Using Event Models in Grid Design

Anthony Finkelstein Joe Lewis-Bowen
Giacomo Piccinelli
Department of Computer Science, University College London,
Gower Street, London, WC1E 6BT, UK
{A.Finkelstein, J.Lewis-Bowen, G.Piccinelli}@cs.ucl.ac.uk

Abstract

Virtual observatories, the goal of the EGSO and AstroGrid projects, exemplify demanding grid requirements. We present a process for rapidly developing event models of architectural designs, using the existing LTSA tool. We report on the models used throughout the early lifecycle of the projects; these faithfully evaluated and clearly demonstrated whether candidate designs fulfill grid requirements. We therefore recommend using the demonstrated technique to evaluate other grid projects' designs.

1 Introduction

A method for evaluating grid system designs with event models is presented in this chapter. Grid projects must satisfy demanding requirements by combining the functionality of distributed components. In the AstroGrid and EGSO projects innovative architectural designs have been proposed to achieve this. We have assessed these before implementation by developing formal models.

The modelling language, FSP (Finite State Process), and its analysis tool, LTSA (Labelled Transition System Analysis), are well established [27]. Enough detail of the dynamic modelling language is presented here for readers to apply this method themselves. We also report on our experience of modelling astronomy grid systems; models proved valuable throughout the early project lifecycle.

Our models of astronomy data-grids bridge requirements and design to validate the planned systems. Before discussing the modelling method and experience, we introduce our projects' requirements and design solutions to demonstrate the relevance of our architecture models.

1.1 Data-grid requirements

The European Grid of Solar Observations (EGSO [11]) for solar physicists and AstroGrid [2] for night-side astronomers both provide an initial framework for 'virtual observatories'. Their similar requirements are typical of data-grid projects, which enable access to and analysis of widely distributed complex data and aid knowledge generation.

Astronomers need on-line data and analysis tools to effectively address their scientific problems. However, it is often difficult to locate and match these

[9]. Existing on-line archives of observations (for example NASA SDAC [31] and Strasbourg CDS [8]) have diverse, labor intensive access methods. Data organization standards are not generally followed as different instrument teams work within different physical parameters. There is also a variety of specialist software available (for example, SolarSoft [4] and Starlink [33]), and much larger datasets are planned.

A virtual observatory should provide a common infrastructure to federate resources. As well as enabling transparent access to diverse datasets and automated analysis, collaborative investigations should be possible. It should also maximize the benefit derived from collected data and accelerate the growth of knowledge, in line with the e-science vision [23].

At an abstract level, these requirements are also shared by grids in which diverse distributed computational resources are the critical resource. Both must share resources in transparent infrastructure across traditional domain boundaries to support flexible, efficient services – enabling virtual organizations, essential to the grid vision [15].

The description of EGSO's requirements, phrased in a general way below, are exemplars for the domain; they may be used a checklist for other data-grid projects' requirements. Their detail supports the validity of the models discussed in section 3. A general review of data-grid requirements are given elsewhere [21]. The techniques used to elicit requirements are also presented to demonstrate that they accurately capture user needs.

EGSO requirements. The classified essential system requirements follow. They emphasize operational and maintenance aspects (as classified by [3], also called non-functional or quality of service requirements). As such behavior cannot be implemented by an isolated component, they must be considered when planning the general system architecture.

Data and metadata. The system should enable users to gain access (subject to a security policy) to data and non-data resources. Cache space, computation resources and data processing applications are examples of non-data resources.

To achieve this the system should support a framework of metadata structures that incorporate all resource attributes in the current solar physics archives. It should include administrative, structural and descriptive information. The framework should be capable of supporting semi-structured and incomplete data and metadata.

The system should be able to translate between metadata structures and correlate multiple data resources as required. Metadata structures should not be dependent upon references to other information resources for their use, wherever possible.

When accessing data, the user should also be able to view the corresponding metadata.

Data processing. The system should be enable users to access computing facilities to prepare and analyze data, and execute user processing tasks.

The system should support the migration of existing and user uploaded software and data to these facilities, binding user parameters to tasks interactively. Interfaces should be provided to promote increased uniformity of access to computing resources independent of underlying mechanisms.

Monitoring and management. The system should include components to monitor the state of resources, infrastructure and submitted user tasks. Tasks should be managed so that users may be notified of their state changes.

Security. The infrastructure should enable both authorization and authentication to uphold security. These mechanisms should support policy for different types of request at different granularity (from the whole system to parts of a dataset).

The security infrastructure should protect the resources available via the system. At the same time, scientific users and the providers of resources should find the security mechanisms easy to use.

Interoperability. The system should be interoperable with other grid projects (in solar physics and related domains); it should make use of their standards for metadata and protocols.

Within EGSO, uniform standards for data management, access and analysis should be used by all system entities. Common interfaces support the incorporation of multiple, heterogeneous and distributed resources.

Requirements analysis. The technical EGSO requirements were derived from a wider user requirements investigation conducted during the first step in the project. EGSO's vision was illustrated with informal system diagrams and usage scenarios, which formed the basis of the models described in section 3.1.

The methodology adopted for eliciting firm requirements involved established techniques [22]. Direct sources of information included interviews, group discussions, small workshops, questionnaires, and scenario-based feedback. Indirect sources of information included domain-specific documents, analysis of similar projects, and analysis of existing systems (as described in [16] [35]).

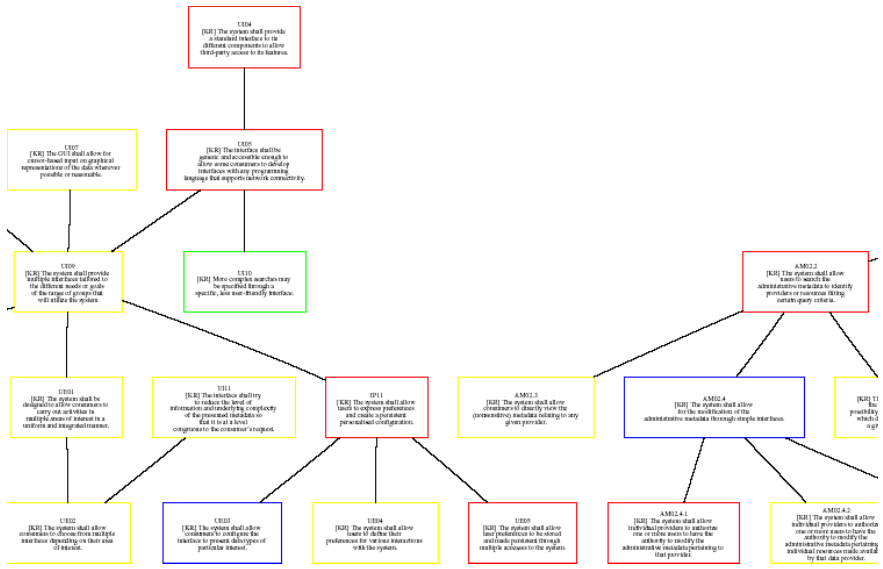
The requirements, including domain knowledge of existing working practice and future goals, were presented tree-like relations (fig. 1). This representation aided requirements reviews in feedback sessions. Separate branches of the tree covered different areas of concern for the system. The depth of a node within the tree (its distance from the root) captured the scope of the concern addressed. Node color was used to categorize requirements. The tree was encoded in XML and a tool was developed for its automated management (which generated fig. 1).

This representation greatly helped various stakeholders gain an immediate perception of the relations between different requirements (related to 'viewpoints' [13]). In particular, the tree-based format played a crucial role in requirement prioritization. Situations in which a narrow requirement, believed to be important, was within the scope of a wider requirement area, accepted as less important, were immediately exposed.

Also, the tree format enabled a clear view of areas of concern for which an adequate level of detail had not been achieved. Such situation was highlighted by shallow branches including nodes of high priority. Areas such as security and user interface were expanded based on this technique.

The requirement engineering activity generated EGSO's NSR (Negotiated Statement of Requirements [22]). Detailed scenarios were also derived, which provided input for the models described in section 3.1.

Figure 1: A view of part of the EGSO requirement tree showing relationships, priority and hidden requirements. Note the clear representation of these 18 nodes from a graph of 171 (the detailed text is not relevant).



1.2 Astronomy data-grid designs

As the EGSO requirements were refined, the envisioned system was captured in a formal architecture. Following MDA (Model-Driven Architecture [17]) principles, different levels of refinement were used for multiple layers; the infrastructure middleware components were specified between user interfaces and local resource applications. Unambiguous architecture diagrams were defined with UML (Unified Modelling Language [7]) profiles, exploiting the language's flexible notation. For example, fig. 2 shows the architecture of one sub-system.

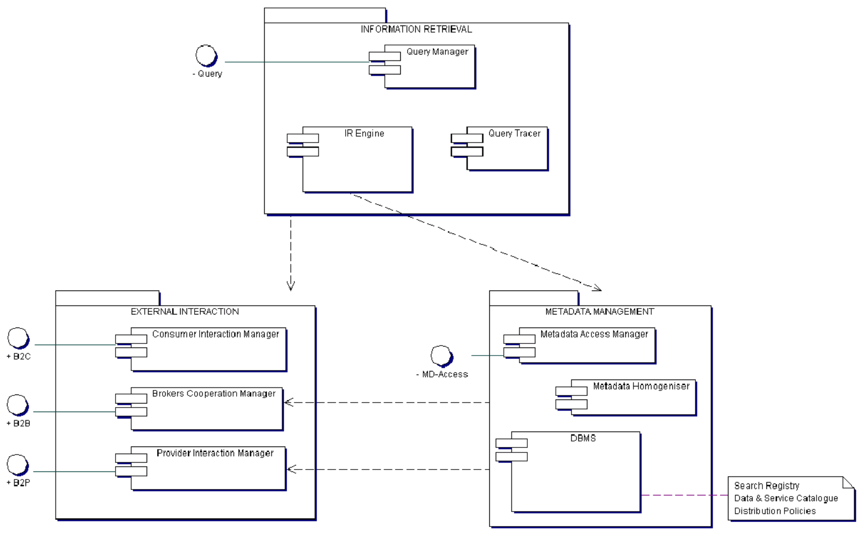
The components of the EGSO architecture are described below, with notable features of the whole system. The architecture of AstroGrid and other data-grids are presented too; their solutions to similar problem domains are compared with EGSO's.

EGSO: EGSO resolves the heterogeneous data and metadata of scattered archives into a 'virtual' single resource with a unified catalogue. This broad catalogue provides a standardized view of other catalogues and allows richer searches with information on solar events and features.

Resources are accessed via connectors for diverse protocols, and information is exchanged using adaptors that homogenize different formats. The EGSO framework for creating connectors and adaptors enables access to a wide range of software system.

The EGSO system architecture distinguishes three roles: data consumers, data providers, and brokers. Note that an organization which hosts an EGSO node can play multiple roles, and that all broker instances behave consistently.

Figure 2: UML component diagram of the EGSO broker high-level architecture (rendered by Together). Subsystem component groups, interfaces and static dependencies (not communication paths) are shown.



The roles are best understood by their interaction, apparent in design walk-through, so several usage stories follow.

A consumer submits its initial requests to a broker to find which providers hold the data or services specified. The broker provides the consumer with references to providers and information to help selection. The consumer then refines its request with one or more providers to receive the data or service directly.

A provider publishes information on its available data and services by contacting a broker. They agree what information is provided (for example: data format, resource ontology, update frequency, and access policy). A provider may also use a broker when contacted by a consumer (for example: to get information on the consumer).

Brokers monitor the interaction of consumers and providers, and manage information about resource availability. They interact with each other (in a decentralized peer-to-peer relationship), sharing this information to present consistent behavior. Brokers can therefore manage the state of user tasks and resource availability, and ensure security policies are upheld.

Supporting functionality (including: caching, logging, auditing, format transformation, and workflow management) are modelled as provider services. For example, if a broker saves queries or results, it is presented as a caching service provider.

The roles are reminiscent of the tiered architectural style with client, back-end and middle tiers. However, each acts as a server in a middleware layer that cuts across the system. Diverse user interfaces are served by the consumer, and there are clients for the broker and provider administrators. The provider wraps the primary back-end resources, but the broker and consumer roles also have back end interfaces to databases and other local operating system resources.

The EGSO system architecture therefore meets the requirements. Rich metadata (in the catalogues) is provided to facilitate data and data processing resource discovery (via brokers) and access (via provider connectors). Interoperability is enabled (using adaptors to homogenize information) in a secure, monitored framework (maintained by the brokers).

AstroGrid: The AstroGrid architecture has different components to EGSO, but their essential interaction is strikingly similar. Users initially contact a 'registry' of available services to locate their required data and data processing capabilities. A 'job control agent' acts on behalf of users to submit requests directly to resource providers. Also, a special class of registry accepts updates to service availability and distributes the update. Requests (and their results) are represented in a homogenous format, where necessary via a provider adaptor.

However, unlike EGSO, results are not returned directly to the user – instead the user is notified that they are available in a shared data area. This behavior fits well with the AstroGrid philosophy for asynchronous stateless communication and collaborative working practices.

This architecture does not have an analogue to the EGSO broker, though the registry and job control components partially fulfill its function. Without a component that coordinates resource access and user tasks, the AstroGrid system has less emphasis on infrastructure management. This architecture may prove more scalable, but may be unable to provide a consistent service.

Other projects: EGSO and AstroGrid alone illustrate grid scale adaptations of general architectural styles; EGSO's broker is a tiered solution, whilst AstroGrid's decentralized functionality has an asynchronous service model.

The following paragraphs survey other data grid projects' key architectural components. It is apparent that their architectures provide some of same functionality as EGSO, without clearly abstracting responsibility. Note that quality and quantity of information about these projects in the public domain varied significantly, so their review may be misrepresentative.

In the European Data Grid (EDG [12]), early project architecture documents describe organizations playing more than one role. A 'consumer' interacts with a 'registry' to locate 'producers'. The Consumer then contacts a Producer directly to obtain data. A 'metadata catalogue' is present to store attributes of logical file names.

In the Grid Physics Network (GriPhyN [20]), the focus is on a Virtual Data Toolkit (VDT). The VDT provides a data tracking and generation system, to manage the automatic, on-demand derivation of data products. A Metadata Catalog Service (MCS) contains information about logical files. User applications submit queries to the MCS based on attributes of the data. The MCS returns the names of logical files that satisfy the query. The user application then queries a Replica Location Service (RLS), to get handles for physical files before contacting the physical storage systems where the files reside.

In the Biomedical Informatics Research Network (BIRN [5]), a 'data mediator' component provides a semantic mapping, creating the illusion of a single domain from a user perspective. BIRN uses the Metadata Catalogue (MCAT) and associated Storage Resource Broker (SRB) to perform basic data retrieval functions. The Data Mediator liaises with associated ‘domain knowledge bases’ in response to queries. ‘Handles’ for data resources that satisfy the query are returned to the User Application. The MCAT then enables refinement of the query based on attributes of these data resources.

These projects have defined their available architectural models in terms of physical components or tools, rather than functional roles. Where comparisons can be drawn with the roles of the EGSO model, it appears that for most projects, queries and requests for information are refined between the entities playing the part of the ‘consumer’ and the ‘broker’. Two projects provide an inference to the ‘provider’ for refining requests. In nearly all projects, the ‘two-step’ nature of information retrieval is made explicit, with the discovery of logical file names being a process distinct from the discovery of physical file names and locations.

1.3 Overview

This introduction to EGSO demonstrates a well engineered design to fulfill rigorously gathered requirements that exemplifies data-grid projects. In the remainder of the chapter we discuss how to verify designs for this class of system requirements.

Sections 2 and 3 describe our method for developing dynamic models and report on our experience of using them. Section 2, on methodology, advances an existing event transition modelling language and tool to a reliable process. It should be especially interesting to those who’d like to learn how to practically apply dynamic modelling techniques. Section 3, our experience report, demonstrates the value of models developed at 4 stages in the projects’ lifecycles, from initial envisioning to detailed design. This should interest software engineers who wish to evaluate our method.

The concluding section, 4, summarizes our findings, draws attention to related work and proposes the direction of future developments. It is hoped this chapter will inspire others to model their systems using our method.

2 Methodology

This section introduces the method that we developed to evaluate the EGSO architecture (presented above, subsection 1.2) and the AstroGrid detailed design. It may model other novel distributed systems to judge whether requirements are met.

Our process for generating event models builds on the established FSP language (and the associated LTSA tool) and its creators’ techniques. The next section 2.1 introduces its purpose and scope. The remainder of this section introduces our process (section 2.2), and then demonstrates it with a worked example (section 2.3). This walk-through may be used as a tutorial introduction to FSP specification for readers who wish to reuse our modelling process.

2.1 Event modelling

Throughout engineering, models are used to test system properties before building the product for live use. Using models early in the development lifecycle improves understanding and reduces the risk of project failure, with relatively little cost and effort. Models typically use an abstract view that ignores all details of the system except those being studied (see [34] for a detailed overview).

Event models are used in software engineering to examine the interaction of partially independent concurrent processes. Each process may represent an operating system's thread, a user application, or a complete sub-system on a network. The dynamic operation of a process is represented by its state model – a graph in which the process's states are connected by events. When processes communicate, they share the same events and their state transitions are synchronized; such interaction introduces risk.

Concurrency verification tools analyze paths through the combined state space of all the processes in a system. They flag paths to any state in which no process can get make further transitions – a 'deadlock' – and circular paths from which there are no possible transitions to a system's target state – called 'livelock'. When these undesirable concurrency conditions are found, software engineers typically employ well known methods; guarding access to shared resource can prevent deadlock, and adjusting process's priority can prevent livelock (as discussed and illustrated using LTSA and Java by [27]).

LTSA is an event modelling tool that detects deadlock and livelock by composing the state space of process events specified in FSP. It detects other negative emergent system properties by determining whether undesirable states can be reached. As well as automatically detecting errors, the tool allows the user to manually step through a system's composed state space – choosing from those events that are possible in the current system state.

LTSA is freely available ([25]), and as it is used as a teaching tool, it is easy to use. By graphically showing state models and animating state transitions, the tool helps users to understand the complex consequences of simple processes' combined events. LTSA extensions (that are not used in the work presented here) support code generation from message sequence charts (including negative scenarios), prototype animation to assess an application's usability, and quantified stochastic annotation to generate simulation statistics.

Event models are therefore an established method for evaluating concurrency risks. Historically then have been applied to low level, critical system designs (for example, in operating systems and embedded real-time systems). In contrast we apply them to high level, abstract designs of grid systems to assess whether the operational requirements (discussed in subsection 1.1) are met. Our models therefore mitigate the risk of failing to meet requirements for the general operation of the whole system.

2.2 Modelling process

This section introduces a reliable, repeatable process for specifying event driven models of grid systems. The technique has evolved through our experience of developing models in FSP, described below (section 3). A complete iteration of the model lifecycle should take 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 needn't 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 reevaluation 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. Instead, we wish to emphasis the effort involved in identifying faults and modifying models at stage 4 – discussed for each model version in the worked example. We also stress that step 5 is essential for the intelligent application of conclusions derived from the modelling effort.

2.3 Worked example

The process described above (section 2.2) is used to develop a demonstration model system in this section. Though simple, the system is non-trivial and includes design elements used in grid systems. The FSP code for the model is presented for 3 steps: the serial implementation, the parallel implementation and a refined implementation. Modifications to the code between model versions are highlighted by marking the unchanged code in grey.

Each model is discussed in four parts. First, the operational target and general design concerns of the modeler at the given step are described. Next, the language features introduced in the model version are explained. 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.

Step 1 – Intention of modelling a service

The tutorial model represents a service that actions asynchronous user tasks; this may represent a database query or a computation job. It is required to serve several users. Users may submit several tasks. It is also required that the

Figure 3: Initial FSP service 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 ).
```

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 the brokers of the EGSO models, 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 in a genuine system. By hiding back-end complexity in this way, grid systems can manage dynamic resource 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 our data-grid models.

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 version in fig. 3.

Language features: In FSP (capitalized) processes are defined by a sequence of (lower case) events. The state model loops, indicated by returning to the process name. Termination at a 'STOP' keyword prevents the analysis tool identifying deadlocks.

In our 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). LTSA composes the simple processes' state spaces to analyze their complex interactions, testing for safety and progress. It supports manual animation of possible event paths, with graphical presentation of the processes' states.

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). Shared synchronous events may be prefixed by the letters of the processes communicating, indicating direction; for example, 'us_request' and 'su_result' in the above model. 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.

Step 3 – Concurrent users and tasks

Model goal: 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 in fig. 3 are implemented in the second version in fig. 4.

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 in curly bracers after the composed process, with equivalent event pairs separated by a slash; these are necessary to indicate the prefixes

Figure 4: Service model with concurrent users and tasks.

```
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 }.
```

are equivalent to the suffixes for events synchronized between pairs of processes that both have multiple instances.

Debugging: Errors when matching event prefixes are common, and cause unexpected events. These should 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` , 'request' would be possible before the user process had made the 'submit' event.

Event matching errors can also be introduced easily 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 in fig. 6). 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.

If errors are made in the range of indexed events, processes can carry out inappropriate events. Specifically, if a process should only use a subset of an event index range but is defined for the full range, it can make a state transition that should be under the control of another process. In this tutorial, if task events were only distinguished by an index, such an error could represent the service process submitting a task.

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 [16]). This pattern scales well when several functions are required to complete a task, and service points

Figure 5: Refined FSP service model.

```
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 ] }
```

action several task types. It therefore models a grid system's flexible workflow management, with dynamic resources supporting heterogeneous applications.

Step 4 – Refinement with a semaphore.

Model goal: Analyzing the model shown in fig. 4 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 of fig. 4, is avoided by adding a semaphore as in fig. 5. 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.

Language features: Process state suffixes (for example, `SEMA[ b:0..1 ]`) and conditional event paths (using the keyword 'when') are introduced in this model. These suffixes allow a process to hold different states between event sequences. The initial semaphore state is false, so a claim will successfully raise it and changes 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 synchronized 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, we may have exposed possible problems in the two component design; 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, 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 $N \times M$ 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 our task process in fig. 6. As well as determining transitions representing 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 build using this pattern.

Figure 6: Task process with integer state parameter.

```
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 ] ).
```

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 modeler may highlight features of the listing in fig. 5. 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. They 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 listed in fig. 5 has 2^{27} states. (LTSA cannot compose the 2^{41} states for 3 users and 4 tasks on our small development machine; the Java 1.4.1 run-time environment runs out of memory at 133MB.) In this case, the modeler 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.

3 Evidence

This section discusses models developed at 4 different design stages – 3 within the lifecycle of the EGSO project, the last in AstroGrid. It demonstrates that dynamic models have been understood by our colleagues and have improved our grid projects' designs. Those who are interested in software engineering methods can use this discussion to evaluate our technique.

The narrative presentation of this concrete evidence reflects the previous section's abstract introductory material. For each stage we describe: the project state, the model implementation, language limitations noted, other observations and details of how the model was communicated to colleagues. These indicate the model development method being applied – from the system specification input to presentation, via model refinement.

Before presenting this material for each lifecycle stage, we revise the purpose of modelling – rehearsing the criteria by which our technique must be judged.

3.1 Model interpretation

The general purpose of modelling was presented above (section 2.1). We now discuss further why software engineers are motivated to model systems before implementing them. This reveals what we hoped to achieve by representing data-grids in FSP.

As a system's performance and maintenance qualities may be determined by the design of the whole system, it may be impossible to correct endemic weakness after deployment by replacing components. Therefore, failing to satisfy requirements for quality early in the project lifecycle may drastically reduce the long-term utility of the system. The benefit of risk management and prompt resolution of faults rises with the complexity of the system, with distributed and legacy systems presenting special challenges.

System models may be implemented before the main development effort. Software models should reflect the real system's essential properties and capture its fundamental operation in a simple way. Modelling should expose weak design, guide improvement and help common understanding. The modelling technique itself may be judged on its demonstrable link to reality, its representation of properties that introduce risk, and its clarity. If these criteria are met, stakeholders can accept evidence from the model with confidence to amend their designs (or expectations).

Stage 1 – *EGSO* Vision

Project state Initially the solar physics community's vision of their data-grid was based on elements of existing distributed systems. Generic use cases and informal component diagrams were envisioned to enable transparent access to distributed data stores. When the EGSO system was imagined, it was thought that data resources would be described in distributed catalogues (databases of metadata) and that a peer-to-peer network would allow data products to be shared. It was unclear whether distributed independent entities would be able to work together as envisioned, so 4 models were implemented to demonstrate the sketched systems could do essential data-grid tasks.

Implementation Model development took 5 working days. Each of the 4 models tackled a specific challenge, as listed below, to remove the risk of unfamiliar designs failing as early as possible.

1. A 'layer' model demonstrated that a common type of service state could 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.

2. A 'queue' model demonstrated that multiple clients could concurrently submit queries to a shared queue, whilst tasks were fairly scheduled to a back end service provider. A middle tier broker therefore managed a shared resource.

3. A 'secure' model demonstrated how requests through a service portal could be guarded with a check of clients' identities by a third party. Each client's access status was held and administered by the independent authority.

4. A 'tier' model demonstrated how static and dynamic metadata records of data providers' resources and status enabled voluntary location (and migration) transparency. The client could specify a preferred provider, but would be routed to another with the same content if their first choice was unavailable. It became apparent from the model that transparency was not symmetrical; providers necessarily maintained the consumers' identities associated with queries.

Limitations Shortcomings in the methodology were noted at this stage. It was found that the models represented the performance of scheduling and security concerns weakly. Stochastic LTSA or another simulation language that supports continuously variable annotation would be better suited to evaluate algorithm performance. Specialist annotated object analysis, as described in [24] and carried out for data-grids (including EGSO) in [14], would provide greater confidence that a distributed design upheld security constraints. Additionally, these models did 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.

Observations Despite the shortcomings, the simple fact that these models could be implemented and animated to reach target states, such as the client receiving a resolved query, was sufficient justification of the project goals and informal system architecture at this stage. The models did not reach end states that would prevent progress, proving that there was no logical restriction to the scalability and reliability of these basic designs.

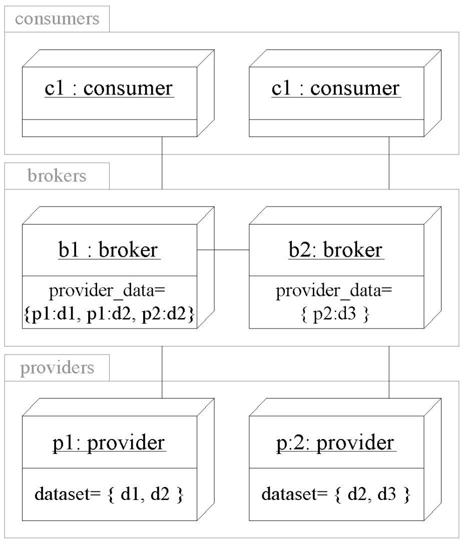
Communication When the models were animated in the LTSA tool, the scientific users could understand how the design met their requirements. Project managers and engineers recognized that the models captured views of the conceptual system represented by the informal diagrams. By having operational models very early in the development lifecycle, stakeholders therefore gained confidence that the system being envisioned was valid.

Stage 2 – *EGSO* Architecture

Project state The EGSO requirements were finalized at the same time as the initial system architecture was presented. The top-level design used UML diagrams to describe 12 sub-systems and 21 components within 3 architectural roles: the consumer, broker and provider (as described for EGSO in section 1.2). The design elements' static dependencies and interfaces were given, but communication methods were not specified.

To generate a dynamic model that represented the architecture, we derived 47 detailed scenarios from the documented requirements. These instances of system activity highlighted different types of functionality and demonstrated non-functional properties through operation and maintenance. 8 core scenarios were identified; others merely refined the behavior of these or captured lower

Figure 7: Informal UML deployment diagram of the EGSO architecture model, showing the initial metadata state used in tests.



priority behavior. 3 of these represented system behavior (optimization, security and protocol flexibility) that we already had learnt could not be modelled satisfactorily. Another used an analysis service that could only be modelled in the same way as the data location scenario.

Implementation The 4 remaining core scenarios represented: transparent data location, query resolution by distributed metadata, dynamic resource growth, and query rerouting on provider failure. From these we generated a single model with concurrent instances of the architectural elements, 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 processes. The final models' 32 types of event were associated in a many to many relation with architectural components; the majority represented interaction between pairs of components. Sub-systems in the architecture that represented internal mechanisms (hidden by dependent components with interfaces) were not modelled, reducing complexity; they couldn't affect the safe concurrent progress of the system.

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 7.

Limitations With multiple instances of every role shown in figure 7, LTSA could not analyze the combined state space of 2^{78} transitions. Safety could only be checked by modifying the model parameters so that there were duplicate instances of just one role at a time (though 2 brokers were necessary to represent query forwarding). By testing safety for each role independently, it could be shown that communication semaphores would be needed to prevent broker deadlock.

Observations 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 (these could even be crudely associated with well known patterns, as in [18]).

The differences between the architecture components and model events, and their complex relationship, emphasizes 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.

By animating the core scenarios, we demonstrated that the implemented architecture hid 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 decentralized, dynamically scalable and robust. In these operations, the represented data-grid tasks made concurrent progress without interfering with each other's states.

Communication The scenarios derived from the requirements that helped the development and testing of the model at this stage may be reused as system test descriptions. This reinforces the relationship between software lifecycle stages whereby earlier design stages are associated with later testing, working in towards the central implementation stage (the V-diagram [34]). By basing the model on both scenarios and the architectural components, it also 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, and all project stakeholders accepted its demonstration that the design would behave well.

Stage 3 – *EGSO* Interface Design

Project state The EGSO architecture was refined to 3 design documents for the consumer, broker and provider roles, with supporting documentation for the scientific data model and shared interaction sub-system. The broker was perceived to be the sub-system that was essential for the reliable operation of the system, and its design was finalized ahead of the others. Its design included message sequence charts for interaction with the other roles, therefore encompassing much of the system architecture whilst having little domain specific content and few isolated components. For these reasons, its design was the primary input to the next state of modelling.

Implementation The 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. We did not use the LTSA extension for drawing message sequence charts [37], as its generated FSP code was complex and hard to manually modify.

3 models were developed over 9 working days. We initially modelled a single instance of each role (with a slave broker used as the target of all broker to broker interaction) and 77 event types representing the message sequence charts. 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 not included in the second model. The behavior expressed was still richer than the models of previous stages, as 76 event types were implemented in paths conditional on query state.

The third model was implemented to evaluate an alternative design still being considered; distributed metadata implied brokers should forward unresolved queries until all peers had failed. Only the broker nodes (with reduced functionality) and the semaphores were represented with just 16 event types. This was sufficient to capture the contrasting properties of the alternative design.

Limitations Message sequence charts are not precisely captured by encoding FSP directly; synchronous events indicate a message exchange, but not its direction (from the source process to the sink). However, an extension to LTSA can be used to graphically present such charts. Even without this, events that are not synchronized between processes (and may be hidden at composition) represent work done by the message originator or consumer. The latter may also represent the functional work of the server, necessary to model concurrency.

Observations 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 parameterization with the forwarding node, extra query history was required – this could not be easily represented in FSP.

The modelling completed at this stage therefore validated the interfaces and evaluated alternative designs. These designs are also domain independent, and demonstrate behavior common to many data-grids – so they again indicate design patterns; the second model represents a generic resource metadata management solution, and the third model a peer to peer service discovery network.

Communication We successfully used the model to argue for some modifications to the EGSO design. 8 undocumented messages had been added to the model. These included 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 symmetrical entities that act both as client and server, at risk of deadlock or requests loss. The second model from this stage may be maintained in parallel with interface development, so that future design changes can also be validated against the design original goals.

Stage 4 – *AstroGrid* Detailed Design

Project state 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' interaction via public methods; these were complex and subject to change. However, by discussing the distributed interaction and exploring design risks with the projects' software engineers, we extracted a message sequence chart that spanned the system elements for the essential data-grid task of query resolution and delivery of an analyzed data product.

Implementation Just 2 working days were spent developing 2 models based directly on the message sequence chart. As at the previous stage, we created process synchronization events for documented messages and added hidden events for 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 second model introduced concurrency to explore a risky circular dependency, implementing the 3 classes involved with a job state process and a job factory. These 5 entities shared 12 types of events.

Observations The first model animated the essential message sequence chart, demonstrating that the message set was complete as no more messages were needed to prompt or guard required process activity. The second model did reveal a possible deadlock in the circular dependency of 3 job control processes when there are as many active jobs as entities.

Communication Discussion with the engineers distinguished stateful from asynchronous communication interfaces. Therefore, the demonstrated deadlock should not be a risk, 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 behavior 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; this has yet to be modelled to demonstrate the logical reliability of the design. The cycle of design, modelling and design review will therefore need at least one more iteration in this project.

In fact, the engineers are 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 higher priority risk than component interaction failure, we plan to use stochastic FSP instead of the basic language described in this chapter.

General Limitations An advanced EGSO model safeguarded communication with semaphores, whilst the AstroGrid model fails to synchronize the distributed state of tasks across components. AstroGrid's design side-steps synchronization failure by accommodating lost messages, whilst EGSO must implement blocking communication with its critical brokers. It is therefore clear that reliable progress in a decentralized, scalable data-grid architecture is sensitive to connector design. However, FSP has only represented connection oriented communication with synchronized event in our models.

Some ADLs have rich semantics for connector typing, which may capture different data-grid component connection strategies. However, despite FSP's simplicity and the experience given 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. Using such connector template processes adds further complexity though, restricting the representation of complex systems.

4 Conclusion

The experience (presented in section 3) of applying the modelling technique (demonstrated in section 2) shows how our method for assessing data-grid architecture is applied within the early software lifecycles. The process and our findings are summarized below in section 4.1; we conclude that our method helps the reliable application of LTSA. Section 4.2 contrasts our method with other modelling techniques, and 4.3 discusses the direction of our work.

4.1 Summary

Formal representation of components' interaction verifies proposed system designs. By combining and analyzing the interacting events of simple processes, LTSA can check complex concurrent system components work together, avoiding communication conflicts and reaching desirable system states. We built models from requirements scenarios, interface specification and message sequence charts, then refined their concurrency and demonstrated test event sequences. In this way, we could reliably use LTSA to faithfully represent systems, verify their designs and demonstrate our findings.

We have shown that data-grids are a challenging domain that need front-loaded development to reduce risk. By applying our modelling technique to 2 projects at 4 stages in their lifecycle, we've demonstrated that our method is feasible. The conclusions drawn for each stage, that demonstrate the method is also valuable, are listed below.

1. The 4 conceptual models demonstrated that: service providers can be protected against infrastructure faults and insecure access, a broker enables fair scheduling, users can reliably access hidden distributed resources, and there is no logical constraint to these entities scalability.

2. The architectural scenario model also demonstrated how users could simply use a decentralized network; metadata management (for both the persistent association of providers to data sets and their volatile availability) enabled dynamic growth and robust service.

3. The 3 interface design models demonstrated that the design reliably made progress; they also uncovered hidden complexity of one design option, found missing messages in interactions, and highlighted the protection required for symmetric component dependencies.

4. The 2 models of the object interaction for data-grid query resolution demonstrated the risk of stateful connection deadlock (where there was a circular dependency) – avoided by non-deterministic asynchronous messaging.

The technique was especially useful in the early project, bridging requirements and architectural design; its formal demonstration of the initial concepts increased confidence.

Grid systems are characterized by complex, concurrent transactions where user progress and resource management depends on the system's dynamic distributed status. LTSA captures the resulting non-deterministic interaction that makes grid design error prone. Our method reliably and rapidly captures systems' emergent behavior, so it can evaluate grid designs and communicate findings well.

4.2 Related work

Models are widely used in software engineering; 5 common ways of applying models are listed below.

1. Informal box and line diagrams are easy to generate and understand, and are common early in project's life. Their components may be generated for the major envisioned tasks of the conceptual system. Several different sketches may be used to guide imagination and discussions between customer and engineering representatives (applied in object modelling by [28]). Their ambiguity means that they cannot strongly support analysis of system properties or guide implementation.

2. Architecture description languages support formal analysis of a high level design's components and connectors. An ADL may be chosen because it expresses a high-risk area well, for example by supporting heterogeneous connector protocols or domain specific performance properties. Models generated may be analyzed by tools that employ formal methods to prove constraints or goal satisfaction, but they are typically hard to generate and understand. ([29] reviews many ADLs, and [32] introduces architectural modelling.)

3. Object modelling may be employed from the initial system conceptualization and architectural design (where MDA is applied [17]), but most widely used at detailed design (where UML is widely used [7]). The abstract and concrete properties and relationships of objects in the system can be captured unambiguously and intelligibly. Basic analysis (for example, for design consistency) and quality guidance (for example, for decoupled encapsulation) is ensured, and extra mechanisms (such as OCL formal methods) may be applied.

4. Prototypes may be implemented to demonstrate a sub-set of the system's behavior or as a stepping-stone toward full implementation (in evolutionary development [6]). They may tackle high-risk system components and interfaces, or be generated automatically by development tools. The prototype's accuracy may therefore converge on the actual system representation, so it can be analyzed by increasingly realistic system tests.

5. Simulation typically focuses on one view of system behavior, such as performance. Its implementation may include a model of the system's domain, but shouldn't be reused in the real system – in contrast with a prototype. The statistical analysis of a valid simulation's operation unambiguously demonstrates the real system's properties.

All of these methods may represent both static and dynamic views of the system; object modelling and prototyping capture dynamic behavior better than typical informal diagrams and ADL specifications, but only simulation is targeted at exposing the emergent properties of live operation.

The modelling technique that we have used for data-grids is strictly dynamic, based on formal analysis guided at combinatorial state space analysis. It is designed to work with a traditional ADL, Darwin [26], that represents static components with diverse connectors hierarchically. As it is also a teaching aid, it is easier to understand than other formal methods and tools.

All of the 5 modelling techniques introduced above have been applied in grid projects, and all but simulation have been used specifically in AstroGrid and EGSO. This section will review this effort, making reference to our projects where possible. Other artifacts of the astronomy data-grids' early software lifecycle will be mentioned, as these give the background against which our modelling effort is judged. The overview should also demonstrate that our models represent a unique investigation.

1. Initially the EGSO project was specified by its proposals. To help elicit requirements and conceptualize the system implementation, informal component diagrams were drawn. These were inspired by the current state of on-line archives, distributed system architecture and peer to peer networks.

   These informal models were used in discussions with the scientific community to refine requirements. The most advanced model was documented in a systems concepts document that supported requirements analysis.

2. Our preliminary modelling effort for EGSO attempted to describe the informal diagrams in an ADL. We used the flexible framework of ACME [19] with the intention of specifying detail in Wright [1] (which supports the flexible connector typing required for a complex distributed system). Generic types were identified; 3 connectors were defined for stateful interaction, message dispatch, and high volume data stream, and 4 components (defined by their connectors) were descriptively named origin, consumer, filter and data.

   Other work has distinguished grids of computation resources from traditional distributed systems using formal specification [36]. This work emphasizes the transparent access, also essential for data-grids. It uses unambiguous rules for mapping application processes to abstract resources and defining scalable access policies. These would be useful to prove a middleware design offers a genuine grid solution, or as the foundation for formal analysis of specific systems' architecture.

However, formal methods' complexity makes them labor intensive and hard to apply correctly. In our experience, their awkwardness is aggravated by poor tool support and the languages' obscurity. Stakeholders also had great difficulty understanding the method and the significance of our preliminary findings. For these reasons, such heavyweight analysis is not as useful as our method in the rapid evaluation of project designs.

3. Use cases, the first step of object modelling, were generated for EGSO and AstroGrid. They guided domain modelling for EGSO, whilst AstroGrid developers went further to generate preliminary class and message sequence diagrams. The EGSO architecture was specified in UML component diagrams, using MDA techniques.

   As design proceeded, AstroGrid developed detailed message sequence charts and class diagrams for components, whilst EGSO used message sequence charts to design interfaces. In all cases, expressing the static and dynamic views of the system was feasible and generally comprehensible. These 2 projects therefore demonstrate that state of the art object modelling can be successfully used for data-grids; the method is applied hierarchically, documenting abstract and concrete design decisions as they are agreed.

   Object models usefully assign functional responsibility to decoupled elements and may demonstrate design patterns that are known to support desirable properties (for example, from [18], a factory may generate new services and a mediator may hide heterogeneity). However, UML models have not been used in either project to test the quality of service properties that were noted to be at risk in section 1.

4. Both AstroGrid and EGSO have generated prototypes for diverse reasons. User interfaces that access minimal scientific data demonstrated (to users) how searches may be guided and data-sets joined, and (to engineers) how existing technology may be used to implement flexible thin clients. These cosmetic prototypes were documented with requirements and technology reviews, but their implementation did not demonstrate the required reliability and scalability of real data-grid applications.

   In EGSO, prototypes were also implemented to demonstrate the operation and interaction of architectural components, but these were straw-men used to guide further design rather than skeletons of a genuine implementation that demonstrated testable system properties. All these prototypes can therefore only remove uncertainty in existing early project artifacts.

   In AstroGrid though, genuine development challenges were faced by prototypes; for example, a web service accessed a simple registry to demonstrate remote querying of a wrapped data interface using a scalable, reliable technology (Apache Axis). By developing functionally incomplete but testable components before the design was finalized, AstroGrid demonstrate how prototypes can reduce risk towards the start of the project lifecycle. However, such prototypes must become almost as sophisticated as the final system before they can demonstrate some of the required system performance qualities.

5. Simulation has yet to be used for AstroGrid or EGSO, but is being used to evaluate generic grid tools. For example, stochastic discrete event simulation has tested coupled scheduling and data migration strategies [30], and Internet topology simulation has tested a non-deterministic resource query mechanism [10].

   Such models demonstrate dynamic system properties (notably performance and scalability) well, but are weakened by the current lack of real data to base the simulation parameters on. They have also only been applied as one off experiments to demonstrate a given tool's algorithm; a generally applicable methodology has not yet emerged.

4.3 Direction

Our method may be generally applied to model informal system descriptions, static architectural designs and the message sequence charts of detailed design. The method may benefit any innovative software system with distributed processes that are at risk of failing to make the coordinated progress required to uphold functionality and quality of service. Relatively little effort is required to derive an accurate and understandable dynamic model that explain and validate common software design artifacts. Our experience of reusing model elements indicates data-grid model patterns (like those highlighted throughout section 2) could be abstracted.

Our own investigation will continue as the AstroGrid and EGSO projects develop. If current designs are faithfully implemented, we will observe whether the models' properties are reproduced in the deployed systems. We will also maintain the models to track and validate design changes, verifying whether FSP can capture the real world complexities that caused the modifications. We will monitor how the designs of other data-grid projects meet this new domain's special challenges, hoping that architectural styles and design patterns can be abstracted and modelled to enable the reliable production of high quality systems.

5 Acknowledgements

EGSO is funded by the European Information Society Technologies (IST) Framework Programme 5 (FP5), grant IST-2001-32409. Thanks to all of the EGSO team, managed by Bob Bentley. Special thanks to Clare Gryce and Nathan Ching for contributions to the requirements analysis. AstroGrid is funded by the UK Particle Physics and Astronomy Research Council (PPARC), grant PPA/G/S/2001/00438. Many thanks to Keith Noddle and his development team leaders. Joe Lewis-Bowen's PhD is also funded by PPARC, grant PPA-/S/E/2001/03334; he thanks his supervisor, Louise Harra.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[2] AstroGrid http://www.astrogrid.org/.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[4] R. D. Bentley and S. L. Freeland. SolarSoft – An Analysis Environment for Solar Physics. In *A Crossroad for European Solar and Heliospheric Physics – SP 417*, page 225. ESA Publication, March 1998.

[5] BIRN (Biomedical Informatics Research Network) http://www.nbirn.net.

[6] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[7] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1998.

[8] CDS (Centre de Données astronomiques de Strasbourg) http://cdsweb.u-strasbg.fr/.

[9] A. Csillaghy, D. M. Zarro, and S. L. Freeland. Steps Towards a Virtual Solar Observatory. *IEEE Signal Processing Magazine*, 18(2):41–48, 2001.

[10] P. Dinda and D. Lu. Nondeterministic Queries in a Relational Grid Information Service. In *Proceedings of Supercomputing 2003 (SC2003)*, November 2003.

[11] EGSO (European Grid of Solar Observations) http://www.egso.org/.

[12] European Data Grid http://eu-datagrid.web.cern.ch.

[13] A. Finkelstein, J. Kramer, B. Nuseibeh, and M. Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, 1992.

[14] I. Flechais and M. A. Sasse. Developing Secure and Usable Software. In *Proceedings of OT2003*, March 2003.

[15] I. Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[16] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer*, 35(6), 2002.

[17] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley and Sons Ltd, 2003.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns : Micro-architectures for Reusable Object-oriented Software*. Addison Wesley, 1994.

[19] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[20] GriPhyN (Grid Physics Network) http://www.griphyn.org.

[21] C. Gryce, A. Finkelstein, and J. Lewis-Bowen. Relating Requirements and Architectures : A Study of Data-grids. Submitted to Journal of Grid Computing, 2003.

[22] M. Jackson. *Software Requirements and Specifications.* Addison Wesley, 1995.

[23] K. G. Jeffery. Knowledge, Information and Data, 2000.

[24] J. Jürjens. UMLsec: Extending UML for Secure Systems Development, 2002.

[25] LTSA (Labelled Transition System Analyser) http://www.doc.ic.ac.uk-/ltsa/.

[26] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of 5th ESEC*, pages 137–153, 1995.

[27] J. Magee and J. Kramer. *Concurrency : State Models and Java Programs.* John Wiley and Sons Ltd, 1999.

[28] L. Mathiassen, A. Nuk-Madsen, P. A. Nielsen, and J. Stage. *Object Oriented Analysis and Design.* Marco, Hasseris Bymidte 21, 9000 Aalborg, Denmark, 2000.

[29] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[30] K. Ranganathan and I. Foster. Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids. *Journal of Grid Computing*, 1(1), 2003.

[31] SDAC (Solar Data Analysis Center) http://umbra.nascom.nasa.gov/sdac.-html.

[32] M. Shaw and D. Garlan. *Software Architectures : Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[33] Starlink Project http://www.starlink.rl.ac.uk/.

[34] R. Stevens, P. Brook, K. Jackson, and S. Arnold. *Systems Engineering: Coping With Complexity).* Prentice Hall, 1998.

[35] H. Stockinger. Distributed Database Management Systems and the Data Grid. In *Proceedings of 18th IEEE Symposium on Mass Storage Systems*, 2001.

[36] V. Sunderam and Z. Nemeth. A Formal Framework for Defining Grid Systems. In *Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[37] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios. In *Proceedings of 9th TACAS Conference*, April 2003.