Williams P J S, and Breen A R. 'Evolution with heliocentric distance of turbulent-scale irregularities in the solar wind', Steel MIST/ UK Solar Physics 2002 2002
37. Finkelstein Anthony, Gryce Clare, and Lewis-Bowen Joe. 'Relating Requirements and Architectures: A Study of Data-grids', Journal of Grid Computing 2-2 2004
38. Finkelstein Anthony, Lewis-Bowen Joe, and Piccinelli Giacomo. 'Using Event Models in Grid Design'.In 'Grid Computing: Software Environments and Tools', Cunha J C and Rana Omer, Springer Verlag 2006.
39. Flechais Ivan and Sasse M Angela. 'Developing Secure and Usable Software', Proceedings of OT2003 2003
40. Foster I and Kesselman C. 'The Grid: The Globus Tookit'.In 'The Grid: Blueprint for a New Computing Infrastructure', Foster Ian, Morgan Kaufmann 1998.
41. Foster I, Kesselman C, Nick J, and Tuecke S. 'The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration', Global Grid Forum (GGF) 2002
42. Foster I, Williams D N, and Middleton D. 'Earth System Grid II; Turning Climate Datasets into Community Resources', DOE Collaboratory Pilot Project proposal 2001
43. Foster Ian. 'The Grid: Blueprint for a New Computing Infrastructure', Morgan Kaufmann 1998
44. Foster Ian. 'The Anatomy of the Grid: Enabling Scalable Virtual Organizations', Lecture Notes in Computer Science 2150-2001
45. Foster Ian and Iamnitchi Adriana. 'On Death, Taxes and the Convergence of Peer-to-Peer and Grid Computing', 2nd International Workshop on Peer-to-Peer Systems (IPTPS) 2003
46. Fowler Martin. 'Refactoring: Improving the Design of Existing Code', Object Technology Series 1999
47. Fox G C and Furmanski W. 'High Performance Commodity Computing'.In 'The Grid: Blueprint for a New Computing Infrastructure', Foster I, Morgan Kaufmann 1998.
48. Frey J, Tannenbaum T, Livny M, Foster I, and Tuecke I. 'Condor-G: A Computational Management Agent for Multi-Institutional Grids', IEEE High Performance Distributed Computing (HPDC10) 2001
49. Gamma Erich, Helm Richard, Johnson Ralph, and Vissides John. 'Design Patterns: Micro-architectures for Reusable Object-oriented Software', Addison Wesley 1994
50. Gannon D, Bramley R, Fox G, Smallen S, Rossi A, Ananthakrishnan R, Bertrand F, Chiu K, Farrellee M, Govindaraju M, Krishnan S, Ramakrishnan L, Simmhan Y, Slominski A, Ma Y, Olariu C, and Rey-Cenvaz N. 'Programming The Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications', Extreme! Computing, Indiana 2001
51. Gannon Dennis and Grimshaw Andrew. 'The Grid: Object Based Approaches'.In 'The Grid: Blueprint for a New Computing Infrastructure', Foster Ian, Morgan Kaufmann 1998.
52. Garlan D, Monroe R, and Wile D. 'ACME: An Architecture Description Interchange Language', Proceedings of CASCON 1997

53. Garrido Jose. 'Object-oriented Discrete-event Simulation with Java: A Practical Introduction', Kluwer Academic 2001
54. Goble C. 'MyGrid: Personalised e-Science on the Grid', Global Grid Forum (GGF) 2002
55. Gorlatch Sergei and Alt Martin. 'Grid Programming with Java, RMI and Skeletons'.In 'Grid Computing: Software Environments and Tools', Cunha J C and Rana Omer, Springer Verlag 2004.
56. Grimshaw A and Wulf W. 'Legion - A View from 50,000 Feet', IEEE International Symposium, High Performance Distributed Computing (HPDC-5) 1996
57. Gurman Joseph. 'A White Paper Concerning the Virtual Solar Observatory', VSO birds of a feather AAS/SPD (American Astronomical Society, Solar Physics Division) July 2002 2000
58. Harra L K and Sterling A C. 'Material Outflows from Coronal Intensity "Dimming Regions" during Coronal Mass Ejection Onset', The Astrophysics Journal 561-2 2001
59. Hoschek Wolfgang, Jean-Martinez J, Samar Andrea, Stockinger Heinz, and Stockinger Kurt. 'Data Management in an International Data Grid Project', First IEEE ACM International Workshop on Grid Computing 2000
60. Jeffery Keith G. 'Knowledge, Information and Data', Office of Science and Technology (OST) Briefing, Central Laboratory of the Research Councils (CLRC) Information Technology Department 2000
61. Johnston W, Simon H D, Bair R, Foster I, Geist A, and Kramer W. 'Enabling and Deploying the SciDAC Collaboratory Software Environment', DOE Science Grid proposal 2001
62. Johnston William. 'Realtime Widely Distributed Instrumentation Systems'.In 'The Grid: Blueprint for a New Computing Infrastructure', Foster Ian, Morgan Kaufmann 1998.
63. Jurjens Jan. 'UMLsec: Extending UML for Secure Systems Development', http://www4.in.tum.de/~jurjens/papers/uml02.pdf 2002
64. Kennedy Ken. 'The Grid: Compilers, Languages and Libraries'.In 'The Grid: Blueprint for a New Computing Infrastructure', Foster Ian, Morgan Kaufmann 1998.
65. Krauter Klaus, Buyya Rajkumar, and Maheswaran Muthucumara. 'A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing', Wiley Interscience - Software: Practice and Experience 32-2 2002
66. Lane TG, Asada T, Swonger R, Bounds N, and Duerig P. 'Architectural Design Guidance'.In 'Software Architectures: Perspectives on an Emerging Discipline', Shaw Mary and Garlan David, Prentice Hall 1996.
67. Lang Kenneth. 'The Sun from Space', Springer Verlag 2000
68. Ledlie Jonathan, Shneidman Jeff, Seltzer Margo, and Huth John. 'Scooped, again', 2nd International Workshop on Peer-to-Peer Systems (IPTPS) 2003
69. Linde Tony. 'AstroGrid Architecture: Vision, Overview, Detail', http://wiki.astrogrid.org/bin/view/Main/TonyArchVision 2003
70. Linsolas Romain and Soldati Marco. 'EGSO Consumer Specification and Implementation Document', EGSO-WP3-CONS1-20030310 2003

71. Lopez Isaac, Follen Gregory J, and Gutierrez Richard. 'NPSS on NASA's IPG: Using CORBA and Globus to Coordinate Multidisciplinary Aeroscience Applications', Advanced Computational Concepts Laboratory and NASA Glenn Research Center http://accl.grc.nasa.gov/IPG/CORBA/NPSS_CAS_paper.html 2000
72. Luck M, McBurney P, Shehory O, and Willmott S. 'A Roadmap for Agent Based Computing', AgentLink Community 2005
73. MacLean A and McKerlie D. 'Design Space Analysis'.In 'Scenario Based Design: Envisioning Work and Technology in Systems Development', Carroll John M, John Wiley 1995.
74. Magee J, Dulay N, Eisenbach S, and Kramer J. 'Specifying Distributed Software Architectures', 5th European Software Engineering Conference (ESEC) 1995
75. Magee Jeff and Kramer Jeff. 'Concurrency: State Models and Java Programs', John Wiley 1999
76. Mathiassen Lars, Nuk-Madsen Andreas, Nielsen Peter Axel, and Stage Jan. 'Object Oriented Analysis and Design', Marco 2000
77. Medvidovic Nenad and Taylor Richard N. 'A Classification and Comparison Framework for Software Architecture Description Languages', IEEE Transactions on Software Engineering 26-1 2000
78. Messina Paul. 'Distributed Supercompting Applications'.In 'The Grid: Blueprint for a New Computing Infrastructure', Foster I, Morgan Kaufmann 1998.
79. Moore Reagan W, Baru Chaitanya, Marciano Richard, Rajasekar Arcot, and Wan Michael. 'The Grid: Data-Intensive Approaches'.In 'The Grid: Blueprint for a New Computing Infrastructure', Foster Ian, Morgan Kaufmann 1998.
80. Oram Andy. 'Peer-to-Peer: Harnessing the Power of Disruptive Technologies', O'Reilly 2001
81. Phillips Kenneth. 'Guide to the Sun', Cambridge University Press 1995
82. Piccinelli Giacomo and et al. 'EGSO Architecture v2.0', EGSO Report EGSO-WP1-D4-20040308 2004
83. Preiss Bruno. 'The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environment', SCS Multiconference on Distributed Simulation 1989 1989
84. Priestley Mark. 'Practical Object Oriented Design with UML', McGraw Hill 1996
85. Putzer A. 'HEP Applications', EU DataGrid Project WP8 2000
86. Ranganathan Kavitha and Foster Ian. 'Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids', Journal of Grid Computing 1-1 2003
87. Reardon K, Ching N, Bentley B, Gryce C, and Giordano S. 'EGSO System Requirements Table', EGSO-WP1-ID1-20030128 2003
88. Reardon K, Giordano S, and Antonucci E. 'User and Science Requirements Document', EGSO-WP1-D2-20021031 2002
89. Reardon Kevin. 'Unified Model of Solar Metadata', EGSO-DE01_01-D02-021001 2003
90. Rector A, Kalra D, and et al. 'CLEF - Joining up Healthcare with Clinical and Post-Genomic Research', UK e-Science All Hands Meeting 2003

91. Reinefeld A and Schintke F. 'Concepts and Technologies for a Worldwide Grid Infrastructure', Zuse Institute Berlin 2002
92. Royce W W. 'Managing the Development of Large Software Systems', IEEE WESCON 1970
93. Sang Janche, Kim Chan, and Lopez Isaac. 'Developing Corba-Based Distributed Scientific Applications from Legacy Fortran Programs', NASA Ames Research Center HPCC CAS Workshop 2000
94. Shaw Mary and Garlan David. 'Software Architectures: Perspectives on an Emerging Discipline', Prentice Hall 1996
95. Snelling David. 'Unicore and the Open Grid Services Architecture'.In 'Grid Computing, Making the Global Infrastructure a Reality', Berman F, Fox G C, and Hey A J G, Wiley 2003.
96. Soldati Marco and Csillaghy Andre. 'Notes on Visual Interfaces to Specify Queries', EGSO User Interface Review 20020710 2002
97. Sommerville Ian. 'Model-based Specification'.In 'Software Engineering 5th edition', Sommerville Ian, Addison Wesley 1995.
98. Sommerville Ian. 'Software Engineering', Addison Wesley 2000
99. Stevens Richard, Brook Peter, Jackson Ken, and Arnold Stuart. 'Systems Engineering: Coping with Complexity', Prentice Hall 1998
100. Stix Gary. 'The Triumph of Light', Scientific American Jan-2001
101. Strogatz Steven H. 'Exploring complex networks', Nature 410-2001
102. Sunderam Vaidy and Nemeth Zsolt. 'A Formal Framework for Defining Grid Systems', 2nd IEEE ACM International Symposium on Cluster Computing and the Grid 2002
103. Surendranath Vineeth. 'Slithering through molecules: an overview of Python in bioinformatics', Py magazine 8 2005
104. Thompson B J, Newmark J S, Gurman J B, Neupert W, Delaboudiniere J P, St Cyr C, Stezelberger S, Dere K P, Howard R A, and Michels D J. 'SOHO/EIT Observations of the 7 April 1997 Coronal Transient: Possible Evidence for Coronal Moreton Waves', Astrophysics Journal Letters 517-L151-154 1999
105. Tuecke S, Czajkowski K, Foster I, Frey J, Graham S, and Kesselman C. 'Grid Service Specification', Globus Alliance draft research paper http://www.globus.org/research/papers/gsspec.pdf 2002
106. Uchitel Sebastian, Chatley Robert, Kramer Jeff, and Magee Jeff. 'LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios', Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2003
107. Venugopal Srikumar, Buyya Rajikumar, and Kotagiri Rarnarnohanarao. 'A Taxonomy of Global Data Grids', http://www.chinagrid.net/dvnews/upload/2005_04/05042623219594.pdf 2005
108. Verma Snigdha, Gawor Jarek, von Laszewski Gregor, and Parashar Manish. 'A CORBA Commodity Grid Kit (CoG)', 2nd International Workshop on Grid Computing/ Supercomputing (SC2001) 2001

109. Wallin C, Ekdahl F, and Larsson S. 'Integrating Business and Software Development Models', IEEE Software 19-6 2002
110. Wang Y and et al. 'A High-Throughput X-ray Microtomography System at the Advanced Photon Source', Review of Scientific Instruments 72-4 2001
111. Watson Paul. 'Databases and the Grid', UK North-East Regional e-Science Centre technical report http://homepages.cs.ncl.ac.uk/paul.watson/home.formal/DBandGrid3.pdf 2002
112. Zahn Jean-Paul and Stavinschi Magda. 'Advances in Solar Research at Eclipses from Ground and from Space', NATA Advanced Research Institute conference 1999
113. Zharkova Valentina and Ipson Stan. 'Survey of Image Processing Techniques', EGSO-5-D1_F03-20021029 2002
114. 'About SIMSCRIPT II.5', http://www.simprocess.com/products/simscript.cfm
115. 'AstroGrid architecture documents', http://wiki.astrogrid.org/bin/view/Astrogrid/ArchitectureDocs
116. 'AstroGrid pattern catalogue', http://wiki.astrogrid.org/bin/view/Astrogrid/PatternCatalogue
117. 'AstroGrid Science Problems (The AstroGrid 10)', http://wiki.astrogrid.org/bin/view/Astrogrid/ScienceProblems
118. 'AstroGrid workgroup team responsibilities', http://wiki.astrogrid.org/bin/view/Astrogrid/TspMinutes03
119. 'BETA language homepage', http://daimi.au.dk/~beta/
120. 'Big Bear Solar Observatory', http://www.bbso.njit.edu/
121. 'CDS (Centre de Donnees astronomiques) Strasbourg', http://cdsweb.u-strasbg.fr/
122. 'Committee on Space Research (COSPAR)', http://www.cosparhq.org/
123. 'COST: Component Oriented Simulation Toolkit', http://www.cs.rpi.edu/~cheng3/sense/cost.html
124. 'CSP archive', http://vl.fmnet.info/csp/
125. 'EBML-EBI European Bioinformatics Institute Toolbox', http://www.ebi.ac.uk/Tools/
126. 'Global Oscillation Network Group, National Solar Observatory', http://gong.nso.edu/
127. 'GOES Space Environment Monitor', http://www.ngdc.noaa.gov/stp/GOES/goes.html
128. 'GPDK (Grid Portal Development Kit)', NLANR DAST project, DOE Science Grid, http://doesciencegrid.org//projects/GPDK/
129. 'GPSS World Computer Simulation', http://www.minutemansoftware.com/simulation.htm
130. 'JINI Architecture Overview', Sun Microsystems whitepaper http://www.sun.com/jini/whitepapers/architecture.html
131. 'Kitt Peak National Observatory', http://www.noao.edu/kpno/
132. 'l'Observatoire de Paris/ Astronomical Paris-Meudon-Nancay Observatory', http://www.obspm.fr/
133. 'MERLIN/ VLBI National Facility (Multi-Element Radio Linked Interferometer Network)', http://www.merlin.ac.uk/
134. 'National Geophysical Data Center, NOAA Satellite and Information Service, Solar Data Services', http://www.ngdc.noaa.gov/stp/SOLAR/solar.html

135. 'PARSEC Parallel Simulation Environment for Complex Systems', http://pcl.cs.ucla.edu/projects/parsec/
136. 'Project: SPADES', http://sourceforge.net/projects/spades-sim/
137. 'Prolog: the ISO standard documents', http://pauillac.inria.fr/~deransar/prolog/docs.html
138. 'PVS Specification and Verification System', http://pvs.csl.sri.com/
139. 'RHESSI home page', http://hesperia.gsfc.nasa.gov/hessi/
140. 'SimPy homepage', http://simpy.sourceforge.net/
141. 'SOHO LASCO CME catalog', http://cdaw.gsfc.nasa.gov/CME_list/
142. 'Solar and Heliospheric Observatory', http://sohowww.nascom.nasa.gov/
143. 'Solar Data Analysis Center at NASA Goddard Space Flight Center', http://umbra.nascom.nasa.gov/
144. 'Starlink', http://www.starlink.rl.ac.uk/
145. 'SwarmWiki', http://www.swarm.org/
146. 'The IBM WebSphere Application Server for web services website', http://www-3.ibm.com/software/webservers/appserv
147. 'The NSO (National Solar Observatory) Component of the VSO (Virtual Solar Observatory) website', http://vso.nso.edu/
148. 'Transition Region and Coronal Explorer', http://vestige.lmsal.com/TRACE/
149. 'Yohkoh outreach', http://www.lmsal.com/YPOP/homepage.html

Appendix A. Solar data-grid use cases

The first section of this appendix describes how the solar physics community uses the Internet for research currently - expressed as use cases, and discussed for activities only loosely related to data-grid ambitions. Requirements for enhancements to the existing process are then described. Other speculative directions for Grid based solar activities further into the future are considered briefly.

'The system' is taken to be the network that all observational solar physicists share (not the computer resources of a single research group), matching the scope of the Internet and data-grid. Network interactions are captured by the use cases - required interfaces are between the system and 'actors' (users and external systems). 'Primary' and 'secondary' actors divide those whom the system is intended to serve and others who only have a supporting or incidental role. Each use case describes the sequence of events that should happen when the actors interact with the system, with optional routes for conditional events.

A.1. Use cases for current activity on the Internet

Compiling space instrument data

Ground control operators supply primary solar observation data from spacecraft to the solar physics community. The operators are responsible for marshalling data from the telemetry received and generating the files for the archives.

Actor: Mission control centres (primary)

Use case: Files of data are generated on a public server from the data received from the spacecraft. (The raw telemetry data is not made available.) Header information in the file indicates the time and type of observation, following the Flexible Image Transport System (FITS) file standard.

Example: An image from a telescope may be mixed with framing and error correction information as well as data from other instruments, or fragmented over several transmissions that lack time sequence. The image data is reconstructed and associated with information such as the time over which the observation was made.

Data server administration

The servers on which the data from space borne instruments is stored must be installed and maintained. The fundamental solar observation package to enable the "using solar observations" use case (below) is SolarSoftware (SSW or Solarsoft). Otherwise, system maintenance tasks for administrators are similar to those for any other distributed network.

Actor: Institution system administrators (secondary)

Use case: Perform network maintenance tasks to support computing resources required by solar research institutions (for example, installing and configuring machines, adding data storage devices, removing unused resources, maintaining installed software, administering user accounts, configuring network connections and routers, installing and policing security policies).

Using solar observations

Solar physicists have been making use of recent observations of the sun from spacecraft, notably Yohkoh, SOHO and TRACE. The data from all instruments on each mission are made readily available on-line. Publishing and sharing the primary observation data has greatly helped the validation and evolution of theories in solar physics. Data from future missions including HESSI, Solar B and STEREO are anticipated to be available in the same way.

Actor: Solar physics research groups (primary)

Use case: Scientists locate and then access solar observation data on public data archive servers. The data files are typically organised in a directory tree, by mission, then instrument and then possibly by observation time. The data is viewed using scientific visualisation applications (primarily SolarSoft and IDL, described below in the "sharing SolarSoft" use case).

Conditional: For some instruments, a database may also be searched to help interpret the observation data. (See also "using ancillary solar data" use case.)

Conditional: Applications for an instrument may provide an interface to browse the files (as part of the SolarSoft instrument specific routines).

Conditional: More complicated actions associated with using solar observations are presented as separate use cases (below). They are: "deriving secondary data", "using non-solar data", "using other solar observations", "using solar summary data" and "using ancillary solar data". These may be seen as conditional extensions of this use case.

Example: A scientist interested in the solar atmosphere above active regions may load images captured at the same time from Yohkoh SXT and SOHO EIT for soft X ray and ultra violet images of related phenomena at different altitudes. Once they have located the primary data files, they may then download them for further analysis (see "deriving data" below).

Using ancillary solar data

Scientists require additional data beyond the basic satellite observations. Such data may help tasks such as calibrating instrument observations and searching for observations using a catalogue. Calibration data could represent a light curve data - indicated the intensity associated with the observed images. Such ancillary data is typically stored with the observations, possibly in database records of the SolarSoftware Database (SSWDB) rather than files. (See the related "using solar summary data" use case below as well.)

Actor: Solar physics research groups (primary)

Use case: A scientist wishes to evaluate downloaded observation data from the archive. To prepare the observation for mathematical treatment, data to calibrate the instrument results is also downloaded for analysis preparation.

Example: Two early examples of catalogues are from Yohkoh and original GOES missions that detected X-ray light curves. The data from the GOES missions were analysed to generate an event list catalogue to help the association X-ray surges with active sun phenomena. The initial Yohkoh database (deployed before Solarsoft) included a copy of the

catalogue of active regions from NOAA observations to help identify the photosphere regions underlying the observed corona features.

Using solar summary data

Synoptic information about the sun (i.e. coordinate systems mapping its differential rotation) and observation summaries are required for planning detailed observations and understanding the context of the primary observations data. For example full-disc solar images are available for magnetograms and hydrogen-alpha which provide information on the development of active regions.

Actor: Solar physics research groups (primary)

Use case: A scientist downloads synoptic and summary data from the archive from which they have also downloaded observation data from the same time.

Example: A scientists wishes to direct an instrument toward an active region (possibly at a special location, such as the solar limb). They require synoptic information to direct the instrument to the likely position where an observation will be made. Alternatively, a scientist already has detailed observation of an event. They may then use synoptic information to understand the context of the event (such as its preceding history and neighbouring solar features).

Using other solar observations

Data from ground-based observatories (for example, additional visible light images, magnetograms, and radio based observations) supports research based on space based observations. Other sources of solar data that are less accessible than that characterised by the "using solar observations" case above may be included with this case (for example, data from short missions on rockets and high altitude balloons, which may carry particle, gamma ray and other detectors that do not form images).

Actor: Mission Solar physics research groups (primary)

Actor: Ground-based observatory staff (secondary)

Use case: Scientists request observatories for information of a certain type (that the observatory is known to provide) for a given period. The observatory provides the information either electronically or physically.

Conditional: The observatory cannot provide the data (for example if weather prevented observations).

Using non-solar data

As well as data from direct solar observations, solar researchers may use supporting data from other sources. These include space missions that detect the solar field and the solar wind *in-situ* (such as ACE and Ulysses), and satellites that detect the earth's magnetic field (notably CLUSTER).

Actor: Solar physics research groups (primary)

Actor: Non-solar mission control centres (secondary)

Use case: Mission control centres for satellites detecting magnetic field and particles provides an on-line archive of data from its instruments. A scientist that has downloaded solar observation data then accesses these archives. They search for observations made shortly after the observations (for example, allowing for the solar wind travel time that they have calculated), and download the records found.

Deriving secondary data

Much of the research work of observation based solar physicists involves analysing the raw data to extract structures and patterns. The result of this analysis may be: a traditional graph relation, images corrected for rotation or with instrument artefacts removed, composite images layered from several instruments (typically for different temperatures), movies of dynamic events, correlations of integrated light to images (such as emission curves and spectra for features), or inferred data plots (for example overlaying an image with magnetic field lines). This analysis is typically done on local copies of data downloaded from the archives with direct user interaction (i.e. the scientist logs in to the server hosting downloaded data and analysis software to run scripts). Scientists use the derived data to guide their investigation, and eventually to validate theories (being published in technical papers alongside the described scientific findings.) The "workings" that let up to the published results are not typically shared (and may even be destroyed, leaving only the original data and final method used).

Actor: Solar physics research groups (primary)

Use case: A scientist performs mathematical transformations on downloaded solar observation data and generates graphical representations.

Conditional: An interactive cycle of adjusting analysis routines and their parameters may be entered, before results are finally generated that apply to the scientist's developing theory.

Sharing SolarSoft

SolarSoft contains a collection of routines (and a few objects) that extends the Interactive Data Language (IDL). It is interpreted (rather than compiled) with strong data structure and library support for scientific analysis and visualisation. A suite of routines is typically associated with each type of instrument data.

Actor: Package programmers (secondary)

Actor: Package users (as solar physics research groups, above)

Use case: Programmers design and implement functions for the data that an instrument generates. The functions are made available in modules, forming part of the SolarSoft package with the directory structure and SSWDB (if used). Users configure their environment for SolarSoft and load the modules from the server to enable solar data analysis.

Conditional: The programmers may return to their software for maintenance tasks. Once the enhancements are completed, modified and new functions are made available.

Recording event catalogues

There is a continuous record of major solar events. The sunspot (and solar cycle) catalogue goes back several hundred years. Classification of sunspots and records of magnetograms and flares are more recent. On-line access to electronic copies of this data is patchy.

Actor: Solar institutions (secondary)

Use case: Data from observations is analysed for new events. Such events are categorised (for sunspots and flares) and catalogued. The catalogues of events are then published (physically and electronically).

A.2. Other existing solar network applications

Examples of successful applications on the Internet build on the use cases presented to support demonstrate the value of worldwide data sharing in solar physics. Each of the following examples could be expanded into their own systems for use case analysis, though this is beyond the scope of this document. Here they illustrate that networks are already being used for Grid-like applications.

Global Oscillation Network Group

GONG is a worldwide network of ground-based observatories dedicated to heliosiesmology. Observations of Doppler shifts in the photosphere are collected for longer periods than each single observatory's viewing time. The captured information of vertical movement in the chromosphere may be analysed for long period repetitions that indicate deep oscillations passing through the sun. As a network of cooperative data sharing and analysis which has produced new scientific findings, GONG is a paradigm for Grid style solar applications (though it is a closed group of dedicated resources).

Space weather reporting

The solar science community's interest in space weather largely concerns solar wind and radiation detected at the earth. This affects the earth's atmosphere, and human activities such as satellites, aircraft crews and power distribution (clearly of interest to other communities). Current information is made available on the Internet primarily by NOAA, which makes solar and atmospheric data available with information such as anticipated climate effects. This is also a candidate example for an emerging Grid-like application, where data is analysed from different sources and information derived is distributed in near real-time.

Amateur and educational interest

Space science is of great interest to school children, amateur astronomers and other groups. NASA has a major commitment to public outreach, and other institutions involved in solar science also benefit from publicity. Popular science television programs and books now use images generated by solar research directly. The Internet also provides amateurs with a great deal of accessible information. For example, NASA's image of the day frequently shows

results from solar research and provides links for people to find out more. Though this is just an example of public data resources on the Internet, it indicate how a wider public may be interested in supporting a solar Grid network.

A.3. Use cases making better use of the existing network

Distributing data archives

Currently the storage network of satellite archives is fixed, and scientists initially need to identify which archive provides the data they require. Typically, if the archive required turns out to be remotely located, subsequent search and download may be further delayed. The system may become more responsive if existing and new stores can be more dynamic, so that data can be copied and moved for optimised access. The system would automatically maintain caches and backups along with information on data availability. Migration transparency is provided, as the actual location of the data in the network would not be required by the user (though typically a definitive archive of data would be maintained). Peer-to-peer networks exemplify this style of distributed data access.

Actor: Solar network administrators (secondary)

Use case: An administrator allows their server on the solar network to be classed as a data node (possibly representing a cache or mirror as well as a data store). As information is requested and transferred via the server, the system uses available cache space for frequently used data. When a copy is made, the system records that the data is cached (with other catalogue information used for searching the network). When the same data is accessed again, the users would experience quicker response.

Conditional: If the data does not have a constraint that copies must be left on the original data store, then the system may delete the original copy as it moves it.

Discussion: The principle of a distributed archive is closely related to existing Internet technology for proxy caches and mirrors. However, there is a slightly different principle here, borrowed directly from peer-to-peer networks, as the archive is fundamentally distributed. This fits well with the "easier access to data" use case below, where the real source of the data is irrelevant to the user. The principle also fits well with the Grid paradigm of flexibility between computing resource providers and users. This paradigm applies strongly in solar physics, where those providing the archive resources are likely to remain closely tied to the observatories and mission control centres. If there is a widespread need for institutions to make the definitive archive data for their instruments highly available, then this requirement may become simply a need for standard Internet caches. Proxies caching data for local institutions will end up storing local copies of the frequently accessed catalogues discussed above without special provision. Meanwhile, isolated observatories would use distributed network caches to save unnecessary traffic via their limited network resources (see "greater access to other data" below).

Greater access to other data

Grid technology offers control network traffic allowing observatories to provide restricted public access to data that is currently privately held (and only available on request). This would

allow the researcher community easier access to ground-based observations, allowing data from many more instruments to be included in scientific investigations. The system should ensure that publishing the private data (which may be in proprietary formats) in generally accessible formats should require very little manual effort. Additionally, small isolated observatories should make use of the distributed data network (described above) to federate their archives, perhaps with the system's large central data servers acting as proxies for the observatories stores. Risks of forcing extra work and high volumes of network traffic must be avoided if open access to observatories' archives is to be successfully accomplished.

Actor: Observatory record administrators (secondary)

Use case: An observatory administrator registers their archive with a public server for the solar network (without providing a dedicated server themselves). The solar data-grid system is then able to access this information without directly exposing it to the Internet. The system may also copy data to its public servers if it is frequently requested (as a distributed data archive, below). The system may enforce a limited access policy if the observatory requires it (with logged, authorised access possibly generating charging information).

Conditional: An observatory's data may well not be in a format that can be used by the solar system's search routines (nor by typical solar scientists' research applications) and may have no catalogue information. In this case the data must be ported to standard formats when accessed (or ported to solar system caches), and marked up with standardising meta-data for catalogue search compatibility. These translation functions are similar to the "generate information" use case below, and therefore represent conversion scripts or procedures. Package programmers drawn from the data-user community (rather than the observatory that logs the data) would provide translation operations. Note that the original data should not be transformed itself (as this is time consuming, unnecessary and risky).

Discussion: The main goal of this use case is to overcome the obstacles that currently prevent ground observatories making their observations publicly available. Institutes operating small observatories typically have less investment and international interest than space missions. Therefore, it should be as easy as possible for their existing data archives to be adapted to a solar grid. There must be no requirement for the existing record structure to be mapped some standard, and the system administration burden (for example moving files, policing fire-walls, protecting local resources) should be absolutely minimal.

It has also been assumed that small observatories are reluctant to publicise data by making it available to absolutely anyone via the Internet. Observatories such as Learmonth and Big Bear only provide summary data, for example. If large archives were directly available, the observatories' network may be choked by wasteful public requests. By using request based distribution they may also retain more control of their data. However, though the Internet is characterised by anonymous public access, authorisation and traceability primary concerns in Grid applications - so resources may be protected and users can be billed for service. Therefore, observatories should easily be able to enforce the control they desire, such as limiting data to trusted users, limiting the quantity of data that may be used and even tamper-proofing data.

These use cases ignore commercial implications and possibilities of solar institutions providing Grid services, assuming open community policies. Auditing user access would be intended to impose fair access by providing quota allocations. However, there are several economic models for Grid services that could be deployed based on genuine money or usage credits. These typically associate the identity validated by authorisation services with credit channels. If smaller observatories (or other institutions) wish to use the Grid to generate revenue, requirements in this area could be greatly expanded.

In describing the use case above it was assumed that each observatory's data server would be a normal node on the solar system network. If they were equivalent peers to other servers, federation and proxy formation should arise naturally as central servers cache frequently accessed data. However, an alternative would be to actively hide the observation servers. The central system data servers that they are directly connected to would then become formal proxies, and Grid service transmission of data would not go to the observatory. In this case, the data server may have a special on demand protocol for the observatory connection, or may pull data periodically and cache as much as possible. Alternatively the central servers may represent asynchronous gateways that forward unsatisfied requests to the observatory system, so the observatory may publicise specific data (in a similar way to current request based systems).

Ground observatory based data provides more sporadic coverage than satellite data (as night and weather prevent observation). Additionally much 'data' may be only recorded on photographic plates, making the effort to provide on-line access insurmountable. Such detailed data would very rarely be required by scientific investigations (such as long term luminosity or solar cycle trend studies). These features make it essential that the data network catalogues provide clear information on what observations are available (see "generating information for data" below). Collating information from such sporadic archives has been very successfully achieved for GONG.

Easier access to data

Selecting the right observational data for solar physics could be made far easier. Searching could be more automated, allowing researchers to find appropriate data without knowing its source (location transparency) or the way that data is stored (platform transparency). A distributed data network supports such transparency, as it maintains internal meta-data about archive content and availability. Scientists should also avoid downloading everything and then examining it by directing searches to extract exactly the sub-set of observations required. Note that the range of archives queried and the richness permitted in the queries may be initially limited, but become broader and more complex as the system is developed.

Actor: Solar physics research groups (primary)

Use case: Scientists submit a semantically rich query to their local server. The system finds files (or other data records) that provide matching information from the whole network. The system filters the data records before returning them so that just the required sub-set is presented.

Conditional: Before the data is returned to the user, the system may present a summary of the matching data and confirm that it should proceed with downloading.

Example: Specialist examples illustrate what may eventually be possible in such an environment. A query of the 20 strongest hard X ray events over a 2 month period may examine data from both Yohkoh and SOHO archives, and present images for the peak of each event. In another case, a query for the most detailed chromospheric image of a certain active region may return partial disc images from several wavelengths throughout the lifetime of the active region from ground-based observatories and spacecraft (or full disc images if detailed images when these are not available).

Discussion: The assumption above is that a rich query would be possible (ideally close to natural language). In fact, existing search applications use text matching (for example, Internet search engines and UNIX filters - that allow regular expression) and logical expressions (for example, SQL and GUI forms). These are the only options likely to be achievable for solar archive searching in the foreseeable future, and they are limited. To construct an SQL query the user must know something of the data records - for example column names and data format to query over a date range. Regular expressions are also limited - for example a date range search may fail to match some formats and match unwanted data (such as angles or three dimensional coordinates). However, by using synonyms and conversion functions for the user and marked up schema for catalogues, more flexibility is permitted.

Other ways of querying solar data archives may also be considered. It may be beneficial to represent the problem as a set of relational databases with unknown schema, or objects with hidden members and different methods. These complex representations fit some of the data directly, and simpler data stores (notably files with headers) can be mapped to them easily. Advanced database techniques and object-oriented patterns may then be applied. Possible natural query methods may come from artificial intelligence. If the data to be searched could satisfy a grammar, Prolog-like propositional queries may be possible. It may even be suitable to use an evolutionary technique such as neural networks to discover semantic structure in the data. See the discussion for speculative "data resource to information resource" section below for other possibilities that add semantic value which could help natural searches.

Progress on the searching issue can be made by case studies. By finding many diverse example queries, representations of the raw data and methods to search these representations (and maintain them) can be found. The structures and software algorithms may then be tested against further case studies before being implemented. Other related scientific areas should also be studied so that data mark up and distributed catalogue-based search strategies are not tied in to the solar domain.

Generating information for data

To support easier access to data, more information should be made available about the data at its source. This would extend FITS header information and database catalogues, for example. Such information catalogues allow a searching application to skim the lightweight meta-data before accessing the relevant raw data files. This information fits closely with the catalogues that record data availability in the "distributing data archives" case above.

Generating such information should be automatic, minimising the overhead to institutions providing the data.

Actor: Package programmers (secondary)

Use case: Programmers (drawn from solar research institutions) write scripts to generate information tags for certain types of data, depending on the data source. The package programmers, who also write instrument specific routines, know best what interesting tags may be applied to the data. They should follow guidelines (and use validation tools) so that the tags are useable by the search procedures described above. Their scripts are then run automatically (as batch jobs, or at the time the data is first queried) to generate additional information on all data.

Conditional: Alternatives could be: self generation, catalogue design by base stations / mission software engineers. Also issue of enforcing standards here, or free for all where anything ever considered useful is available

Conditional: Users (typically solar research scientists) note that information can be extracted from a certain data store. As a data store is queried in certain ways, the server notes that its data contains a certain class of information. The system tags the data to indicate the information for future queries.

Example: Yohkoh records where local areas of bright X-ray are seen may be tagged as likely active region loops. A script may search data for intense areas statistically, or use another catalogue of active regions to generate cross references.

Discussion: Package programmers have been described designing the scripts to generate meta-data above. They seemed the most reliable agents for generating useful information reliably. Though it represents extra effort for people that are already possibly on tight time scales, it seems a good one-off investment of effort for long-term benefit. It is anticipated this group is likely to overlap with research scientists, who may well generate these information scripts too (just as they also contribute to the existing SolarSoft).

Alternatives to package programmers were considered. Catalogues could be generated automatically, perhaps directed by data users notes on information they are extracting from data recovered from the archive or guided by archive search patterns. This is likely to generate little useful information, and generate it unreliably. Alternatively mission control centres and observatories that provide the archives (or mission software engineers that write the routines that generate the instrument's data) could provide it, but it would not be so close to their interests.

Indexes are related to catalogues, but serve only to provide quicker data retrieval. Design of indexes relies on understanding data lookup and usage patterns, and relates to caching strategies that would be employed for the "distributing data archives" use case. Files are currently ordered by observation date, which provides one *de facto* index. Other orders and groupings could be by such things as solar latitude, the currently catalogued sunspot index or a derived flare frequency.

Remote pre-analysis

Pre-analysis is closely related to the above requirement for easier access, where analysis (as well as selection) of the data is performed before it is downloaded. Rather than requesting a subset of data, scientists could requests abstractions from the raw data. (Note, the purpose of such a request is for remote treatment to save network and local computation resources. The scientific investigation still relies on apprehension and qualitative evaluation of the underlying data.). For example, basic statistical operations could generate averages and trends, or coarse resolution images. Alternatively, interesting events may be identified and extracted from the data.

Actor: Solar physics research groups (primary)

Use case: Scientists submit a query with functional elements to local server. Once the system has identified suitable data (which may be remote, as above), the functions are applied on the data at the nearest available computing resource (see below for "distributing analysis tasks"). The results are then returned to the user.

Conditional: The system may indicate the input data found for the function before the data is retrieved and processed. The system may also indicate initial results before the function is carried out on all data. At both points the user may interrupt if the query or function prove unsuitable.

Example: (This example is a very specialist case, intended to test the extreme limit of what might be available.) A scientist wishes to find the average area of the chromosphere covered by bright plage for each month over an 11 year solar cycle. They generate a function that can identify bright areas of helium emission images and divides the bright area by the total area of the solar disk. The system finds daily images for the period. If the images come from several different sources, the system may also need to normalise all data initially. The plage function is then performed on computational servers (possibly including idle workstations) at the institution where the data is stored. The averaged results for each month are then returned to the user by the system.

Discussion: The type of analysis required is highly domain specific. Though more examples would help system implementation (as for "easier access to data" above), the requirements for the environment where pre-analysis can be done must be distinguished from requirements for specific operations. For the purpose of this use case and the requirements and design derived from it, just the generic features of the context in which analysis can be run should be identified (rather than specialist functional operations).

The existing SolarSoft structure allows separation of the analysis environment from the user's operations (though it is not clear whether SolarSoft was considered as a model that could be used for other (even non-scientific) disciplines outside observational solar physics). A researcher knows that any server with SolarSoft will have an environment where their scripts can run - the IDL interpreter, key environment variables and modules used by their script will be found. However, the researcher is still free to code whatever operation they desire into their script - they are not limited to the functionality provided by the existing modules (in IDL, SolarSoft or the specific instrument being used).

Despite good commonality in the current network, any system for dispatching data analysis tasks to distributed resources should be flexible enough to cope with variation in the environment. Different versions of the software and variations in the availability of modules will be encountered at best. (At worst, different types of modules will do the same tasks, requiring dynamic mapping of service requests to component interfaces. This is possible if services and interfaces are described, permitting component "introspection" in the system.) The run time environment should also be expandable for developments in solar applications and for adaptation to other environments.

Distributing analysis tasks

Some analysis tasks of solar data are computationally intensive. Rather than accessing local computing resources (high power servers or networked workstations) directly to carry out these tasks, scientists should be able to submit jobs to distributed computing schedulers. These would automatically find suitable available resources for the task. The main benefit for the researcher should be access to greater computational power. Also, pooled resources may operate more efficiently as intensive tasks are rare, and a large community should be adequately served by a just few powerful computing centres.

Actor: Solar network administrators (secondary)

Actor: Solar physics research groups

Use case: The administrator registers servers (and possibly workstation clusters) as computing resources. The task management aspect of the system monitors these resources for availability and current loading. When a researcher intends to run a task (possibly by submitting a query with an analytic component as above), the system allocates resources to the task.

Conditional: Computer time used may be accounted for users. If the user is unauthorised to use distributed computation resources or has exceeded their allowance, the task may fail (or the client may run the job locally).

Example: Distributed computing tasks have been one of the first successful deliveries of Grid technology. Task distribution is established in small networks of fairly homogenous resources (especially clusters of workstations sharing an operating system). Typically tasks may be started as batch jobs, or a central server may maintain a job dispatch queue. Now, Grid enabled systems like Condor-G can distribute tasks across large, decentralised and quite heterogeneous networks (with resources from supercomputers to Linux farms). The key difference in a Grid like distributed system is there is no central point of control, whilst local policies to protect and share resources are still guaranteed.

However, though it is likely that this use case could be met by existing technology, there is still need to identify the bespoke elements of the solar system. If IDL is to be used as the common environment (as discussed in the "remote pre-analysis" case above) a strategy to break up and control distributed IDL processes would be required.

Though this use case is introduced primarily for large analysis tasks initiated by scientists, the same infrastructure may be used for the distributed tasks generated by the system itself. The most significant of these would be the catalogue searches described under

the three data use cases above. Also deriving from those use cases would be the tasks that generate the catalogues and manage the data movement strategies for caches and replicas.

A.4. Speculative future grid applications

There is a long history to the argument that making information readily available via technology enables more rapid scientific advancement (predating the Internet and digital computers). Information may be seen as a stepping stone from data to knowledge, and that providing shared information resources enables a new class of reflective investigation. It is also the case that computing time may simply be saved by sharing derived data and caching frequently required results on a network. The following two sections describe ways in which the goal of open information may be accomplished in the system of solar data.

Publicising analysis

Scientists should be able to make analysed data more readily available. These results may be made available before the associated papers (or equivalent) are published. Derived data that is not directly published may also be made available. This allows results to be analysed easily, may save duplication of effort in analysing data. These possibilities should improve the quality and scope of scientific investigations.

It is imagined that information and knowledge repositories could improve scientific method by allowing clearer objective review of data quality, and allowing new types of investigation. Scientific research may expand in the area between a team's own finding and received wisdom from others' published work. Easy online access to analysed data may allow scientists to find unexploited data, patterns in research and innovative new directions. It may also make interdisciplinary research easier.

For example, following from a distributed analysis task of the type described in the previous section, the analysis data and the procedure used could be published as well linking the paper to the archives that the system acquired raw data from. Mathematical models used to derive information about temperature, magnetic fields, and other conditions from the data may be applied to future data. Links to other data stores that could be analysed in a similar way could also be provided.

Where necessary, individuals' and institutions' intellectual property should also be preserved. Services to "watermark" derived data may be used to associate the information with the original scientist that generated it. Also, public work may be protected by reservation meta-data (perhaps applying economic bidding models); in this way the scientist's claim to the analysis that they may use in forthcoming publications is declared. Policies that allow different levels of access to the data should also be upheld, for example providing summaries for anyone, detailed workings for known peers, and methodology to immediate colleagues.

As analysis of data may be accomplished in the remote and distributed environment of the system, and data availability is automatically managed, it is a simple step to publicise analysis. Rather than destroying results of analysis once delivered to the user, the system could mark up, catalogue, cache and distribute the derived information as it does the observation

data. There may be an explicit role for a publisher, who validates the information, associating the results with the source data and methods used.

Note that in the current scientific environment of solar physics, analysis methods are not normally provided and shared. This is typically because the scientists that develop the methods are aware that their methods are rough and not of open source software grade (rather than intellectual jealousy). Hiding analysis using tiered access policies is one solution, whilst others may allow summaries of methods to be automatically generated to a common standard, or provide tools to compile rough analysis procedures to higher quality code (as with source code "beautifiers").

Data resource to information resource

To support the capability for distributing published derived information there may be additional methods the system employs to transform data resources to information resources. This overlaps with the discussion of possible ways to automatically generate catalogue information in the "easier access to data" use case.

Software agents may automatically generate meta-data by (either working automatically like Internet robots or being dispatched when a search was done). These may be guided by finding shortcuts for previously successful searches, or be unguided, finding their own associations (a knowledge base generation task). It is likely information generated in this way could be interesting, but would not necessarily be useful. Closely related is enabling the server to record the history of searches (this may simply be a cache, like that used by a good relational database server). The first researcher using the query type would be using a slow, serial search (such a file tree walk) but the subsequently a similar search could use a quicker, directed search (perhaps like a database cursor). Agent searches can achieve similar results by leaving marker trails for other agents to detect.

The possible content of catalogues has been discussed above, and represents the initial information resource. Catalogues may become increasingly complex as package programmers develop new ways to generate them. It is possible to imagine a catalogue entry for an active region, which would indicate records from different instruments that detected data for its duration. It may also indicate other information, such as catalogued sunspots for the active region and citations for papers that the event was analysed for (as part of "publicising analysis" case above). The catalogues should be flexible enough to be expanded for such possibilities (and others not currently imagined). Catalogues may also be flexible enough to cover quite different types of records from non-solar archives (for example, enabling solar-stellar collaborations, currently proceeding on spectra and luminosity comparisons).

Meta-data may also refer to parts of a data record within a file (using random file access reference, and information on how a file is composed). A search could perform tasks like picking out the a few bytes evenly distributed, or even tracking the pixels for a region on the surface as the sun, corrected as the sun rotates in each image.

Arrays (or vectors and matrices) are frequently and widely applied to the data set for solar observations. The data has obvious dimensionality in time and space (either the two dimensions of a telescope's field of view or the surface of the sun, or the three dimensions of

the interior and atmosphere of the sun). Observations may also be placed on dimensions of temperature, electrical current and magnetic field strength, pressure and composition, electromagnetic wavelength and energy, velocity and other physical properties.

Information resources may use such arrays naturally. In catalogue generation and searching, divided tasks may operate over different ranges of a shared array. It may be necessary to map different formats of data into a common array. An example application of this may be developing a unified representation of the solar atmosphere by processing data from different instruments that observe different altitudes. IDL, with its strong array functionality, may be well suited for running these tasks.

Models

This discussion has been focussed on observational based solar physics (and the observation archives). Solar physics also advances thanks to models of the sun (especially of the interior only indirectly observed, but also for the atmosphere and dramatically observable phenomena). These are typically computer based now, and the data produced by these models (as well as the models themselves) may be added to the solar data network. Observation based scientists would be able to download (or even generate) simulation data alongside data from instruments, whilst theoretical scientists could feed observed data directly into their simulations.

Many of the challenges for opening access to model data transfer from observatory archives, as there is a great variety of data (in different formats, providing different information and at different levels of quality) on currently isolated resources that may have special access policy constraints. Some of the comments that apply to the reluctance in the community to share scientific workings above also apply strongly to solar modellers, and where appropriate policies to protect sensitive information and hide methods in development should be guaranteed.

Current solar models typically require significant computing resources to run. Their algorithms take advantage of parallel computing strategies (for example, simulations are designed to be decomposed to layers or cells that can be modelled partially independently). This type of application has been very successfully transferred to Grid distributed computing environments previously (in particle physics, biochemistry and climatology). Therefore, it may be that the solar data network may serve also as a solar computer modelling network if sufficient computing resources are available. To accomplish this, requirements for the "distributing analysis tasks" use case above would need to be extended. Notably, performance and coordinated scheduling of distributed tasks would become a key concern (as the "high throughput" task of analysing distributed data is transferred to a "high performance" parallel computing problem).

Additionally, solar physics has some overlap with theoretical stellar modelling. A closer relationship between these two groups may be possible if stellar simulations were also generally available. Once again, a clear interface would be required, with automatic conversion tools and synonym mapping at gateways. The distribution of solar and stellar simulations provides another class of information resources. The same system would be used for refining models and understanding observed data.

Space weather prediction

Rather than passively displaying current space weather conditions, with short term predictions, background processes could apply forecasting heuristics to predict conditions into the near future. Such agents could be extensions of the software that analyses data in catalogue generation. The agents would require a model of the sun that would infer future weather from the observed current state (possibly forming part of the solar model data described in the previous sub-section). They would require access to the data source (providing the latest observation data) and a forum for presenting forecasts (extending existing web pages or broadcasting a news service, as described below) as well as computing resources. The models may generate confidence indexes with their predictions (for example, indicating a percentage chance that an emerging active region will become eruptive on the next solar rotation).

Terrestrial climate models have benefited from continuous simulations that take regular input from observed data (in real-time) whilst they run. These models are now also beginning to take account of space weather conditions. Therefore, there may be merit in combining terrestrial and space weather models with observed earth and space weather together. This should improve both the theoretical understanding behind the models, and weather forecasting.

Live news service

Closely related to the area of space weather is continuous solar monitoring. Solar conditions for the last day are typically available to all interested groups on the Internet currently. Data could be made publicly available in near real-time if available current observation data were analysed automatically by software agents. Details of flares and other rapidly evolving events could be broadcast within minutes. This may be of interest to amateur groups (perhaps making their own observations at the same time) as well as professional scientists looking out for specific events and others with a professional interest in solar activity (such as air crew).

Overlap with other physics research

As information (as well as data and published material) associated with solar data becomes more available, scientists from related fields may make more use of it. Stellar physicists have been assumed to be the initial group that could take advantage of solar observations, but the intersection with earth scientists has also been highlighted. Beyond these groups, high energy physicists could examine solar data for behaviour equivalent to that observed in particle accelerators or theoretical models, and cosmologists could make greater use of solar data for their theories of the evolution of the universe. Additionally, as scientists from these related disciplines develop their own Grid infrastructure, interesting information and resources (both data and physical) may be available to solar physicists.

Appendix B. Solar data-grid goals

B.1. Abstract, technical goal decomposition

The 47 abstract solar physics data-grid goals that were generated before the EGSO project started are listed below. Extra digits are added to each goal's parent goal identifier, so the number of digits indicates their position and depth in the tree. None have multiple parents, and, unless indicated by an 'or' prefix of the second goal, there is an 'and' relation between all goals at the same level ('or' pairs represent alternative goals to satisfy).

Tree ID	Requirement
1.	The data-grid should provide an infrastructure for solar physics operations on information resources.
1.1.	The data-grid should provide an infrastructure for distributed resources.
1.1.1.	Data resources should be co-operational.
1.1.1.1.	Heterogeneous data systems should provide similar service.
1.1.2.	The network should be transparently scaleable.
1.1.2.1.	Any reconfiguration required to use new resources should be dynamic.
1.1.2.2.	Resource discovery should be automatic.
1.1.3.	There should be no central point of failure.
1.1.3.1.	Control should be distributed.
1.1.3.2.	Interaction should be characterised by co-dependencies rather than slaved relations.
1.1.3.3.	Authority should be elective and dynamic.
1.1.4.	It should be possible to use resources that are not dedicated to the application.
1.1.5.	The user community should be open.
1.1.5.1.	Network components should not require configuration changes to include new users.
1.1.5.2.	Single user authority should be sufficient for all resources to enforce policy.
1.1.6.	Resources should be available for arbitrary computation.
1.1.6.1.	Compute resources should advertise what services they make available (environment and performance).
1.1.6.2.	Compute tasks should be distributed to available resources based on policy.
1.2.	The data-grid should meet the domain specific needs of solar physics.
1.2.1.	Data should be represented with its Solar observation context.
1.2.2.	Queries should allow Solar semantic discovery.
1.2.3.	It should be possible to process data (e.g. using SolarSoft) before download.
1.2.4.	Scientists should be able to extract information from data.
1.2.4.1.	Information discovery should be automatic based on procedures designed by scientists.
1.2.4.2.	Information should contribute to meta-data used in guided information discovery.
1.2.5.	Heterogeneous data representations should be comparable.

1.2.5.1. Conversion of data representation should be automatic based on procedures designed by scientists.
1.2.6. Observatories and mission control centres should not be required to perform additional tasks to make their data available.
1.2.7. Scientists should be able to publish generated data (e.g. analysis results, simulation) and information.
1.3. The data-grid should operate well.
1.3.1. Storage media should provide data quickly.
1.3.2. Redundancy should be supported at the storage level to protect disk failure.
1.3.2.1. Redundant data should be accessible immediately in case of failure.
1.3.2.2. (or) The data store should be unavailable while backups are restored.
1.3.3. The storage capacity should be scaleable.
1.3.3.1. The data store should dynamically detect and use new capacity.
1.3.3.2. (or) Configuration of the data store to use new capacity should be straightforward and require minimal downtime.
1.3.4. User access should be limited to known, authorised users.
1.3.4.1. Unauthorised access should be prevented.
1.3.4.2. It should be possible to categorise users to enable intelligent policies (such as limited write access).
1.3.4.3. User activity should be logged to support service agreements (accessibility, payment, user license).
1.4. The data-grid should include information resources.
1.4.1. The user should be able to query for data and for results to be returned.
1.4.1.1. The user interface should be intuitive and easy to use.
1.4.2. Data mining based on characteristics and patterns should be possible.
1.4.2.1. Metadata should be used to organise the data.
1.4.2.2. Categories and hyperlink graphs should be used to enable data-mining.
1.4.2.3. Patterns used in the metadata structures should be presented to the user.

B.2. EGSO goal analysis

Nathan Ching decomposed EGSO's finalised 156 requirements into a goal heirarchy, identifying 13 missing parent requirements. All 169 goals are listed below, as they are not definitively published elsewhere. Numbers, prefixed with 'R' for documented EGSO requirements and 'A' for added goals, identify the requirements, listed with their parent goal identifier.

ID	Parent	Requirement
A01	NONE	We need a European Grid for Solar Observations
A02	A01	The system shall facilitate user access to resources
A03	A02	The system shall control manage and track the usage of all resources.
A04	R001	The system shall be able to include as wide a range of users as

		possible.
A05	R001	The system shall allow users to submit queries.
A06	R001	The system shall allow users to manage queries.
A07	R001	Users must be able to view metadata.
A08	R001	The data shall deliver data to the user in accordance with their needs.
A09	R006	The system shall not discriminate against data providers
A10	A04	The system shall provide interfaces to suite a variety of user requirements
A11	A10	The system shall not discriminate against users on the basis of the computational resources they have access to.
A12	A10	The system shall allow users to express preferences and create a persistent personalised configuration.
A13	A05	The system shall allow a variety of types of query.
R001	A02	The system shall facilitate the end users' access to the resources made available by the data providers
R002	R018	The system shall maintain information describing the data provider's holdings and access methods for all forms of data and metadata
R003	A05	The system shall allow users to search for data providers who meet certain query criteria.
R004	R003	The system shall allow users to directly view the (non-sensitive) metadata relating to any given provider.
R005	R003	The system shall allow authorised parties to modify the data provider description metadata thorough a simple interface.
R006	R001	The system shall be able to include a wide range of data providers as possible
R007	A09	The system shall not exclude data providers on the basis of the way in which their data is archived and stored.
R008	A09	The system shall not exclude data providers on the basis of the way in which their data is accessed.
R009	A09	The system shall not exclude data providers on the basis of the completeness of their data or metadata holdings.
R010	R001	The system shall provide a means for data providers to register with the system in order to indicate their willingness to make their resources available.
R011	R121	EGSO should, as far as that information is accessible to the system, maintain information about the history of datasets.
R012	R016	EGSO should provide the means to know, within the restraints placed by the individual users on disclosure of their personal activity, the history of any dataset and its usage.
R013	A02	The system should be able to facilitate access to additional, non-data resources
R014	A02	Where possible, similar access methods should be used for all resource

		interactions
R015	R134	The system should be able to handle additional information provided by the resource providers that may allow the system to better monitor and allocate those resources
R016	A03	The system should provide users with complete acknowledgments for all resource providers utilised in a specific session
R017	A07	The system should allow metadata catalogues to be located according to a common classification of their contents.
R018	R001	The system should provide the means to perform the mapping of multiple datasets to a common framework
R019	R002	The system should permit the identification of relevant instruments through the referencing to a common classification of their data products or operational modes.
R019 (R022?)	R025	The system should support the definition and implementation of a community metadata standard.
R020	A04	The system should permit, where possible, the use of common terminology or definitions to define the input search parameters.
R021	A01	EGSO should seek to maintain interoperability with other similar projects in the solar or related domains.
R023	R002	The system should support multiple representations of metadata as supplied by data providers.
R024	R052	The system should allow for the conversion of multiple metadata representations to a common format based on a defined data model.
R025	R002	The data model should be consistent with community metadata standards.
R026	R006	The data model should be valid for all known solar physics observations.
R027	R026	The data model should be usable for observations made from observatories at a fixed position on the surface of the earth.
R028	R026	The data model should be usable for observations from earth orbiting satellites, balloons, or other moving observation platforms.
R029	R026	The data model should be usable for observations made from a fixed position in a heliocentric coordinate system.
R030	R026	The data model should be usable for observations made from observatories moving within the heliocentric coordinate system
R031	R006	The data model should be flexible enough to accommodate new instrumentation or modifications to its structure in a dynamic way.
R032	R031	The data model should allow for the addition of new metadata elements
R033	R031	The data model should not make assumptions about contentious scientific matters in its structure.
R034	R002	The system should only handle metadata that is freely available.
R035	R53, R054	The system should provide to the users a Unified Observing Catalogue, which is the result of merging the metadata from multiple data providers

		into a single common representation.
R036	R035	The UOC should contain information on observations made of the Sun and of the heliosphere.
R037	R035	The UOC should contain complete information about all known observations, regardless of the availability of the underlying data.
R038	R035	The UOC should contain sufficient information to facilitate a broad range of searches for solar data.
R039	R035	The UOC should organize information according to the predefined data model.
R040	R035	The UOC should not contain information that has dependencies on proprietary software.
R041	R156	The system should allow portions of the Unified Observing Catalog to be exported to the users for access on their local machine
R042	R041	The exported portions of the UOC should attempt to be as self-contained as possible
R043	R041	The system should export portions of the UOC in a format that is easily human readable.
R044	R041	The system should export portions of the UOC in a format that allows for efficient machine reading.
R045	R042	The system should export portions of the UOC in a format that is self-describing
R046	A07	The system should provide fragments of the Unified Observing Catalogue containing only the information relevant to the parameters provided by the user.
R047	R53, R054	The system should minimize the dependencies on ancillary data that may not be part of the common data model.
R048	A07	The system should assist users in accessing observational metadata that is not part of the common data model
R049	A07	The system should provide the means to track the creation date and revision level of the portion of the UOC being utilised
R050	A07	The system should provide the means to check the reliability of a given fragment of the UOC against a known reliable version of that same fragment
R051	R60, R106	The system should provide access to sources of metadata that provide additional input for the user in the selection of data of interest.
R052	R023	The system should support the interaction with multiple sources of metadata with non-consistent metadata formats.
R053	R052	The system should support one or more metadata models that are able to provide a common framework for multiple metadata representations.
R054	R052	These metadata models should be sufficiently flexible to incorporate new or richer metadata content as it becomes available.
R055	R078	The system should provide access to existing catalogues of derived

		metadata
R056	R078	The system should provide access to new catalogues of derived metadata as generated through feature recognition software.
R057	R059	The system shall provide the means to access annotations, corrections, and additions to the metadata for a given dataset
R058	R53, R054	The system shall provide the means to store metadata annotations that are not incorporated into the authoritative metadata for those data
R059	A07	The system shall provide the means for the users to access the metadata annotations when reading in data files for which such annotations might exist.
R060, R106	A05	The system should provide for the use of multiple representations of a given dataset as part of the data selection process.
R061	R060, R106	The system shall allow for the use of existing representations of the data as provided by a data provider in the user interface
R062	R060, R106	The system shall be able to generate additional representations of the data, with the aim of aiding data selection or reducing the transferred data volume, for use in the user interface or as part of the data reduction process.
R063	A11	The system should allow data requests to be made through online web-based forms.
R064	R090	The system should provide the interface to allow automated predefined data request.
R065	R066	The system should provide a standard interface to its different components to allow third-party access to its features.
R066	A10	The interface should be generic and accessible enough to allow some users to develop interfaces with any programming language that supports network connectivity.
R067	A11	The GUI should not be tied to a particular operating system
R068	A10	The GUI should rely on cursor-based input on graphical representations of the data wherever possible or reasonable
R069	A11	The GUI should maintain a high level of interactivity even in cases of large network latencies between user and remote servers
R070	R006	The interface with resource providers should be designed so as to minimize requirements on the providers.
R071	R070	The system should not require any modifications of existing data and metadata products made available by the providers.
R072	R070	The system shall be capable of accessing remote data sources through currently existing query gateways.
R073	A03	The interface shall not expose resource providers to additional system security risks, nor risk compromising the resources accessed through the system
R074	A83	Allow data resources to be selected based on recentness of desired

		data.
R075	A83	EGSO should provide the means to execute joint correlative searches across multiple data resources.
R076	R078	The system should allow search criteria to be supplied in the format of metadata catalogues.
R077	A82	The system should be capable of identifying overlapping catalogue entries over user-selected dimensions.
R078	A05	The system should allow searches taking as inputs the data available in compiled lists of solar events or features.
R079	R076	The system should be able to accept user-supplied catalogues in a standard format as input criteria for a data query
R080	R079	Create user-defined catalogues from supplied parameters using EGSO standard metadata.
R081	R079	Create user-defined catalogues from supplied parameters in external format .
R082	A83	Classify identified datasets according to quality of overlap.
R083	A13	The system should allow ad hoc content-based queries on data
R084	A05	Allow results of user-defined reduction to be used in query evaluation.
R085	A83	EGSO should provide the capability to execute content-based queries across datasets contained in multiple data archive
R086	A83	Prioritize data according to chosen "quality" metrics
R087	A13	Allow queries with (partially) fuzzy boundaries.
R088	A13	Allow queries to require a minimum number of matches.
R089	A05	EGSO should be designed to handle from hundreds up to thousands of data requests per day
R090	A06	The system should allow results of previous queries to be recalled.
R091	R090	The system should identify software and metadata version for a recalled query.
R092	R090	The system should, when possible, indicate when the results of a recalled query may no longer be consistent with the results when query was originally executed.
R093	A06	The system should allow users to monitor or automatically be notified of progress of query.
R094	A06	The system should allow queries to be paused while user confirms query actions.
R095	A06	The system should allow users to save search criteria so that they can return and reuse them, with modification if required.
R096	A06	The system should allow for queries that will notify users whenever new data are made available to the system that match the query criteria.
R097	A06	The system should allow datasets to be calibrated prior to downloading.
R098	R097	The system shall allow access to calibrated data as provided by the data provider itself.

R099	R097	The system shall allow the data to be calibrated independently from the data provider using additional resource providers capable of calibrating given datasets.
R100	R097	The system should provide the resources to calibrate data, in so far as the calibration software is available for the given dataset.
R101	R013	The system should provide the mechanisms by which data can be processed remotely from the end user.
R102	R101	The system should allow execution of standard pipeline processing techniques to produce standard data products
R103	R013	The system should allow execution of user-defined workflows constructed from software components made available by the system.
R104	R101	The system shall allow the execution of user-provided code on EGSO administered resource providers.
R105	R101	The system shall allow the execution of user-provided code on selected third party nodes.
R107	R100	If no calibration or analysis software is available or applicable for a given dataset, the system should opt to return the raw data to the user.
R108	R097	If multiple options are available, the system should allow the user to select the level of calibration or reduction desired
R109	A06	The system should try to minimize the amount of data that needs to delivered to or managed by the end user.
R110	A06	Where possible, the system should allow multiple options for data delivery methods.
R111	R110	For a given data provider, the system should support the data delivery methods made available by that provider.
R112	R110	The system should attempt to provide HTTP-based online delivery of requested datasets.
R113	R110	The system should attempt to provide FTP-based delivery of requested datasets.
R114	R110	The system should support delivery of data sets via physical media.
R115	A06	EGSO should allow for all requested data to be delivered in a choice of several standard formats.
R116	R115	For a given data provider, the system should support the data delivery formats made available by that provider.
R117	R115	The system should provide the means to convert data furnished by multiple data providers in different file formats into a common data format, as chosen by the user.
R118	R115	The user should be given the choice of which data format they prefer and where possible the data should be provided in that format
R119	R115	The system should avoid rewriting large quantities of data solely for the purpose of changing the output format
R120	A06	The system should be designed to accommodate from tens up to

		hundreds of GBytes per day of data transfers from data provider to users
R121	A03	Selection of data source for a particular data set should be on the basis of quality of access (interface and network), provenance of the data, etc.
R122	R121	The selection of the source for a particular dataset should take into account the quality of the network connection between the provider and user (or other resource providers to which the data will be transferred as part of the intermediate processing).
R123	R003	If multiple data sources are available and the user indicates the desire, the system should provide the user with the possibility to manually select from which data source to transfer the data.
R124	A12	The system should allow users to choose from multiple interfaces depending on their area of interest.
R125	A10	The system should be designed to allow users to carry out activities in multiple areas of interest in a uniform and integrated manner.
R126	A12	The system should allow users to configure the interface to present data types of particular interest.
R127	A12	The system should allow users to define their preferences for various interactions with the system.
R128	R127	The system should allow user preferences to be stored and made persistent through multiple accesses to the system
R129	A03	The system should have the capability to estimate resources required to execute any given request
R130	R134	The system should provide the means to control resource utilisation through quotas.
R131	R130	The system should provide the means to manage quotas at a user or group level.
R132	R134	The system should allow requests to be held while awaiting additional authorisation
R133	R134	The system should provide the means to prioritize requests to reduce impact of resource-intensive queries on more interactive queries.
R134	A03	The system should be capable of tracking resource usage in order to identify and possibly correct bottlenecks in the system.
R135	R139, R073	The system should provide the means to allow users to gain access to resources through user authentication.
R136	R139, R073	The system should provide the means to allow users to gain access to resources through user authorisation
R137	R012	The system should provide the means that the users' usage of the system can be protected from individual disclosure.
R138	R073	Authentication and security should be implemented so as to be as unobtrusive as possible.

R139	R152	The system should be able to control access to data that are proprietary, based on access policies provided to the system by data providers.
R140, R151, R157	A09	The system shall include data for which there is no wholly unrestricted access.
R141	R139	The system shall provide a means for the data providers to define and publish the access restrictions and required access conditions for the different portions of their data holdings.
R142	R139	Access control should be done at a level that accommodates providers who have not made their access policies known to the system.
R143	R139	The system shall attempt to facilitate the authentication and authorisation requests necessary to receive access to restricted datasets.
R144	A02	The system should be based on the data, metadata, and other resources provided by multiple, independent resource providers.
R145	A09	The system should utilize data sources distributed around the world.
R146	A04	The system should allow near real-time access to certain data resources
R147	R006	There may be one or more copies of the data stored at multiple locations
R148	R147	There may exist no single complete copy of a given data set at any single site
R149	R147	The multiple copies of a dataset may represent differing revision levels of that dataset
R150	R147	For a given instrument and observing period it is possible to define a primary data source that can be considered authoritative
R152	R140, R151, R157	The system shall not exclude data providers who have restricted access conditions on all or part of their data.
R153	R002	The system should be able to include metadata catalogs containing well over one million records.
R154	A01	The system should provide the means for enhanced collaboration among the users
R155	R154	The system should allow users to share with other users the results of their searches through simple identifiers.
R156	A11	EGSO should provide for users to interact with locally available data and metadata through the same interfaces even in the absence of connectivity to other central resources of the system.

Appendix C. Solar data-grid domain model

Below entities associated with the solar physics Grid requirements are listed. It is intended to aid EGSO design, whilst being sufficiently generic for any solar virtual observatory (indeed, only few changes would be required for these objects to suite any virtual observatory). The object definitions align with the architecture specification of the system; detailed design may use some of these objects and relations, but is also anticipated to generate many more with much clearer specification.

A name and short description are given for each object. Comments and points for discussion are shown in italics. In addition, relations to other objects are given, classified into the 5 stereotypes given in the following table. Relationships between objects have been classified in the following stereotypes (note that these are not strict relationships that are enforced, as these objects are not intended to be implemented directly).

O	contains	Each instance contains the indicated objects. Other entities will get to information and facilities of the contained object via this object.
A	aggregate	A special case of the 'contains' relationship where the instance is a collection of the aggregated objects, and typically just has the role of servicing the aggregation.
K	knows	The instance of the knowing object is aware of the referred object (and may use it) - in contrasts with the 'contains' and 'aggregate' relations, other entities would not access the known object via this object.
R	creates	The object serves as a factory that generates instances of the indicated type. Unlike the 'contains' relation (which may also construct the contained objects), the objects generated are independent of their creator.
S	specialise	The object is derived from the general type indicated, containing all the information and functionality of the parent as well as additional characteristics.

C.1. Data resources

Many relations in this group are missing. The breakdown of where meta-data, catalogue fragments, derived data go - into archives or different specialisations (or generic) storage resource - and relations between them is a big issue and architecturally sensitive

ID	Name	Relate	Description, notes
1	internet	A:2	The public network for data distribution (connecting storage resources and supporting communication).
2	storage resource		An internet network node where data (including observations, catalogues, results, ancillary data etc.) is held. This data host represents the physical part and underlying software (e.g. the OS), not the logical store or aggregate of the scientifically interesting data.

			(Storage nodes not on the network - private areas used by scientists on site at institutions or observatories - are invisible to the internet and grid. Access to them must be via agents external to the network, and are represented in our model only by institution owning them.) (Archive and database could be specialisations of this, but they are different as logical repositories - not really a better specified type of physical repository.)
3	archive	A:5	An electronic store for observation data (a directory of files is the paradigm example of this type of store).
4	database	K:3	An organised store for any data that has meaning only in its context (i.e. relations to other data). Typically the data stored in the database will reference primary observation data (e.g. a catalogue, calibration data). (A relational database with non-trivial normalised tables is the paradigm for this).
5	observation		An instance of gathered data that records an aspect of the sun's (or its atmosphere's) state. This covers many types: images, light curves (e.g. for hard X-ray or radio), magnetograph, Doppler and seismograph observations, slit spectra, *in-situ* field and wind measurements etc.
6	observation meta-data	K:5,	A conglomeration of all data associated with an observation that adds basic information (for interpreting the data). 7 possible classes within the meta-data are identified (entities 7 to 13).
7	time		The time observation was made (may be the start and end time if cadence information is not also given, it may take different forms that can all be translated to UTC).
8	coordinate		The position in on sun of the observation (typically as a 2 dimensional area on the plane perpendicular to the line of site which would need to be translated to solar surface latitude and longitude or synoptic volume).
9	frequency		The wavelength the observation was made at (typically as a range, which may be associated with temperature). (This term could be used for energy of particles too, as very short wavelengths are typically expressed in energy anyway. This term would probably not be applied to helioseismology or other observations of flux or shock waves though, as these are derived information.)
10	cadence		The time accuracy of observation.
11	quality		Any indicator of observation quality (e.g. signal-noise ratio, background perturbation level, or subjective ranking).

12	resolution	The scale for feature distinction (this may be derived from the pixel count of the instrument with its field of view to be translated to an area size at solar surface, or wavelength resolution for spectrum).
13	intensity	The signal strength (possibly radiation emission for image, field strength for a magnetograph, movement rate for Doppler etc.), possibly integrated for an observation or an array of observations (e.g. for all image coordinates or light curve data points).
14	calibration data	Reference data used to interpret observation (e.g. luminosity, sensitivity spectrum).
15	summary data	Less detailed data derived from one or several observations (e.g. an average trend, low resolution image, representative instance image from a long period) or observation context (e.g. whole sun image or coordinates grid for an observation site to be mapped on).
16	preview data	Temporary low detail information e.g. the result of a search or an intermediate analysis result. (This is for interactive control of large download / analysis tasks and associated with specific observation data rather than summary data which is stable, may support a catalogue and should be generally applicable).
17	synoptic data	Information on the rotation of the sun (including rotation for each latitude or the evolution of a coordinate system, which allows correction for earth orbit (observation point movement) and determining proper motion at the solar surface).
18	feature catalogue	Information of when active sun phenomena occurred, typically associated with a classification index for type and scale (e.g. active region, large flare, CME).
19	derived data	Any product of the analysis routines applied to basic observation data (e.g. image cubes and movies, light curves correlated with images, composite and layered images of data from different instruments, inferred data plots over images (especially magnetic field lines, but possibly inferred density, temperature, velocity or cloud contours), post processed observations with instrumentation artefacts removed or corrected for rotation, differentiated images indicating differences and movement (identifying coronograph CME or surface and sub-surface flow), possibly mathematical model output, etc.).
20	result	A scientifically significant (as judged by scientist) piece of derived data (may be a summary of other derived data, e.g. several data points that support a relation predicted by

			hypothesised models).
21	generic catalogue	A:22	The worldwide collection of information about observations expressed in a common framework to guide the resolution of users' queries.
22	catalogue entry		A copy of an observation meta-data instance (either translated to a generic format or containing original information).
23	catalogue fragment	A:21	A chunk of the generic catalogue. By dividing the whole worldwide catalogue it can be distributed and replicated more easily.) The fragment is likely to include versioning information (or timestamp at origin) to ensure the must recent changes are available.
24	search history		A log recording data looked up (which may be used to identify desirable data, based on popularity).
25	cache		A temporary store of data (typically copied from remote nodes when they were targeted by users queries). It enables efficiency gains by avoiding repeated transfer of popular data).
26	ancillary data-set		Other data of different types to the main solar observations. This has wide variety, so this term may be decomposed in further analysis. Examples include instrument calibration data, reference observations, look-up tables to map observed measurements to scientific quantities.
27	automatic observation information	S:6	Exactly as the observation meta-data, but generated about an observation by automatic process (this may be simple – e.g. identifying where the centre of the sun is in an image based on the exact pointing of an instrument – or more complex, derived from image processing and feature recognition techniques (which may include classification) – e.g. an automatically identified flare from a rise on x-ray light curve).
28	observation reference		A globally unique observation identifier. This allows users to keep a definitive link back to the observation (or other raw-data) for quoting with derived scientific results.
29	published work		Any complex document that is not just observational or derived data, including: papers, web-sites, textbooks, presentations, graphics that use observations and scientific results. This entity has other bibliographic properties, for example the authoring scientists.

C.2. Computation resources

ID	Name	Relate	Description, notes

30	grid	A:31	A public network for computing activity – coordinating distributed data access and transformation tasks – overlying the Internet.
31	compute resource	K:32	A network node where analysis (or other computation tasks) can be performed. This entity represents the physical resources and fundamental support for analysis packages (toolkits, procedures, models etc.) including the operating system.
32	analysis toolkit	A:33	A programming toolkit for generic computational analysis of data to mathematically transform and graphically represent observations (e.g. optimised array storage and operations, image rotation and contrast enhancement, Fourier transform, statistical analysis including line fitting).
33	analysis procedure		A single programming routine (or very small set of functions) that are composed in a toolkit (e.g. an array class if the toolkit is object-oriented, or a library of geometry operations).
34	instrument software	S:33	Software that is required to access the data for a specific instrument (which may use generic analysis procedures, or overlap their function). It is assumed this is at the same programming resolution as other analysis procedures (routines, classes or function libraries), so these may also be composed in toolkits.

(The following fit less well into computation - models can be seen as a type of data, translation operations can be seen as part of the process.)

35	model	Any mathematical expression of a relation that predicts physical properties (matched against observations to generate a result that validates a hypothesis).
36	simulation	A complex computation system with interacting parts representing a physical system and programming infrastructure (e.g. process event list). This is likely to encode models, as defined above. (This could be decomposed if the purpose of the grid were distributed management of high-performance simulation execution).
37	translation function	A specialist routine that transforms data – typically used to translate non-standard representations into generic information format for interoperability (e.g. insertion into the global catalogue to permit user searches across data that was originally in diverse forms).
38	synonym	A relation between terms used in data models that allows mapping between them, permitting them to be evaluated as equivalent.

C.3. Process control

Most of these may be classed as into computation and data objects. However, they are here in their own group because they do are not represent anything scientific (nor anything exposed to grid users), they are just defined to keep the grid moving. This set could include interface objects, listed below.

ID	Name	Relate	Description, notes
39	request		A specification of the search criteria that identify a subset of records (typically instantiating values that can be matched in properties of observations). (This is likely to be specified as fields in an electronic form, but could conceivably be well-formed SQL or structured email).
40	access permission		Information about the users (individual scientist and groups) that are able access resources.(This could be an aggregate of accounts, but that implies an account stands for a trusted institution or a third party recommended by a trusted organisation as well as an individual. This is imagined as a permission table or guest list for the server to check, not a general pool of user information – which could be defined elsewhere.)
41	access policy		A description of trust relations between resources and institutions (i.e. this is likely to be at an abstraction level above individual user access to enable virtual organisations).
42	middleware		The infrastructural processes that enable generic network communication (including authorisation, distributed processing etc.).(This could be lumped with the compute resource description; a valid analysis node should include this as well as its own operating system).
43	account	O:44	A user's record with usage and quota information (held on the users' side; should not be the providers' responsibility to maintain this in distributed copies). (It could also include the users' authentication signature key.)
44	key		A pass-code that identifies the user. (It is imagined this is PKI public-key that allows signatures to be validated to prevent user imitation, prevents message tampering, and encodes sensitive data. It may be used for proxy-authorisation where access policies support single-sign on).
45	access log		The record of a user's (or institution's) use of resource (for quota allocation, billing, fraud investigation). (This is held on the server side, unlike account information).

46	audit trail	This entity is used as a more heavy weight record of usage trail across networked resources than either the account or access log alone.
47	protected space	Reserved storage capacity, only accessible to a specific user or group (possibly from remote institution).This may be used to store temporary results (workflow intermediaries) that will be used in further analysis, or derived data that is to be shared in collaborations, or just as reliable storage of users' work.
48	reservation	A statement of access rights of user (group or institution) to resources (space, compute, or data) that guarantee availability. This would be useful if activities on distributed resources must be coordinated. (Reservation may be time bound, and may follow an economic model even if real money is not used. It may also be associated with grid scheduling and security privilege revocation processes).
49	gateway	A process controlling access to a proxy for another resource (which may be used to checking policies before forwarding requests, or performing searches on passive provider storage resources).
50	proxy	A proxy server in public domain for private archive (enabling protection of the archives whilst making them visible on the network, or providing a catalogue that maps structured record references to unstructured data etc.).
51	version	Information to resolve the associated entity's currency (this may be attached to routines, toolkit, calibration data, catalogue fragment, policy, raw data etc.).
52	analysis task	A processing job which runs on compute resource to transform and analyse data using available procedures.
53	task queue	A collection of tasks (with implied priority for a given scheduling strategy etc.).

C.4. Interface

ID	Name	Relate	Description, notes
54	search client	R:39	The access point at which scientists submit requests to the grid of resources (may be GUI or CLI).
55	client configuration		A description of client parameters preferred by a scientist or group. This allows a client to flexibly fit different search paradigms, for example, presenting different parameters to fill in for images, spectra or events (these fields should always map to generic catalogue properties though).

56	client field	An element of a search client (for example, the box of a GUI that holds a value, or the argument on a command line).
57	access control	Tools on an administration interface that allow administrators to modify accounts and policies (and view access and audit logs). (These may not actually be dedicated tools e.g. any file editor can be used to view and search text logs).
58	resource administration	Tools for observatory and resource administrators to register and control their resources on the grid. (At its most basic this could be a script that runs a web server on a domain registered machine, but should also permit an archive to be registered as a grid object, describing its catalogue and the hooks for mapping data).

C.5. External entities

ID	Name	Relate	Description, notes
59	spacecraft	A:61	A collection of unmanned instruments that automatically generate data, especially satellites in earth orbit (that may observe the sun for long periods without interruption). As typically funded by international collaborations, the data collected by spacecraft is likely to be widely distributed.
60	observatory	A:61, 69, R:5	A site with manned instruments collecting data when conditions allow. Observations made may go into a public archive (especially if the observatory's instruments collect electronic images by CCD cameras, rather than photographic film), or may be stored at the observatory for access by request only.
61	instrument		A device that collects a certain type of observation. It determines some properties of the data collected (e.g. wavelength and resolution).
62	ground station	A:59, 69, R:5	A manned site that collects and manipulates automatically generated observation data before putting them into an archive.
63	region	A:64	An aggregation for several scientific groups (as well as observatories and satellite ground stations). This may be national or defined by an agency e.g. ESA. (Other aggregations exist, e.g. collaborations for a specific spacecraft mission outside agency boundaries. These may be region's sibling specialisation of a broader definition of institution groups, which represent real instances of the virtual organisation supported by the grid.)

64	institution	K:60, 62, 65, 68	A location for computer resources and users (typically scientists and administrators). An institution may be directly associated with an observatory (though more commonly it is loosely associated via a region-scale agreement, especially as defined by universities) and typically contains science groups.(Institutions could be said to have observatories and ground stations, or these could be two specialisations of institutions, but this seems unnatural and less flexible.)
65	group	A:66	A collection of scientists that share work in progress (especially derived data in collaborative investigations). These are commonly collocated at the same institution, but may be a disparate group associated by a specific project (again, following the virtual organisation model).
66	scientist		A human agent that analyses observations and generates results.(Not all relations on these external objects are defined yet; scientists are obviously attached to institutions and may have direct relation to administrators and instruments. In general there is great scope for associating external entities with data, computation etc. too.)
67	software author	S:66	Someone that authors and updates analysis procedures (that may either be manipulating data from a specific instrument or additions to the generic analysis toolkit).
68	administrator		A person who maintains data and computation resources (notably installing and making hardware available, installing and upgrading software, maintaining authorised users information and giving them access to the resources).
69	observatory staff		An operator or administrator at an observatory who knows activity and available records for observatory or satellite.(This may be split this into observatory versus satellite ground station staff, with roles caricatured as a librarian versus a technician. These roles could be specialisation of administer too. For the purpose of the system, this role is really a gateway between the network (with scientists) and an observatory with hidden data).
70	member of public		Anyone with internet access. (These people may have a general solar interest and be outreach targets - students, amateur astronomers - or be anonymous scientist looking at top level data without analysis).

Appendix D. EGSO scenarios

D.1. Consumer exposed scientific functionality

1.1. A query is resolved by joining data slices from multiple archives

The users selection criteria determine that more than 1 archive is needed to resolve the query. The correct result is resolved in a reasonably efficient manner and returned.

Where there are more than 2 archives, the system may minimise data transfer by estimating which is the minimising criteria and resolving that first. For example, where 1000 records must be crossed with 100 and 10 from other tables, the 100 by 10 join should be done first. This method holds whether the queries may be resolved at any of the archives, just one, or at computation resource remote from all archives. However, in all cases there should be improved performance if the maximal data is reduced closer on the network to the archives, further from the user.

A data slice is a straightforward operation for a database. However, for a file store it may require procedures to extract just the required data from large files. See 1.5.

1.2. A query is resolved by computationally determined data

Using similar search criteria to that used for observational data, the user queries information that must be resolved computationally. For example, a model may be used to predict the date when an active region on the far side of the sun would reappear. Such information may be used to guide further queries, for example if the query is for spacecraft observation times based on orbit. Such a query hides the details of computation toolkits from the user.

In a related way, the query may be resolved by calibration data lookup. This may be functionally derived or static ancillary observation data, for example representing reference spectral lines.

1.3. A query is resolved by metadata on the types of data available

A user request for the types of data available is possible, such that the search criteria are resolved at the data description level rather than the data itself. This is equivalent to querying the dictionary of a relational database.

1.4. A query for basic observation data is returned with relevant metadata

The data that matches a user's query is found, but supporting information associated with the data is also returned. Though the user may suppress this information, they should not be required to specify it in their query. This allows the user to learn information that is necessary for interpreting the data.

Examples include:

- Instrument characteristics - static properties. Such notes indicate the worst case instrument behaviour (for example, the centring margin of error), and

observation interpretation traps. In some complex cases, the note may need to indicate who to contact to get information.

- Observation notes and annotations (at best equivalent to observatory logs) - contingent local conditions for observation. (These do apply to satellite observations too - for example, SOHO LASCO images that include comets or radiation static).
- The data confidence level (for any selection dimension). This should allow results to indicate error bars or allow selection based on a quality threshold. Where similar matches are made at alternative archives (for different instruments), the confidence level should also allow the observations to be ranked on quality.
- Provenance information regarding the raw data. This may include a history of major archive updates (for example, when all data was corrected because of instrument recalibration), which catalogues were used to identify the data (indicating metadata quality) and what analysis may already have been done (for example, removing artefacts or storage via lossy compression algorithm). Such information should guide the user who is trying to reproduce earlier results (if they cannot correct for errors, they should at least understand why their result is different).

1.5. A query is resolved by a pointer to a data archive

A user query does not generate the actual data, but a pointer to where the data is. The user must then arrange their own data path to get the results (for example, contacting an archive and asking for a tape, or negotiating a bespoke protocol that is not recognised by the system). To achieve this, the system must resolve the query against its catalogues (or other metadata, such as an event list), which may held at a different location to the data. At its simplest, such a query may be for records within a time range, the system resolving when observations were made within the range.

1.6. A query uses basic data as metadata to resolve a query on related data

A user queries a data product directly at one point, then uses the same data as a data description in a subsequent query. The system allows such vague data treatment as both data resource and metadata. An example of such a metadata 'grey case' may be a GOES light curve - as well as being raw data itself, its peak values indicate the time of flares that may be observed by other instruments such as SOHO EIT (the flare locations in turn guide selection for active regions).

1.7. A query is resolved by analysis tests on the basic data elements

A user query can only be resolved by examining the raw data (rather than by its metadata description). The system therefore resolves the query by applying indicated tests to basic data elements. If the properties in the query are represented in a relational database, this

is straightforward. Alternatively, simple tests or specialist toolkit routines may need to be applied on a sequence of candidate data elements (such as files). When the low level data query is laborious, the user keeps visibility of progress by receiving an estimate of the time and effort scheduled by the system, updated as data is resolved - this supports user interaction with their query 'in flight'. The system must maintain the query state so that interrupted schedules can be resumed, and the user can recover their results if their own system connection is interrupted.

Examples include:

- A software agent task is scheduled to open files and examine their headers or other records within the files.
- A data analysis process processes image files, prepares and normalises them, then runs a specific scientific routine (for example, to identify prominences on the limb).
- A quick query is run when the user indicates speed is more important the detail (for example, 1 in 10 records are tested, or a subset of archives with possible matches are queried).
- A complete query is run when the user requires definitive results (in a converse example to the above, every possible archive is queried, the system queuing and rescheduling queries where archives are temporarily unavailable.)
- The user manually interacts with the system once a preliminary query returns quick look data to identify which of the records summarised are actually required (note, this may be functionally equivalent to the previous case).

1.8. A query is not resolved, but generates information to generate a successful query

In the same way that the user is provided with unrequested information to help their interpretation of returned data, in cases where their query fails, additional information about the failure is provided by the system. Alternatively, a user may include a minimum success rate in their query so that results are not returned if less than a specified number of data are found.

Examples of failure cases include:

- An archive is indicated to have records in the metadata, but it cannot be reached. This may include instructions of how the user can manually request data. See 1.5.
- When a join using more than 2 criteria fails, the criteria that describe the null set are indicated.
- When security restrictions are in place on a user and these cause the failure, the restriction and how to re-negotiate it are indicated. Such restrictions include exceeding a quota of system resource allowance, or lack of privilege for a private data set.

1.9. Queries are resolved in diverse parameter spaces

A user can phrase similar queries that identify records by diverse dimensions, including primary observation and operationally derived properties (including functional transformation including rate of change and integrals of primary parameters). The same observation is identified by diverse properties in different queries. Note that some observation dimensions are relevant for specific observation paradigms, such as spectral analysis and helioseismology. Example selection dimensions include:

- observation time interval
- cadence
- most recent observation
- location
- coverage
- resolution
- feature size
- event identification
- feature class (especially for flares and sunspot groups)
- spectral band range
- temperature (i.e. different spectral lines)
- anisotropic temperature profile (especially for solar wind)
- plasma density
- velocity (i.e. Doppler shift)
- oscillation mode (for helioseismology)
- energy (especially for radio, x-ray and radiation monitors)
- polarisation
- magnetic field polarity and strength
- other instrument specific criteria.

The user query can combine an arbitrarily rich set of selection criteria, representing the dimensions of the target observation and the parameters of the instrument. For example, the range of parameters currently supported by the RHESSI data interface should be feasible. However, the instrument details should be hidden from users that do not wish to specify its parameters. In the above set of query dimensions, therefore, user queries that select on derived properties are implied - these should guide resource discovery and specific instrument sensitivity parameters for unfamiliar users. See 1.4.

1.10. A query is resolved for fuzzy criteria

Some user selection criteria may describe a fuzzy set of matching data (i.e. the data properties are only partially held). This should allow results to be selected on a confidence level threshold. Examples include:

- plasma temperature range (identified by spectral bands which are insufficient alone to isolate all plasma of the temperature associated with the relevant ion, given abundance, velocity and pressure side-effects)
- the identification of s-shaped twisted coronal loops (a vague class of object)

- coronal hole extent (imperfect feature recognition by manual or automatic techniques)
- coronal mass ejections directed toward earth (a challenging property to determine from available instruments).

Rather than quantified selection criteria, a user may submit a semantically rich query, using natural language or a specific propositional calculus for a solar physics ontological model. If employed such a query should be able to be mapped to associated parameters that isolated the semantically described data set, matched to system information resources - possibly directing the user to another query type that can be resolved to specific data. For example, a query for coronal waves associated with CME uplift may direct the user to a query examining the speed of regular changes in extreme ultra-violet bands, correlated in time with a CME list.

1.11. A query is given without specifying where it might be resolved

The user interfaces virtualises the data stores by allowing a query by data type without addressing the origins of the data. A preliminary result, which may be used to guide further queries, is returned based only on physical properties in the observation (as listed in 1.9.) when the archive or instrument types are not specified. See 1.4. and 1.9.

1.12. The response to a query is given in a format chosen by the user

The result of a user's query can be presented in a way specified by a user, both for the data organisation structure and the delivery protocol. For example, a user may specify a certain FITS file format delivered via FTP, or an IDL object created by a Solarsoft API function. As a minimum, the user should be able to chose between the original archive standard format and interface and the EGSO generic common description format.

1.13. A user initiates a pipeline of data recovery and analysis tasks

A user query may be specified within an analysis pipeline, so that raw data identified in an archive is transformed via manipulative, mathematical or scientific operations before delivery. This should be possible by graphically linking computation elements or a scripting environment. Client functionality should allow workflow validation and control (including monitoring, synchronisation, interaction and completion notification). The computations should be scheduled by the system, controlling live process interaction or batch submission as appropriate, taking advantage of parallelisation where possible. Examples include:

- Applying routine manipulative or mathematical operations, such as extracting a record slice or determining the maximum rate of change.
- Calibrating the original data using dedicated routines and reference data provided for the instrument.
- Performing domain specific scientific operations, such as constructing a Doppler shift overlay - at the most complex, determining internal structure using helioseismology principles.
- Integrate observation analysis with model virtual sun information, as may be used for field line reconstruction (linking the virtual sun to magnetograms and

high wavelength images) and active centre emergence (additionally using helioseismology analysis products).

- Correlating data from different sources (normalising and then combining them), for example in a composite image.
- Applying operations from an automated image recognition toolkit, where extracting the limb then identifying prominence arcades may be separate operations, for example.

Note that the pipeline results (and intermediate products) may take of the order of days to derive. The system must therefore reliably accumulate and hold the results and protect access to them. In some cases, the data product should be accessible to a closed group of collaborators rather than just the query instigator.

1.14. An inexperienced user is guided through necessary pipeline analysis tasks

A user who is not able to specify their own pipeline through lack of system knowledge or of expertise about the scientific workflow required should still be able to integrate their queries with pipelines. Such usability may be enabled by the system (providing help resources equivalent to tutorials or a wizard) and by provision of a library of reusable pipelines. Common pipelines may be associated with a specific instrument, such as the selection and analysis steps required to construct a RHESSI image. The pipeline library should also contain basic workflow templates that specific procedures may be hung on for novel scientific analysis.

1.15. A user programs an interactive workflow

A user may interact with an executing workflow both with interrupt signals and at predefined stages where intermediate results are examined by the user. The user may guide subsequent flow by choosing the data product that should form the input of the subsequent flow, or making other decisions having evaluated preliminary results.

Advanced users should also be able to register new model templates with the library available to all users. Such new models may represent workflows required for a specific type of scientific investigation. See 1.13. and 1.14.

1.16. A user adjusts their system interface to maximise its suitability for their area of interest

An advanced user should be able to modify their interface for a specific type of scientific investigation. For example, investigation of radio signals or *in-situ* radiation measurements requires energy and intensity against time as primary selection criteria rather than the synoptic images of traditional observations. This implies that where a graphical user interface is used, the window organisation (input and display areas, menus and dialogues) should be determined by a modifiable configuration. The configuration should be flexible enough to support adaptation to new types of scientific investigation, which may rely on novel types of input and display areas.

Note that there may be a default configuration, or a small library of basic standard interfaces that are available to inexperienced users. This mechanism should operate in a similar way to the workflow templates. See 1.15.

The configuration of user interface should extend to diverse graphical and non-graphical interfaces. Therefore it should be possible for an interface to guide query construction (generating the same query) for a web form using an application server, an IDL client using local APIs, a batch interface used by a daemon process, and a portal that wraps the system interface in its own arbitrary interface. Note that these latter cases imply the same behaviour should be accessible to affiliated data-grids, providing a way that external users meeting EGSO requirements can specify searches and analysis pipelines to use system data and infrastructure resources.

1.17. A user receives recommended resources based on their area of interest

Configuration profiles may be used to group users into types of scientific investigator. This allows the system to recognise what the similar research interests of users classified in the same group are. Therefore, system may recommend (on request) related areas of interest to the user based on the types of queries they have configured. This may guide the user to relevant resources - either instruments and archives, or specific data resources such as popular observed events.

Such interest grouping may be further aided if it is possible to examine users' query histories. Storing and mining successful system tasks is more complex than just classifying users on their configuration. Dissemination of interesting resources would also be helped by broadcasts to subscription lists. For example, this may allow identification of interesting observations to be disseminated to those interested in the type of captured event rapidly, maximising data use.

1.18. A user of a data resource is notified of its update

A data archive may be wholly updated occasionally (perhaps once every few years), possibly due to error correction, recalibration, reorganisation or analysis procedure upgrade. A user who previously accessed and analysed such a data resource may benefit from this update. Using access logs, the system should automatically notify users of the update (possibly using the same mechanism as the interest list broadcasts). This applies to the case where the user has accessed a resource transparently (in which case their previous query may no longer be able to reach the data. See 1.11. and 1.17.

1.19. A user is identified by the system to gain greater access

A user may access the system anonymously or using an authenticable unique identity. The user identity supports accountability and supports advanced access possibilities, for example:

- using more computational resources for queries and pipelines requiring intensive analysis

- accessing data and data products (possibly within an analysis pipeline) restricted to a closed group (restricted to listed users or institutions)
- precedence in query scheduling over anonymous users
- registration for resource usage (the user data access history), allowing interest grouping and update notification
- the users own information (for example, on interface configuration, custom analysis pipelines, and derived data) may be protected (for example, by encryption, digital signature and watermarking).

Note that once a user has used their secure identity to protect their owned system information, they may control how that information is used. For example, they may set up a closed access group for that data, remove bad associations the system has determined, and support their intellectual property rights to information they provide to the public domain. See 1.13. and 1.18.

1.20. Data is mined for hidden relations

The user should be able to specify a data-mining type of query, where the related properties within the data are to be discovered and therefore cannot be represented by the user initially. To find such hidden relations, it may be necessary to specify agent tasks within an analysis pipeline. A processing agent should be able to access large parts of data archives in a managed way (possibly sequentially accessing records), and generate data markers and relation evaluation state (such as weighted coefficients or neural network topology) as intermediate data product. See 1.13.

1.21. Different observations of the same target allow instrument accuracy evaluation

A user may deliberately select for observations as similar to each other as possible, then normalise and analyse them for differences. For example, visible light telescopes may be compared for the exact position of simultaneously observed sunspots. Such comparative analysis should use automated pipelines. By identifying common differences shared by many sets of observations the user will be able to identify inaccuracies in each instrument. By comparing at least 3 observations for comparison the instrument with the error can be readily identified, but other techniques to compare confidence may be used - for example if CCD and film observations are compared. If such are operations used by an instrument team, they may annotate their data or analysis procedures published results to improve their value. See 1.4. and 1.13.

D.2. Provider and administrator operations

2.1. A data archive is registered with the system

An administrator of a data-resource should be able to add an archive easily. This would typically represent enabling access to a new provider. They are likely to have a novel schema that can be mapped to EGSO catalogue standards using a Archive added. They may be expected to use one of a small number of access methods (e.g. HTTP or FTP), but it should be

easy for the provider to specify their connection protocol requirements. The information of a new data resource should become visible to the whole system, so that users may find it with their existing client configuration.

2.2. Metadata for an archive is specified, enabling mapping to standard data descriptors

A provider should be able to advertise the semantic content of their archive (just as they do for the low-level schema and access method in 2.1). This is likely to be in the form of keywords applicable to the archive (both its data and metadata, so this represent metadata about metadata). The mark-up should allow EGSO to automatically use a wrapper to translate the archive content to a common standard; for example, mapping planar coordinates to longitude and latitude or custom date-time formats to the ANSI C library standard. Such automatic filter generation would only apply to simple providers; data resources with rich complexity may require more effort, such as the administrator writing and advertising their own filter as a data analysis service.

A variation of this use case is where an observatory registers their data but does not provide a service host for users accessing their service. In this case the EGSO grid should hide the provider's location and just advertise a general description of the type of data available. The EGSO infrastructure should also be able to uphold guarantees about bandwidth and registration restrictions that the provider has requested.

2.3. Metadata is embellished

Just as scientific metadata is advertise in scenario 2.2. it should be possible for a administrators to modify providers' existing metadata. This may be necessary in a number of cases:

- When the data classification resolution is refined the EGSO ontology is extended. For example, feature lists may be reconciled (when it is accepted they have a shared underlying physical process) or split (when advances in the analysis or modelling of phenomena enables similar observed effects to be distinguished).
- When a new catalogue format is advertised, or an existing format is modified. In such cases, the self-describing nature of metadata should enable existing search engines to automatically make use of the new catalogue schema.
- When the observation teams or scientists (perhaps solar analysis package programmers) write applications that generate new information tags on observation data. Such novel metadata may be generated in batch processing or on user request, and should be immediately available to other users, alongside existing EGSO catalogue metadata, to help their semantically defined queries.

2.4. A query paradigm is introduced by the provider and becomes available to users

If a provider advertises resources that go beyond those already existing in the system, and therefore outside current EGSO ontology (as covered by scenario 2.2.), they should also be able to advertise a new query paradigm. For example, if typical data access were by parameterised relational queries (such as SQL joins on numeric values), and a provider had a resource that could only be queried by keyword (such as an archive of observation log comments), the EGSO infrastructure should be able to route the appropriate Google style phrase or regular expression to the provider. This scenario may apply when new scientific methods are taken up, and is related to the modifiable user interface scenario 1.16.

2.5. A portal is provided to an equivalent data resource organisation

EGSO should be able to interface to other data-grids. A portal service should be exposed along with other provider resources (either as a data or analysis service provider, depending on the available functionality).

An equivalent data resource, outside the scope of EGSO's core services in exposing on-line electronically recorded observations, would be very long term historic information; examples could be ancient Chinese astronomers records of sunspots, or mineral radiology data that indicates changes in solar radiation.

The anticipated difference in a service interface to such a resources makes this scenario a special case of 2.4.

2.6. Metadata is corrected to allow more accurate query resolution

As well as allowing for scenario 2.3. - modifying the structure of metadata - administrators should be able to modify the metadata content. Such modifications should be cumulative, so that even if information about observations is more accurate than the provider's own metadata, the original annotation is not lost. Such modifications should be recorded with the date they were made and their provenance, so that user queries may be guided by constraints they specify on the metadata to be trusted.

A clear example of metadata correction has been encountered in prototype automated feature recognition, where the analysis generates better information for the observed centre, oblateness and orientation of the sun's disc than given in observation logs.

2.7. A provider advertises a data cache area and it is used by the system to optimise query resolution

A cache may be deployed to improve performance by staging popular data closer to users or on parts of the network with broader bandwidth. The EGSO infrastructure operations should automatically make good use of this resource, populating it and recording its content as a data resource in the system catalogue. Users should then be able to discover and transfer data from the cache. When cache content is recorded, the time it was populated and the definitive source of the data should be recorded, so that updates to data replicas may be managed.

If a service node within the EGSO infrastructure is itself advertised as a data node (for example, holding unified catalogue metadata) and has unused storage capacity, the system

should be able to make use of this resource as a cache in the same way and a provider's dedicated resource.

2.8. Analysis tasks are scheduled with fair allocation and respect to required workflow

Users may request tasks of the EGSO system that require workflow management when they run across several resources in parallel or a pipelined sequence of operations, as when they wish to calibrate observed data against ancillary test data or execute their own analysis code. In such cases the system should allocate processing resources in a fair way, so that progress is guaranteed on all tasks submitted. Where user tasks only require basic resources, they should not use high performance resources (that are in high demand).

2.9. Credit for data resource usage is recorded

When users access resources, their activity should be recorded to allow billing or fair access. A fair credit paradigm would prevent a minority of users hogging services even if no accounting of real money were used. The archived usage information would allow the system to prepare feedback to be prepared for users.

The usage records should also allow providers to track the usage of their resources. This is important where institutions' funding depends on whether they can prove their value to the community. It may be necessary to anonymise usage records to protect user privacy when providers access accounting information.

Accounting records may make it easier for scientists to generate their acknowledgements for published work. The metadata that users can query would include the identity of data providers and those that provided the analysis functionality or generated the catalogue information, if appropriate, and indicate which other users are accessing the resources.

2.10. A data archive with limited resources is encapsulated, allowing full controlled access

Where a provider only has limited resources to host query services against their data, EGSO should wrap the interface and protect it from excessive volume. This is critical where the provider uses their custom analysis software to query observation data. Applications written in C and Forth exist for analysing the data of ground-based observatories. In such cases, providers should trust EGSO to manage proxies for its users on the limited local resources that host such interfaces.

2.11. A closed group shares a data product

A data resource may be accessed by a just those people named on the provider's policy. This is especially useful where some special scientific effort has gone into producing the data being provided. This may be the case, for example, where the data is the product of a complicated query that supports an unpublished model relating observation criteria, where specialist scientific metadata is provided in a novel event list laboriously compiled by a junior researcher, or where the derived data is output by an experimental analysis routine. Equivalent

protection would also be required for small data-sets entered by users that which to gain maximum benefit for their effort before opening the resource to the community.

Users should be able to modify and annotate such guarded data, so that the resource can be the focus of user feedback on analysis, or be evolving with the lifecycle of the associated process. It should be noted that the EGSO roles allow a data product (generated by queries or analysis) to easily become a data resource (advertised via the normal simple provider registration mechanism) in such examples. This is in contrast to other projects which use a special case service for collaborative examination of data products (such as the AstroGrid MySpace entity).

2.12. Analysis code generated by a user is hosted by resource provider

EGSO is required to host user code, following the current paradigm of distributing the SolarSoft toolkit. Resource providers may advertise their nodes as capable of hosting code, specifying what software is provided (with other metadata such as the version number and host architecture). User requests should also be validated against the provider's policy (allowing exclusive collaboration between institutions, for example). Though EGSO guarantees user identity, it must be up to the provider to specify a policy and organise its local deployment to manage safe user code execution, for example, with sandboxes.

If the code is used to generate a data product, it may be that the user's code can also be registered with metadata about its output's semantics. For example, user code may reuse feature recognition functionality to identify coronal waves or opening field lines in observations. In this way, EGSO may make best use of available data in matching future user queries to its catalogue. This scenario therefore overlaps with 2.11. regarding data product advertisement and 2.2. regarding catalogue metadata extension.

2.13. A provider advertises an analysis service and becomes available to users

When a provider registers its services (rather than data, described in scenario 2.1.), EGSO should manage the advertisement so that users at any point may make use of the new resource. Examples of non-data service resources include:

- Calibration analysis routines to normalise data recovered from a store before it is transmitted (possibly being forwarded to future workflow operations or complex query actions).
- Computation in a commodity pool (such as a Condor managed PC cluster with spare-cycles), suitable for low-priority. This example is a special case of scenario 2.12 with its own safety issues.
- Ancillary data may be hosted by a data provider, for example, containing test instrument data that allows the observation data to be calibrated. In this case, the provider may also host services to calibrate the data, as in the first example. In this case, EGSO should be able to guide a user query that identifies the data to the associated ancillary data.
- Specialist processing resources may be encapsulated and advertised in the EGSO framework, possibly seeming like a portal as in scenario 2.5. Examples

of specialist applications include helioseismology (reconstructing the sub-surface density and movement from photosphere oscillation) and image reconstruction from gamma photon detection (as provided by the R-HESSI team).

5 classes of computation hosting have been identified. Using the appropriate policy may minimise latency by reducing network resource usage (reducing data before its transmission). The 5 classes are:

- no processing,
- processing is always done at the source (e.g. for necessary calibration),
- processing is optionally done at the source (e.g. R-HESSI image construction),
- the processing is done in the query pipeline, but cannot be done at the source,
- processing is done at a specific host affiliated to the data source.

2.14. Access is controlled by institutions

Providers must advertise a policy for EGSO to test validated user identities against when they register if they wish to limit access to their resources. Policy content is most likely to be at the resolution of institutions (rather than countries or named individuals, for examples). Different policies may be applied to different types of user operation, for example:

- accessing quick-look data,
- transferring a large amount of data (over a parameter defines size)
- running a computation task,
- updating (writing) metadata (either concerning low-level infrastructure information or semantic scientific catalogue information),
- accessing data that is less than a certain age (allowing only named institutions to access data gathered in the last 12 months, for example),
- accessing data on a closed group list (see the shared derived data scenario 2.11.),
- indicating that an executed query should be hidden or advertised to other users,
- hiding a data resource's content, perhaps whilst advertising its existence (permitting EGSO users to make private arrangements to use an archive, for example),
- upholding a related organisations policy (for example, defined at a national level),
- providing a timeout on temporary session certificates,
- showing or hiding intermediate results abd derived data (for example: workflow checkpoints, partial results of a joining query),
- showing or hiding users' history of generated data products (along with search criteria and interface preferences), which also concerns sharing data as described in scenario 2.11.

- viewing and modifying users' provisioned resource usage quota (account credit), which may also be aggregated to a group or institution, or divided into user roles.

When EGSO revokes a user identity, that user should no longer be able to use the resources that they had previously had access to. This means that authentication policy changes should cascade around the network, and that provider validation must be done by the infrastructure for each request without compromising the requirement for usable security.

D.3. Hidden middleware (or middle-tier) operations

3.1. Data result is minimised using analysis computation resource

When it is necessary to process data identified by a consumer query, the EGSO infrastructure should transparently make use of computational resources close to the data archive. Ideally, computation resources co-hosted with the data should minimise the data before it is transmitted at all. Though this scenario is closely related to 2.13. (with its 5 classes of computation strategies) it emphasises the infrastructure's automated computation management behaviour, that need not be presented to users, rather than advertised computation capability. In this scenario the computation may still be hosted by a third party, rather than using just those resources within the deployed EGSO network that are directly managed.

3.2. Query resolution time is minimised using metadata replacement for popular functional transformation

In cases where a filter is used to modify or analyse data required popular queries, there may be the opportunity to optimise response times by caching the data product. EGSO administrators would need to specify the configuration to indicate how valuable traffic volume and processor load is compared to cache storage space. Then, when popular derived data was identified from users' query histories, the system should start populating look-up tables.

A simple example of such cached analysis would be the good observation times for a satellite, derived from its orbital parameters. A more complex example, that is already done to automatically generate classified flare indexes, would be the differentiation of GOES x-ray light curves to find sharp rises.

This may be implemented as an extension of the functionality that automatically generates catalogues, which are also abstract data related to raw archive content, or the functionality that caches popular data. The generated look-up tables may represent a semantically significant metadata set themselves, and represent a refinement, reorganisation or extension to the ontology used to guide intelligent query resolution. In this case, the scenario is related to 2.6.

3.3. Query resolved indirectly using data held at staging resource

Where a user query cannot be directly satisfied at one data resource, it may be necessary for the infrastructure to manage sub-queries to first collect the required data together at an staging point independent of the archives. The query is then resolved against the

aggregated data slices rather than directly via a data provider interface. The staging area may be a cache provider, ideally using resources close to the largest data store to minimise network resource use.

As an example of this scenario, a join between 2 archives may be done by testing the joined data at the staging resource. The join criteria may be on matching observation times, then the user query may be resolved by testing combined emission intensity values. In this case, further minimisation would be possible with good data management scheduling by only moving data that is both within the time range and that may be above the test value. Note EGSO should manage distributed query resolution and the data routing that requires for an arbitrary number of step, not just 1 staging point between 2 archives.

In a variation of the scenario, a data staging resource may be use to federate an EGSO provider data set with one outside the EGSO infrastructure. This becomes an alternative view of scenario 2.5. where the portal functionality exposes a data-set used for very specific queries.

3.4. Pipeline tasks managed with interactive schedule estimate & quota allocation (with synchronisation etc.)

When a user request must be satisfied by a sequence of query and analysis operations, in a workflow, EGSO should allow the user to interact with the tasks it manages. It should be able to report to the user which provider resources have been allocated to satisfy which tasks; if possible, in advance with estimated completion time. A user should then be able to modify, or at least cancel, the tasks; ideally, confirming the system's workflow breakdown and allocated resources once an anticipated cost against the user's credit quota of the search is calculated.

It may also be feasible to indicate the status of user authorisation, as proxies generated by the query are validated against provider policies. If an increased credit quota or entry to a list of permitted users must be negotiated, the query and workflow may be temporarily suspended and saved. A user may also have the option of using different roles (or otherwise making the best use of policy restrictions) to improve the prioritisation or avoid provider refusal in the workflow determined by EGSO.

EGSO must also record the resource activity done on behalf of users and their queries so that consumer and provider institutions can manage their quotas. By grouping use into roles, it may also be apparent to network managers which resources are most in demand. Such information would help to guide future maintenance, development and deployment.

3.5. Data result matched minimally using data slice method

Where a provider's records are large and complex, containing more information than was requested, it should resolve the query by just returning the appropriate part of the record. Though easy for database records, as much solar data is stored in structured files, local routines dedicated to the data schema must be used. In a typical example, many instances of observations are stored in one FITS file under a shared header (for a time interval). The data slice will return just the needed 'frames' in the file. These may not be simply directly specified by time interval; an index may be used to locate a catalogued event of interest.

3.6. Query resolved at mirror on primary archive route failure

Users' queries should be resolved to logical classes of data resources, not physical instances. When a satisfying instance is located and found to be unavailable (either because it returns an error recognised by the infrastructure, or because the query forwarded on behalf of the user is not acknowledged or resolved at all - perhaps after a timeout) and other instances exists, the query should automatically be retried by forwarding it to another resource (and on again until all fail, in the worst case). The alternative data resource instances may be mirrors of each other, organising the synchronisation of all their records (this is common in solar physics for valuable, widely used observations received from satellites such as SOHO).

There is scope for using 'intelligent' procedures to determine which order the resource instances are tried in. The user may have known preferences that should be respected (for example, a national archive may be preferred by researchers in that country, as their use encourages its continued funding). Observed network traffic may lead the lightest loaded resource or the one with the least predicted latency between itself and the user being chosen (this involves a degree of network 'weather' observation, modelling and forecasting); regularly failing resources (or those resources dependent on unreliable network links) may simply be blacklisted, so no queries are sent to them; very popular overloaded resources may be identified so that more replicas are made of them. Alternatively, queries may simply be dispatched on a round robin rota of chosen options.

3.7. Metadata update cascades from updated archive regularly

Archives typically have regular updates to their content, and therefore must regularly update their metadata. An archive providing data for a quiet observatory may only need to change their catalogue every few days, but where data timeliness is especially important - notably in space weather - updates may be regular and frequent (every few minutes).

These metadata updates must be advertised, and therefore shared across the data-grid. Diverse communication models can implement this: a provider may broadcast the update to all other nodes (such an unscalable solution, this is undesirable), or they may post the update to a shared blackboard or agent that broadcasts the new metadata to registered subscribers for changes (which may be a brittle solution as it depends on some degree of centralisation), or all those nodes interested in catalogue updates may periodically pull them directly from the provider (at the risk of using stale metadata until they do), or the updates may be passed along chains of connected nodes (which therefore have a responsibility to propagate received updates, and recognise previously seen or otherwise out of date metadata updates; this is a strong but potentially slow solution).

3.8. Query diverted from out of date mirrors when primary archive indicates major update

Though the scenario of 3.6 - routing some queries to data mirrors - is normally desirable, there are occasional circumstances when an entire archive gets updated at once (for example, when experimental determination of better instrument calibration data forces an entire

data set to be recalibrated). In these circumstances, the data replicated at mirrors is suddenly all wrong, and there should be a mechanism to stop users' queries being sent to them. (This may be indicated by flagging the mirrors as out of date and due an update in the resource catalogue, or by tagging the correct source of the update in such a way that it is always chosen in preference to its mirrors. Replica management design is a topic that extends beyond this use case; the prioritisation and scheduling of the mirrors updates and the removal of the special avoidance tagging are not considered).

In a variation of this scenario (that would require extra audit functionality of the infrastructure), when such an update occurs all users that had previously accessed the out of date archives could be notified. This 'product recall' could also be issued when an analysis procedure was changed to such an extent that the previous results generated by it (which scientists may be relying on to demonstrate or falsify their hypotheses) have become erroneous.

3.9. Query resolution accomplished decentrally using distributed metadata (/data) matches

As the data-grid scales up, it will become unfeasible for the whole catalogue of available data and analysis resources to be reproduced everywhere that it is needed. There must therefore be a way for the infrastructure to divide the catalogue into chunks that can be distributed to different locations. This is made more complex as the catalogue itself is also changing as resources are extended and their availability changed. Just finding a way that user queries can be resolved in such an uncertain distributed environment is a challenging scenario in itself. It may be necessary for queries to be resolved in parts, so that the strategy for resolving them is composed at different nodes on the infrastructure that have the appropriate catalogue chunks. Metadata that the infrastructure maintains about queries managed in this environment must be kept up to date with the distributed routes the query has taken too.

It is not clear whether this technique arrives a definitive result; the same query posed from a different starting point may be resolved in a different way. This non-deterministic behaviour should be expected of large data-grids that have decentralised management. In this distributed architecture, there should be scope for performance levelling, though, opportunistically taking advantage of known resources.

3.10. User analysis code is isolated in sandbox and cannot invoke illegal operations

In a data-grid, code from unknown users may be run on resources that are only designed as general service points (and therefore do not know what capabilities they provide to specific analysis programs). Security may be upheld by isolating users' tasks. If they are submitted as non-interactive batch jobs, there is scope for static analysis and planning safe execution times (when user tasks with not conflict with critical administration actions). Interpreted languages (scripts and Java) may use their virtual machine as a sandbox. Either way, safeguards can prevent the user's programming invoking illegal operations or making other users' activities unsafe.

3.11. Query parameter validation (type checking) achieved close to client to minimise waste searches

Borrowing a design from website form design, wasted user queries can be avoided if errors in requests are identified at (or close to) the client interface (rather than only failing when queries are presented to the remote provider). GUI forms that self-describe their fields can help clients to check parameter types (and possibly even ranges validity) before the query is submitted to either the infrastructure or the providers' search engines. Queries may be stopped at the infrastructural tier (before the provider is reached) too for semantically higher-level query failures (for example, requesting time intervals when the observatory did not have a view of the sun). The overhead of the required 2-tier resolution is still efficient if it saves use of providers' limited connection and processing resources for well-formed queries from other users.

3.12. Approximately 1000 queries and 100 GB of data are handled per day

As a ball-park figure for EGSO data-grid loading, it is assumed that most scientists make few requests each day, so load evens out to less than 1 query per minute. However, given the nature of the data, even at this rate there is significant data load. Though not accounting for 100 MB per result, additional network maintenance load and, more significantly, provider updates to catalogue content must be factored in.

Appendix E. Requirements architecture matrix

In this appendix the 83 detailed requirements of data-grids within the 18 requirements areas discussed in Section 4.1 are given (with index numbers and priority). For each requirement, the architectural styles that may influence whether or not the requirement can be met are given. Up to 5 styles are scored for suitability with reasons as described in Section 4.1. The 83 by 5 matrix has been flattened into 83 records in the table below; the meaning of the values is given in this short template for the records:

Req. number	A data-grid must, should or could provide the behaviour described here or have this given property.		Priority
Style 1	Score	Reason style 1 supports this requirement.	
Style 2	-Score	Reason style 2 undermines this requirement's satisfaction.	

1.1	A data-grid must be able to include distributed, heterogeneous data resources, to form one logical resources that crosses organisational and administrative boundaries.		1
Layer:	2	Layers help abstraction, narrow protocols hide heterogeneity.	
Tier:	2	Tiers help transparency including format heterogeneity and present single point of entry.	
Peer:	1	Peer networks are often flexible enough for host diverse types of data.	
Agent:	-1	Blackboard must be central for agents to write to (read from), compromising distribution.	

2.1	The users of a data-grid must be able to discover and gain location transparent access to resources.		1
Tier:	2	Tiers provide transparency, of which location transparency is about the simplest.	
Peer:	2	Peer networks enable discovery and may location anonymisation (beyond transparency).	
Agent:	1	Agent based resource discovery should help subsequent look-ups (e.g. used by Google).	

3.1	A data-grid must be able to include data of various format and structure.		1
Layer:	1	Layers are intended to abstract diverse formats (for data and signalling).	
Tier:	1	Tier systems may provide framework for type mapping (lining up with OSI presentation layer) as in CORBA object marshalling.	

3.2	A data-grid should be able to include raw, processed and annotation data.		2

3.3	A data-grid should be able to include multiple copies of individual data files and data sets.	2
3.4	A data-grid should allow data to be assigned both logical and physical identifiers.	2
3.5	A data-grid should enable users to create bespoke logical views on data.	2
3.6	A data-grid could include data stored on tape as well as data stored on disc.	3
3.7	A data-grid could include multiple copies of a data file/set in different formats, at a single location	3
4.1	A data-grid must support domain-specific metadata standards.	1
4.2	A data-grid should support a metadata framework that includes multiple metadata schema.	2

Layer:	1	Layers may allow meta-data abstraction (e.g. diverse lower level to homogenous higher level presentation).
Tier:	1	Tiers could help metadata transparency (though typically rely on their own metadata to provide other transparencies).
Agent:	1	Agents may be able to crawl across diverse standards of metadata (with individual processes tuned to extract different metadata) as for unreliable html meta-tags

4.3	A data-grid should support a metadata framework that enables translation between metadata schema.	2

Layer:	1	If layers can hide heterogeneous metadata, their connecting protocols may form a standard for translation.
Pipe:	1	Filters may be suitable for metadata transformation.
Agent:	1	Agents reviewing diverse metadata content can write to a common format in the shared data-structures of the blackboard.

4.4	A data-grid should support a metadata framework that is flexible.	2

Layer:	1	If layers provide abstraction, they will support lower layer heterogeneity and therefore flexibility.
Tier:	1	Service networks have demonstrated flexible meta-data (e.g. WSDL).
Peer:	1	Peer infrastructures separate discovery from content, and should therefore allow discovery against arbitrary metadata standards.

Agent:	1	As agents can cope with arbitrary input, they should handle flexible metadata.

4.5	A data-grid should support a metadata framework that is extensible.	2
Layer:	1	If layers provide abstraction, they should allow for changes at the lower levels.
Peer:	1	Peer infrastructures separation of discovery from content should also allow metadata extension.
Agent:	1	As agents can cope with arbitrary input, they should handle changes to metadata.

4.6	A data-grid could enable automatic extraction and generation of metadata.	3
Peer:	1	Peer networks generate metadata about the network in a decentred way by the way each node is used.
Agent:	2	Agent technology has been most successfully applied for automatic data analysis.

5.1	A data-grid should support queries based on attributes of data (pull).	2
Tier:	1	The traditional solution to attribute based queries is client-server, which evolved into tiers for distributed systems.

5.2	A data-grid should support queries based on pattern matching (push).	2
Agent:	1	Agents may generate directory based look-ups / data mining result production, and so push matched patterns as described.

5.3	A data-grid should enable users to construct complex queries, based on atomic query components.	2

5.4	A data-grid could support queries that span multiple data resources.	3
Tier:	2	A middle tier is required to fork then join queries for a single client request.
Pipe:	1	Pipelines divide work amongst resources and allow synchronisation points for return.

5.5	A data-grid could support access to data at a level of granularity below that of an individual file.	3
Layer:	1	Layers are a suitable way to mark top-level data elements independent of low level storage structure.
Agent:	1	Agents have been used to examine data within files.

5.6	A data-grid could support frequent and rapid data access.	3

Tier:	2	Tiered systems allow parallel session management and may be used to prevent blocking.
Pipe:	1	If rapid access means that high throughput query pipeline must keep going, pipe and filter decomposition may be preferred to black-box query resolution requires.

6.1	A data-grid must include data processing resources.	1

6.2	A data-grid should include a variety of data processing resources.	2
Layer:	1	Layered interaction with resources hide variety.
Tier:	1	Tiers should help maximise use of variety.
Pipe:	2	Pipeline scheduling can make best use of a variety of computing resources in non-trivial parallel decomposition.

6.3	A data-grid could support remote code execution.	3
Tier:	1	Tiers decouple client interaction from server-side activity, and this applies to remote code execution management too.
Peer:	1	Peer networks do use mobile code, though typically tailored for a specific task.
Agent:	1	Agent technology may spread tasks across different platforms as mobile agents

6.4	A data-grid could include data processing resources that are not local to data or users.	3
Layer:	1	Virtual network communication provided by layers may help local control of remote resources, hiding intermediary control mechanisms.
Tier:	1	Tiers may also support marshalling between local and non-local tasks.
Peer:	2	Peer networks are successfully used for highly distributed computing tasks.
Agent:	1	Mobile agents may work in unrelated locations.

6.5	A data-grid could support data processing across multiple data resources.	3
Layer:	1	Virtual communication paths may also help coordination across data resources.
Tier:	1	A middle tier may support forked execution of tasks spanning data resources.
Pipe:	1	If multiple data resources can be connected in a workflow, pipeline management may help.
Agent:	1	Tasks required for multiple data sets may be divided into agent responsibilities.

6.6	A data-grid could support parallel data processing.	3

Layer:	1	Virtual communication paths should also help coordination across compute resources.
Tier:	1	A middle tier may help to coordinate parallel tasks.
Peer:	2	The peer network topology makes parallel progress a primary operation.
Pipe:	2	Pipeline processing is well established for parallel task execution.
Agent:	1	Agents may work on divided tasks in parallel.

6.7	A data-grid could support lengthy batch processing.	3
Tier:	1	A middle tier may supervise the state of tasks while a user goes offline, queuing requests for batch execution.
Peer:	1	Some tasks on peer networks are massive batch tasks (e.g. SETI at home).
Pipe:	2	Tasks decomposed into a pipeline schedule may successfully be run for very long, discontinuous computation times.
Agent:	1	Agents typically work autonomously.

6.8	A data-grid could include data storage resources that are local to remote processing resources.	3
Tier:	1	Middleware task coordination could include same-site remote resource use.
Peer:	1	Some peer networks download data with tasks for analysis.

7.1	A data-grid should be able to support the transfer of entire datasets.	2
Layer:	1	The OSI layers help reliable delivery by assigning responsibility for error checking, ordering etc.
Pipe:	1	Parallel pipeline may be used for high-volume data flow.

7.2	A data-grid could support continuous network traffic from data sources to data resource nodes.	3
Layer:	2	Layers are essential to uphold quality of service across a network.

8.1	A data-grid should enable access by users in variety of roles.	2
Layer:	1	Layered abstraction will help keep alternative roles hidden at application from lower implementation.
Tier:	1	Tiers allow abstraction of client types.
Peer:	1	Pier networks allow different node roles

8.2	A data-grid should enable data selection and querying.	2

8.3	A data-grid should enable local visualisation of data.	2

8.4	A data-grid should enable browsing of analysis services.	2

Tier:	2	Service based middleware supports capability browsing.
Peer:	1	Peer networks should support service discovery, though this cannot be as reliable as a centralised directory based model.

8.5	A data-grid should enable access to analysis services.	1
Tier:	2	Middleware should permit transparent service access.
Peer:	1	Peer networks should be able to forward service requests to nodes that can do work.

8.6	A data-grid should enable users to upload code.	2
Layer:	1	Layered abstraction may be applied to mobile code (possibly separating presented specification, parsed bytecode for virtual machine, and the deployed executable).

8.7	A data-grid should enable users to manage data.	2
Peer:	1	Peer networks decentral management means user responsible for their work.

8.8	A data-grid should enable users to manage their accounts.	2
Tier:	1	The middle-tier may provide a point where accounting can be reliably managed.

8.9	A data-grid should enable users to organise active jobs.	2
Tier:	1	Middleware services should include task management, which may be exposed to the client.
Peer:	1	Peer networks typically provide handles through which tasks can be identified, which may support user task control in an uncontrolled network.

8.10	A data-grid should offer an interactive and integrated workbench.	2

8.12	A data-grid could enable collaborative work between users.	3
Tier:	1	A middle tier may coordinate activity between users.
Peer:	2	Peer networks encourage collaboration as control is decentral, and anonymous sharing is enabled.

8.13	A data-grid could enable users to disconnect and leave jobs running.	3
Tier:	2	Tiers separate clients from back-end activity, enabling backend state to be maintained independently of user.
Peer:	1	Execution on a remote peer is possible without instigator connection.
Pipe:	2	Pipeline workflow management allows offline progress

Agent:	1	Agent progress may be possible without a client (as long as the client does not host blackboard).	
9.1	A data-grid should support a range of existing applications and tools.		2
Layer:	1	Layers may hide underlying differences by abstraction.	
Tier:	2	Tiers should be able to transparently wrap diverse back end tools.	
9.2	A data-grid could allow users to create new applications or tools through an API.		3
Tier:	1	Middle tier metadata should be flexible enough to add new services.	
Peer:	1	Peer networks should allow easy registration of new services.	
9.3	A data-grid could allow users to create new applications or tools through composition of existing services.		3
Layer:	1	Abstraction of implementation to service descriptions may be helped by layers.	
Tier:	1	Middle tier middleware also provides abstraction and mechanisms for generic service description with their IDLs.	
Pipe:	1	Services may be composed in a pipeline description.	
9.4	A data-grid could support visualisation tools for browsing data.		3
10.1	A data-grid must enable users and administrators to access information about the static state of the system.		1
Tier	1	Monitors may relatively easily be included in the middle tier to use static metadata.	
Peer:	-1	Peer networks have little static structure, as they should dynamically organise themselves.	
10.2	A data-grid must enable users and administrators to access information about the dynamic state of the system.		1
Tier:	1	Monitors may relatively easily be included in the middle tier to use dynamic data about the network.	
Peer:	-1	Peer networks typically hide user activity (due to their application context), but if monitor hooks were included they could only give local information reliably.	
Agent:	1	The central blackboard that agents write to may be viewed by administrators to determine their current working state.	
11.1	A data-grid must enable the management of work over distributed resources.		1
Tier:	2	The middle-tier exists to manage distributed systems.	

Peer:	1	Peer networks make good use of distributed resources, though central management is not typical.
Pipe:	2	Pipeline scheduling should control tasks on distributed resources (with staging and synchronisation).
Agent:	1	Work may be divided across resources in an agent-based architecture.

11.2	A data-grid should enable jobs to be matched to available resources.	2
Tier:	2	The middle tier controls dispatch to distributed resources.
Peer:	2	Peer networks may fit resources to requested needs well.
Pipe:	1	Schedulers that use pipeline view may dynamically determine resources used.

11.3	A data-grid should enable jobs to be prioritised.	2
Tier:	1	Middle-tier meta-data may include queued task priority for dispatch.
Peer:	1	Priority may be determined by “time to live” style tags on work dispatched to a peer network.
Pipe:	1	Queues at service points in a pipeline may be prioritised.

11.4	A data-grid should enable bottlenecks in the system to be identified and corrected.	2
Tier:	2	Middle tier enables progress monitoring and provides control mechanisms to reorganise the network.
Peer:	1	Peer networks may be designed to be free from bottlenecks by sharing and automatically avoid over use as all nodes are servers (but if bottlenecks form they may be hard to identify).
Pipe:	1	The scheduler should avoid bottlenecks based on task decomposition and staging.

11.5	A data-grid could enable re-negotiation of resources for jobs already running.	3

11.6	A data-grid could enable checkpointing of active jobs.	3
Agent:	1	Pipelines should provide checkpoints for recovery of flow around point of failure.

12.1	A data-grid should support intercommunication and interoperation with other grids in related domains.	2
Layer:	2	Different levels of abstraction allow mapping to diverse protocols.
Tier:	2	Tiered networks allow service via a portal to be presented in the same way as intra-system controlled resources.
Pipe:	1	Pipe and filter may allow transformation from service within one system to another (as compute grids link specialised HPC to CPU farms).

13.1	A data-grid must support the authentication of users/resources.		1
Tier:	1	Middle tier can organise certification before back end access.	
Peer:	1	Peer networks may insist of signatures within exchanges (though typically the same technology is used for the opposite purpose - anonymisation).	
13.2	A data-grid must support the authorisation of users/resources.		1
Layer:	2	Security layer can police authorisation requirements.	
13.3	A data-grid should support the auditing of actions carried out by system entities.		2
Tier:	1	Middle tier metadata could be used as the basis of an audit trail.	
Peer:	1	Some peer networks may generate ‘cookie trails’ that could be used for auditing.	
13.4	A data-grid should enable users/resources to be made accountable for their actions within the system.		2
Tier:	1	Middle tier metadata could be used as the basis for accountability.	
13.5	A data-grid should support the enforcement of individual security policies.		2
Tier:	1	Tiers can help to wrap local heterogeneous policies and technologies.	
Pipe:	-1	Smooth pipeline operation may be inhibited by diverse methods of boundary negotiation.	
13.6	A data-grid should support individual security policies that are subject to rapid change.		2
Pipe:	-1	Changing methods of boundary negotiation may inhibit smooth pipeline operation.	
13.7	A data-grid should support individual security policies that vary in strength and granularity.		2
Tier:	1	Tiers can help to wrap local heterogeneous policies and technologies.	
13.8	A data-grid should accommodate the existing security mechanisms of individual resources.		2
Tier:	1	Tiers can help to wrap existing heterogeneous mechanisms.	
13.9	A data-grid should support single-step and multi-step sign on.		2
Tier:	2	Tier decoupling may allow single application side sign event to be associated with multiple back end checks.	
Pipe:	1	Workflows may possibly specify sign on as interaction points.	

13.10	A data-grid should support mobile users.		2
Agent:	1	Agents may be mobile, and may therefore support mobile users too (possibly in the role of an agent writing to the shared area).	
13.11	A data-grid should allow users to confirm the integrity of data after transfer or processing.		2
Layer:	1	Layer technology is used to verify integrity of messages.	
Tier:	1	Tier management could support independent validation.	
Peer:	1	Peer networks typically provide mechanisms that guarantee integrity.	
13.2	A data-grid should not employ security mechanisms or processes that significantly reduce the availability of resources.		2
Layer:	-1	Layers add overhead as each part unpacks and verifies messages according to its responsibilities.	
Tier:	-1	Tier mechanisms add an overhead.	
14.1	A data-grid must scale to be able to include 10TB - 1PB new data per year.		1
Tier:	-1	Management via tiers may introduce bottlenecks that have scaling limits.	
Peer:	2	Peer networks scale very well (though typically rely on many small nodes).	
14.2	A data-grid must be able to include data files and sets of variable size.		1
Layer:	-1	Layers typically provide the same types of frame for all traffic, which may add overhead when only small messages pass through a high capacity protocol.	
Tier:	1	Tiered networks may separate data from message signal traffic, enabling equal management and control for a range of data scale.	
Peer:	1	Peer networks typically provide a way to separate data from its metadata, enabling large resources to be publicised in an equivalent way to small ones.	
14.3	A data-grid should be able to include a total volume of data of PB scale.		2
Tier:	1	Separation of control to higher tiers from back-end storage helps interaction with demanding resources.	
Pipe:	1	Handling very large scale data migration may be helped by parallel streaming that may be described in a workflow.	
14.4	A data-grid should be able to scale to include new data resources.		2
Peer:	2	Peer networks are designed to grow as data providers are added.	
14.5	A data-grid could support up to 10,000 simultaneous processes.		3

Tier:	0	Though tiers support distributed state, they may also represent a bottleneck that may impede progress of large number of parallel activities.
Peer:	2	Decentred peer networks may make progress on millions of concurrent tasks.
Pipe:	1	Some high performance computing monitors of parallel tasks approach this scale.

15.1	A data-grid should be enable use of resources to be managed for optimum performance.	2
Tier:	1	Middleware metadata should provide monitoring information as basis for optimisation by reconfiguring resources.
Pipe:	1	Workflow schedules may be tuned to make optimal use of resources (though this is non-trivial).

15.2	A data-grid could enable a query response time of 5-10s.	3
Peer:	-1	Peer network topology implies constrained response time cannot be guaranteed.
Pipe:	-1	Pipeline components are best suited for long end-to-end tasks, and therefore not for rapid interaction.
Agent:	-1	Agent based methods are not typically designed for responsiveness, and may take a arbitrarily long time to compose a result.

15.3	A data-grid could support near real-time data processing.	3
Pipe:	2	A data stream may go directly into pipeline processing (e.g. Regular analysis of data from an instrument), and filters may work to a central clock to ensure synchronise progress.

16.1	The security services of a data-grid should not have a single point of failure.	2
Tier:	1	Tiers may coordinate shared responsibility, so a validation task may fail over to a redundant resource when the primary service point fails.
Peer:	2	Decentred peer networks are ideal for elastic degradation of service as sub-sets of the network may continue to make progress on node failure.
Agent:	-1	If a blackboard were used for any part of the security process, this would be potential single point of failure.

16.2	The data access services of a data-grid should be faulty tolerant to some degree.	2
Tier:	1	A middle tier may handle transfer to redundant nodes on primary failure to ensure continued data access service.
Peer:	2	Peer networks are highly fault tolerant with respect data routing.

16.3	A data-grid should have capabilities for job recovery in the event of system failure.		2
Tier:	1	Middleware may coordinate transfer of task state from a failed resource to store or another resource.	
Pipe:	1	Workflows may include checkpoints that allow for job recovery.	
17.1	A data-grid should allow new functionality or services to the system once deployed.		2
Layer:	1	Abstraction provided by layers may facilitate low level extension.	
Tier:	1	In a component service network, the middle-tier meta-data descriptions of function and discovery method should scale to include new resources.	
Peer:	1	Peer networks typically allow flexible description of service at nodes.	
17.2	A data-grid should support the portability of system components local to users and data resources.		2
Layer:	2	Portability is greatly helped by layers (hiding hardware or other low-level dependencies from the application).	
Tier:	2	Tiers provide transparency, and platform transparency is fundamental.	
18.1	A data-grid must allow existing heterogeneous components to be successfully integrated, as necessary.		1
Layer:	1	Layered abstraction of low-level platforms enable component integration.	
Tier:	2	A primary aim of the transparency enabled by a middle tier is heterogeneous component integration.	
Peer:	1	Peer networks typically integrate heterogeneous nodes (which may host heterogeneous components).	
18.2	A data-grid could allow heterogeneous components that are not yet available, to be successfully integrated.		3
Layer:	1	Layer abstraction also enables integration with future diverse low-level elements.	
Tier:	1	Middle-tiers should enable future integration, but may enforce component responsibilities to allow compatibility.	
Peer:	1	Peer network flexibility should extend to future uses.	

Appendix F. ACME model

F.1. EGSO defined by 3 connector stereotypes

This code defines 3 connector types for streams, interaction and fire-and-forget protocols. The 4 component types defined first are classified by their interaction style. The candidate EGSO architecture is built from these types with 22 component instances and their connectors. Grouping components into 3 tiers is not formally encoded.

```
// file: egso_tier_my_go.acme author: J Lewis-Bowen updated: 2002.07.08
// content: Hand crafted Acme definition of Grid architecture

// ABSRACT TYPE TEMPLATES
// Studio generates 'Family' not 'Style', and uses 'Type' instead of 'Template'

Style grid =
{
   // 4 COMPONENT TYPES

   // parameter for creating a grid_origin node - just its unique submit source
   // I'm not sure how 0-n request source ports can be added later
   Component Template grid_origin
      ( grid_submit_source : Ports ) =
   {
      Ports grid_submit_source, grid_request_source;
      // I can't think of any properties for now
      // - rules about ports for external 1NF parser?
      Properties{}
      // NB unexpected syntax - no ';' after {} block
   }

   // consumer's got to have one input and at least one write output
   Component Template grid_consumer
      ( grid_submit_target, grid_write_source : Ports )
   {
      Ports grid_submit_target, grid_write_source;
      Properties{}
   }

   // filter's got to have one input and output (and 0-n request points)
   Component Template grid_filter
      ( grid_submit_target, grid_submit_source : Ports )
   {
      Ports grid_submit_target, grid_submit_source,
         grid_request_source;
      Properties{}
   }

   // data's got to have at least one request target port
   Component Template grid_consumer
      ( grid_request_target : Ports )
   {
      Ports grid_request_target, grid_write_target;
      Properties{}
   }

   // 3 CONNECTOR TYPES
   // all of these have constructors with the ports at either end
   // use defining for auto-naming based on end points (I think)
   // NB unexpected syntax - constructor lists Ports type for comp, Port now

   Template request( request_source, request_target : Port )
      defining ( this_conn : Connector ) =
   {
      this_conn = Connector
      {
         // to be honest, I'm not sure what these connector roles
         // (for attached components) add...
```

```
            Roles
            { grid_source_comp, grid_target_comp }
            // Like components, I can't think of any properties for now
            // - parameters that distinguish the 3 types of communication?
            //   e.g. paradigm = { packet | exchange | stream };
            //   what is going to understand this?
            Properties{}
        }
        // Attachments link roles and ends of connector
        // - no surprises here, seen all these tokens before
        Attachments
        {
            this_conn.grid_source_comp to request_source;
            this_conn.grid_target_comp to request_target;
        }
        // NB unexpected syntax - constructor init here, not component roles
    }

    Template submit( submit_source, submit_target : Port )
        defining ( this_conn : Connector ) =
    {
        this_conn = Connector
        {
            Roles
            { grid_source_comp, grid_target_comp }
            Properties{}
        }
        Attachments
        {
            this_conn.grid_source_comp to submit_source;
            this_conn.grid_target_comp to submit_target;
        }
    }

    Template request( write_source, write_target : Port )
        defining ( this_conn : Connector ) =
    {
        this_conn = Connector
        {
            Roles
            { grid_source_comp, grid_target_comp }
            Properties{}
        }
        Attachments
        {
            this_conn.grid_source_comp to write_source;
            this_conn.grid_target_comp to write_target;
        }
    }
}


// SYSTEM DEFINITION
// I just want to define all the components and connections for now
// I expect to add the optional ports to components for some connectors

System egso : grid =
{
    // SERVANT TIER ENTITIES
    // (first 4 are one of each type!)
    // I'm unimaginatively naming ports as component at other end and direction
    gui = origin( search_source );
    template = data( gui_target );
    search = filter( gui_target, parse_source );
    download = consumer( process_target, result_source );
    // this is a bit of bodge - primary reader of data result is invisible user, not peer...
    result = data( peer_filter_target );
    peer_origin = origin( peer_filter_source );
    peer_filter = filter( peer_origin_target, download_source );

    // CONNECTIONS internal to servant tier
    // (first 3 one of each type again!)
    // so, thanks to previous naming source & target pretty trivial
    // but note, many ports not defined in constructors above...
```

```
    request( gui.template_source, template.gui_target );
    submit( gui.search_source, search.gui_target );
    write( download.result_source, result.download_target );
    request( peer_filter.result_source, result.peer_filter_target );
    submit( peer_origin.peer_filter_source, peer_filter.peer_origin_target );

    // GRID TIER ENTITIES
    parse = filter( search_target, broker_source );
    broker = filter( parse_target, wrapper_source );
    wrapper = filter( broker_target, request_source );
    locate = data( broker_target );
    describe = data( wrapper_target );
    request = filter( wrapper_target, match_source );
    // queue is the target of lots of things, process really needs it to read
    queue = data( process_target );
    receive = consumer( send_target, cache_source );
    cache = data( process_target );
    process = origin( download_source );
    plugin = data( process_target );
    // notice how 3 ad-hoc ports are the target of process,
    // which is constructed with none?
    share_origin = origin( share_filter_source );
    // I picked locate, not describe, as primary sharing data arbitrarily
    // - this is probably the most important shared information
    share_filter = filter( share_origin_target, locate_source );

    // CONNECTIONS internal to grid tier
    submit( parse.broker_source, broker.parse_target );
    request( broker.locate_source, locate.broker_target );
    submit( broker.wrapper_source, wrapper.broker_target );
    request( wrapper.describe_source, describe.wrapper_target );
    submit( wrapper.request_source, request.wrapper_target );
    write( request.queue_source, queue.request_target );
    write( receive.queue_source, queue.receive_target );
    write( receive.cache_source, cache.receive_target );
    request( process.queue_source, queue.process_target );
    request( process.plugin_source, plugin.process_target );
    request( process.cache_source, cache.process_target );
    request( share_filter.locate_source, locate.share_filter_target );
    request( share_filter.describe_source, describe.share_filter_target );
    submit( share_origin.share_filter_source, share_filter.share_origin_target );

    // ARCHIVE TIER ENTITIES
    match = filter( request_target, send_source );
    send = filter( match_target, receive_source );
    metadata = data( match_target );
    record = data( send_target );

    // CONNECTIONS internal to archive tier
    submit( match.send_source, send.match_target );
    request( match.metadata_source, metadata.match_target );
    request( send.record_source, record.send_target );

    // GLOBAL CONNECTIONS system wide inter-tier
    submit( search.parse_source, parse.receive_target );
    submit( process.download_source, download.process_target );
    submit( request.match_source, match.request_target );
    submit( send.receive_source, receive.send_target );
}
```

F.2. Comparing envisioned architectures' components

This table associates components of the 2 informal EGSO architecture visions with each other and with the components of 22 component ACME model. Blank cells indicate no association, italics represent tiers that aggregate components (which are only conceptual, and, though marked in the ACME code, not formally constrained).

Decentral	Hierarchical	ACME	Association
scientist	GUI	gui	Clear
servent		share	Clear
		servant	Likely; decentralised model may be at a higher level than functional sharing.
	user-input	template	Clear
	ancillary		None
query-agent	search-engine	search	Unclear; 'search-engine' is not user-side, 'query-agent' user delegate could be.
		parse	Unclear; 'search-engine' and 'parse' have different roles to 'query-agent' manager.
trading-service	request-broker	broker	Unclear
		grid-tier	Likely; informal models may be at a higher level than functional 'broker'.
	resource-registry		Unlikely; 'trading service' is a daemon service, 'resource-registry' is just a gateway to distributed data.
	filemap-registry		
wrapper		wrapper	Clear; presuming data transform is not the same as query transform.
archive	source	*archive*	Clear
repository	unified-observations		Unclear; 'unified-observations' is an EGSO property, 'repository' is 3rd party metadata.
metadata-description		describe	Unclear; 'unified-catalogue' is not intended to be an ontology.
	catalogue		Unclear; 'catalogue' is intended as EGSO derived metadata avoiding need for 'wrapper'.
	features-events		Unclear; 'features-events' are intended to just be novel derived data.
plugin-repository		plugin	Clear; but 'plugin' could be more than a data converter.
metadata	file-map	metadata	Likely; different preconceptions of what data description is may be significant.
time-location	location	locate	Clear
data	data	record	Identity
	queue	queue	Identity
	process	process	Likely; in the hierarchical model, 'process' scope includes toward physical modelling.
		agree	None
		match	None
		receive	None
		cache	None
		send	None
		request	None
		download	None
		result	None
admin	other		Clear
resource-broker	RDBMS		Unclear; 'RDBMS' serves as a search-engine, 'resource-broker' does not support 'trading-service'.

Appendix G. Modelling methodology

This appendix describes a general method to develop models, arising from experience of evaluating EGSO and AstroGrid data-grid designs with FSP, described in Chapter 5. It may model other grid systems to judge whether requirements are met. The iterative, rapid process is introduced, and then demonstrated with a worked example. This walk-through may be used as a tutorial introduction to FSP specification.

There are 5 steps in the process:

1. *Requirements' analysis:* identify the purpose of the model and the events in it.
2. *Sequential implementation:* compose processes that represent single instances of the components and tasks.
3. *Concurrent implementation:* enable multiple concurrent component instances by indexing the processes and events.
4. *Testing:* analyze the composition, debug and refine the model.
5. *Operation:* demonstrate the model system and modify the real system's design.

The process is used to develop a demonstration model system in the remainder of this appendix. Though simple, the system is non-trivial and includes design elements used in grid systems. The FSP code for the model is presented for steps 2 to 4: the serial implementation, the parallel implementation and a refined implementation. Marking the unchanged code in grey highlights modifications to the code between model versions.

Each modelling step is discussed in four parts. First, the operational target and general design concerns of the modeller at the given step are described. Next, the language features introduced in the model version are noted. Notes on debugging follow to highlight some common errors; these cannot be exhaustive but may help those new to FSP to avoid puzzling faults in code successfully compiled by LTSA. Finally, the reusable grid design patterns employed are highlighted.

G.1. Step 1 - Intention of modelling a service

The tutorial system represents a service that actions asynchronous user tasks; these may represent database queries or computation jobs. It is required to serve several users. Users may submit several tasks. It is also required that the service should return the completed task to the correct user. The user should be able to distinguish multiple results even though sequence is not guaranteed.

The level of detail in the worked example captures a system architecture or interface design specification. The model's purpose would be to evaluate whether such a design implements the system requirements.

The two components of the hypothetical system design, 'user' and 'service', are integrated by two messages: users request their tasks to be actioned, and the service returns the task result. The state transitions for a task are distributed between the components: users move tasks from their initial state by submitting them, and the service completes tasks by working on them.

This example is simpler than a genuine grid system architecture. However, the two components are very similar to the consumer and provider interfaces to brokers in EGSO, and component pairs in the AstroGrid model (user message queue and job controller, or data agent and job manager). In the model, the service event for task completion is hidden from the user -- it could be modelled as a complex operation with other components as in a genuine system. By hiding back-end complexity in this way, grid systems manage dynamic metadata and enable transparent access to heterogeneous resources. The distributed state transition model is also an essential feature of grid systems. Therefore, the tutorial system genuinely reproduces the ingredients of real data-grid models.

G.2. Step 2 - Sequential user task service

Model goal

Processes are defined for the user and service components and the task state transitions. Their events are combined in a system process. Component communication is represented by the shared events 'request' and 'result' -- these are paired like communication API operations on the protocol's source and sink. The component activity is defined in the task events 'submit' and 'work' - these represent the functional algorithms that transform state. These are implemented in the first model:

```
TASK =
   ( submit -> queued -> work -> done -> TASK ).
USER =
   ( submit -> request -> USER
   | result -> done -> USER ).
SVC =
   ( request -> queued -> SVC
   | work -> result -> SVC ).
||SYS =
   ( TASK || USER || SVC ).
```

Language features

In FSP (capitalised) processes are defined by a sequence of (lower case) events. The state models loop, indicated by returning to the process name. Termination prevents analysis, as LTSA stops examining the combined state space once a point is found from which no further progress can be made; LTSA can identify safe versus undesirable paths in models with 'STOP' and 'ERROR' terminations, keywords indicating service does not continue.

In this model, alternative state transition sequences are indicated by the pipe symbol. Though the user or service process in isolation would follow the alternative paths in a non-deterministic way, the task event sequence will guarantee the expected order.

Processes are composed in a concurrent higher level process using double pipe operator (which also prefixes the composite process name). Such processes guide LTSA's composition of simple processes' state spaces, whose complex interaction can be analyzed for safety and progress.

Debugging

The 'queued' and 'done' events were added to the task transitions, paired events for the functional transitions. Without the 'queued' event in the service process to go after the user process's 'submit', the model would allow the 'work' event before completing the communication events.

It can be beneficial to employ a naming convention for events (not used here, as it was judged terse code is easier to absorb when new to the language). The letters of the processes communicating may indicate event direction; for example, 'us_request' and 'su_result' in the above model prefix shared 'user' and 'service' synchronous events. Conversely, events that are not shared may be explicitly hidden to reduce the state space for LTSA. Additionally, the events taken directly from the design may be distinguished from those added whilst debugging or refining a model by using different name styles. As LTSA can list the model's event alphabet, such conventions help to highlight design flaws exposed by the model.

Design patterns

This model separates the applications' operation from interaction in the underlying infrastructure. The task events represent the functional transformations, whilst the 'request' and 'result' events represent communication. Note also that without the task events, the user and service processes are identical; with them, communication direction is indicated by distinguishing the messages' sources and sinks. FSP processes can therefore clearly represent a layered architecture.

G.3. Step 3 - Concurrent users and tasks

Model goals

Multiple user instances are created in the system composition at this step; 2 are sufficient to demonstrate concurrent operation. Multiple task instances are also required; 3 are more than sufficient to demonstrate concurrent task submission by a user. Concurrent instances of the user and task processes of the first model are implemented in the second version:

```
TASK =
   ( submit.usr[ u:1..2 ] -> queued ->
      work.usr[ u ] -> done -> TASK ).
USER =
   ( submit.tsk[ new_t:1..3 ] -> request.tsk[ new_t ] -> USER
   | result -> done.tsk[ old_t:1..3 ] -> USER ).
SVC =
   ( usr[ new_u:1..2 ].request.tsk[ new_t:1..3 ] ->
      tsk[ new_t ].queued -> SVC
   | tsk[ do_t:1..3 ].work.usr[ do_u:1..2 ] ->
      usr[ do_u ].result -> SVC ).
||SYS =
( tsk[ t:1..3 ]:TASK || usr[ u:1..2 ]:USER || SVC ) /
   { usr[ u:1..2 ].submit.tsk[ t:1..3 ] /tsk[ t ].submit.usr[ u ],
      usr[ u:1..2 ].done.tsk[ t:1..3 ] /tsk[ t ].done }.
```

Language features

A range of integers - for example `usr[ u:1..2 ]:USER` - index multiple process instances at composition (the name 'usr' is cosmetic). This prefix is applied to all events in the process

instances to ensure they are uniquely named in the composed state space. To index the events in other processes an equivalent suffix is used - for example submit.usr[u:1..2]. In both cases, the variable over the range may be used within the scope of an event sequence -- for example, the task process reuses 'u' to ensure work is carried out for the right user. Synonyms are listed for the composed process, using the '/' operator; these are necessary to indicate the prefixes are equivalent to the suffixes for events synchronised between pairs of processes that both have multiple instances.

Debugging

Errors when matching event prefixes are common, and cause unexpected events. These should be checked for in LTSA by noting inappropriate possible events when manually tracing state transition sequences. For example, if a typographic error made the first service process event prefix user[new_u:1..2].request.tsk[new_t].1..3, 'request' would be possible before the user process had made the 'submit' event.

Event matching errors can also be easily introduced in the synonyms. This risk is mitigated by the naming convention used here, where suffix and prefix values are symmetrically swapped. Note that this is not a hard rule; here it was decided that though the 'done' event needs to indicate the task index for the user processes, the equivalent event in the task process does not need the user suffix.

Named constants and ranges may be substituted for the integers given (using the 'const' and 'range' keywords in declarations, as below). This can make the code easier to understand and enable the number of entities to be changed easily, notably when the combined state space is too large for LTSA to compose.

Errors in ranges are also common (though this model is not especially at risk). For example, a variable may represent a range of states for events, whilst some process sharing the event may only operate on a subset of possible states. If the full range is used in the process definition, inappropriate progress may be made when the process transforms states with incorrect indexes; an example applied to this tutorial's model would be the service process doing the submit event.

Design patterns

The distributed state model is more advanced in this version; the task process instances carry information about the user that submitted them. In this way task metadata is represented independently of a specific component. Therefore the service functionality is kept simple, pending tasks may be actioned in an arbitrary sequence, and completed tasks are returned to the correct user.

The asynchronous session state information represented here is an essential feature of grid services (in contrast with web services [41]). This pattern scales well when several functions are required to complete a task, and service points action several task types. It therefore models a grid system's flexible workflow management, with dynamic resources supporting heterogeneous applications.

G.4. Step 4 - Refinement with a semaphore

Model goal

Analyzing the previous section's model in LTSA demonstrates that the system deadlocks. This is because a user acts as both a client and a server by generating requests and consuming results. If both are attempted simultaneously, neither the user nor the service can make progress.

Deadlock in the concurrent implementation, is avoided by adding a semaphore in the model below. The semaphore ensures safe operation as it must be claimed by the competing components before they exchange a message. A progress check is also added to ensure the system will not reach a livelock and tasks are guaranteed to eventually complete.

At least three other methods could avoid the deadlock. Each user request could block until the result is returned, or connectionless communication could be simulated by allowing messages to be lost between in transmission. Alternatively, existing tasks could be shared by multiple user and server processes, dividing responsibility for message generation and consumption -- this may be implemented as concurrent threads within a sub-system. These solutions are unacceptable, as the hypothetical requirements demanded that multiple asynchronous tasks for each user should be possible with just two reliable components.

```
SEMA = SEMA[ 0 ], SEMA[ b:0..1 ] =
   ( [ x:{ usr[ u:1..2 ], svc } ].claim ->
      ( when ( b ) [ x ].fail -> SEMA[ b ]
      | when ( !b ) [ x ].raise -> SEMA[ 1 ] )
   | when ( b ) [ x:{ usr[ u:1..2 ], svc } ].drop -> SEMA[ 0 ] ).
TASK =
   ( submit.usr[ u:1..2 ] -> queued ->
      work.usr[ u ] -> done -> TASK ).
USER = USER[ 0 ], USER[ t:0..3 ] =
   ( when ( ! t ) submit.tsk[ new t:1..3 ] -> USER[ new_t ]
   | when ( t ) claim ->
      ( raise -> request.tsk[ t ] -> drop -> USER[ 0 ]
      | fail -> USER[ t ] )
   | result -> done.tsk[ old_t:1..3 ] -> USER[ t ] ).
SVC = SVC[ 0 ], SVC[ u:0..2 ] =
   ( usr[ new_u:1..2 ].request.tsk[ new_t:1..3 ] ->
      tsk[ new_t ].queued -> SVC[ u ]
   | when ( ! u ) tsk[ do_t:1..3 ].work.usr[ do_u:1..2 ] -> SVC[ do_u ]
   | when ( u ) svc.claim ->
      ( svc.raise -> usr[ u ].result -> svc.drop -> SVC[ 0 ]
      | svc.fail -> SVC[ u ] ) ).
||SYS = ( tsk[ t:1..3 ]:TASK || usr[ u:1..2 ]:USER || SVC || SEMA ) /
   { usr[ u:1..2 ].submit.tsk[ t:1..3 ] /tsk[ t ].submit.usr[ u ],
      usr[ u:1..2 ].done.tsk[ t:1..3 ] /tsk[ t ].done }.
progress PROG = { usr[ u:1..2 ].done.tsk[ t:1..3 ] }
```

Language features

Process state suffixes (for example, `SEMA[ b:0..1 ]`) and conditional event paths (using the keyword 'when') are introduced in this model with the new semaphore process. Process state suffixes allow a process to hold different states between state transition sequences. The initial semaphore state is false - the first claim will be successful and change the process state, the next claim would fail.

In a similar way, user and service process states are used to hold information on the task to be submitted and the user to return the completed task to respectively. These parameters ensure that events are repeated for the correct task when a semaphore claim fails.

The progress check is indicated by the named set of target events (declared with the 'progress' keyword). LTSA proves that there must be a path to these events from anywhere in the combined state space; the tool gives equal priority to possible event paths, unless otherwise indicated, to determine whether the events can be reached.

Debugging

When introducing the semaphore process, it is easy to overlook the necessary event prefix (for example, `[ x:{ usr[ u:1..2 ], svc } ].claim`). This is necessary to make the event names unique, though the semaphore itself has one state model for all processes using it. The variable 'x' can take values over the user process instance prefix range or the value 'svc' - the prefix used when the service uses the semaphore. Without this, the system quickly reaches deadlock (as each semaphore event is synchronised to every processes that uses it).

State parameters were added to the user and service processes that use the semaphore. Without them, the processes would have to repeat the 'submit' or 'work' events when a semaphore claim failed. This would represent a poorly designed system that has to repeat application functions when communication fails. By adding them, the user and service processes are guaranteed to complete their action on one task before starting another. However, this solution makes the processes more complex and less flexible. These faults would be aggravated if the components performed more than one function. By having to refine the model in this way, possible problems in the two-component design may have been exposed; adding additional staging components may simplify the model and, ultimately, the system.

Design patterns

The semaphore is a generally used pattern in concurrent distributed systems. To be used effectively, it must guard the critical resource - for this model (and decentred grid systems in general), the service communication channel. As there is a single service, a single semaphore instance is sufficient. For protected communication between components in an N to M relation, an N × M semaphore combination may be necessary - requiring complex synonyms to model.

Process state tests, like the semaphore's, can represent the distributed state transitions of a data-grid task. The range of values can be enumerated with named constants to help debugging. This method is applied to the task process below. As well as determining when to do the transitions that represent application functions, other processes can use test events that do not update the task state. Monitoring services and other components that are essential to support data-grid infrastructure can be modelled in this way. Flexible services that support complex task workflows, dependent on shared system state, can also be built using this pattern element.

```
const TS_INI = 1
const TS_QUE = 2
```

```
range TS_R = TS_INI..TS_QUE
TASK = TASK[ TS_INI ], TASK[ ts:TS_R ] =
   ( when ( ts == TS_INI )
      submit -> queued -> TASK[ TS_QUE ]
   | when ( ts == TS_QUE )
      work -> done -> TASK[ TS_INI ]
   | test[ ts ] -> TASK[ ts ] ).
```

G.5. Step 5 - Hypothetical demonstration

If presenting the worked example to the stakeholder who required asynchronous response or the designer who specified the components and interface, the modeller may demonstrate features of the listing in the final model. Communication with the shared service point has been guarded to make it safe, and the components have been modified to prevent progress with other tasks until an actioned task is communicated. These features can be demonstrated by stepping through scenarios, illustrated with LTSA's animation and state transition graphs. The stakeholders may then decide to implement further models to evaluate alternative designs in which the user acts as a blocking client, or that have task staging components.

Commonly, by making a model concurrent at step 3, or by resolving errors at step 4, it becomes too complex for LTSA compose and analyze. The simple composed system of the last model has 2^{27}. (LTSA could not compose the 2^{41}states for 3 users and 4 tasks on the development machine used; the Java 1.4.1 run-time environment runs out of memory at 133MB.) In this case, the modeller must demonstrate a partial model and identify parts at risk to faulty interaction, before repeating the cycle of model development for a reduced system specification. The worked example in this section could be seen as such a simplified system; a single service point for multiple users and tasks would be the risky part of a larger system in which actions behind the 'submit' and 'work' events has been ignored.

Appendix H. EGSO concept models

H.1. Layer

This model demonstrates how layers trap different types of failure. Possible events in the life of a distributed search and analysis task are represented in 4 layers: client, grid portal, reference catalogue, and data store. 2 instances of a generic service interface at the portal and catalogue layers can forward queries to next level or fail. A binary state switch ensures all alternative responses are exercised, determining a looping, unbranched sequence of events; it takes 4 queries before a success.

```
/* file: dist_search_d.lts author: J Lewis-Bowen updated: 2002.09.11
 * conent: FSP model of events in a distributed search-analysis task */

const NWORK = 2
const NREF = 2
range BOOL = 0..1

Client = ( query -> { unavailable, nofind, nomatch, result } -> Client ).

Service = Service[ 0 ],
   Service[ work:BOOL ] =
      ( when ( work ) svcack -> Service[ 0 ]
      | when ( !work ) svcfail -> Service[ 1 ] ).

Portal = ( query -> Worker[ 0 ] ),
   Worker[ w:0..NWORK ] =
   ( when ( w == NWORK ) unavailable -> Portal
   | when ( w < NWORK ) work[ w + 1 ].svcrequest
      -> ( work[ w + 1 ].svcfail -> Worker[ w + 1 ]
         | work[ w + 1 ].svcack -> search -> Portal ) ).

Catalogue = ( search -> Reference[ 0 ] ),
   Reference[ r:0..NREF ] =
   ( when ( r == NREF ) nofind -> Catalogue
   | when ( r < NREF ) ref[ r + 1 ].svcrequest
      -> ( ref[ r + 1 ].svcfail -> Reference[ r + 1 ]
         | ref[ r + 1 ].svcack -> request -> Catalogue ) ).

Store = Store[ 0 ],
   Store[ work:BOOL ] =
      ( when ( work ) request -> result -> Store[ 0 ]
      | when ( !work ) request -> nomatch -> Store[ 1 ] ).

||System = ( Client || Portal || Catalogue || Store
   || work[ w:1..NWORK ]:Service || ref[ r:1..NREF ]:Service ).
```

H.2. Queue

This model demonstrates fair, concurrent distributed task progress. A portal serves several clients; queries are actioned by a trivial worker process. Each client submits a query to the shared portal, which are queued by a scheduler before the single worker uses a dispatch process to take jobs and generates return result.

The client's simple event transitions start and end (in schedule error or success) a query. The user-base is the set of clients - an indexed identifier 'user' prefixes events for clarity (a style used in many subsequent models).

The portal has 2 internal state cycles for query submission and response. Its query, result and unavailable events are directed to the client (being shared synchronous events). Its queued and complete events are shared with the worker (though only the stubs are shown here).

The query queue is made of several slots; each references the user whose job the query is, whilst zero indicates an empty slot. Checking slot state has no side-effect, but allocation can only happen when a slot is empty, whilst freeing only happens when a slot is filled (other processes decide to allocate or free).

The portal uses the scheduler to queue jobs (the client's query event is synchronised). It uses the 'plan' sub-process to loop through slots for user queries. As it checks each slot (using an offset counter for zero to maximum index range convenience), it decides to queue the query if the slot is free or move on to next if its full. When it is checked all the slots it may decide that the service is unavailable. Note that the check and allocate events are synchronised to the slot process, the queued event is synchronized to the portal, and 'skipslot' is a hidden event.

The dispatch process used by the worker mirrors the scheduler - an external 'pollforwork' event prompts a walk through the queue looking for non-null worker assigned slots to free and do work on. The worker passes the start index (here it keeps state for a fair round robin poll - other methods have been tried), so the 'nowork' event may be returned when the queue is not empty. Note that it uses a different check to the scheduler, so there's no irrelevant synchronisation. Also note that an implemented index wrap around has been removed. This was intended to save a worker that has just completed a job from the last slot from receiving 'nowork' and having to poll from zero again. However this is necessary anyway if the queue is empty above last slot used.

The trivial worker takes anything queued (the event 'pollforwork' is synchronised to the dispatch process, which frees it from the queue) and immediately sends the synchronised 'complete' event back to portal. It uses initial queue index polled is stored so that jobs do not get stuck above the first slot (and all clients can make progress). Note that the index has the same offset as the poll loop so that the range matches and indexed events.

```
/* file: queue_c.lts author: J Lewis-Bowen updated: 2002.09.12
 * content: FSP code for client and worker about portal-scheduler */

const NUSER = 3
range USERBASE = 1..NUSER

Client = ( request
      -> { unavailable, result }
      -> Client ).

||Userbase = ( user[ USERBASE ]:Client ).

Portal = ( user[ u:USERBASE ].request
      -> user[ u ].submitjob
      -> ( user[ u ].queued
            -> Portal
         | user[ u ].queuefull
            -> user[ u ].unavailable
            -> Portal )
   | user[ u:USERBASE ].complete
      -> user[ u ].result
      -> Portal ).
```

```
Slot = Slot[ 0 ],
   Slot[ u:0..NUSER ] =
      ( { check_in[ u ], check_out[ u ] } -> Slot[ u ]
      | when( u == 0 )
         allocate[ new_u:USERBASE ]
         -> Slot[ new_u ]
      | when( u > 0 ) free[ u ]
         -> Slot[ 0 ] ).

const NQUEUE = 2
range QUEUE = 1..NQUEUE

||Queue = ( queue[ QUEUE ]:Slot ).

Schedule = ( user[ u:USERBASE ].submitjob
      -> Plan[ 0 ][ u ] ),
   Plan[ slot:0..NQUEUE ][ u:USERBASE ] =
      ( when ( slot == NQUEUE )
         user[ u ].queuefull
         -> Schedule
      | when ( slot < NQUEUE )
         queue[ slot + 1 ].check_in[ slot_u:0..NUSER ]
         -> if ( !slot_u ) then
            ( queue[ slot + 1 ].allocate[ u ]
            -> user[ u ].queued
            -> Schedule )
         else
            ( skip_full_slot[ slot + 1 ]
            -> Plan[ slot + 1 ][ u ] ) ).

Dispatch = ( pollforwork[ start:0..NQUEUE ]
      -> Poll[ start ] ),
   Poll[ slot:0..NQUEUE ] =
      ( when ( slot == NQUEUE )
         nowork -> Dispatch
      | when ( slot < NQUEUE )
         queue[ slot + 1 ].check_out[ u:0..NUSER ]
         -> if ( u ) then
            ( queue[ slot + 1 ].free[ u ]
            -> user[ u ].dowork[ slot ]
            -> Dispatch )
         else
            ( skip_empty_slot[ slot + 1 ]
            -> Poll[ slot + 1 ] ) ).

Work = Work[ 0 ],
   Work[ robin:0..NQUEUE ] =
      ( pollforwork[ robin ]
      -> ( nowork -> Work[ 0 ]
         | user[ u:USERBASE ].dowork[ point:0..( NQUEUE - 1 ) ]
            -> user[ u ].complete
            -> Work[ point + 1 ] ) ).

||System =
   ( Userbase || Portal
   || Queue || Schedule || Dispatch
   || Work ).
```

H.3. Secure

This model demonstrates that client progress may be contingent on authentication by a third party. In the simple authentication service (that may stand for authorisation too) users have accounts, enabled by an administrator. The grid service portal checks for authority before returning success or denying service (lower tiers that would do work to resolve a request are not represented).

As in the previous model, the user-base is made of users who request service then get success or denial responses. User data is made of accounts with a binary state for certified users; a check can decide whether they are valid or invalid. The administration process authorizes user. Ideally, the model would allow users to be added dynamically, but it needs all possible states statically defined. Expiry and withdrawal are also not represented. As only unauthorised accounts may only be authorized, the administration process stops in this model (technically an error for LTSA). The service process shares user start and end events, and uses the check event to synchronize with the account event path for valid and invalid users to determine which result is reached.

```
/* file: authority.lts author: J Lewis-Bowen updated: 2002.09.13
 * content: FSP code for authorisation service and user account admin */

range BOOL = 0..1
range USERBASE = 1..2

User = ( request -> { success, deny } -> User ).
||Userbase = ( user[ USERBASE ]:User ).

Account = Account[ 0 ],
   Account[ allow:BOOL ] =
      ( when ( allow ) check -> valid -> Account[ 1 ]
      | when ( ! allow ) check -> invalid -> Account[ 0 ]
      | when ( ! allow ) authorise -> Account[ 1 ] ).
||Userdata = ( user[ USERBASE ]:Account ).

Admin = ( user[ u:USERBASE ].authorise -> Admin ).

Service = ( user[ u:USERBASE ].request -> user[ u ].check
      -> ( user[ u ].valid -> user[ u ].success -> Service
         | user[ u ].invalid -> user[ u ].deny -> Service ) ).

||System = ( Userbase || Userdata || Admin || Service ).
```

H.4. Tier

This model demonstrates how tiers provide location transparency (and support use of data mirrors) whilst still allowing users to specify a preference. 2 clients share a server for the dataset domain, which has 2 paired data and reference stores (representing data and metadata). Data properties within a store that could match a semantic query are not represented (as is typical for subsequent models). A Boolean switch at the data tier determines availability, flipped at access, which allows searches to fail (an administrator can enable or disable the data store from the opposite state). Note that the switch is not affected if the data availability is just checked. The reference service may therefore be used as a metadata catalogue to check availability. Each client queries with location transparency, as a result from any store is acceptable. The intermediate concurrent system 'domain' helps the naming of shared events with a dataset index.

The client process has a simple query and response cycle. The result, if got, is indexed to dataset used, demonstrating provenance. The server data access layer takes the client query, and decides which dataset to use in the logic of a search sub-process that cycles over the stores (using an offset index). Note, it uses the result event (not the get event) just to save

complex alias relabelling (as used in later models). The server chooses a users' favourite store when there are multiple hits. It uses the 'hitlist' collection of all datasets in a second pass to make the choice; each record on the list has a binary state for a hit.

```
/* file: tier_h.lts author: J Lewis-Bowen updated: 2002.09.13
 * content: FSP code for server transparently chosing data resources */

range BOOL = 0..1
const NDATA = 2
range DATASET = 1..NDATA
range USERBASE = 1..2

Data = Data[ 1 ],
   Data[ hit:BOOL ] =
      ( when ( hit ) disable -> Data[ 0 ]
      | when ( ! hit ) enable -> Data[ 1 ]
      | check[ hit ] -> Data[ hit ] ).

||Domain = ( dataset[ DATASET ]:Data ).

Dataadmin = ( { dataset[ d:DATASET ].enable, dataset[ d:DATASET ].disable }
   -> Dataadmin ).

Find = Find[ 0 ],
   Find[ hit:BOOL ] =
      ( found -> Find[ 1 ]
      | clear -> Find[ 0 ]
      | review[ hit ] -> Find[ hit ] ).

||Hitlist = ( dataset[ DATASET ]:Find ).

Server = ( user[ u:USERBASE ].query -> Search[ 0 ][ u ][ 0 ] ),
   /* search iterates over dataset for user, loop carries hit state */
   Search[ d:0..NDATA ][ u:USERBASE ][ got:BOOL ] =
      /* passed data, no hits, return failure to client */
      ( when ( d == NDATA && ! got )
         user[ u ].nomatch -> Server
      /* passed data, at least one hit, go on to chose which to return */
      | when ( d == NDATA && got )
         search_complete -> Chose[ 0 ][ u ][ 0 ]
      /* still searching data - check, update find token and move on */
      | when ( d < NDATA ) dataset[ d + 1 ].check[ match:BOOL ]
         -> if ( match ) then ( dataset[ d + 1 ].found
            -> Search[ d + 1 ][ u ][ 1 ] )
         else ( dataset[ d + 1 ].clear
            -> Search[ d + 1 ][ u ][ got ] ) ),
   /* chose iterates over hitlist with flag for second pass */
   Chose[ d:0..NDATA ][ u:USERBASE ][ repeat:BOOL ] =
      /* end of list first pass (no favorite found), start repeat pass */
      ( when ( d == NDATA && ! repeat )
         no_favourite -> Chose[ 0 ][ u ][ 1 ]
      /* in repeat pass, look for any match to give client or move on*/
      | when ( repeat && d < NDATA )
         dataset[ d + 1 ].review[ match:BOOL ]
         -> if ( match ) then ( dataset[ d + 1 ].get
            -> user[ u ].result[ d + 1 ] -> Server )
         else ( skip[ d ] -> Chose[ d + 1 ][ u ][ 1 ] )
      /* in first pass, look for match in user's favorite store (match
       * store and user number) to give client or move on */
      | when ( ! repeat && d < NDATA )
         dataset[ d + 1 ].review[ match:BOOL ]
         -> if ( match && ( d + 1 ) == u ) then ( dataset[ d + 1 ].get
            -> user[ u ].result[ d + 1 ] -> Server )
         else ( skip[ d ] -> Chose[ d + 1 ][ u ][ 0 ] ) ).

Client = ( query -> { nomatch, result[ d:DATASET ] } -> Client ).

||Grid = ( user[ USERBASE ]:Client || Server || Hitlist
   || Domain || Dataadmin ).
```

Appendix I. Modelling EGSO architecture

I.1. Roles

This model, based on the 3-role EGSO architecture, allows the animation of several scenarios derived from core requirements. The whole system is composed of several consumers, brokers and providers, clearly labelled. The 3 roles (or tiers) of the system are apparent in the processes (though the architectural components within each role are not, even though the events do line up to components). The client and resource events are non-deterministic choices, allowing the demonstration of possible functionality without decisions based on artificial logic. Decisions are made by the middle broker tier, though, that use its metadata of the datasets hosted by providers to resolve queries, whilst the provider decides whether it is busy using a gateway that blocks multiple work requests. Consumer and provider interfaces do map events, for example from process-1 event for process-2 index to process-2 event for process-1 index; this is non-concurrent bottleneck, but the processes are still genuinely concurrent and this method is clearer than synonyms (used in several models below). The broker cooperation interface also translates events between brokers and includes logic for forwarding queries in a ring, stopping when all brokers have been tried. Note that there is a serious risk of running out of memory when composing this model in LTSA, aggravated by the catalogue, though it is still possible to trace the scenarios. The number of elements in the tiers, defined by ranges up to a constant maximum where iteration is also used, can be lowered for composition (though there must be at least 2 brokers).

The consumer process can create queries, send them to providers or brokers (then block) then get a result. It has options for: generating queries from different sources, sending alternative query types, getting successful results, and receiving various failure cases. Note that variables need to be redeclared, as a provider other than the one addressed may respond (but the index of the responding broker is hidden - it does not matter).

The broker process uses a table of the providers associated with data sets, with a binary index indicating a link. The provider initiated 'set' event creates the association. Note that a conditional (that the association is not already set) is not needed; it is not an error if the flag gets set when it is already true. Also note that the language cannot compose a multi-dimensional event space in one go, so it is implemented in three steps (the final being for one table per broker in the system composition).

The broker accepts notification from a provider of which datasets are hosted - it records the provider to dataset mapping in its metadata catalogue (described above). The broker also answers queries against its metadata, uses it to route transparent data queries to a provider, and handles requests forwarded from other brokers via a broker cooperation interface (that is also responsible for stopping infinite query chains). Provider choice is based on an iterative search process over the catalogue (which always starts at the first provider, then tries to match the requested dataset - this is not intended as fair algorithm). The search holds state for the consumer reference and a flag for whether the query should be forwarded to the provider found (when the query is for data rather than just about availability of data). Therefore the sub-process

'SearchCatalogue' has iteration parameters for: the consumer, provider and dataset indexes, the flag that indicates whether this is a transparent query to forward to provider, and a counter for the number of brokers that have made this search (incremented by the broker cooperation interface). The broker counter can only start at 2 for the broker forwarded query acknowledgement (as the consumer that originated query uses a forward counter of 1), which means this model makes no sense with less than 2 brokers. (There could be a forward transparent query method here in addition to the forward metadata query using the same principle, but it would double the complexity of the broker cooperation interface, and so has not been implemented.) The provider iteration is also required when the first choice is busy. Note that the choice to repeat a search or terminate is made for a busy provider, the broker forwarding is disabled by the forward counter being initialised to the maximum.

Each provider accepts queries from the broker or consumer, giving the same success or failure matches to both (after a hidden work event, needed for concurrent progress to be demonstrated). The provider's notification of dataset availability to a broker is done exclusively by the interface, whilst job control (including the busy semaphore) is done by the gateway. Note that the direct query currently avoids the semaphore to minimize the interface process' complexity.

```
/* file: simple_cbp_final.lts author: J Lewis-Bowen updated: 2003.04.09
 * content: FSP code for 3 roles of EGSO architecture */

range CONS = 1..2
const MAXBROK = 2
range BROK = 1..MAXBROK
const MAXPROV = 2
range PROV = 1..MAXPROV
range DSET = 1..3
range BOOL = 0..1

Consumer =
   ( { input_query, read_stored_query } ->
      { transparent_query.brok[b:BROK].dset[d:DSET],
        availability_query.brok[b:BROK].dset[d:DSET],
        direct_query.prov[p:PROV] } ->
      ( { prov_result.prov[ans_p:PROV],
        broker_result.prov[brok_p:PROV] }
        -> { visualise, store_query } -> Consumer
      | { no_store, no_match } -> Consumer ) ).

ConsumerIF =
   ( consumer[c:CONS].transparent_query.brok[b:BROK].dset[d:DSET] ->
      broker[b].transparent_query.cons[c].dset[d] -> ConsumerIF
   | consumer[c:CONS].availability_query.brok[b:BROK].dset[d:DSET] ->
      broker[b].availability_query.cons[c].dset[d] -> ConsumerIF
   | consumer[c:CONS].direct_query.prov[p:PROV] ->
      provider[p].direct_query.cons[c] -> ConsumerIF
   | provider[p:PROV].prov_result.cons[c:CONS] ->
      consumer[c].prov_result.prov[p] -> ConsumerIF
   | broker[b:BROK].broker_result.cons[c:CONS].prov[p:PROV] ->
      consumer[c].broker_result.prov[p] -> ConsumerIF
   | broker[b:BROK].no_store.cons[c:CONS] ->
      consumer[c].no_store -> ConsumerIF
   | provider[p:PROV].no_match.cons[c:CONS] ->
      consumer[c].no_match -> ConsumerIF ).


CatalogueItem = CatalogueItem[0],
CatalogueItem[b:BOOL] =
   ( read_item[b] -> CatalogueItem[b]
   | set_item -> CatalogueItem[1] ).
```

```
||CatalogueDset = ( dset[d:DSET]:CatalogueItem ).
||Catalogue = ( prov[p:PROV]:CatalogueDset ).

Broker =
   ( notify.prov[p:PROV].dset[d:DSET] ->
      prov[p].dset[d].set_item -> Broker
   | transparent_query.cons[c:CONS].dset[d:DSET] ->
      SearchCatalogue[c][1][d][1][MAXBROK]
   | fwd_available_ack.cons[c:CONS].dset[d:DSET].fwd_co[f_c:2..MAXBROK] ->
      SearchCatalogue[c][1][d][0][f_c]
   | availability_query.cons[c:CONS].dset[d:DSET] ->
      SearchCatalogue[c][1][d][0][1] ),
SearchCatalogue[c:CONS][p:PROV][d:DSET][to_fwd:BOOL][f_c:BROK] =
   ( prov[p].dset[d].read_item[is_set:BOOL] ->
      if ( is_set ) then
         ( when ( ! to_fwd )
            broker_result.cons[c].prov[p] -> Broker
         | when ( to_fwd )
            routed_query.prov[p].cons[c] ->
            ( start_work.prov[p] -> Broker
            | doing_work.prov[p] ->
               ( when ( p == MAXPROV )
                  no_store.cons[c] -> Broker
               | when ( p < MAXPROV )
                  skip.prov[p] -> SearchCatalogue[c][p+1][d][1][f_c] ) ) )
      else
         ( when ( p == MAXPROV )
            fwd_avail_req.cons[c].dset[d].fwd_co[f_c] -> Broker
         | when ( p < MAXPROV )
            skip.prov[p] -> SearchCatalogue[c][p+1][d][to_fwd][f_c] ) ).

BrokerCoop =
   ( broker[b:BROK].fwd_avail_req.cons[c:CONS].dset[d:DSET].fwd_co[f_c:BROK] ->
      ( when ( f_c == MAXBROK )
         broker[b].fwd_available_nack.cons[c].dset[d].fwd_co[f_c] ->
         broker[b].no_store.cons[c] -> BrokerCoop
      | when ( f_c < MAXBROK && b < MAXBROK )
         broker[b + 1].fwd_available_ack.cons[c].dset[d].fwd_co[f_c + 1] ->
         BrokerCoop
      | when ( f_c < MAXBROK && b == MAXBROK )
         broker[1].fwd_available_ack.cons[c].dset[d].fwd_co[f_c + 1] ->
         BrokerCoop ) ).

Provider =
   ( start_work.brok[b:BROK].cons[c:CONS] -> do_work ->
      { prov_result.cons[c], no_match.cons[c] } -> Provider ).

ProviderGate = ProviderGate[0],
ProviderGate[busy:BOOL] =
   ( routed_query.brok[b:BROK].cons[c:CONS] ->
      ( when ( ! busy )
         start_work.brok[b].cons[c] -> ProviderGate[1]
      | when ( busy )
         doing_work.brok[b] -> ProviderGate[1] )
   | prov_result.cons[c:CONS] -> ProviderGate[0] ).

ProviderIF =
   ( broker[b:BROK].routed_query.prov[p:PROV].cons[c:CONS] ->
      provider[p].routed_query.brok[b].cons[c] -> ProviderIF
   | provider[p:PROV].start_work.brok[b:BROK].cons[c:CONS] ->
      broker[b].start_work.prov[p] -> ProviderIF
   | provider[p:PROV].doing_work.brok[b:BROK] ->
      broker[b].doing_work.prov[p] -> ProviderIF
   | provider[p:PROV].notify.brok[b:BROK].dset[d:DSET] ->
      broker[b].notify.prov[p].dset[d] -> ProviderIF ).

||System =
   ( consumer[c:CONS]:Consumer
   || broker[b:BROK]:Catalogue || broker[b:BROK]:Broker
   || provider[p:PROV]:Provider || provider[p:PROV]:ProviderGate
   || ConsumerIF || BrokerCoop || ProviderIF ).
```

I.2. Architectural components' association to events

Process	Event	Component
Consumer	consumer[1..2].input_query	consumer. request-management. request-specification-manager
	consumer[1..2].read_stored_query	consumer. data-management. repository-manager
	consumer[1..2].store_query	
	consumer[1..2].visualise	consumer. data-visualisation
Consumer, ConsumerIF	consumer[1..2].availability_query. ref_brok[1..2].ref_dset[1..3]	consumer. external-interaction. broker-interaction-manager & consumer. request-management. request-execution-manager
	consumer[1..2].transparent_query. ref_brok[1..2].ref_dset[1..3]	
	consumer[1..2].broker_result. ref_prov[1..2]	
	consumer[1..2].direct_query. ref_prov[1..2]	
	consumer[1..2].prov_result. ref_prov[1..2]	
	consumer[1..2].no_match	
	consumer[1..2].no_store	
Broker, ConsumerIF	broker[1..2].availability_query. ref_cons[1..2].ref_dset[1..3]	broker. external-interaction. consumer-interaction-manager & broker. information-retrieval. query-manager
	broker[1..2].transparent_query. ref_cons[1..2].ref_dset[1..3]	
	broker[1..2].broker_result. ref_cons[1..2].ref_prov[1..2]	
Broker, BrokerCoop	broker[1..2].fwd_available_req. *(+suffix)*	broker. external-interaction. broker-coordination-manager & broker. information-retrieval. query-manager & broker. external-interaction. consumer-interaction-manager
	broker[1..2].fwd_available_ack. *(+suffix)*	
	broker[1..2].fwd_available_nack. *(+suffix)*	
	(suffix:) .ref_cons[1..2].ref_dset[1..3].fwd_co[1..2]	
ConsumerIF	broker[1..2].no_store. ref_cons[1..2]	
Broker, CatalogItem	broker[1..2].ref_prov[1..2]. ref_dset[1..3].set_item	broker. information-retrieval. ir-engine & broker. metadata-management. metadata-access-manager & broker. metadata-management. dbms
	broker[1..2].ref_prov[1..2]. ref_dset[1..3].read_item[0..1]	
Broker	broker[1..2].skip. ref_prov[1..2]	
Broker, ProviderIF	broker[1..2].start_work. ref_prov[1..2]	broker. information-retrieval. query-manager & broker. external-interaction. provider-interaction-manager
	broker[1..2].doing_work. ref_prov[1..2]	
	broker[1..2].routed_query. ref_prov[1..2].ref_cons[1..2]	
	broker[1..2].notify. ref_prov[1..2].ref_dset[1..2]	broker. metadata-management. dbms
ProviderIF	provider[1..2].notify. ref_brok[1..2].ref_dset[1..2]	
ProviderIF, ProviderGate	provider[1..2].routed_query. ref_brok[1..2].ref_cons[1..2]	provider. external-interaction. broker-interaction-manager
Provider, ProviderGate	provider[1..2].start_work. ref_brok[1..2].ref_cons[1..2]	provider. data-management. data-presentation-manager & provider. data-management. repository-connector
ProviderGate	provider[1..2].doing_work. ref_brok[1..2]	
Provider	provider[1..2].do_work	
Provider, ProviderGate, ConsumerIF	provider[1..2].direct_query. ref_cons[1..2]	provider. external-interaction. consumer-interaction-manager & provider. data-management. data-presentation-manager
	provider[1..2].prov_result. ref_cons[1..2]	
	provider[1..2].no_match. ref_cons[1..2]	

Appendix J. Modelling component interaction

J.1. Events

The initial model of interaction between the 3 EGSO roles via the broker has no concurrency or stored state. A slave broker that only serves the master broker demonstrates broker peer interaction. The model captures all events in the message sequence charts of the EGSO broker design. Internal 'tau' events are added for hidden work (their names are preceded by 't'); these include message sequence initiation and termination (direction is not otherwise apparent for synchronous events in the language).

The consumer process connects and signs itself with a broker, which may give unified observation catalogue (resource metadata) updates and results from old consumer queries before accepting queries. It may then initiate a data query (then wait, unblocked) and get a result. It may also initiate a query about the status of data queries in progress, and issue commands to remove one or all of them. It may initiate its own disconnection or be disconnected on a timeout signal from a broker. Sub-processes represent state: CO offline, CG connecting, CC connected (ready to query), and CQ query in progress.

The provider and slave-broker processes also initiate connection (with signature exchange) and disconnection events. The broker accepts metadata, statistics and provider connection status updates at its connection. The provider may initiate a query on resource usage statistics. Otherwise, these processes serve the broker; the provider actions consumer data queries and broker metadata queries. The broker accepts data on consumer queries, provider connection status updates, metadata updates and statistics updates. Both respond to liveness polls. Their sub-processes are PO and BO for offline, and PC and BC for connected.

The master broker process BM therefore serves other roles' connections and queries, determining what information to return and forward as a consequence. It also initiates: metadata queries to providers (which update its UOC), consumer disconnection on timeout, and liveness polls. Its sub-processes that represent state are: BA unconnected alone in the system, BP with provider connected, BB with slave broker peer connected, BG with connecting consumer, BW wholly connected to all other processes, and BQ with a consumer query in progress.

Only one sensible connection (and disconnection) sequence is assumed to avoid a combinatorial explosion. Likewise it is also assumed that only one query is in progress at once (making the remove all queries in progress equivalent to remove one identified query). Also, though the consumer query is non-blocking, other queries (notably the statistics query) will block progress on theoretically possible concurrent tasks in this model. Note that this model is still very prone to stopping as processes needed for service can disconnect without error traps failing the service request, and, as each role acts as both client and server, pairs of processes may make requests to each other and block.

```
/* file: simple_subprocess.lts author: J Lewis-Bowen updated: 2003.06.12
 * content: FSP code for EGSO broker interaction, single role instances */

CO = ( tc_bgn -> c_sign -> c_sign_ack -> set_sid -> CG ),
```

```
   CG = ( tcc_stor -> CC
      | { c_uoc_upd, c_res } -> CG ),
   CC = ( tc_ask -> c_qry -> set_qid -> tc_wait -> CQ
      | c_tout -> tc_halt -> CO
      | tc_end -> c_dcon -> CO ),
   CQ = ( c_res -> CC
      | tc_chk -> c_prog -> { pend, wait } -> tc_chkd -> CQ
      | tc_cmd -> { rm_qid, rm_all } -> CC ).

BA = ( b_sign -> b_sign_ack -> tbm_tchk -> mcat_upd ->
         stat_upd -> dscat_xfr -> tbm_blog -> BB ),
   BB = ( p_sign -> p_sign_ack -> tbm_plog -> dscat_upd -> BP
      | b_dcon -> tbm_dblog -> BA ),
   BP = ( c_sign -> c_sign_ack -> set_sid ->
         fwd_sid -> tbm_clog -> BG
      | p_dcon -> tbm_dplog -> BB ),
   BG = ( tbc_done -> BW
      | tbc_send -> { c_uoc_upd, c_res } -> BG ),
   BW = ( rsrc_qry -> tbm_stat -> rsrc_res -> BW
      | tbm_ask -> uoc_qry -> uoc_res -> tbm_uoc -> b_uoc_upd -> BW
      | tbm_ping -> { b_live, p_live } -> tbm_pung -> BW
      | tbm_tout -> c_tout -> BB
      | c_qry -> tbm_qlog -> set_qid -> fwd_stat -> p_qry -> BQ
      | c_dcon -> tbm_dclog -> BB ),
   BQ = ( p_res -> tbm_stor -> BQ
      | tbm_send -> c_res -> BW
      | c_prog -> tbm_cchk -> { pend, wait } -> BQ
      | { rm_qid, rm_all } -> tbm_act -> BW ).

BO = ( tbs_bgn -> b_sign -> b_sign_ack ->
       mcat_upd -> stat_upd -> dscat_xfr -> tbs_blog -> BC ),
   BC = ( fwd_sid -> tbs_clog -> BC
      | dscat_upd -> tbs_plog -> BC
      | fwd_stat -> tbs_stat -> BC
      | b_uoc_upd -> tbs_uoc -> BC
      | b_live -> BC
      | tbs_end -> b_dcon -> BO ).

PO = ( tp_bgn -> p_sign -> p_sign_ack -> PC ),
    PC = ( p_qry -> tp_data -> p_res -> PC
      | uoc_qry -> tp_mdta -> uoc_res -> PC
      | tp_ask -> rsrc_qry -> rsrc_res -> tp_stat -> PC
      | p_live -> PC
      | tp_end -> p_dcon -> PO ).
```

J.2. Interaction

Based closely on the event of the previous model and broker design, here genuine safe concurrency is implemented for multiple instances of each role. Multiple instances mean that events must be indexed by the entity they apply to, and synonyms must match entity prefixed to indexed events. Entity state is represented by an index on the process instead of the sub-process of the previous model, though each entity may still only have one task in progress at any time. Disconnected states are not represented. Query queues like those implemented in previous models are also not present. Both would cause greater complexity, but would not represent any conceptually deeper challenges. Safety is ensured by a semaphore for each broker, which must also be claimed by a broker when it acts as a client (and cannot serve other entities). Broker to broker interaction is represented by multiple instances of the same process and an interface (that translates events and guards against deadlock when a broker forwards events to itself). Progress is represented by consumer queries being resolved (allowing LTSA to

prove livelock is also avoided). This model is therefore a far more realistic implementation than the previous model, and can be used to test the strength of the design.

The constant state codes represent: CC when a consumer is connected and idle, CQ when a consumer has sent a query, BW when a broker is connected and ready for work, BP when a broker has accepted a query which is pending assignation to a provider, BQ when a broker has a query in progress that is being resolved by a provider, BR when a broker has a response to a query ready to return, PW when a provider is ready for work, and PR when a provider has a response to return to a broker.

The event names are composed in two parts. The prefix is: 't' for hidden events, 'o' for interaction events specified in the original design, and 'a' for interaction events added during modelling. The next one or two letters represent the entities involved: 'c' for consumer, 'p' for provider, 'b' for broker, 'l' for local broker in broker to broker interaction, and 'r' for remote broker in broker to broker interaction. The second part of the event is composed of the following abbreviations: 'c' for connect, 'd' for disconnect, 'q' for query, 'r' for result, 'a' for acknowledge, 'f' for forward, 'e' for error, 'b' for busy, 's' for store, 'w' for work, 'k' for kill, 'v' for receive, 'sid' for session identity, 'uoc' for unified observing catalogue, 'to' for time-out, 'mcu' for meta-catalogue update, 'stu' for statistic update, 'dsc' for data and services catalogue, 'dscu' for DSC update, 'st' for query statistic, 'id' for query identity, 'ign' for ignore lost response, 'pro' for progress, 'pend' for pending in broker, 'wait' for waiting on provider, 'neit' for neither of the previous two statuses, 'kid' for kill on identity, 'kal' for kill all, 'liv' for liveness poll, 'rus' for resource usage statistics.

```
/* file: cpb_state_g.lts author: J Lewis-Bowen updated: 2003.06.17
 * content: FSP code for EGSO broker msg, stateful roles for multi-instance */

const N_CON = 2
range CSET = 1..N_CON
const N_BROK = 2
range BSET = 1..N_BROK
const N_PROV = 2
range PSET = 1..N_PROV
const CC = 0
const CQ = 1
range CSTATE = CC..CQ
const BW = 0
const BP = 1
const BQ = 2
const BR = 3
range BSTATE = BW..BR
const PW = 0
const PR = 1
range PSTATE = PW..PR

SB = SB[0],
SB[b:0..1] = ( sb_d -> SB[0]
   | [x:{c[c:CSET],bw,p[p:PSET]}].sb_c ->
      ( when ( b ) [x].sb_b -> SB[b]
      | when ( ! b ) [x].sb_a -> SB[1] ) ).

C = C[CC], C[cs:CSTATE] =
   ( when ( cs == CC ) tc_q -> sb_c.b[b:BSET] ->
      ( sb_b.b[b] -> C[CC]
      | sb_a.b[b] -> ocb_q.b[b] ->
         ( obc_qid.b[b] -> sb_d.b[b] -> tc_sqid -> C[CQ]
         | abc_qb.b[b] -> sb_d.b[b] -> tc_qb -> C[CC] ) )
   | when ( cs == CQ ) obc_r.b[b:BSET] -> tc_rs -> C[CC]
   | when ( cs == CQ ) tc_qk -> sb_c.b[b:BSET] ->
      ( sb_b.b[b] -> C[CQ]
      | sb_a.b[b] -> { ocb_qkid.b[b], ocb_qkal.b[b] } -> sb_d.b[b] -> C[CC] )
```

```
   | tc_qpro -> sb_c.b[b:BSET] ->
      ( sb_b.b[b] -> C[cs]
      | sb_a.b[b] -> ocb_qpro.b[b] ->
         { obc_pend.b[b], obc_wait.b[b], abc_neit.b[b] } ->
         sb_d.b[b] -> tc_spro -> C[cs] ) ).

B = B[BW][0][0], B[bs:BSTATE][cq:0..N_CON][bq:0..N_BROK] =
   ( ocb_q.c[nc:CSET] ->
      if ( bs == BW && cq == 0 && bq == 0 ) then
         ( tb_sst -> obc_qid.c[nc] -> olr_fst.rb[rb:BSET] ->
            tb_qs -> B[BP][nc][bq] )
      else ( abc_qb.c[nc] -> B[bs][cq][bq] )
   | olr_vst.lb[lb:BSET] -> tr_sst -> B[bs][cq][bq]
   | when ( bs == BP ) tl_q -> sb_c ->
      ( sb_b -> B[BP][cq][bq]
      | sb_a ->
         ( obp_q.pr[p:PSET] ->
            ( apb_qa.pr[p] -> sb_d -> B[BQ][cq][bq]
            | apb_qb.pr[p] -> sb_d -> rb_qb -> B[BP][cq][bq] )
         | olr_q.rb[rb:BSET] ->
            ( arl_qav.rb[rb] -> sb_d -> B[BQ][cq][bq]
            | { tl_qe, arl_qbv.rb[rb] } ->
               sb_d -> tl_qbe -> B[BP][cq][bq] ) ) )
   | olr_qv.lb[nlb:BSET] ->
      ( tl_qve -> B[bs][cq][bq]
      | when ( bs == BW && cq == 0 && bq == 0 )
         arl_qa.lb[nlb] -> tr_qs -> B[BP][cq][nlb]
      | when ( bs != BW || cq != 0 || bq != 0 )
         arl_qb.lb[nlb] -> B[bs][cq][bq] )
   | when ( bs == BQ ) opd_r.pr[p:PSET] -> tb_rs -> B[BR][cq][bq]
   | when ( bs == BQ ) orl_rv.b[rb:BSET] -> tb_rfs -> B[BR][cq][bq]
   | when ( bs == BR ) tb_r -> sb_c ->
      ( sb_b -> B[BP][cq][bq]
      | sb_a ->
         ( when ( bq != 0 ) orl_r.lb[bq] -> sb_d -> B[BW][cq][0]
         | when ( cq != 0 ) obc_r.c[cq] -> sb_d -> B[BW][0][bq] ) )
   | opb_qrus.pr[p:PSET] -> tb_wrus -> obp_rrus.pr[p] -> B[bs][cq][bq]
   | tb_qliv -> { olr_qliv.rb[rb:BSET], obp_qliv.pr[p:PSET] } ->
      tb_sliv -> B[bs][cq][bq]
   | tb_quoc -> sb_c ->
      ( sb_b -> B[BP][cq][bq]
      | sb_a -> obp_quoc.pr[p:PSET] -> opb_ruoc.pr[p] -> sb_d -> tb_suoc
         -> olr_fuoc.rb[rb:BSET] -> B[bs][cq][bq] )
   | olr_vuoc.lb[lb:BSET] -> tr_suoc -> B[bs][cq][bq]
   | ocb_qpro.c[c:CSET] -> tb_wpro ->
      if ( bs == BP && c == cq ) then ( obc_pend.c[c] -> B[bs][cq][bq] )
      else if ( bs == BQ && c == cq ) then ( obc_wait.c[c] -> B[bs][cq][bq] )
      else ( abc_neit.c[c] -> B[bs][cq][bq] )
   | { ocb_qkid.c[c:CSET], ocb_qkal.c[c:CSET] } -> tb_qk -> B[BW][0][0]
   | when ( bs != BQ ) { opd_r.pr[p:PSET], orl_rv.rb[rb:BSET] } ->
      b_rign -> B[bs][cq][bq] ).

PC = PC[PW][0], PC[ps:PSTATE][bq:0..N_BROK] =
   ( obp_q.b[nb:BSET] ->
      if ( ps == PW && bq == 0 ) then ( apb_qa.b[nb] -> tp_qs -> PC[PR][nb] )
      else ( apb_qb.b[nb] -> PC[ps][bq] )
   | when ( ps == PR && bq != 0 ) tp_qw -> sb_c.b[bq] ->
      ( sb_b.b[bq] -> PC[ps][bq]
      | sb_a.b[bq] -> opd_r.b[bq] -> sb_d.b[bq] -> PC[PW][0] )
   | obp_quoc.b[b:BSET] -> tp_wuoc -> opb_ruoc.b[b] -> PC[ps][bq]
   | tp_qrus -> sb_c.b[b:BSET] ->
      ( sb_b.b[b] -> PC[ps][bq]
      | sb_a.b[b] -> opb_qrus.b[b] -> obp_rrus.b[b] ->
         sb_d.b[b] -> tp_srus -> PC[ps][bq] )
   | obp_qliv.b[b:BSET] -> PC[ps][bq] ).

BBIF = ( b[lb:BSET].olr_fst.rb[rb:BSET] -> b[rb].olr_vst.lb[lb] -> BBIF
   | b[lb:BSET].olr_fuoc.rb[rb:BSET] -> b[rb].olr_vuoc.lb[lb] -> BBIF
   | b[lb:BSET].olr_qv.rb[rb:BSET] -> ( when ( lb == rb ) tl_qve -> BBIF )
   | b[lb:BSET].olr_q.rb[rb:BSET] ->
      if ( lb != rb ) then ( b[rb].olr_qv.lb[lb] -> BBIF )
      else ( b[lb].tl_qe -> BBIF )
   | b[rb:BSET].arl_qa.lb[lb:BSET] -> b[lb].arl_qav.rb[rb] -> BBIF
   | b[rb:BSET].arl_qb.lb[lb:BSET] -> b[lb].arl_qbv.rb[rb] -> BBIF
```

```
   | b[rb:BSET].orl_r.lb[lb:BSET] -> b[lb].orl_rv.rb[rb] -> BBIF ).

||SYS = ( c[CSET]:C || b[BSET]:B || sb[BSET]:SB || p[PSET]:PC || BBIF )
/{    b[b:BSET].sb_c /sb[b].bw.sb_c,                  // SEMAPHORE - broker's own
   b[b:BSET].sb_b /sb[b].bw.sb_b,
   b[b:BSET].sb_a /sb[b].bw.sb_a,
   b[b:BSET].sb_d /sb[b].sb_d,
   c[c:CSET].sb_c.b[b:BSET] /sb[b].c[c].sb_c,       //           - consumer
   c[c:CSET].sb_b.b[b:BSET] /sb[b].c[c].sb_b,
   c[c:CSET].sb_a.b[b:BSET] /sb[b].c[c].sb_a,
   c[c:CSET].sb_d.b[b:BSET] /sb[b].sb_d,
   p[p:PSET].sb_c.b[b:BSET] /sb[b].p[p].sb_c,       //           - provider
   p[p:PSET].sb_b.b[b:BSET] /sb[b].p[p].sb_b,
   p[p:PSET].sb_a.b[b:BSET] /sb[b].p[p].sb_a,
   p[p:PSET].sb_d.b[b:BSET] /sb[b].sb_d,
   c[c:CSET].ocb_q.b[b:BSET] /b[b].ocb_q.c[c],       // I/F - consumer-broker
   b[b:BSET].obc_qid.c[c:CSET] /c[c].obc_qid.b[b],
   b[b:BSET].abc_qb.c[c:CSET] /c[c].abc_qb.b[b],
   b[b:BSET].obc_r.c[c:CSET] /c[c].obc_r.b[b],
   c[c:CSET].ocb_qkid.b[b:BSET] /b[b].ocb_qkid.c[c],
   c[c:CSET].ocb_qkal.b[b:BSET] /b[b].ocb_qkal.c[c],
   c[c:CSET].ocb_qpro.b[b:BSET] /b[b].ocb_qpro.c[c],
   b[b:BSET].obc_pend.c[c:CSET] /c[c].obc_pend.b[b],
   b[b:BSET].obc_wait.c[c:CSET] /c[c].obc_wait.b[b],
   b[b:BSET].abc_neit.c[c:CSET] /c[c].abc_neit.b[b],
   b[b:BSET].obp_q.pr[p:PSET] /p[p].obp_q.b[b],    // I/F - provider-broker
   b[b:BSET].apb_qa.pr[p:PSET] /p[p].apb_qa.b[b],
   b[b:BSET].apb_qb.pr[p:PSET] /p[p].apb_qb.b[b],
   p[p:PSET].opd_r.b[b:BSET] /b[b].opd_r.pr[p],
   b[b:BSET].obp_quoc.pr[p:PSET] /p[p].obp_quoc.b[b],
   p[p:PSET].opb_ruoc.b[b:BSET] /b[b].opb_ruoc.pr[p],
   p[p:PSET].opb_qrus.b[b:BSET] /b[b].opb_qrus.pr[p],
   b[b:BSET].obp_rrus.pr[p:PSET] /p[p].obp_rrus.b[b],
   b[b:BSET].obp_qliv.pr[p:PSET] /p[p].obp_qliv.b[b], // broker peer live ping
   b[lb:BSET].olr_qliv.rb[rb:BSET] /b[rb].olr_qliv.lb[lb] }.

progress CRES = { c[c:CSET].tc_rs }
```

J.3. Contention

A simpler model demonstrates the complexity of peer-to-peer semaphore locking. Each entity must first raise a semaphore for itself (so indicating it no longer serves), then raise the semaphore on the entity it wants service from. An entity knows it is being asked for service when its semaphore is raised. This model does not require a centralised supervisor interface to avoid an entity requesting service from itself - when an entity attempts this, it finds the semaphore already raised and backs down. This pattern may be used to replace the centralised broker to broker interaction control in the previous model.

```
/* file: peer_node_f.lts author: J Lewis-Bowen updated: 2003.06.23
 * content: FSP code for safe service by multiple EGSO brokers */

const READY=1
const WORK=2
const ASKED=3
range STATE=READY..ASKED
const N_NODE=3
range NSET=1..N_NODE

NODE = NODE[READY][0], NODE[s:STATE][old_n:0..N_NODE] =
   ( raise.from_n[new_n:NSET] ->
      ( submit.orig_n[new_n] ->
         if ( s == READY && old_n == 0 ) then
            ( accept.orig_n[new_n] -> NODE[WORK][new_n] )
         else ( busy.orig_n[new_n] -> NODE[s][old_n] )
```

```
      | reply.remote_n[new_n] ->
          if ( s == ASKED ) then ( got_result -> NODE[READY][0] )
          else ( spurious_in.[s] -> NODE[s][old_n] ) )
   | when ( s == WORK && old_n != 0 ) request_out ->
      ( success_out -> do_work -> request.for_n[old_n] ->
          ( success.for_n[old_n] -> reply.orig_n[old_n] ->
              release.for_n[old_n] -> release_out -> NODE[READY][0]
          | fail.for_n[old_n] -> release_out -> NODE[s][old_n] )
      | fail_out ->  NODE[s][old_n] )
   | when ( s == READY && old_n == 0 ) request_out ->
      ( success_out -> start_job -> request.for_n[new_n:NSET] ->
          ( success.for_n[new_n] -> submit.remote_n[new_n] ->
              ( accept.remote_n[new_n] ->
                  release.for_n[new_n] -> release_out -> NODE[ASKED][new_n]
              | busy.remote_n[new_n] ->
                  release.for_n[new_n] -> release_out -> NODE[s][old_n] )
          | fail.for_n[new_n] -> release_out -> NODE[s][old_n] )
      | fail_out ->  NODE[s][old_n] ) ).

SEMA = SEMA[0], SEMA[n:0..N_NODE] =
   ( when ( n > 0 ) lower.from_n[n] -> SEMA[0]
   | ask.from_n[new_n:NSET] ->
      if ( n == 0 ) then ( raise.from_n[new_n] -> SEMA[new_n] )
      else ( refuse.from_n[new_n] -> SEMA[n] ) ).

||SYSTEM = ( node[n:NSET]:SEMA || node[n:NSET]:NODE )
/{    node[by:NSET].request.for_n[for:NSET] /node[for].ask.from_n[by],
   node[by:NSET].success.for_n[for:NSET] /node[for].raise.from_n[by],
   node[by:NSET].fail.for_n[for:NSET] /node[for].refuse.from_n[by],
   node[by:NSET].release.for_n[for:NSET] /node[for].lower.from_n[by],
   node[by:NSET].request_out /node[by].ask.from_n[by],
   node[by:NSET].success_out /node[by].raise.from_n[by],
   node[by:NSET].fail_out /node[by].refuse.from_n[by],
   node[by:NSET].release_out /node[by].lower.from_n[by],
   node[n1:NSET].submit.remote_n[n2:NSET] /node[n2].submit.orig_n[n1],
   node[n2:NSET].accept.orig_n[n1:NSET] /node[n1].accept.remote_n[n2],
   node[n2:NSET].busy.orig_n[n1:NSET] /node[n1].busy.remote_n[n2],
   node[n2:NSET].reply.orig_n[n1:NSET] /node[n1].reply.remote_n[n2] }.
```

Appendix K. AstroGrid object interaction models

K.1. State

As at the previous stage, the interaction is initially modelled for a single instance of each designed process. The processes and events are copied directly from the objects and messages of the design's message sequence charts. The model therefore just has single user in the user domain, and only allows a single query to be tracked and resolved by the job management and dataset objects. The processes are: UI for the user interface, UP for the user portal, UR for the user domain registry that would hold dataset metadata, US for the user domain storage space manager that would store query results, UQ for the user message queue that holds notification of completed queries, J for the job entry subsystem, JC for the JES job controller that accepts query workflows, JD for the JES job scheduler that dispatches tasks, JM for the JES job manager that determines subsequent tasks in a workflow, and DA for the storage repository's data agent that actions queries against a dataset. The process representing the job entry subsystem is used to maintain the query state (idle or busy) across its three internal processes. The user message queue also holds state, moving from empty to having a new message when a workflow completes.

A query makes progress through the system via synchronous (stateful) interaction about the user portal (where queries are acknowledged) and asynchronous (fire and forget) interaction through subsequent query events. Each query may take several tasks; therefore after the data agent resolves a dispatched task (recording the result in the user domain storage space), the monitor may submit another or indicate to the user message queue that the query is complete.

The event naming convention is similar to the main model of the previous stage, where the 't' prefix represents hidden events and other letters represent the entities involved. Event suffix code letters used are: 'a' for acknowledge, 'b' for busy, 'd' for done, 'e' for error, 'q' for query, 'r' for result, 's' for submit, 't' for task, 'w' for workflow, 'x' for next. The same codes are reused in the next model.

```
/* file: ag_1b.lts author: J Lewis-Bowen updated: 2003.08.06
 * content: FSP code for AstroGrid job lifecycle design */

const JI = 0
const JB = 1
range JSTATE = JI..JB

const UQE = 0
const UQN = 1
range UQSTATE = UQE..UQN

UI = ( tui_q -> uiup_q ->
       ( upui_eq  -> tui_eq -> UI
       | upui_aq -> tui_aq -> uiup_sw -> UI )
   | tui_r -> uiuq_q[uqs:UQSTATE] ->
       if ( uqs == UQE ) then
          ( tui_re -> UI )
       else if ( uqs == UQN ) then
          ( tui_ra -> uius_q -> usui_a -> tui_qa -> UI ) ).
```

```
UP = ( uiup_q -> tup_q -> upur_q ->
       ( urup_eq -> tup_eq -> upui_eq -> UP
       | urup_aq -> tup_aq -> upui_aq -> UP )
   | uiup_sw -> tup_sw -> upjc_sw -> { jcup_aw, jcup_b } -> UP ).

UR = ( upur_q -> tur_q -> { urup_eq, urup_aq } -> UR ).

US = ( daus_r -> tus_r -> US
   | uius_q -> tus_q -> usui_a -> US ).

UQ = UQ[UQE], UQ[uqs:UQSTATE] =
   ( jmuq_dw -> UQ[UQN]
   | uiuq_q[uqs] -> UQ[UQE] ).

J = J[JI], J[js:JSTATE] =
   ( tj_q[js] -> J[js]
   | when ( js == JI ) tj_aw -> J[JB]
   | when ( js == JB ) tj_dw -> J[JI] ).

JC = ( upjc_sw -> tj_q[js:JSTATE] ->
       if ( js == JI ) then
          ( tj_aw -> jcup_aw -> jcjs_sw -> JC )
       else ( jcup_b -> JC ) ).

JD = ( { jcjs_sw, jmjs_xt } -> tjs_st -> jcda_st -> JD ).

JM = ( dajm_dt -> tjm_dt -> { jmjs_xt, jmuq_dw } -> JM ).

DA = ( jcda_st -> tda_st -> daus_r -> dajm_dt -> DA ).

||SYS = ( UI || UP || UR || US || UQ || J || JC || JD || JM || DA ).
```

K.2. Deadlock

This model adds some concurrent process instances to the previous model, according to the area at highest risk. It therefore demonstrates a deadlock in the triangular dependency of three processes: the job dispatch scheduler, the job monitor and the data agent. It ensures each query workflow has two tasks by using job states (an initial state is also used as a place holder to be assigned a workflow). A workflow factory 'WF' is used by the job scheduler to generate workflows with a unique index (assigned in a sub-process loop, allowing a job for each entity). This model's deadlock is avoided in the design (and the previous model) by the asynchronous interaction - an entity always has the 'choice' of ignoring a message (or, in the previous model, a subsequent workflow task). If synchronous communication were necessary, the deadlock could be also be avoided with semaphores like those used in the previous stage on each leg of the communication triangle.

```
/* file: ag_2b.lts author: J Lewis-Bowen updated: 2003.08.07
 * content: FSP code showing AstroGrid job scheduling circular dependency */

const MAXJD = 2
range JDSET = 1..MAXJD
const MAXJM = 2
range JMSET = 1..MAXJM
const MAXDA = 2
range DASET = 1..MAXDA
range BOOL = 0..1
const JNULL = 0
const JSTART = 1
const JFINAL = 2
range JSTATE = JNULL..JFINAL
const MAXJ = 6
```

```
range JSET = 1..MAXJ

J = J[JNULL], J[js:JSTATE] =
   ( tj_q[js] -> J[js]
   | when ( js == JNULL ) wfjs_swt -> J[JSTART]
   | when ( js == JSTART ) tjm_dt[js] -> J[JFINAL]
   | when ( js == JFINAL ) tjm_dt[js] -> J[JNULL] ).

WF = WF[MAXJ], WF[j:JSET] =
   ( when ( j == MAXJ ) jdwf_sw.jd[jd:JDSET] -> WFL[jd][1]
   | when ( j != MAXJ ) jdwf_sw.jd[jd:JDSET] -> WFL[jd][j+1] ),
WFL[jd:JDSET][j:JSET] =
   ( j[j].tj_q[js:JSTATE] ->
      if ( js == JNULL ) then
         ( jd[jd].wfjd_swt.j[j] -> WF[j] )
      else
         ( when ( j == MAXJ ) twf_se -> WFL[jd][1]
         | when ( j != MAXJ ) twf_se -> WFL[jd][j+1] ) ).

JD = ( jdwf_sw -> wfjd_swt.j[j:JSET] -> jdda_st.da[da:DASET].j[j] -> JD
   | jmjd_xt.j[j:JSET] -> tjd_st.j[j] -> jdda_st.da[da:DASET].j[j] -> JD ).

JM = ( dajm_dt.j[j:JSET] -> tjm_dt.j[j][js:JSTART..JFINAL] ->
      if ( js == JFINAL ) then ( tjm_dw -> JM )
      else ( jmjd_xt.jd[jd:JDSET].j[j] -> JM ) ).

DA = ( jdda_st.j[j:JSET] -> tda_st -> dajm_dt.jm[jm:JMSET].j[j] -> DA ).

||SYS = ( j[JSET]:J || WF || jd[JDSET]:JD || jm[JMSET]:JM || da[DASET]:DA )
/{    jd[jd:JDSET].jdwf_sw /jdwf_sw.jd[jd],
   jd[jd:JDSET].wfjd_swt.j[j:JSET] /j[j].wfjd_swt,
   jd[jd:JDSET].jdda_st.da[da:DASET].j[j:JSET] /da[da].jdda_st.j[j],
   jm[jm:JMSET].tjm_dt.j[j:JSET][js:JSTATE] /j[j].tjm_dt[js],
   jm[jm:JMSET].jmjd_xt.jd[jd:JDSET].j[j:JSET] /jd[jd].jmjd_xt.j[j],
   da[da:DASET].dajm_dt.jm[jm:JMSET].j[j:JSET] /jm[jm].dajm_dt.j[j] }.

progress DW = { jd[jd:JDSET].tjd_dw }
```

Appendix L. Generic connection models

L.1. Stateful connectors and the interaction triangle

3 processes, Q, R and P, use the same connector process to communicate. The process instances are indexed, counting from zero. The connector process type, SC, represents stateful request and response. 4 events are necessary for the client and server side representation of both the query and acknowledgement.

The connector is composed in the system using N by M coupling between each type of pairing. So for 3 nodes in a client role and 2 in a server role, there would be 6 connector instances. In the system configuration given, with 3 sets (prefixed qr, rp and pq) of 2x2 pairings there are therefore 12 connector instances. The connectors are numbered according to the indexes for the nodes at their client and server connection ends; the server role index multiplies the number of clients, and the client index is added. As demonstration, this table shows the connector number associated with a 2 by 2 client-server array - effectively a binary numbering (though the scheme works for arbitrary numbers):

		client	
		0	1
server	0	0	1
	1	2	3

When giving the synonyms for connectors to the process' events, the prefixed process with the index of the node with which it communicates must be given first (defining the range for the variable). The indexed connector prefix is then given mathematically.

The sequence of states for the tasks carried out in a distributed way across the node types is given as a separate process type, named T. Task indexes are reused to avoid an infinite sequence. The number of tasks that can progress is given by the total number of node instances. Tasks in the configuration given, with 2 instances of each node type, are therefore indexed from 1 to 6. Reusing task indices also avoids stopping states for tasks, simplifying genuine deadlock discovery.

As tasks are repeated several times to complete a job, a super-process, named J, is also given. This uses a conditional over the T sub-process to avoid a repetitious encoding of the event sequence after the initial event assigns a task to a Q node. The conditional therefore enforces the constraint that after a task is initially assigned to a Q node, subsequent iterations must begin at the same node; conversely when the task index is reused for another job, indicated by increment wrapping round to zero after the modulo operation, it may be assigned to any Q node.

The event sequence for each task has 3 essential parts - they represent: a query for data or service, a registry look-up for hosts that could satisfy the query, and work on the query at the chosen provider of data or service.

Within each node, work is divided so that task progress can pause and the node perform concurrent actions. Therefore the task pairs represent: look-up 'lk' and assign 'an' for the registry, start work 'wk' and work done 'dn' for the provider, and next task 'nx' and begin task 'bn' for the query agent.

Additional events that are synchronous between nodes are also necessary to represent the transition from one node to the next in the task state. Without this a node could action a task which it should not know off given the communication link events, as the task would not be bound to a node and the node would not be bound to a task index until the next task event happened. The synchronisation events are simply named after the connector: 'qr', 'rp', and 'pq'.

Therefore though each task's iteration represents just 3 actions, 9 events are needed to model its distributed state transitions in a way that allows concurrent progress. Note that the task state transitions are independent of the communication state sequence; the synchronisation events represent information being passed in a message at the application level, whilst the communication events represent lower layers (maybe session or the reliable transport or the basic data-link layer).

Task events must by made synonymous to node events, making the task prefix a node's event suffix. Each synchronisation event have a pair of synonyms. The task index is used by the node to ensure it performs its action on the task just communicated to it. Note that as tasks events are suffixed with the node index, they hold a temporary state until the next event at that node; therefore no extra mechanisms are needed to ensure a node can return to suspended tasks for which it is responsible.

The node processes themselves simply tie the connectors to the task transition sequence. Ideally they, like the aliases, could be automatically generated from the task events; the node responsible for each action is given by the task events' suffixes. Where an event is handed from one node to the next, the exchange must be wrapped in the client and server side communication path events in the process pair transferring the task.

```
/* file: fsp_connect-stereotype-note.txt baseline 5
 * author: J Lewis-Bowen updated: 2003.09.26
 * content: FSP code separating grid connection and task layers */

const QX = 2
range QR = 0..(QX-1)
const RX = 2
range RR = 0..(RX-1)
const PX = 2
range PR = 0..(PX-1)
range QRR = 0..((QX*RX)-1)
range RPR = 0..((RX*PX)-1)
range PQR = 0..((PX*QX)-1)
const IX = 2
range IR = 0..(IX-1)
const TX = PX+QX+RX
range TR = 1..TX

J = J[0][0], J[i:IR][oq:QR] =
      ( when ( i==0 ) bn[i].q[q:QR] -> T[i][q]
      | when ( i!=0 ) bn[i].q[oq] -> T[i][oq] ),

T = T[0][0], T[i:IR][q:QR] =
   ( qr.q[q].r[r:RR] -> lk.r[r] -> an.r[r]
      -> rp.r[r].p[p:PR] -> wk.p[p] -> dn.p[p].q[q]
      -> pq.p[p].q[q] -> nx.q[q] -> J[(i+1)%IX][q] ).
```

```
Q = ( s_q.p[p:PR] -> pq.p[p].t[t:TR] -> s_a.p[p] -> nx.t[t] -> Q
   | bn[i:IR].t[t:TR] -> c_q.r[r:RR] -> qr.r[r].t[t] -> c_a.r[r] -> Q ).

R = ( s_q.q[q:QR] -> qr.q[q].t[t:TR] -> s_a.q[q] -> lk.t[t] -> R
   | an.t[t:TR] -> c_q.p[p:PR] -> rp.p[p].t[t] -> c_a.p[p] -> R ).

P = ( s_q.r[r:QR] -> rp.r[r].t[t:TR] -> s_a.r[r] -> wk.t[t] -> P
   | dn.q[q:QR].t[t:TR] -> c_q.q[q] -> pq.q[q].t[t] -> c_a.q[q] -> P ).

SC = ( c_q -> s_q -> s_a -> c_a -> SC ).

||Sys = ( t[t:TR]:J || q[q:QR]:Q || r[r:RR]:R || p[p:PR]:P
   || qr[qr:QRR]:SC || rp[rp:RPR]:SC || pq[pq:PQR]:SC ) /{
   t[t:TR].bn[i:IR].q[q:QR] /q[q].bn[i].t[t],
   t[t:TR].qr.q[q:QR].r[r:RR] /q[q].qr.r[r].t[t],
   t[t:TR].qr.q[q:QR].r[r:RR] /r[r].qr.q[q].t[t],
   t[t:TR].lk.r[r:RR] /r[r].lk.t[t],
   t[t:TR].an.r[r:RR] /r[r].an.t[t],
   t[t:TR].rp.r[r:RR].p[p:PR] /r[r].rp.p[p].t[t],
   t[t:TR].rp.r[r:RR].p[p:PR] /p[p].rp.r[r].t[t],
   t[t:TR].wk.p[p:PR] /p[p].wk.t[t],
   t[t:TR].dn.p[p:PR].q[q:QR] /p[p].dn.q[q].t[t],
   t[t:TR].pq.p[p:PR].q[q:QR] /p[p].pq.q[q].t[t],
   t[t:TR].pq.p[p:PR].q[q:QR] /q[q].pq.p[p].t[t],
   t[t:TR].nx.q[q:QR] /q[q].nx.t[t],
   q[q:QR].c_q.r[r:RR] /qr[(r*QX)+q].c_q,
   q[q:QR].c_a.r[r:RR] /qr[(r*QX)+q].c_a,
   r[r:RR].s_q.q[q:QR] /qr[(r*QX)+q].s_q,
   r[r:RR].s_a.q[q:QR] /qr[(r*QX)+q].s_a,
   r[r:RR].c_q.p[p:PR] /rp[(p*RX)+r].c_q,
   r[r:RR].c_a.p[p:PR] /rp[(p*RX)+r].c_a,
   p[p:PR].s_q.r[r:RR] /rp[(p*RX)+r].s_q,
   p[p:PR].s_a.r[r:RR] /rp[(p*RX)+r].s_a,
   p[p:PR].c_q.q[q:QR] /pq[(q*PX)+p].c_q,
   p[p:PR].c_a.q[q:QR] /pq[(q*PX)+p].c_a,
   q[q:QR].s_q.p[p:PR] /pq[(q*PX)+p].s_q,
   q[q:QR].s_a.p[p:PR] /pq[(q*PX)+p].s_a }.
```

L.2. Stateless connector

The task lifecycle, in process T, is simpler than the previous example, with bn, begin, and wk, work in progress, events representing its lifespan between 2 components. The client in the stateless connection begins the task, and the server works on it. The stateless connection, process SC, is modelled as moving from c_q, client sends query, to s_q, server receives query, or er, representing the message loss error event. The client and server processes C and S are composed of these events.

The error event is contrived, as it is unlikely to have an analogue in the real implementation of a fire-and-forget protocol like UDP. The stateless server is also modelled as choosing to pick up the message, as LTSA represents the connection message delivery outcome non-deterministically. An improved model may model a server busy state, forcing a forget event on the connection, but this would also introduce events that would have no analogue in implemented operations.

A further refinement to the model, fitting data-grid designs (especially AstroGrid), would be a sweeper daemon that retries lost queries. This may make the model less generic as it is forced to implement a message logging strategy, for example, representing metadata stores of service requests at client and server sides. Such model implementations may run into LTSA's limitations regarding open ended sequences. The modeller also may be required implement

safe concurrent database update; though this is significant to LTSA, it is unlikely to be an architectural level concern of a data-grid engineer.

Also, as the model always allows the error event to occur, eventually all possible task indexes in a finite sequence will be lost, so that the model deadlocks.

```
/* file: fsp_connect-stereotype-note.txt baseline 6
 * author: J Lewis-Bowen updated: 2003.09.26
 * content: FSP code separating grid connection and task layers */

range TR = 1..3

T = ( bn -> wk -> T ).

C = ( bn.t[t:TR] -> c_q.t[t] -> C ).

S = ( s_q.t[t:TR] -> wk.t[t] -> S ).

SC = ( c_q.t[t:TR] -> { s_q.t[t], er.t[t] }-> SC ).

||SYS = ( t[t:TR]:T || C || S || SC ) /{
   t[t:TR].bn /bn.t[t],
   t[t:TR].wk /wk.t[t] }.
```

Appendix M. Stochastic FSP models AstroGrid

M.1. Simple task model

The first non-stochastic model of the AstroGrid job scheduling sub-system is listed below. It is represented just by the states that a task moves through in its lifespan, and the tasks of a provider, which must also to the work of selecting a task when its in an idle state. Integer constants are to make it clearer how events are indexed with task and provider state.

```
// file: sim_ltsa_notes.txt author: J Lewis-Bowen updated: 2004.02.27
// content: FSP model of data-grid tasks.

const T_NEW = 0
const T_READ = 1
const T_DONE = 2
range T_STATE = T_NEW..T_DONE
const T_MAX = 2
range T_RANGE = 1..T_MAX

TASK = TASK[ T_NEW ], TASK[ ts:T_STATE ] =
   ( when ( ts == T_NEW ) read -> TASK[ T_READ ]
   | when ( ts == T_READ ) remove -> TASK[ T_DONE ]
   | when ( ts == T_DONE ) submit -> TASK[ T_NEW ]
   | test[ ts ] -> TASK[ ts ] ).

const P_IDLE = 0
range P_STATE = P_IDLE..T_MAX

PROV = PROV[ P_IDLE ], PROV[ ps:P_STATE ] =
   ( when ( ps == P_IDLE ) task[ t:T_RANGE ].test[ ts:T_STATE ] ->
      ( when ( ts == T_READ ) start -> PROV[ t ]
      | when ( ts != T_READ ) idle -> PROV[ P_IDLE ] )
   | when ( ps != P_IDLE ) task[ ps ].remove -> PROV[ P_IDLE ] ).

||SYS = ( PROV || task[ t:T_RANGE ]:TASK ).
```

M.2. Timed task model

In adding clocks to the above model, it has been simplified further to help debugging. The task transitions of input and done are equivalent to the states T_NEW and T_DONE above. The push and pop events together represent T_READ, separated to avoid presumed errors in LTSA analysing infinitely small event time.

```
// file: sim_ltsa_notes.txt author: J Lewis-Bowen updated: 2004.02.27
// content: FSP timed task simulation.

range TR = 1..3
TASK = (
   input <ct:exp(3)> ->
   ?ct? push ->
   pop <st:exp(1)> ->
   ?st? done -> TASK ).
SVR = (
   task[ t:TR ].pop ->
   task[ t ].done -> SVR ).
timer WAIT { forall[ t:TR ] <task[ t ].push, task[ t ].done> }
||SYS = (
   SVR || task[ t:TR ]:TASK || WAIT ).
```

M.3. AstroGrid job dispatch model

A process for unreliable communication has been added to the simple task model, CHAN. The term job is used instead of task in this model to reflect the AstroGrid convention of activity elements, composed into a task workflow. This has separate event sequence cases for reading query job metadata, resending queries, responding with a job action result message, and resending a that result. Note the consistent abbreviations JM for job metadata and JA for job activity. The DAEMON process tests job metadata to see if either queries or answers have been sent and not received.

```
// file: sim_ag_notes.txt author: J Lewis-Bowen updated: 2004.03.08
// content: FSP for AstroGrid unreliable job messaging and recovery.

abbreviations JM job metadata JA job activity

const JM_NEW = 0
const JM_SENT = 1
const JM_FIN = 2
range JM_STATE = JM_NEW..JM_FIN
const JA_NONE = 0
const JA_READ = 1
const JA_DONE = 2
range JA_STATE = JA_NONE..JA_DONE
const J_MAX = 2
range J_RANGE = 1..J_MAX

JOB_MD = JOB_MD[ JM_NEW ], JOB_MD[ s:JM_STATE ] =
   ( when ( s == JM_NEW ) read -> JOB_MD[ JM_SENT ]
   | when ( s == JM_SENT ) remove -> JOB_MD[ JM_FIN ]
   | when ( s == JM_FIN ) submit -> JOB_MD[ JM_NEW ]
   | jm_test[ s ] -> JOB_MD[ s ] ).

JOB_ACT = JOB_ACT[ JA_NONE ], JOB_ACT[ s:JA_STATE ] =
   ( when ( s == JA_NONE ) ack -> JOB_ACT[ JA_READ ]
   | when ( s == JA_READ ) work -> JOB_ACT[ JA_DONE ]
   | when ( s == JA_DONE ) result -> JOB_ACT[ JA_NONE ]
   | ja_test[ s ] -> JOB_ACT[ s ] ).

CHAN =
   ( j_md[ j:J_RANGE ].read -> query ->
      { j_act[ j ].ack, q_lost } -> CHAN
   | q_resend[ j:J_RANGE ] -> query ->
      { j_act[ j ].ack, q_lost } -> CHAN
   | j_act[ j:J_RANGE ].result -> answer ->
      { j_md[ j ].remove, a_lost } -> CHAN
   | a_resend[ j:J_RANGE ] -> answer ->
      { j_md[ j ].remove, a_lost } -> CHAN ).

DAEMON = (
   j_md[ j:J_RANGE ].jm_test[ jms:JM_STATE ] ->
   j_act[ j ].ja_test[ jas:JA_STATE ] ->
      if ( jms == JM_SENT && jas == JA_NONE )
         then ( { q_resend[ j ], a_resend[ j ] } -> DAEMON )
      else ( allclear -> DAEMON ) ).

||SYS = ( CHAN || DAEMON ||
   j_act[ j:J_RANGE ]:JOB_ACT || j_md[ j:J_RANGE ]:JOB_MD ).
```

M.4. Simulating AstroGrid job dispatch

Clocks are added to the above model for the timing between job event transitions and the daemon's interval between checking for lost jobs. Probability indicators are added to the options for losing jobs on the communication channel. One statistic is collected here, the time between a job being read from the queue and its removal.

```
// file: sim_ag_notes.txt author: J Lewis-Bowen updated: 2004.03.08
// content: Simulation of AstroGrid job scheduling.

const J_NEW = 1
const J_SENT = 2
const J_TODO = 3
const J_DONE = 4
range J_STATE = J_NEW..J_DONE
range J_RANGE = 1..2

JOB = JOB[ J_NEW ], JOB[ js:J_STATE ] =
   ( when ( js == J_NEW ) <? exp(2.0) ?> read -> JOB[ J_SENT ]
   | when ( js == J_SENT ) ack -> JOB[ J_TODO ]
   | when ( js == J_TODO ) <? exp(1.0) ?> work -> JOB[ J_DONE ]
   | when ( js == J_DONE ) remove -> JOB[ J_NEW ]
   | test[ js ] -> JOB[ js ] ).

CHAN =
   ( job[ j:J_RANGE ].read -> query ->
      ( (0.9) ( job[ j ].ack -> CHAN ) | (0.1) ( q_lost -> CHAN ) )
   | redo_q[ j:J_RANGE ] -> query ->
      ( (0.9) ( job[ j ].ack -> CHAN ) | (0.1) ( q_lost -> CHAN ) )
   | job[ j:J_RANGE ].work -> answer ->
      ( (0.9) ( job[ j ].remove -> CHAN ) | (0.1) ( a_lost -> CHAN ) )
   | redo_a[ j:J_RANGE ] -> answer ->
      ( (0.9) ( job[ j ].remove -> CHAN ) | (0.1) ( a_lost -> CHAN ) ) ).

DAEMON =
   ( <? exp(10.0) ?> job[ j:J_RANGE ].test[ js:J_STATE ] ->
      ( when ( js == J_SENT ) redo_q[ j ] -> DAEMON
      | when ( js == J_DONE ) redo_a[ j ] -> DAEMON
      | when ( js != J_SENT && js != J_DONE ) allclear -> DAEMON ) ).

timer T_RESPONSE { forall[ j:J_RANGE ] <job[ j ].read, job[ j ].remove> }

||SYS = ( CHAN || DAEMON || job[ j:J_RANGE ]:JOB || T_REPSONSE ).
```

M.5. AstroGrid task management

In this final attempt to capture the AstroGrid job control sub-system, more processes are added to represent explicitly the task management. The MAN_TASK process represents the task state transitions visible to the scheduler, whilst PROV_TASK captures the transitions exposed to the service provider. Clocks are used to introduce, submit, read and work on tasks. A simplified unreliable communication channel is modelled by the SUBMIT process.

```
// file: ag_naked_task.txt author: J Lewis-Bowen udpated: 2004.05.27
// content: FSP simulation of AstroGrid job submission interface.

const M_NEW = 0
const M_SENT = 1
const M_WAIT = 2
const M_RETURN = 3
range M_STATE = M_NEW..M_RETURN
```

```
MAN_TASK = MAN_TASK[ M_NEW ], MAN_TASK[ ms:M_STATE ] =
   ( when ( ms == M_NEW ) submit_in -> MAN_TASK[ M_SENT ]
   | when ( ms == M_SENT ) check_prov ->
      ( not_seen -> submit_in -> MAN_TASK[ M_SENT ]
      | seen -> MAN_TASK[ M_WAIT ] )
   | when ( ms == M_SENT || ms == M_WAIT ) result_out -> MAN_TASK[ M_RETURN ]
   | when ( ms == M_SENT || ms == M_WAIT ) check_done -> not_replied -> MAN_TASK[ ms ]
   | when ( ms == M_RETURN ) check_done -> done -> MAN_TASK[ M_NEW ] ).

MANAGER = ( <? fixed(10) ?> submit_in -> MAN_WAIT
   | check_done -> done -> MANAGER ),
MAN_WAIT = ( <? fixed(2) ?> check_prov ->
      ( not_seen -> submit_in -> MAN_WAIT
      | seen -> MAN_WAIT )
   | result_out -> MANAGER ).

const P_IDLE = 0
const P_BUSY = 1
const P_DONE = 2
range P_STATE = P_IDLE..P_DONE

PROV_TASK = PROV_TASK[ P_IDLE ], PROV_TASK[ ps:P_STATE ] =
   ( when ( ps == P_IDLE ) submit_out -> PROV_TASK[ P_BUSY ]
   | when ( ps == P_BUSY ) result_in -> PROV_TASK[ P_DONE ]
   | when ( ps == P_BUSY | ps == P_DONE ) check_prov -> seen -> PROV_TASK[ ps ]
   | when ( ps == P_IDLE ) check_prov -> not_seen -> PROV_TASK[ P_IDLE ]
   | when ( ps == P_DONE ) check_done ->
      ( not_replied -> result_in -> PROV_TASK[ P_DONE ]
      | done -> PROV_TASK[ P_IDLE ] ) ).

PROVIDER = ( submit_out -> PROV_BUSY
   | check_prov -> not_seen -> PROVIDER ),
PROV_BUSY = ( <? fixed(5) ?> result_in -> PROV_WAIT
   | check_prov -> seen -> PROV_BUSY ),
PROV_WAIT = ( <? fixed(2) ?> check_done ->
      ( not_replied -> result_in -> PROV_WAIT
      | done -> PROVIDER )
   | check_prov -> seen -> PROV_WAIT ).

SUBMIT = ( submit_in ->
   ( (0.4) ( submit_out -> SUBMIT )
   | (0.6) ( submit_loss -> SUBMIT ) ) ).
REPLY = ( result_in -> { result_out, result_loss } -> REPLY ).

||SYS = ( MAN_TASK || MANAGER || PROV_TASK || PROVIDER || SUBMIT || REPLY ).
```

Appendix N. SimPy models and results

N.1. First EGSO simulation model

The code used for simulation was intended for rapid script development, not quality development of production-line reusable components. The style is of poor quality and must be maintained with attention to traditional coding traps: global variables, unprotected class member variables, hard-coded configuration, side effect updates to objects passed by reference, modification to aggregation data-types being iterated over, etc.

Classes for each EGSO role and its critical message, C_Req, P_Upd and B_All, are derived from the SimPy Process class, so represented consumer, providers, and brokers can queue events for themselves and each other. They call the parent constructor and overwrite the go function that is run when a Process instance is popped off the event queue by SimPy.

The go method for consumers and providers simply contains an infinite loop of interrupting a broker at intervals (when they also report). Their members represent their identity (as a sequential integer passed to the constructor and a reported string), a number in a sequence representing the data they wish to query or publish metadata about, and a pointer to the broker that serves them (passed to the constructor).

The broker has members for its identity, a list of queries it has seen and metadata it knows about, a cursor (for metadata or queries) for its current task, and a list of the peers it is connected to (to whom it can send messages). Each broker instances peers are set after construction - once the complete network is built - with the set_peer method.

The broker's go method is the simulation core, but does not yet represent true EGSO operation. When a broker is interrupted, it must identify what type of process it was interrupted by to decide on a suitable action. Actions in this model are simply passing on consumer queries and provider updates, or just reporting a forwarded message has been received.

The main program initialises instances of each role, uses the SimPy activation function to enqueue them, and starts the simulation (with the SimPy simulation method). The activated consumers and providers will drive the simulation; when they are taken off the event queue they will interrupt brokers, and requeue themselves.

This skeleton simulation is used in the subsequent refined models used for simulation experiments. Features that appear in this code will not be re-described below.

```
# file: cbp_simplest.py author: J Lewis-Bowen updated: 2004.05.24
# content: Simple SimPy simulation of EGSO data-grid architecture.
# NB: Rapid code - bad object member references style, risky recursion.

# preamble
from __future__ import generators
from SimPy.Simulation import *

class C_Req( Process ):
   def __init__( self, id_ix, b_serve ):
      self.id_str = "C_Req_" + str( id_ix )
      Process.__init__( self, name = self.id_str )
      self.id_ix = id_ix
      self.query_ix = 0
      self.b_serve = b_serve
```

```
    def go( self ):
        # send request every 2 sec.
        while 1:
            yield hold, self, 2
            self.query_ix = self.query_ix + 1
            print "%.2f %s request %d to %s" % \
                ( now(), self.id_str, self.query_ix, self.b_serve.id_str )
            self.interrupt( self.b_serve )


class P_Upd( Process ):
    def __init__( self, id_ix, b_serve ):
        self.id_str = "P_Upd_" + str( id_ix )
        Process.__init__( self, name = self.id_str )
        self.id_ix = id_ix
        self.update_ix = 0
        self.b_serve = b_serve

    def go( self ):
        # send update every 4 sec.
        while 1:
            yield hold, self, 3
            self.update_ix = self.update_ix + 1
            print "%.2f %s update %d to %s" % \
                ( now(), self.id_str, self.update_ix, self.b_serve.id_str )
            self.interrupt( self.b_serve )


class B_All( Process ):
    def __init__( self, id_ix ):
        self.id_str = "B_All_" + str( id_ix )
        Process.__init__( self, name = self.id_str )
        self.id_ix = id_ix
        self.query_db = []
        self.mdata_db = []
        self.query_now = None
        self.mdata_now = None
        self.b_peer = []

    def set_peer( self, b_peer ):
        self.b_peer.append( b_peer )

    def go( self ):
        # sleep until interrupted, act on cause.
        # (use hold instead of passivate/reactivate so can identify interruptor)
        while 1:
            yield hold, self, 1000
            if self.interrupted():
                if type( self.interruptCause ) == type( C_Req( 0, self ) ):
                    self.query_now = ( self.interruptCause.id_ix, self.interruptCause.query_ix )
                    self.mdata_now = None
                    print "%.2f %s queried %s:%d" % \
                        ( now(), self.id_str, self.interruptCause.id_str,
                            self.interruptCause.query_ix )
                    self.interrupt( self.b_peer[ 0 ] )
                elif type( self.interruptCause ) == type( P_Upd( 0, self ) ):
                    self.mdata_now = ( self.interruptCause.id_ix, self.interruptCause.update_ix )
                    self.query_now = None
                    print "%.2f %s updated %s:%d" % \
                        ( now(), self.id_str, self.interruptCause.id_str,
                            self.interruptCause.update_ix )
                    self.interrupt( self.b_peer[ 0 ] )
                elif type( self.interruptCause ) == type( B_All( 0 ) ):
                    print "%.2f %s updated %s state query %s update %s" % \
                        ( now(), self.id_str, self.interruptCause.id_str,
                            self.interruptCause.query_now, self.interruptCause.mdata_now )
                # if not-end-of-req-chain: reactivate( self.b_peer )
                else:
                    print "%.2f %s unknown intuerrupt ignored" % ( now(), self.id_str )

initialize()
print "%.2f simulation starts" % now()
b1 = B_All( 1 )
```

```
b2 = B_All( 2 )
b1.set_peer( b2 )
b2.set_peer( b1 )
activate( b1, b1.go() )
activate( b2, b2.go() )
c1 = C_Req( 1, b1 )
p1 = P_Upd( 1, b2 )
activate( c1, c1.go() )
activate( p1, p1.go() )
simulate( until = 10 )
print "%.2f simulation over" % now()
```

N.2. Simulating broker message forwarding

Like the previous model, this simulation contains 3 objects derived from the SimPy Process class. Provider and consumer process instances interrupted broker instances that they have references to with simulated queries and updates (which they count in a member variable) at regular intervals of the simulation clock. A global list representing the metadata published by the providers is used to help consumers pick queries and providers publish unique metadata about the data records they are supposed to have.

Parameters for the simulation are coded as global variables in the script - to be adjusted for different experiments. They represent the size of the network, the relative interval between consumer and provider queries, the number of user messages that the simulation runs for, and the broker forwarding design to be used. The only simulation statistics are the number of user and broker messages, also global variables; their ratio indicates network performance. Either providers or consumers may stop the simulation when enough messages have been sent.

When a simulated broker is activated to receive a message, it determines what type of role it was interrupted by. In any case (except the error trap) it looks at the interrupters member data to see the message content (the metadata to be resolved for a query target or recorded for an update); this poor quality method simplifies the code, avoiding implementation of an intermediate message object or complex interrupt interface arguments. Brokers only take action on queries or updates not seen before, and either type of message is only forwarded to a broker's peers if the current design permits it. Reciprocally, when a broker is interrupted by a peer, the configured design helps it decide the type of message. Queries are resolved when integers representing data are matched in the Broker metadata.

In the main program, multiple instances of brokers then consumers and providers (which are connected to the brokers) are created in separate loops (though all experiments actually use one consumer and provider per broker). Roles are activated as they are created, consumers and providers being initially queued with slightly different start times, with providers acting first to populate the metadata. Brokers are activated to start immediately, but will all yield straight away and become inactive. Once created, brokers must be joined to peers, with a special case for the first two brokers, whose neighbours are the last two in the list (modulo operations on indexes could also have been used for wrap-round).

The simulation is programmed to run for a very long time, but will be stopped when sufficient user messages are sent. In normal experiments, only the final statistics are output.

```
# file: cbp_cheat.py author: J Lewis-Bowen updated: 2004.05.27
# content: Simulating candidate EGSO broker message forwarding strategies.

# SimPy preamble, also import regular Python random library.
from __future__ import generators
from SimPy.Simulation import *
import random
import time
new_seed = int( ( time.time() * 100 ) % 1000000 )
random.seed( new_seed )

# Experiment - ratio user:broker messages for network size (broker), update rate (per user query).
#             network size (.2)    update rate (20)
#             20     100    500    0.2    1      5
#            ----------------------------------------
# fwd_query  | 0.000  0.000  0.000  0.000  0.000  0.000
# fwd_update | 0.000  0.000  0.000  0.000  0.000  0.000

# Parameters: Consumer Provider Broker number, Consumer Provider message interval, run duration,
# broker interaction design ("fwd_query" if don't know all, "fwd_update" if know all).
c_num = 500
p_num = c_num
b_num = c_num
c_sec = 20
p_sec = 100
msg_stop = 1000
design = "fwd_query"

# 2 simple integer measures of message volume.
# class Msg():
#     def __init__( self ):
#         self.user_msg = 0
#         self.broker_msg = 0
user_msg = 0
broker_msg = 0

# Here's the cheat - a global list of all known data records (will the sequence 1,2,3..).
known = []


# Consumer user just requests.
class C_Req( Process ):
    def __init__( self, id_ix, b_serve ):
        self.id_str = "C_Req_" + str( id_ix )
        Process.__init__( self, name = self.id_str )
        self.id_ix = id_ix
        self.query_ix = 0
        self.query_data = 0
        self.b_serve = b_serve

    def go( self ):
        # Send request regularly.
        global user_msg
        while 1:
            yield hold, self, c_sec
            if known != []:
                # As long as data in the grid not empty, current query a random record.
                self.query_ix = self.query_ix + 1
                self.query_data = known[ random.randrange( 0, len( known ) ) ]
                #print "%.2f %s request %d for %d to %s" % ( now(), self.id_str, self.query_ix,
self.query_data, self.b_serve.id_str )
                user_msg = user_msg + 1
                if user_msg > msg_stop:
                    stopSimulation()
                self.interrupt( self.b_serve )


# Provider user just sends updates.
class P_Upd( Process ):
    def __init__( self, id_ix, b_serve ):
        self.id_str = "P_Upd_" + str( id_ix )
        Process.__init__( self, name = self.id_str )
        self.id_ix = id_ix
        # self.update_ix = 0
```

```
        self.update_data = 0
        self.b_serve = b_serve

    def go( self ):
        # Send update regularly.
        global user_msg
        while 1:
            yield hold, self, p_sec
            # This provider's next update is next globally known record (or 1 if first).
            # self.update_ix = self.update_ix + 1
            if known == []:
                self.update_mdata = 1
            else:
                self.update_mdata = known[ len( known ) - 1 ] + 1
            known.append( self.update_mdata )
            #print "%.2f %s update %d to %s" % ( now(), self.id_str, self.update_mdata,
self.b_serve.id_str )
            user_msg = user_msg + 1
            if user_msg > msg_stop:
                stopSimulation()
            self.interrupt( self.b_serve )


# Broker receives and forwards messages from Consumers Providers and Broker peers.
class B_All( Process ):
    def __init__( self, id_ix ):
        self.id_str = "B_All_" + str( id_ix )
        Process.__init__( self, name = self.id_str )
        self.id_ix = id_ix
        self.query_db = []
        self.mdata_db = []
        self.query_now = None
        self.mdata_now = None
        self.data_now = 0
        self.b_peer = []

    def set_peer( self, b_peer ):
        self.b_peer.append( b_peer )

    def go( self ):
        # Sleep until interrupted, act on cause.
        # (Use hold million sec instead of passivate/reactivate so can identify interruptor).
        global broker_msg
        while 1:
            yield hold, self, 1000000
            if self.interrupted():
                if type( self.interruptCause ) == type( C_Req( 0, self ) ):
                    # Received a Consumer query - update current state from that source.
                    self.mdata_now = None
                    self.query_now = ( self.interruptCause.id_ix, self.interruptCause.query_ix )
                    self.data_now = self.interruptCause.query_data
                    #print "%.2f %s queried by %s %d for %d" % ( now(), self.id_str,
self.interruptCause.id_str, self.interruptCause.query_ix, self.interruptCause.query_data )
                    # If not seen this query before, log and maybe forward it.
                    #print "  checking %s in query log:\n  %s" % \
                    #    ( self.query_now, self.query_db )
                    if self.query_db.count( self.query_now ) == 0:
                        self.query_db.append( self.query_now )
                        # If this is query forwarding design, forward if no local metadata match.
                        if design == "fwd_query" and self.mdata_db.count( self.data_now ) == 0:
                            for i_peer in self.b_peer:
                                self.interrupt( i_peer )

                elif type( self.interruptCause ) == type( P_Upd( 0, self ) ):
                    # Received a Provider update - update metadata from it.
                    # (Don't save provider identity with record - essential in reality.)
                    self.query_now = None
                    self.data_now = None
                    self.mdata_now = self.interruptCause.update_mdata
                    #print "%.2f %s updated by %s with %d" % ( now(), self.id_str,
self.interruptCause.id_str, self.interruptCause.update_mdata )
                    # If not seen this data record before, log and maybe forward it.
                    #print "  checking %d in mdata:\n  %s" % \
                    #    ( self.mdata_now, self.mdata_db )
```

```
                if self.mdata_db.count( self.mdata_now ) == 0:
                    self.mdata_db.append( self.mdata_now )
                    # If this is update forwarding design, forward mdata to all peers.
                    if design == "fwd_update":
                        for i_peer in self.b_peer:
                            self.interrupt( i_peer )

            elif type( self.interruptCause ) == type( B_All( 0 ) ):
                self.query_now = self.interruptCause.query_now
                self.data_now = self.interruptCause.data_now
                self.mdata_now = self.interruptCause.mdata_now
                #print "%.2f %s forward from %s state query %s %s update %s" % ( now(), self.id_str,
self.interruptCause.id_str, self.query_now, self.data_now, self.mdata_now )
                # Decide what forwarded update was likely to be from design.
                # If this is query forwarding design, forward if no local metadata match.
                if design == "fwd_query":
                    #print "  checking %s in query log: %s" % ( self.query_now, self.query_db )
                    if self.query_db.count( self.query_now ) == 0:
                        self.query_db.append( self.query_now )
                        broker_msg = broker_msg + 1
                        if self.mdata_db.count( self.data_now ) == 0:
                            for i_peer in self.b_peer:
                                self.interrupt( i_peer )
                # If this is update forwarding, if not seen record before, log and forward it.
                if design == "fwd_update":
                    #print "  checking %d in mdata: %s" % ( self.mdata_now, self.mdata_db )
                    if self.mdata_db.count( self.mdata_now ) == 0:
                        self.mdata_db.append( self.mdata_now )
                        broker_msg = broker_msg + 1
                        # Always forward mdata to all peers.
                        for i_peer in self.b_peer:
                            self.interrupt( i_peer )


                # if not-end-of-req-chain: reactivate( self.b_peer )
                # Decide to forward - measure user msg + 1 only on receipt (when know updating) - a
fudge that hides volume esp. update forwarding?
                # In coding it seems crucial determinant of relative message volume is extra check in
query forwarding whether got local data - can reduce volume all other things being equal.

            else:
                print "%.2f %s unknown intuerrupt ignored" % ( now(), self.id_str )

            # NB Consumer Provider interupt seen-before checks redundant.

initialize()
print "%.2f start %d C (query %d s) %d P (update %d s) %d B (%s) seed %d" % \
    ( now(), c_num, c_sec, p_num, p_sec, b_num, design, new_seed )

cs = []
ps = []
bs = []
# Create brokers in ring topolgy (ensures full connection, don't care about topology for now).
for i_b in range( b_num ):
    new_b = B_All( i_b + 1 )
    if i_b > 1:
        new_b.set_peer( bs[ i_b - 1 ] )
        new_b.set_peer( bs[ i_b - 2 ] )
        activate( new_b, new_b.go() )
    bs.append( new_b )
# (Neighbors of first 2 brokers the last 2.)
bs[ 0 ].set_peer( bs[ b_num - 1 ] )
bs[ 0 ].set_peer( bs[ b_num - 2 ] )
bs[ 1 ].set_peer( bs[ b_num - 1 ] )
bs[ 1 ].set_peer( bs[ 0 ] )
activate( bs[ 0 ], bs[ 0 ].go() )
activate( bs[ 1 ], bs[ 1 ].go() )
for i_b in bs:
    b_str = "B %s connected " % i_b.id_str
    for i_bp in i_b.b_peer:
        b_str = b_str + i_bp.id_str + " "
    #print b_str

# Create consumers and providers; if their number matches brokers asign 1:1, otherwise random.
```

```
# (Consumer activation times after providers on different even numbers, providers on odd.)
for i_c in range( c_num ):
    if c_num == b_num:
        new_c = C_Req( i_c + 1, bs[ i_c ] )
    else:
        new_c = C_Req( i_c + 1, bs[ random.randrange( 0, b_num ) ] )
    activate( new_c, new_c.go(), at = ( 2 * p_sec ) + ( 2 * i_c ) )
    cs.append( new_c )
for i_p in range( p_num ):
    if p_num == b_num:
        new_p = P_Upd( i_p + 1, bs[ i_p ] )
    else:
        new_p = P_Upd( i_p + 1, bs[ random.randrange( 0, b_num ) ] )
    activate( new_p, new_p.go(), at = 1 + ( 2 * i_p ) )
    ps.append( new_p )

simulate( until = 1000000 )
print "%.2f end %d C/P %d B messages" % ( now(), user_msg, broker_msg  )
```

N.3. Broker network peer scaling

This program was used to demonstrate that more randomly connected networks (of brokers) minimised the number of hops between pairs of nodes. Instances of the B_Node class, representing brokers, fill the B_Network class' member list a_bnodes.

The main program calls the network constructor in a loop to pass a sequence of different values for parameter pair b_num and net_sigma, representing the network size and randomness of connectivity. The constructor creates every broker nodes, then makes the first maximal ring of connections with the node class' set_peer method, before making the random connections. Broader randomness to connections is indicated by the index of the new neighbour of a node having a greater difference to the node's own index. A normal distribution is used, of the width indicated by the randomness parameter, but as the new neighbour cannot be the same as the node or its first neighbour, the distribution is not preserved.

Once the main program has created a network with the given parameters, it calls the network crawl method to set the statistics of how well connected the network is, members mean_con and max_con. These represent the average of every node's average distance to every other node, indicated by the number of hops between nodes, and the maximum number of hops of any node pair in the network. They are derived from each node's own member statistics, mean_hop and max_hop.

To calculate the statistics, the crawl method invokes each node's reach_all function, updating the network max_hop statistic if necessary and summing the average hops before finally calculating the overall average. The reach_all method itself works by following links through the graph of nodes, starting with the local's node 2 neighbours, until all nodes have been reached. The number of hops to each node is put in the member dictionary d_reach. The graph is iterated over by following the neighbours of nodes currently in cursor, a list of node references, recording the distance to them if they've not already been reached, then making them part of the new_c, the cursor to use on the next iteration. The maximum number of hops is given by the number being recorded just before the dictionary was full, and the average can be found by summing the dictionary entries' value.

Some intermediate workings in this implementation, for a different method of traversing the network and reporting the details of connections made, are included but commented out.

```
# file: broke_net_gen.py author: J Lewis-Bowen updated: 2004.05.30
# content: Simulation demonstrating connection strategy for rapid propogation.

import random
import time
new_seed = int( ( time.time() * 100 ) % 1000000 )
random.seed( new_seed )

class B_Node:
    def __init__( self, id_ix ):
        self.id_ix = id_ix
        self.b_peer = []
        self.d_reach = {}
        self.max_hop = 0
        self.mean_hop = 0.

    def set_peer( self, b_peer ):
        self.b_peer.append( b_peer )

    def reach_all( self ):
        hop = 1
        self.d_reach[ self.id_ix ] = 0
        cursor = [ self ]
        # Try reaching each node by building tree.
        while len( self.d_reach ) < b_num and len( cursor ) > 0:
            new_c = []
            for i_c in cursor:
                for i_p in i_c.b_peer:
                    if not self.d_reach.has_key( i_p.id_ix ):
                        self.d_reach[ i_p.id_ix ] = hop
                        new_c.append( i_p )
            hop = hop + 1
            cursor = new_c
        self.max_hop = hop - 1
        hop_total = 0
        # Doing this as not sure will always be b_num nodes reachable.
        for i_b in self.d_reach.keys():
            hop_total = hop_total + self.d_reach[ i_b ]
        self.mean_hop = float( hop_total ) / float( len( self.d_reach ) )

        # Recursive call version - thought adding history to save loops.
        #    self.d_reach[ i_b ] = self.reach( i_b, 0 )
        #    hop_total = hop_total + self.d_reach[ i_b ]
    #def reach( self, b_node, hop, origin ):
        # Recursive function - stop if this is node searched for or hopped too long.
        # Problem - parts explosion, stack gets too big.
        #print "%d asked to reach %d at %d" % ( self.id_ix, b_node, hop )
        #if self.id_ix == b_node:
        #    return hop
        #elif hop == b_num:
        #    return b_num + 1
        #else:
        #    a_peer = []
        #    for i_peer in self.b_peer:
        #        a_peer.append( i_peer.reach( b_node, hop + 1 ) )
        #    return min( a_peer )


class B_Network:
    def __init__( self, b_num, net_sigma ):
        self.a_bnodes = []
        self.b_num = b_num
        self.mean_con = 0.
        self.max_hop = 0
        # Create brokers in one-way graph - set 2 connections per node.
        for i_b_ini in range( self.b_num ):
            self.a_bnodes.append( B_Node( i_b_ini ) )
        # Only set peers after all nodes created so can randomly connect to any.
        for i_b in range( self.b_num ):
```

```
            # "Outer ring" connect next along necessary property of full connectivity.
            near_peer = ( i_b + 1 ) % self.b_num
            far_peer = i_b
            # Find other peer on normal distn. from here, random as long as not self or near_peer.
            while far_peer == i_b or far_peer == near_peer:
               far_peer = int( random.normalvariate( i_b, net_sigma ) ) % self.b_num
            self.a_bnodes[ i_b ].set_peer( self.a_bnodes[ near_peer ] )
            self.a_bnodes[ i_b ].set_peer( self.a_bnodes[ far_peer ] )

    def crawl( self ):
        # Only look up hops to all neighbours after fully connected.
        total_con = 0.
        for i_b_crawl in self.a_bnodes:
            i_b_crawl.reach_all()
            total_con = total_con + i_b_crawl.mean_hop
            if i_b_crawl.max_hop > self.max_hop:
               self.max_hop = i_b_crawl.max_hop
        self.mean_con = total_con / float( self.b_num )

# report on each nodes connectivity
#for i_b in a_bs:
#    b_str = "Node %s connected" % i_b.id_ix
#    for i_bp in i_b.b_peer:
#        b_str = "%s %d" % ( b_str, i_bp.id_ix )
#    print "%s hop mean %.3f max %d reach" % \
#        ( b_str, i_b.mean_hop, i_b.max_hop ) # :\n  %s, i_b.d_reach )


max_b_num = 500
b_num = 4

print "%d to %d nodes (%d seed)" % \
    ( b_num, max_b_num, new_seed )

a_b_num = []
while b_num <= max_b_num:
    a_b_num.append( b_num )
    b_num = 5 * b_num
for i_b_num in a_b_num:
    a_sigma = [ 1 ]
    i_sigma = 2
    while i_sigma <= i_b_num:
        a_sigma.append( i_sigma )
        i_sigma = 2 * i_sigma
    for i_sigma in a_sigma:

        # Create and explore networks.
        b_network = B_Network( i_b_num, i_sigma )
        b_network.crawl()
        print "%d nodes %.1f spread %.3f connectivity %d max-hop" % \
            ( i_b_num, i_sigma, b_network.mean_con, b_network.max_hop )

# demonstrated more randomness always helps (did do just to sigma = size / 2):
# Also did 2500 (12500 too much?)
```

N.4. Scalability with recall messages

The most sophisticated simulation of the EGSO broker network tested whether recall messages could improve performance by stopping the forwarding of consumer queries that had been resolved. The parameters for this model are the relative speed of query and query-stop messages, t_msg_query and t_msg_stop, as well as the network size, the maximum interval between consumer queries (much greater than simulated message forwarding times) and a flag indicating whether stop messages should be sent.

In this model, brokers were not implemented as SimPy processes; they just represent static network for queries (and stop messages) to propagate through. Instances of Broker in the global brokers list are given maximally random connections to peers with an even distribution of second neighbour indexes (unlike the near normal distribution of distance used in the previous network experiment). Broker class members recorded the messages seen in a log dictionary, and a highly abstract representation of the metadata held - simply the broker's own index.

A dictionary enumeration, query_state, is used in the log dictionary entry object, QueryLog. Each log entry, keyed on the consumer that initiated the query (on the assumption a consumer can only have one live query in progress at a time), has a query sequence identifier (unique for that consumer), and pointers to the brokers from which the query was immediately forwarded and originally stopped (initially undefined), as well as the latest query state seen by the broker.

Consumer processes, as previously, generate queries at regular intervals of the simulation clock to their local broker, the query represented by the index of the broker that can resolve the query. Providers are not modelled at all, as this simulation does not concern metadata updates.

Classes derived from the SimPy class Process are also used to represent the messages being passed around the broker network. This allows simulation time to pass, perhaps representing network latency or implied broker message queue action time, to give stop messages a chance to overtake query propagation.

A consumer creates a QueryMsg instance each time it is activated by its go method, passing its broker pointer to the constructor, with other logged query metadata, as the first node to query. That query will wait until the simulation time represented by the experiment's t_msg_query parameter has passed. If the target broker can match the query, the state is updated and, if design configuration permits, a stop message is created to run back down the propagation chain of brokers. In other cases, when the broker target has not logged the current query, a new query message is created and activated for the target broker's neighbours. The broker log is also updated in this case so that back propagation of stop messages is possible.

Each stop message behaves in a similar way (once it has waited for a shorter time), but in this case it checks its target broker's log to decipher whether the query has already been stopped and which node is the previous link in the chain. If the broker has not already seen the stop message, and the previous link in the chain was not the node the stop message has just been routed through, and the design allows stop message forwarding, a new stop message process is constructed targeted at the earlier node. Even when the stop is not propagated (in which case it must be occurring on the broker where the query was resolved) the target broker's query state is updated to save further messaging were a copy of the query message to be forwarded via a different route. Once the stop message reaches the consumer, it can trigger it to submit a new query immediately.

The simulation runs for a fixed number of user messages, as before, and reports the same statistics which indicate performance by giving the ratio of user to broker message.

```
# file: cb_msg.py author: J Lewis-Bowen updated: 2004.06.01
# content: Simulating EGSO data-grid broker message forwarding and recall.
```

```
from __future__ import generators
from SimPy.Simulation import *
initialize()

import random
import time
new_seed = int( ( time.time() * 100 ) % 1000000 )
random.seed( new_seed )

t_msg_query = 16
t_msg_stop = 1

n_consumer = 20
n_broker = n_consumer
t_c_pause = 1000

design = None # or "fwd_stop" # or None
broker_msg = 0
consumer_msg = 0
total_msg = 10000

# Query state 'enum' - fwd_query and fwd_stop not used.
query_state = { "seen_query": 0, "fwd_query": 1, "seen_stop": 2, "fwd_stop": 3 }

if design == "fwd_stop":
   print "%.2f sim start %d seed %d broker network stop %d times faster" % \
      ( now(), new_seed, n_broker, t_msg_query / t_msg_stop )
else:
   print "%.2f sim start %d seed %d broker network no stop messagees" % \
      ( now(), new_seed, n_broker )


# Query message waits, looks at broker data, propogates 2 more queries or backwards stop.
#   See message in mid flow (with unique identity for the consumer, and its content):
#   C origin (ID string) ---...---> previous (B or C ref) --- msg ---> target (B ref)
#   may create 2 more new messages (if not already forwarded or seen stop for this messages):
#   target (B ref) --- new-msg ---> target's peers (B ref)
class QueryMsg( Process ):
   def __init__( self, cont, orig, iden, prev, targ ):
      Process.__init__( self )
      self.content = cont
      self.origin = orig
      self.msg_id = iden
      self.previous = prev
      self.target = targ

   def go( self ):
      # ACTION introduce more randomness to this. Wait (as if in transit) then log and act.
      t_wait = t_msg_query
      #print "%.2f QueryMsg %s-%d from %s to %s pause %.2f" % ( now(), self.origin, \
      #    self.msg_id, self.previous.identity, self.target.identity, t_wait )
      yield hold, self, t_wait
      global broker_msg
      broker_msg = broker_msg + 1
      # Found match at target - log query stop, start sending stop messages back up the path.
      if self.target.mdata.count( self.content ) == 1:
         if not self.target.log.has_key( self.origin ):
            self.target.log[ self.origin ] = QueryLog( self.msg_id, self.previous )
         self.target.log[ self.origin ].state = query_state[ "seen_stop" ]
         self.target.log[ self.origin ].stopper = self
         if design == "fwd_stop":
            new_stop = StopMsg( self.origin, self.msg_id, self.previous, self.target )
            activate( new_stop, new_stop.go() )
      # Test that target hasnt seen this query, then log it and propogate it to neighbours.
      # (Wont propogate again if logged query no matter what state).
      elif not self.target.log.has_key( self.origin ) or \
               self.target.log[ self.origin ].query_id < self.msg_id:
         self.target.log[ self.origin ] = QueryLog( self.msg_id, self.previous )
         for i_peer in self.target.peers:
            new_query = QueryMsg( \
               self.content, self.origin, self.msg_id, self.target, i_peer )
            activate( new_query, new_query.go() )
```

```
# Stop message looks at broker query state, may propogate to peer that didn't come from.
#   Arguments represent entities in flow as QueryMsg, except working back towards origin;
#   i.e. C may be target, forwarded messages created may be against or with flow of old query.
class StopMsg( Process ):
    def __init__( self, orig, iden, targ, prev ):
        Process.__init__( self )
        self.origin = orig
        self.msg_id = iden
        self.previous = prev
        self.target = targ

    def go( self ):
        # ACTION introduce more randomness to this. Wait (as if in transit) then log and act
        t_wait = t_msg_stop
        #print "%.2f StopMsg %s-%d from %s to %s pause %.2f" % ( now(), self.origin, \
        #    self.msg_id, self.previous.identity, self.target.identity, t_wait )
        yield hold, self, t_wait
        global broker_msg
        broker_msg = broker_msg + 1
        # If node a client, query completed.
        if type( self.target ) == type( Consumer( 0, None ) ):
            reactivate( self.target )
        # Otherwise assume another broker in chain to pass stop through - propogate toward
        # origin and to neighbour that stop didnt come from (if it was remembered).
        elif self.target.log.has_key( self.origin ) and \
                self.target.log[ self.origin ].query_id == self.msg_id and \
                self.target.log[ self.origin ].state != query_state[ "seen_stop" ]:
            # Log query as stopped before activating messages (to avoid repeat).
            self.target.log[ self.origin ].state = query_state[ "seen_stop" ]
            self.target.log[ self.origin ].stopper = self.previous
            # If the original query origin is not where seen the stop from, forward the stop.
            if self.target.log[ self.origin ].stopper != self.target.log[ self.origin ].sender:
                new_stop = StopMsg( self.origin, self.msg_id, \
                    self.target.log[ self.origin ].sender, self.target )
                activate( new_stop, new_stop.go() )
            # For each peer, if peer is not where seen stop from, forward the stop.
            for i_peer in self.target.peers:
                if i_peer != self.previous:
                    new_stop = StopMsg( self.origin, self.msg_id, i_peer, self.target )
                    activate( new_stop, new_stop.go() )
        else:
            # Still log query as stopped even if don't need to forward (in case see query).
            if not self.target.log.has_key( self.origin ):
                self.target.log[ self.origin ] = QueryLog( self.msg_id, self.previous )
            self.target.log[ self.origin ].state = query_state[ "seen_stop" ]
            self.target.log[ self.origin ].stopper = self.previous


# Consumer waits then generates original query, suspends until stop (answer) received.
class Consumer( Process ):
    def __init__( self, id_ix, brok ):
        self.identity = "C%03d" % id_ix
        Process.__init__( self, name = self.identity )
        self.broker = brok
        self.query_id = 0

    def go( self ):
        while 1:
            # Wait (for imaginary user input) before posing another query, then update statistic.
            yield hold, self, t_c_pause
            global consumer_msg
            consumer_msg = consumer_msg + 1
            if consumer_msg > total_msg:
                stopSimulation()
            # Increment query msg id, pick mdata queried at random ACTION not on broker range.
            self.query_id = self.query_id + 1
            query_what = random.randrange( 1, n_broker + 1 )
            #print "%.2f Consumer %s request %d for %d to %s" % ( now(), self.identity, \
            #    self.query_id, query_what, self.broker.identity )
            new_query = QueryMsg( query_what, self.identity, self.query_id, self, self.broker )
            activate( new_query, new_query.go() )
            # Don't do anything while query in progress.
            yield passivate, self
```

```
# Broker not process, just place where metadata held (message objects do its work).
class Broker:
    def __init__( self, id_ix ):
        self.identity = "B%03d" % id_ix
        self.peers = []
        # ACTION remove this cheat where broker only know about mdata defined by own index.
        self.mdata = [ id_ix ]
        # Query log has dictionary of QueryLog objects States: 0 seen query, 1 forwarded query,
        #   2 seen stop (may reach before 1), 3 forwarded stop. E.G. 6th query from C1,
        #   forwarded from B2 { "C001", [ 6, B002*, None, 0 ] }
        self.log = {}
        # Stop log has C1 completed query identites as: { "C001", [ 1, 2, 3, 5 ] }.
        #self.stopped = {}

    def set_peer( self, peer ):
        self.peers.append( peer )


# Broker uses QueryLog objects to conveniently store its view of each consumers query in progress.
class QueryLog:
    def __init__( self, id_ix, send ):
        self.query_id = id_ix
        self.sender = send
        self.stopper = None
        self.state = query_state[ "seen_query" ]


brokers = []
# Create network, assigining brokers' second peer neighbours at random.
for i_broker in range( n_broker ):
    brokers.append( Broker( i_broker + 1 ) )
for i_broker in range( n_broker ):
    next_ix = ( i_broker + 1 ) % n_broker
    far_ix = i_broker
    while far_ix == i_broker or far_ix == next_ix:
        far_ix = random.randrange( 0, n_broker )
    brokers[ i_broker ].set_peer( brokers[ next_ix ] )
    brokers[ i_broker ].set_peer( brokers[ far_ix ] )

# Simulation starts with a lot of consumers activated to pose queries.
# (Consumers contact broker given on index - MUST change if n_consumer != n_broker )
for i_consumer in range( n_consumer ):
    new_consumer = Consumer( i_consumer + 1, brokers[ i_consumer ] )
    activate( new_consumer, new_consumer.go(), at = i_consumer )

simulate( until = 10 * total_msg * t_c_pause )
print "%.2f sim end %d consumer %d broker messages" % ( now(), consumer_msg, broker_msg )
```

N.5. Experimental results

The programs above reported the following results, presented as graphs in Chapter 6. Throughout, performance is measured by the ratio of maintenance messages (broker to broker traffic forwarding queries or metadata updates) to user messages (both from the provider and the consumer).

Performance does not change as metadata grows.

Metadata size	Query fwd 1	Query fwd 2	Metadata fwd
196	14.402	14.421	3.724
360	15.067	15.162	3.116
523	15.01	15.333	3.097
696	14.934	15.029	3.287
860	14.858	15.143	3.116

1023	15.01	14.972	3.097
1196	14.991	14.915	3.287
1360	15.143	15.314	3.116
1523	15.105	15.276	3.097
1696	14.934	14.991	3.287

The ratio of broker-broker messages to user-broker messages is consistent as broker metadata on provider data resources grows, for both algorithms. All experiments used a 20 broker ring network with a provider update rate of 0.2 with respect to the consumer query rate.

Performance degrades in proportion to network size for both algorithms.

Network size	4	20	100	500
Query fwd 1	1.8735	14.978	80.21	309.88
Query fwd 2	1.8732	14.906	80.269	308.88
Metadata fwd	0.5016	3.2224	17.82	111.38

Linear rise in the ratio of broker-broker messages to user-broker messages with respect to network size (the number of brokers connected in a ring). All experiments use a provider update rate of 0.2 with respect to the consumer query rate. Note, broker message volume is deterministic when provider updates forwarded, so a single simulation run is sufficient.

Effect of changing ratio of consumer to provider depends on algorithm.

Ratio C:P	0.04	0.2	1	5
Query fwd 1	17.307	14.978	8.9414	2.983
Query fwd 2	17.275	14.906	8.987	2.9849
Metadata fwd	0.798	3.2224	9.538	15.85

Converse linear rise and fall of broker-broker messages to user-broker messages for provider update forwarding algorithm versus consumer query forwarding with respect to provider update rate (with respect to the consumer query rate). All experiments use a 20 broker ring networks.

Linear growth of distance between nodes with logarithm of network size.

Network size	4	20	100	500
Mean hops	1	3.235	5.529	8.023

The minimum average number of hops between arbitrary pairs, for the most random broker network connection, grows in proportion to the logarithm of network size.

Efficient connections between nodes do not much affect performance.

Connectivity randomness	Query fwd 1	Query fwd 2	Metadata fwd
0	15.029	15.021	3.222
0.1	14.899	13.373	3.222
1	13.518	13.824	3.222

There is only a slight decrease in the ratio of broker-broker to user-broker messages for the consumer query forwarding algorithm as the randomness of network connection between brokers grows. All experiments use a 20 broker network with a provider update rate of 0.2 with respect to the consumer query rate.

Effort of stopping queries decrease performance further.

Network size	4	20	100	500
Query fwd 1	10.4	67.9	331.9	1445.8

Query fwd 2	10.8	66.9	333.9	1405.9

Simulation results showing significantly more broker-broker messages in proportion to user-broker messages, growing in a linear way as network size increases. Only consumer query submission (not provider content update) messages was simulated on a randomly (fully) connected broker network.

Appendix O. Commercial development

Despite the demonstrated value of dynamic models in scientific data-grid projects, it does not follow that modelling is likely to be taken up in similar commercial products, as evidence from the author's previous experience as a developer and programming team leader at Logica Mobile Communications shows. Though these reported observations do not represent a survey of industrial practices, specific examples of the way software engineering practice are carried out in commercial development are listed below.

- *Lifecycle gates*. Product releases' timely delivery was controlled with formal gate meetings at waterfall development milestones, where deliverables were checked. This mitigated against uncontrolled delays to release experienced in earlier product releases, caused by requirements creep and reliance on libraries that were both critical and at risk from their complexity. However, project managers did not clearly distinguish between business and software processes (as noted for other businesses [109]), whilst developers and testers were not aware of wider standards being followed.
- *Refactoring*. Developers significantly improved the scalability and administration functionality of legacy code (the previous rapid development of daemons, which had become critical to customers' high-throughput business operations, had generated chunks of thousands of lines of unstructured code without decomposition structure, documentation or reference tests). However, these improvements to software's maintainable quality did not follow published solutions (such as refactoring patterns [46]). Developers used their own experience; success therefore relied on tacit knowledge, whilst the implemented functionality of live reconfiguration controls were poorly linked to customer needs.
- *Iterative development*. In some cases, features were developed with close customer interaction (as in Extreme Programming [7]); very early prototypes were shared and guided by customer feedback. These tended to be small high-risk components, such as gateways that joined systems with novel protocols.
- *Simulation*. Simulations of users' behaviour were developed to evaluate designs, notably for new libraries of complex operations that several sub-systems relied on. These were unplanned, were not packaged for customer delivery, only being written for developers' convenience (though colleagues reused them in system tests). Harnesses simulating user actions' were similarly developed for automated testing (of diverse command formats directed to the applications using the new libraries; they were later adapted to performance and regression testing too). As they were designed and evaluated before the applications themselves were modified, this activity shows evidence of test first planning (another ingredient of Extreme Programming).
- *Paired programming*. Some engineers worked very closely, so that development on difficult parts of systems was completely collaborative. This

again was unplanned in project management task assignment, and lead to some uncertainty of responsibility, whilst delivering the higher quality promised by Extreme Programming and allowing expert programmers to share their knowledge.

- *Rapid development*. In major components, scripting languages (including Python and the in-house Flexible Services Development Language) were used to customise high performance daemons (written in C and C++). This successful rapid development strategy also emerged, without planning for maintenance of the customisation code (of hundreds of times greater volume than initially expected).
- *Process metrics*. Sophisticated fault tracking and revision control computer aided software engineering systems were used (including ClearCase), following documented processes that evolved for the business. These tools could generate sophisticated metrics for managers (with scripted query design and analysis, for example, to track sub-systems' fault rates). However, such advanced capability was not systematically used, and did not guide strategy (for example, targeting effort to improve productivity).

This experience of commercial practice therefore shows how high quality engineering practices emerge naively. The emphasis on timely delivery meant business processes did not support ongoing product line investment, though code to support this was developed by developers for their own convenience. Also, the waterfall model, implicitly imposed by the gate-driven process for release traceability, was strengthened through Extreme techniques intuitively when engineers faced risky tasks. In this context, it is hard to imagine a business case for the modelling techniques investigated for data-grids and presented in this thesis being recognised at the level of corporate strategy, though developers may readily adopt it as a technique to reduce risk of quality failure.