The software infrastructure for a Distributed System Factory

by Walt Scacchi

This paper describes an innovative approach to the construction, application and deployment of software factories. Based on experience in creating and evolving the System Factory project at USC, we present a new experimental project, whose technological and organisational objectives are wide-ranging. This effort is called the Distributed System Factory (DSF) project. The DSF project is intended to provide a software infrastructure suitable for engineering large-scale software systems with dispersed teams working over wide-area networks. This software infrastructure is the central focus of this paper. As such, this paper describes the information structures that can be used to model and create the infrastructure, as well as target software applications. It also describes an electronic market-place of logically centralised software services which populate and execute within this infrastructure. Finally, it describes a brief view of how the DSF project can grow to accommodate academic and industrial research groups.

1 Introduction

In general, our interest is in applying and deploying software factories in industrial settings. In particular, our approach is to apply and deploy software factories in large and expanding academic settings, as a way to realise our general interest. This is a report on the objectives and status of development in the Distributed System Factory (DSF) project. The aim of this DSF project is to create, apply and deploy a software engineering infrastructure that can support a distributed wide-area network of software factories. Such an infrastructure is needed to support what might be called as a national software engineering 'collaboratory' [1, 2]. This report focuses on the DSF software infrastructure. It does so in order to emphasise that greater opportunities for deploying and applying software factories in industrial settings lie in establishing large-scale collaborations between industrial and academic efforts.

1.1 *Going beyond industrial software factories*

In general terms, software factories in Europe, Japan and the US face a common set of problems. These problems include how to

- support the distributed engineering of software systems across multiple organisation locations.
- support the concurrent engineering of large software systems throughout their life-cycle.
- support rapid large-scale software R&D efforts.
- facilitate software technology transfer and transition.
- educate and train new software engineers.
- assess and evaluate current software development practices and new software technologies.
- demonstrate new software engineering technologies on practical applications.
- make effective use of local-area and wide-area networks of heterogeneous computer systems.
- create national or international markets for software tools.
- integrate various internally and externally developed software engineering tools, techniques and management strategies to form coherent software engineering environments.

Clearly, these are all 'big' problems that do not have simple well defined solutions. However, there are differences between the various software factory projects, to the extent that they either seek to find effective solutions to these problems or view these problems as being separable versus inter-related.

There are some substantial problems that cannot be easily addressed by industrial software factory projects. Principal among these are the dilemmas faced in conducting large-scale experiments with new software engineering technologies or teamwork structures. These experiments require the deployment and application of new technologies in real full-scale development projects. Current industrial practice in the US indicates that an effort of five–seven software

engineers per year represents a $1 million cost to the host organisation. Large-scale development projects often involve project staffs of 10–100+ engineers, grouped into teams in one/many location(s), who work on development projects that take months to complete, and whose resultant programs are usually in the range of 25000 to 500000 source-code statements. Therefore, there is a substantial problem in the high cost and associated high risk of major economic losses that can arise if the experimental technologies fail, or if the project that employs them fails to yield a viable product.

The ongoing presence of these costs and risks seem to make such large-scale *in situ* experimentation with new software technologies unlikely for most industrial organisations. However, in our view, these same kinds of experiments can be undertaken in academic settings on a large scale without comparable high costs and risks. This has been demonstrated most convincingly in the USC System Factory project [3]. This project has, over the past ten years, involved between 20–90 graduate computer science students at a time in the engineering of large software systems, with development schedules that typically span 4–12 months. In turn, this project has engineered software tools and domain-specific applications whose size vary between 5000 to over 100000 source-code statements. In many cases, this software has been developed under contract to, delivered to and applied by software technologists in industrial organisations and government laboratories [3].

Our experiences in the USC SF project indicates that academic groups can undertake large-scale experimentation with new software engineering technologies and teamwork structures that will not, in practice, be undertaken in industrial settings. To us, this means that there is a potential to realise substantial benefits when industrial organisations collaborate with academic groups to conduct large-scale software engineering experiments. Further, we believe that it is possible for such collaborations between, and among, academic and industrial organisations to occur on an even larger scale, in forms similar to what we call a Distributed System Factory, a network of system factory-like projects. Accordingly, the DSF is conceived as an experiment in distributed large-scale software engineering and application. This paper provides a description of the software infrastructure that we are developing which can support such large collaborations.

1.2 Overview

In this paper, we provide a brief summary of the USC SF project, and we describe our view of a DSF, in order to substantiate the development objectives of the DSF instructure that we are now building. The software infrastructure itself is described in terms of the technologies that we have already prototyped, as well as the status of those now in construction. We also present our view of the opportunities and barriers that exist in the widespread deployment and application of DSF-like projects.

2 The USC System Factory project

Started in 1981, the USC SF project has been one of the pioneering efforts focused on conducting exploratory and applied research of large software systems *in situ*.* As part of this project, we have undertaken a number of empirical studies and surveys of software engineering practices in a variety of industrial, commercial, government and academic settings, including the USC SF project itself [4–9]. This focus on the organisational and social dimensions (the people side) of the software engineering process, together with the technological side, has often given us unique insights into what can be accomplished with advanced software engineering processes, given limited resources and other circumstantial constraints. However, our interests go beyond this.

2.1 Organisational engineering in the USC System Factory project

The USC SF project is also centrally focused on providing software engineering education for a large number of graduate students [10]. Over a period of ten years, we have had more than 600 MSC and PhD computer science students participate in the research, development and training activities that we regularly conduct. In our view, this is an essential component of the SF research effort, since the educational experience provides hands-on involvement in the specification, prototyping, design, implementation, testing, demonstration, use and evolution of advanced software engineering tools and techniques. Such involvement in the technical decision-making and workgroup interactions that these engineering processes entail generally gives these people a clear understanding of what is possible, what problems arise, what trade-offs are made, and what actions improve or degrade the quality of the software technologies they produce. This form of involvement is significant in our view for three reasons.

First, the active participation of our student developers and users regularly leads to new insights about the software engineering technology that we develop and use. These insights, in turn, give rise to new versions of the software components or integration mechanisms that we subsequently develop. Making these insights available to subsequent development efforts is realised through deliverable documents, which specifically address future enhancements, and through the five–ten people who persist within the USC SF project across multiple SF development cycles. Thus, this represents an organisational mechanism that supports the evolutionary growth of the SF instructure through intra-organisational technology transfer and transition [3]. In turn, this same mechanism is also contributing to the development of the DSF infrastructure.

Secondly, the vast majority of the USC SF graduate students take positions in industrial firms, where they will be called on to provide comments or advice on what new software technologies to use or acquire in order to support the development of industrial software applications. The hands-on involvement, participatory development and 'knowledge transfer' that these students experience have been found, in both the USC SF project and elsewhere, to be effective in the transfer of advanced software technologies into industrial practice. Similarly, this facilitates the organisation transitions that integrate new technologies into workplaces in

* Coincidentally, these are also among the actions now recommended by the Computer Science and Technology Board in the US for what modes of research in software engineering should be pursued in the coming decade [2].

ways that enhance working conditions, workgroup collaboration and job satisfaction [3, 7, 9, 11–15].

Finally, the experiences and insights that we have gained through the ongoing cyclic development and evolution of the SF infrastructure have helped to motivate us to more systematically codify our knowledge into more accessible forms. For example, we have analysed and documented the strategies that we find successful for managing both small groups and large teams of software engineers [3, 8, 16]. In recent years, we have also developed a process specification language and environment to model, simulate and analyse how people work with the various software tools and development techniques that are part of the USC SF [17, 18]. This effort seeks to

- provide prescriptive software process support.
- model and formalise the empirically observable processes [19, 20], including what happens when organisatonal software processes breakdown or fail.
- support the accommodations or negotiations that people engage in to try to sort things out in response to unexpected conditions or breakdowns [6, 18, 21].
- simulate and replay these models to help explore alternative conditions that might improve current practices [18, 21, 22].

2.2 Software engineering in the USC System Factory project

Our concern with software engineering technology in the USC SF project is focused on the development, use and evolution of large software system tools, applications and environments. Each USC SF development effort produces inter-related multi-version documents that describe the life-cycle (the requirements, functional specifications, designs, test plans, implementations, deployment guides, user manuals and maintenance guides) for the system being developed. As such, we have found it more efficient for the structure and, to some extent, the content of these documents to be standardised in order to facilitate parallel/multi-person development, review and quality assurance. However, the form and contents of these document standards has evolved over time to reflect emerging needs and technological advances.

Software systems developed in the USC SF project are documented using both fully structured descriptions (e.g. functional specifications, designs and source code), whose syntax and semantics can be formally defined and automatically analysed, and weakly structured descriptions (narrative requirements, user manuals and maintenance guides), whose content can be text-processed and understood by people but which may be ambiguous and incomplete. We have found that software hypertext mechanisms are particularly well suited for organising, accessing, browsing through and inspecting these kinds of online documents [23, 24]. In addition, we have added automated mechanisms to identify and trace relationships across multiple semi-structured descriptions to varying degrees, in order to configure, validate and maintain the consistency of inter-related software descriptions as they evolve [25, 26].

Large software engineering projects in the USC SF project produce encyclopaedic volumes of semi-structured and inter-related descriptions. As such, the production of system life-cycle documentation represents a substantial fraction of the system development effort. However, this is often necessary, since the large scale of the system development effort often implies that the system's documentation will be the focal medium for co-ordinating engineering tasks and information flows among developers working at different times and places. In general, software development requires the substantial co-ordination of people (with different skill levels), automated tools, multiple system descriptions and other organisational resources. The USC SF project is no different in this regard. Accordingly, our use of electronic mail and bulletin boards to help co-ordinate development activities is increasingly essential, but these tools lack knowledge of the software development products, processes, workplaces and their inter-relationships that are discussed when using these media. However, we have begun to experiment with knowledge-based software technologies to represent and manipulate this kind of knowledge [17, 18]]. Similarly, to increase the rate and quality of software system production, we have been investigating other technologies, including rule-based message management systems [27, 28], groupware technologies [29], and on-line catalogues of reusable software components [23, 30, 31].

2.3 Forming a superset of the SF project: a network of software factories

Given the ten years of research, development and educational accomplishments that we have achieved in the USC SF project, we have become convinced that other academic and industrial organisations can benefit in similar ways, by putting system factory-like projects in motion. Having said this, we are trying to help realise this objective by evolving the USC SF infrastructure into one that can support and integrate a wide-area network of system factory-like projects. This objective is in anticipation of advances in communications networks that can be expected to provide orders of magnitude greater bandwidth and throughput, compared to what is widely available today. More importantly, we are interested in increasing the scale of participation in system factory-like experiments and therefore hope to similarly increase the scale of the accomplishments possible.

Thus, we now focus our attention on the requirements and capabilities of the software infrastructure for a DSF project that we have been developing since 1989.

3 The Distributed System Factory project

The principal requirement for the DSF project is to support a software engineering process, where everything is potentially distributed over a wide-area, through a loosely coupled infrastructure of logically centralised services.† Thus, this requires an explicit framework that models what software entities can be distributed, as well as what services process them. Further, a major requirement is that such a software infrastructure must not be subject to global control from a single administrative authority. Instead, our intention is that such an infrastructure must be loosely coupled,

† Logically centralised indicates that computational services appear to a user (or another program) as local, rather than distributed, across a wide-area network. Thus, particular implementations of logical service centralisation may involve accessing these services through a database that maintains information about the location, interfaces and access protocols to remote services.

extensible and converge control on local authorities through logically centralised servers. Thus, the requirement here is that the DSF infrastructure should eventually be able to operate like *an open electronic market of software services*, rather than just a centrally controlled hierarchy [32]. Finally, the DSF project should provide opportunities for large-scale collaboration between software engineering researchers, educators and practitioners. In the following Sections, we further describe these requirements.

3.1 Supporting distributed everything

What does it mean for a large-scale software engineering process to be distributed over a wide-area setting? In our view, there are two kinds of answers to this question:

- identifying what software objects and activities can be distributed but still effectively accessed and used.
- identifying what mechanisms are needed to integrate these distributed objects and activities into a logical, coherent software process workspace.

The kinds of things that can be distributed in a DSF are *software objects*, *software tools*, *software process tasks*, *software engineers*, *target software applications* and *other components of the DSF infrastructure*. *Software objects* are the data entities and executable control routines that are assembled into either external software engineering products or internal software environment mechanisms. *Software tools* include compilers, testing systems, application generators, language-directed editors etc. *Software process tasks* include

- □ recurring development tasks, requirements analysis, software specification and design, implementation etc.
- □ process improvement activities, measurement, modelling, simulation and evaluation.
- □ process (and product) co-ordination activities, planning, team-building, configuration management, review meetings etc.
- □ technology transfer and transition activities between software producers and consumers.

Software engineers are the people who perform various software process tasks at different times and workspaces to create, update or disseminate software products. *Target software applications* are the packaged software products that software engineers produce to operate sequentially, concurrently or in parallel on distributed processors and workplaces. *Other DSF infrastructure components* can include a geographically dispersed network of host computers, as well as office and computational workspaces that can be accessed synchronously or asynchronously. But, if any, or all, of these elements can be distributed, how can they be co-ordinated and integrated?

3.2 An electronic market of logically centralised services

In the DSF, we assume there is no single global authority that can choose the best or optimal set of software objects, mechanisms or activities needed to most efficiently produce high-quality software. Instead, our objective is to provide a series of interfaces and protocols that enable software service mechanisms to be co-ordinated, selected, configured or extended, as necessary. Thus, instead of a single global authority, we seek to support an electronic market of software services, where external developers can join and introduce new or competing services, or where coalitions of software technologists may form to develop highly specialised service mechanisms or package service mechanisms that span multiple layers of the DSF infrastructure described below.

What categories of software services do we expect to be required by the DSF project? At a minimum, we expect that the same categories of services being investigated by software engineering environment researchers are required. To us, these categories of software engineering environment services include

- *operating system and network encapsulation services*, which abstract the operations of the underlying hardware and software platforms through sets of higher order interfaces.
- *object management services*, which provide for the organisation, storage, retrieval and update of typed, attributed software objects created or manipulated by people or programmed tools.
- *collaboration services*, which provide mechanisms for allowing multiple users synchronous or asynchronous access to software objects or lower level services.
- *tool integration services*, which provide mechanisms for rapidly configuring or integrating newly developed tools into coherent software engineering environments that access and manipulate underlying objects and services.
- *tool services*, which provide for different classes of processing tasks that software environment users will employ.
- *user interface services*, which provide for the construction or use of consistent 'look and feel' user-system interaction, process window layout, command menu selection and other direct manipulation display capabilities.
- *re-engineering services*, which provide mechanisms for rapidly extracting, synthesising, encapsulating and extending the functional capabilities of externally developed software systems, so that they can henceforth be supported within the DSF infrastructure.
- *quality assurance, configuration management, validation and verification services*, which are intended to assure that newly engineered and evolving software applications, and accompanying documentation, satisfy a formalised set of practical quality or integrity constraints.
- *software process modelling services*, which provide support for the capture, representation, query, simulation and analysis of multi-agent software processes performed within resource-limited organisational settings.

An electronic market represents a distributed network computing environment, where clients can choose among many service providers for the processing they require. This implies that clients select and access alternative service providers through interfaces and protocols that are common across servers or service brokers [33]. This also means that the market for services can expand both *horizontally*, as multiple servers emerge to compete to provide a given category of service, as well as *vertically*, as servers emerge that encapsulate or further stratify a set of distinct service cate-

gories. We chose to model and construct the DSF infrastructure from software services that can be either directly accessed as a separate service layer, or constructed from provided mechanisms and underlying services. Note, however, that an electronic market is not a software components integration mechanism, as in Reference 34. Instead, a market is a forum where software components or services of all kinds can be made available for distribution, exchange, acquisition, experimental use or evaluation. Thus, in the DSF, it remains the responsibility of local development groups to select and integrate software components or services they believe are viable or significant contributions.

3.3 The layers of the DSF infrastructure

The DSF infrastructure represents a multi-layered open system. This openness indicates that evolutionary changes§ can occur at any layer which, in turn, can propagate new services or service failures to adjacent layers. As such, this openness implies that DSF integration is a process that coincides with evolutionary changes. Thus, the more diverse and heterogeneous the DSF becomes, the more extensive the integration work that is allowed or required. However, this process allows opportunities for both horizontal and vertical integration and encapsulation of inter-related servers as a basis for constructing highly specialised (horizontal) services or (vertical) environments. This means that different software infrastructure developers can focus their development activities on the construction of robust single-layer services (e.g. user interface services), or on the packaging of multi-layer services into a package-of-services environment (e.g. an environment that provides collaborative multi-user tool interface and integration services).

Based on our experience in the USC SF, these layers must span from the operating system (OS) and network level to up through the relationships between the developers and end-users of an application. In this way, we can see how the DSF infrastructure interlinks technology and people. At present, the number of infrastructure layers that we work with is eight, including

□ *OS and network interface;* provides access to OS-managed entities such as files, processes, streams, queues, pipes, sockets, utilities, daemons, command shells, memory buffers and device drivers, through local or remote system calls embedded in software objects.

□ *control and data objects;* provide access to entities that either (for control) manipulate OS entities and data objects, or (for data) present the type, attributes, attributes values and realisation contents for software objects manipulated by tool subsystems. In addition, control objects are data objects that can be executed through the OS and network interface.

□ *subsystem services;* provide a medium through which software tools or application programs access and manipulate underlying objects that users select through the user interface.

□ *user interface;* provides the displays and command menus which invoke designated tools on selected objects that may conform to some software process task.

□ *software process tasks;* provides a medium for specifying the possible sets of actions that proceed or follow a displayed event or command selection through the user interface.

□ *software engineering agents and resources;* provides a medium for specifying what roles different people or intelligent mechanisms can play in order to perform a software process task with available resources, and how they might work together.

□ *software engineering teamwork;* provides a medium for planning and co-ordinating how different agents might work together when either software process tasks succeed or breakdown (and require some remedy); or relationships between software application producers and consumers change or shift.

□ *software producer–consumer relations;* provides a notation for specifying how software developers, users and maintainers can be inter-related in order to facilitate effective software technology development, transfer, transition and routine use.

§ Such as reuse, functional enhancements, bug/anomaly resolutions, performance improvements, interface or internal representation restructuring or platform migrations on the technology layers; staff turnover, budget cuts or schedule changes etc. on the topmost people layers.

4 Information structures for modelling the DSF infrastructure

Below is a brief description of the information structures that we have developed for modelling and creating a working DSF infrastructure that accommodates both local-area and wide-area network services. In our view, each of these classes of structures is necessary. Similarly, these structures must be extensible and evolvable, since time and experience (both ours and yours) will support the wisdom of our current judgements. These structures include software objects, compositions of software objects, software engineering processes and software engineering settings. We will describe each in turn, as well as their constituent substructures. Overall, our objective is to create and maintain these information structures for software engineering as a directed graph, which we will refer to as a software hypertext [17, 23].

4.1 Basic software objects (BSO)

BSOs denote semi-structured object descriptions of various length and substructure [23, 28]. Semi-structured means that object descriptions are structured to some degree. All objects are typed and possess descriptive ***attributes***, which characterise the form, contents, purpose and network identity of the object [23, 35]. The range of possible values these attributes take on may be either completely defined (e.g. by formal language specification, type declaration or enumeration) or weakly defined (an alpha-numeric string). However, the presence and formalisation of these attribute definitions determines the degree to which the descriptions can be parsed, analysed and interpreted by processing mechanisms such as language-directed editors, rule-based interpreters, specification and design analysers, or compilers to determine their consistency and completeness [3, 23]. BSOs and object compositions provide a common, typed substrate for representing and managing software data objects in ways that facilitate OS and subsystem integration.

4.1.1 System libraries and communication protocols: the base-level software objects represent procedural code that provides an interface to the underlying operating system entities and a wide-area network. Typically, these are function or procedure libraries that are externally managed and maintained. A common example is the X-Window system libraries that are publicly available, regularly updated and ultimately maintained at MIT. However, we have also developed a collection of low-level utility routines, which provide different kinds of access to OS or network-managed resources, such as interprocess communication sockets or network file servers. With these mechanisms, facilities such as distributed OS command language shells [36], gateways to remote repositories [35] and synchronous multi-user display interfaces can be provided. Further, use of these mechanisms in systems-level programs is often structured into protocols of operations for transparent access to local or remote objects [35]. Such transparency provides a common view (a client-server protocol) across multiple object bases, which may be heterogenous. For example, the following set of entities and operations are part of what we refer to as a 'distributed hypertext protocol', which provides a transport layer and service-level interface to local or remote software objects in a way that hides the details of their physical implementation [35].

- **Entities:**
  - ☐ *objects* are either atomic or composed entities with characterising attributes and contents, such as remote source code files and databases.
  - ☐ *contents* are part of a designated object with a concrete realisation, such as the source code within a source-code file object.
  - ☐ *attributes* are unary relations that indicate the type or value for a set of properties attached to an object, such as its author, date or creation, object identifier, revision timestamp etc.
  - ☐ *links* are binary relations that indicate the source and target object(s) associated through some user-defined mapping.
- **Operations:**
  - ☐ *get* retrieves a designated entity from local or remote servers.
  - ☐ *delete* removes the instance of a designated entity.
  - ☐ *add* creates an instance of a designated entity.
  - ☐ *update* deletes an old instance, then adds a new instance of a designated entity as an atomic event.
  - ☐ *list* retrieves a (sub)set of the attributes of the designated entity from its servers.
  - ☐ *find* searches for a set of objects that matches a predicate.

4.1.2 Relations: relations represent links within or between software objects. For example, *keywords* and *annotations* are special kinds of relations that are supported with processing mechanisms embedded in our software hypertext server [23]. Keywords support tracing the occurrence of index terms across software documents. Annotations represent hierarchically linked narratives that further describe a designated word, phrase or object to indicate its meaning, explanation of its use or decisions pertaining to its definition and purpose. Users can also define semantic relations which, in turn, can be static or operational. Static relations denote a simple logical relation between linked software objects, whereas operational relations invoke a user-defined function or process when linked items are visited or modified.* Overall, relations can be stored and managed by either an object-oriented or relational database management system (dbms), if the dbms mediates access and updates to the object space† [23, 25]. In this way, linked software objects can be indexed, browsed through and relationally queried through the mechanisms of the dbms, including query processors, report generators, pattern matchers and fourth-generation languages. This also allows for the relation (link) servers to exist separately from the BSO servers, thus accommodating more distribution and server heterogeneity [35].

4.1.3 Software product components and tools: BSOs may also represent large-grain software entities. Such objects may take the form of complete executable programs or common documentation unit (e.g. the man page on UNIX). What makes such objects basic is that they are treated as atomic either by a user or by some other program. Tools such as a C++ source-code compiler and debugger may each represent composite programs, but if their parts are inaccessible to users or other tools, these executable programs can be treated as non-decomposable software objects. Thus, the granularity or size of a software objects does not necessarily determine its atomicity.

4.2 Software object compositions

These are networks or graphs of BSOs and other (nested) compositions that denote some user-defined association(s) [37]. For example, transient documents can be composed for printing only the sections or subsections of an article modified by an author since the last revision. In multi-authored documents, such as large software systems, this is a particularly useful feature that facilitates parallel development activities and project status monitoring. Similarly, hierarchical compositions allow software descriptions throughout the life-cycle process to be managed, viewed or evolved in ways that maintain and assure their correctness, or keep track of their inconsistencies [26].

There are three forms of compositions that we have found useful in the USC SF project: linked paths, deliverable software products and project partitions.

4.2.1 Linked paths: linked paths typically represent an aggregate linear sequence of BSOs that a user finds conveys useful information as a linked set. The linearity denotes that the sequence of BSOs should be viewed (on-line or off-line) in the order composed. There are no restrictions as to what objects can be included in a linked path, other than that they be accessible. For example, a source-code file could be linked to its originating requirements, its design and a collection of mail messages among its developers, which discuss the relative merits of alternative implementation strategies. In practice, we find that linked paths are fre-

* For example, qsort( ) IS_A sort-function, where IS_A denotes a static subclass relationship, and qsort.c COMPILES_INTO qsort.o represents an operational relation that invokes the C compiler whenever qsort.c is modified, in order to automatically derive qsort.o.

† This allows the software object repository to be treated as a 'storage manager' or 'object server', and thus does not assume that the software objects must be stored physically within the dbms, nor at only a single network site.

quently used during exploratory development activities [37] and that most linked paths tend to include only a small number of objects.

4.2.2 Deliverable software products: deliverable products represent aggregations of BSOs and/or hierarchical compositions of BSOs [23]. For example, a BSO might be used to denote each section, subsection or paragraph within the requirements analysis document for a project. More generally, deliverable products can be used to define the structure of software system information as a collection of life-cycle document structures that can be standardised, shared and reused across many project teams [23]. This standardisation can therefore support parallel authorship of multiple software documents when the inter-relationships between documents are made explicit *a priori*. This arrangement thus creates an opportunity for improving the co-ordination and productivity of the system development effort [37].

4.2.3 Partitions for projects and teams: partitions provide a structuring mechanism whereby collections of deliverable products can be composed, standardised and shared by classes of users in different development efforts. For example, in the USC SF project, partitions can be organised by work group, team or project site [3]. Partitions represent contexts [38] for structuring access to a hypertext of software objects. Thus, individual product or BSO instances can be stored in a single partition but accessed from multiple partitions. In this way, people working within a partition may browse through, link or compose across multiple partitions. This supports the assignment of standard BSO/product-processing mechanisms to designated types of software objects [17, 23]. Similarly, it helps to minimise getting lost in a software information hyperspace, since a user always works within a known context with defined deliverable products and BSOs.

4.2.4 Software engineering processes: the preceding information structures are used to describe the organisation of software objects as product components that emerge during the software life-cycle. However, it is often the case that *the software life-cycle process* (the sequence of tasks and associated processing mechanisms that create and manipulate software objects) is itself subject to alternative definitions and task compositions. Thus, the tasks which software engineers perform should also be developed, documented, organised, updated and processed in ways analogous to other software objects.

The information structures for specifying software engineering tasks need to describe the sequences of processing mechanism invocations that simplify the routine manipulation and interchange of software documents. Consider source-code program documents, for example. Here, the processing mechanisms for manipulating source-code descriptions include editors, compilers, debuggers, program linkers and loaders, as well as formatters for displaying highlighted ('pretty-printed') program listings. The task of developing a working program usually requires the use of each of these mechanisms. The sequence of their invocation is mostly non-deterministic and non-procedural, but it is easily tracked by individual software engineers for small programs. However, if the programs being developed are large and built by teams developing multi-version program components according to an elaborate life-cycle engineering methodology [3, 23], the program writing task becomes complex and costly, if not well co-ordinated. The description of software object/document processing tasks can be decomposed at many levels of detail and inter-relationship. For example, there must be support for specifying 'the task of task specification'. This ***meta-task*** description is needed for organising tasks, specifying their components and inter-relationships and assigning them to appropriate partitions. Meta-task descriptions must be maintained, since the content and structure of task descriptions may evolve in open system workplaces in unexpected ways [6, 21, 39–41]. Similarly, there are at least two classes of tasks for which many subtypes can be identified. These are the ***management tasks*** of project administrators and the ***engineering tasks*** of technical project staff. Management tasks focus on activities such as decomposing system development projects into subsystems; assigning staff and processing mechanisms; scheduling and budgeting subsystem description development; monitoring project progress and productivity; acquiring and maintaining an adequate supply of staff and computing resources; assuring the quality of the integrated and validated final product document assembly; and redoing any of these when things break down, go wrong, or when external conditions dictate [16]. Engineering tasks are performed by individual or small groups of engineers who create, manipulate and interchange component documents assigned to them among other things [6, 16]. These tasks include analysing system requirements, developing system functional specifications and designs, program development and test, maintaining existing systems, and so on.

Software project management and engineering tasks are inter-related in many ways. For example, in order to confirm that a system is fully operational and ready for delivery, the tasks of assuring that all the right versions of all of the right system components were selected, composed in the right order, and then appropriately tested, must all be successfully performed. Each of these subtasks, in turn, requires management subtasking. For example, each of these subtasks assumes that the required source code programs were developed and run through the assigned set of source-code processing mechanisms. However, they also assume the existence of either an automated processing mechanism or manual administrative mechanism to check that the components fit together in a consistent and complete manner. When the number of components is in the hundreds or thousands, each existing in one or more versions, each potentially fitting into many alternative configurations and each having one or more sets of data for testing, a combinatorial explosion of alternatives must be engineered, configured and managed. The complexity of the emerging system begs for co-ordination and automation rules that simplify and maintain a closed system description, whose consistency and completeness can be directly assessed. However, the complexity of system artifacts requires that the successful performance of the management and engineering tasks must be interdependent 6, 16, 40–42.

All tasks, regardless of level of description, describe a potentially *non-linear sequence of actions*.§ These actions

§ Non-linear sequence indicates a partial ordering of non-deterministic or potentially iterative incremental actions. In other places, these action sequences are called *task chains* [6, 18, 40, 42] or *plans* [43].

affect some concrete or abstract transformation of a software BSO, product or partition. For example, in the task of implementing a software system design as a program, the creation of a successfully compilable program component is a necessary action. Other actions in the sequence include

- understanding the software design.
- developing a program implementation strategy.
- establishing which processing mechanisms are available.
- debugging anomalous program behaviour etc.

In addition, each task or action can be guarded with pre- or postconditions that must be satisfied before forward progress can continue. Further, the actual action sequence traversed by a user can be recorded in a history script for subsequent analysis, replay or reuse [18].

In turn, all actions can be further decomposed into *primitve* actions. These represent commands issued in dialogues between an individual developer and the current tool or processing mechanism in use. For example, understanding the software design can entail

- □ browsing through a design document.
- □ following embedded cross-reference relations.
- □ asking the design dictionary for information about a particular software object.
- □ tracing design details back to the originating system requirements.
- □ searching for an existing program component which performs a similarly designed computation etc.

As before, the sequences of primitive actions are also nonlinear [17].

Overall, primitive actions, actions, tasks and meta-tasks must be carefully aligned and performed when a system of software life-cycle documents are produced. This can become an emerging open-ended activity that we seek to structure, control and close, so that we can assure its consistency and completeness. As such, these software process descriptions at varying levels of detail cannot be guaranteed *a priori* to be consistent, complete or correct under all possible performance circumstances. Hence, the need arises for viewing the creation, manipulation and evolution of software process task descriptions as structured information that should be managed as a domain-specific software object hypertext [17, 24]. To this end, we have developed a knowledge-based software process language and support environment for developing formal models of software tasks structures and their relationships to other development objects [17, 18].

4.2.5 Software engineering settings: a DSF development effort will include people with different skills, processing mechanisms and various shares of organisational resources in dispersed locations. Thus, this information should also be described and linked into the software object hypertext. We will limit our focus here to the project participants.

People in software engineering projects are typically divided in terms of management, engineering, customer, and maintenance skills. Further decompositions (or subtypes) of each can be identified.

For management skills, we sometimes see specialists for process management, quality assurance, scheduling and budgeting, and configuration management. On small projects, these skills might be possessed and put into practice by a single person. However, on large projects divided into many subsystems and small group teams, these management skills will often be distributed among many people. Similarly, on the engineering side, there may be specialists for each software life-cycle activity or a subset of life-cycle activities. For example, the principal software engineer and a small group of trusted senior engineers may be primarily responsible for defining an overall software application architecture and major subsystems, as well as specifying their operational requirements. These specialists, however, will usually not be given the additional responsibilities of performing the detailed design and implementation of source code modules.* Similarly, the majority of system programmers may not have the skills for producing a viable set of system requirements or overall design.

However, there are also other people whose participation and skills can affect a software engineering project. These are the end-users and clients, who define the system's general requirements, and the system's maintenance staff. If the customers have experience in specifying or using diverse software applications, their skills in specifying system requirements will be different from someone acquring or using an unfamiliar application technology. These latter type of customers may, as a result of their inexperience or uncertain knowledge of the new technology, end up frequently changing the specification of their requirements. However, it is widely recognised that changing system requirements is one of the most frequent causes of projects being late, over budget, or otherwise a technical failure or maintenance nightmare[14]. This brings us to another class of project participants, software system maintenance staff.

For a large system, maintenance activites go on for years. Maintenance tasks (adding functional enhancements, resolving anomalies, tuning system performance, migrating the system to other environments) are divided among staff, according to subsystem responsibilities. Software maintenance staff for large systems are generally not the same people who originally developed the system. As such, their knowledge of the system's operational behaviour, function and structure must be derived from either the existing software object descriptions, informal conversations with others or direct experience. One frequent problem here is that the existing descriptions (source code versus system designer) are typically inconsistent, incomplete or otherwise inadequate [6]. Thus, software maintenance staff are often at a disadvantage in keeping operational systems viable, unless there are automated maintenance support systems to assist them [25, 44], or the available software object descriptions were engineered and maintained throughout the project up to this point. As a result, the success of the maintenance staff's tasks depends on their ability to accommodate or negotiate alternative definitions of their tasks or work arrangements, as well as understanding how new requirements can be met by modifying the existing systems [6, 25].

As such, we can identify four classes of participants for

* One possible exception to this might arise for modules designated as critical to overall system performance or integration. In most projects, however, this may be uncommon.

software engineering projects; managers, engineers, customers and maintenance staff, which can be further decomposed into task specialists that are inter-related and interdependent. However, what is critical is the linking of tasks to skills and other resources, rather than the number or names of the participant classes. The task-skill combinations are constrained by the limits of the organisational resources and automated processing mechanisms allocated to their interlinked project partitions. Thus, the structure, content and flow of project participants' task organisation should be described and managed as evolving software object descriptions [17, 18, 24].

Lastly, we should also delineate the configuration of tools and organisational resources, as well as their relations to other setting, process and software object description structures. Accordingly, these descriptive configurations should also be represented within the software hypertext, as suggested elsewhere [3, 17].

5 The software services infrastructure

We now describe the collection of software processing servers which service the information structures that represent and model the DSF infrastructure.

5.1 *A distributed object management server*

Given the preceding model for software information in the DSF infrastructure, we need basic services for accessing, storing, sharing and updating this information across multiple networked sites. At present, these services are realised by what we call the distributed object management server (DOMS).

The requirements satisfied by DOMS are manifold. It provides a richer set of persistent object services than available from a local-area network file server. It provides services for both individual objects and object sets/databases, which are treated as concurrently accessible composite objects. It provides for extensibility through user-defined object types. It provides a means for locating objects distributed over a wide-area network through query mechanisms in ways transparent to a user. Thus, object databases can contain or compose distributed objects. It also supports operations such as read, write, copy, name, find etc. much like a network file system. It provides object-oriented database functionality, together with a persistent programming language. However, it also provides a library of DOMS functons that can be either extended and linked into an application system, or integrated with other mechanisms in the DSF infrastructure through the DHT protocol [35]. Given this, DOMS can be configured as a single, logically centralised repository that integrates a heterogeneous collection of object servers and link servers.

All objects within DOMS have a unique identity, type, behaviour defined by attached methods and complex state. Object types can be determined at runtime, so that control objects can be directly executed or treated as data. In turn, control objects can also access and manipulate DOMS, as well as its objects. Objects can include other objects. Thus, object methods and attribute values may also be objects. Object behaviour can be inherited through a conventional inheritance mechanism. Behaviour methods are polymorphic, so that the same message sent to different object types may invoke different methods. The object state (which may represent a snapshot of an object's current attribute values) is encapsulated and accessible only through the object's methods.

5.2 *Groupware support through collaboration servers*

As noted earlier, teams of engineers work in parallel or concurrently to develop, inspect, assure and manage multiversion software products, their configurations and their component objects. These software objects then become medium for co-ordinating process tasks and information flows across organisational time and space. Conventional email and bulletin boards provide general-purpose asynchronous communication services. However, these conventional communications services can only provide *ad hoc* support as engineering teamwork becomes more dispersed. What we need are communication services that can support synchronous and asynchronous process tasks and tool invocations by multiple users. We therefore refer to functional mechanisms that provide these 'groupware' services [29] as *collaboration servers*.

The collaboration servers we provide as part of the DSF infrastructure represent a multi-level interaction protocol. However, we distinguish between synchronous and asynchronous collaboration services. For synchronous collaboration, we focus on providing concurrent multi-user interfaces to software tools or applications. For asynchronous collaboration, we focus on task mail-message management.

The architectural layers for synchronous collaboration span from the OS network layer up through the user interface application layer. Synchronous collaborations across distributed sites entail a ***virtual network*** of user communications, object manipulations and tool/application command invocations. The virtual network circumscribes a group of users who seek to collaborate through shared objects and processes over a wide-area network in real-time. Thus, this virtual network mechanism is not merely a shared window system, but instead it represents a set of service layers on which various low-level processes, objects, user sessions, tools or applications can be shared over a network. This requires a set of functions that set up the necessary interprocessor communication sockets over the network. This service layer, in turn, manages the virtual network topology through functions that create, remove, join and exit virtual collaboration networks (and sub-networks), and multi-cast messages. The next layer up provides functions for synchronisation of data shared in a collaboration session (e.g. text buffer, fine-grain locking/unlocking for concurrent user access). On top of this, resides what we call the Co-X layer, which inherits the capabilities for X-Window widgets [45] utilised by tools or applications, and adds methods and structures that extend the X client-server protocol to work with the underlying virtual network and data synchronisation services. Finally, the tool/application layer represents the subsystems that offer users collaborative access, such as co-texteditors, real-time conference systems, voting systems and shared drawing tools. Our experience with the development of co-tools such as these indicates that many existing tools that are already compatible with X widgets (e.g. xedit, vi, emacs) can be 'co-ised' through rather modest source-code alterations (changing less than 20 lines of code in some cases). Clearly, the real-time synchronous performance of the virtual network depends on a moderately

high network bandwidth, such as is now available for most local-area networks, for emerging metropolitan-area networks and for proposed high-speed wide-area networks.†

Things are much simpler for the asynchronous collaboration server. Simply put, this server extends conventional email services to handle typed and attributed objects as mail messages. This means that rule-based mail filters can be constructed that can automatically send and receive mail to and from local/remote users [46]. In turn, these messages can transmit or trigger executable control objects on the message receiver's end. This enables, for example, process task-specific communications to be sent from a user-defined object to remote users, whose eventual reply will be linked to the originating object, deliverable product or partition. In this way, mail messages can be sent and received in the process-task contexts where they emerge, rather than being localised to decoupled access facilities of conventional email systems. This asynchronous collaboration mechanism was previously demonstrated in the USC SF with the ISHYS hypertext environment [17]. Thus, using both the synchronous and asynchronous communications mechanisms, we find that we can support the shifting, collaborative work structures that emerge during sustained software engineering efforts [8].

5.3 *Tool configuration mechanisms*

Many software engineering tools are developed in isolation from other tools. Thus, constructing an integrated software engineering environment from these tools is often problematic. Although it is possible to consider building such tools and environments together from the bottom-up, such a strategy imposes a major software development effort, as well as a potential deprivation of access to otherwise useful software engineering aids. Similarly, when possible, we would also like to avoid making extensive modifications to already useful tools in order to incorporate them into an environment. As such, we seek a set of mechanisms that can provide support for integrating both loosely coupled and tightly coupled tool sets [34] and user interfaces into coherent software environments. Further, we want these mechanisms to be configured through declarative specifications to enable rapid (re)configuration of an environment.

Our first effort along these lines led to an extension of our software hypertext environment [23]. This extension incorporated a module-interface language specification [25] and associated interpreter to provide the desired environment integration mechanism. In this way, a collection of environment modules (i.e. tools or shell scripts) could be specified, configured and made executable on hypertext-managed software objects through interactive mechanisms. This environment specification is, of course, another user-defined object type which, in turn, is created, stored, accessed and updated within the software hypertext object base. The interpreter is designed to evaluate this specification whenever a user moves from one partition to another. Thus, this made it possible to be able to construct software tool environments that could be customised to different user roles or access patterns. Based on our success with the ease of use of this specification-driven environment configurator, we next extended the specification language to accommodate user interface modules, which can be used to rapidly configure different types of tool/process task invocation windows and command menus. Accordingly, this facilitates the rapid configuration of domain-specific software hypertext environments [24]. Thus, we now make regular use of these tool configuration mechanisms as part of our environment integration services.

5.4 *Programmable graph editors, browsers and user interfaces*

Graph-based models of information represented in object management systems are gaining more attention for use in software engineering environments. In turn, many software engineering environments use graph editors to manipulate different types of information. For example, Software through Pictures is now a widely recognised commercial software development environment, which makes extensive use of a variety of graph editors as its user interface to underlying software objects and databases. We have developed a programmable tree/graph editor (TGE), which is used to rapidly construct graph editors that can be combined with other tools as part of an integrated software engineering environment [47]. TGE represents a set of three base editors, undirected graph, directed graph and tree editors. Application- or domain-specific tree/graph editors can then be constructed through specialisation of the object classes that are incorporated into the base editors. In simple terms, this means that all base editor functions for node/-edge selection, node relocation, edge redirection, storage operations, panning, zooming, cut-and-paste etc. are inheritable and can be further specialised in an application editor. Similarly, the base TGE graphic entities, such as graph nodes, edges and user command menus, together with their display attributes and layout algorithms, can also be inherited, specialised and bound to designated software objects types or instances within a TGE application editor. This supports the rapid construction of editors for object hierarchies, data-flow diagrams, graphic database schemata, hypertext webs, animated finite-state machines, PERT charts etc. TGE can also be used for rapid prototyping of multi-view user interfaces [47]. Figs. 1–3 provide views of the process model class hierarchy editor (Fig. 1), the database schema editor (Fig. 2) and the software system configuration design editor (Fig. 3), all of which are based on TGE.

5.5 *Re-engineering software for integration*

Software tools and applications to be incorporated into the DSF infrastructure can be expected to come from a variety of organisations, with varying degrees of engineered software development descriptions. Thus, a fundamental problem to be addressed is how to simplify the integration of software systems that have been, or will be, developed without explicit conformance to DSF information structures model. As such, we are providing a set of software re-engineering mechanisms, whose purpose is to assist with the conversion and/or integration of these externally developed software systems into forms compatible with the DSF infrastructure. However, we assume that these external software systems are available in source-code form, in a high-level language (such as C, FORTRAN, Pascal, Ada) and can be executed on a UNIX OS processor.

† USC is currently connected through a cluster of LANs, as well as to ten or so remote sites through a 1.5 Mb metropolitan area network. These are the facilities we used for our experimentation with the collaboration servers.

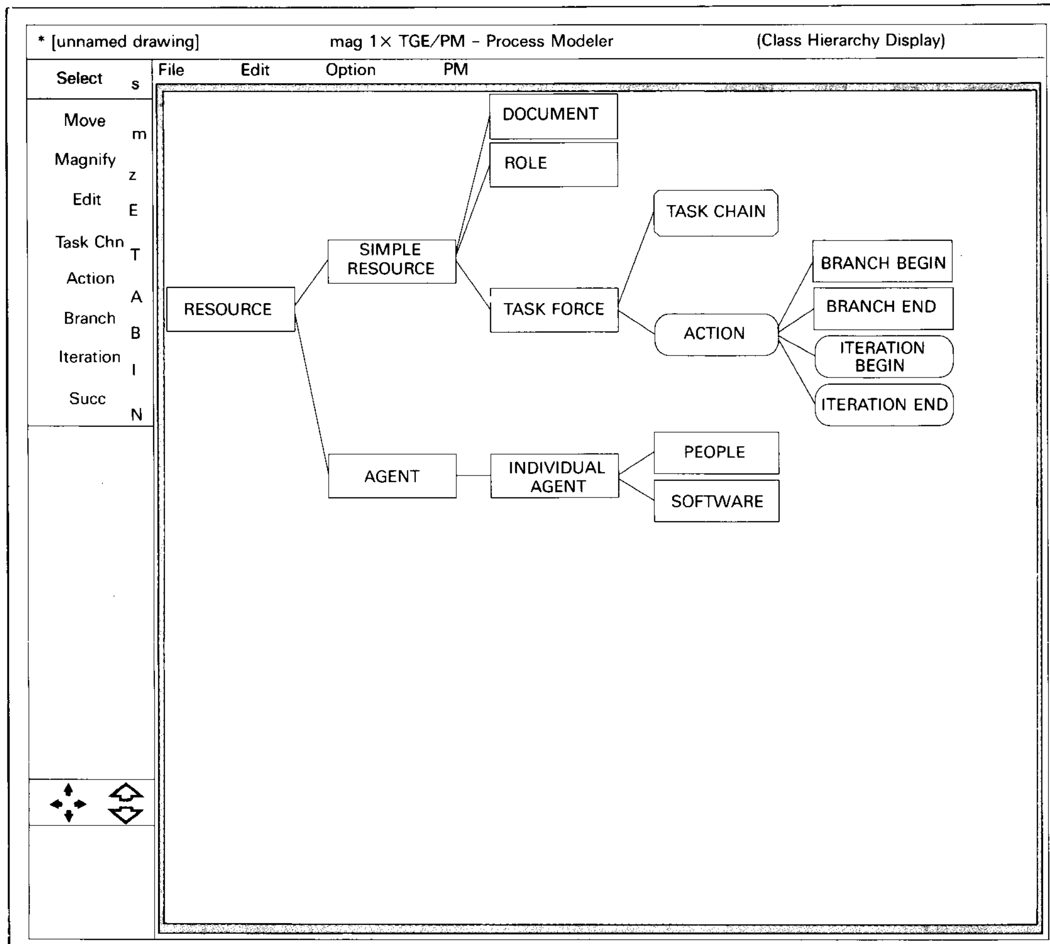

Fig. 1 A process model class hierarchy editor

5.5.1 Re-engineered integration: this Section describes what our objective is in re-engineering a software system to become compatible with the DSF infrastructure. We have been exploring the potential of our tool configuration mechanisms described above as a central component of this service.

In our view, this form of re-engineering entails a multi-stage integration process, which roughly corresponds to the steps that can be taken to couple external software subsystems to their adjacent layers in the infrastructure. The first stage represents what we call 'user interface facade integration'. At this stage, the requirement is to get the external subsystem connected to the system libraries and user interface command menus, so that the subsystem can be invoked separately from an existing user interface. This means that its OS command invocation can be encapsulated and attached to a UI command menu item, its inputs provided manually by the user and its output displays be made compatible with the existing window system.§ The second stage is to integrate recurring user inputs into nested command or object selection menus. The third stage is to integrate the subsystem's inputs and outputs that can persist in the underlying object storage manager. In practice, this often means that the storage manager must either intercept file system calls, provide a file system-like interface or require that the subsystem source code is converted to employ the storage manager in place of the file system. Call interception and conversion are complicated, messy and expensive, and should therefore be avoided when possible. Instead, we provide a file system-like interface at this stage, so that the subsystem still acts as if it was interacting with a file system. The fourth stage is to supplant the subsystem's file system model, with a more object-based model that reflects the object management and access conventions used elsewhere in the DSF infrastructure. As before, the idea is to avoid modifications to the subsystem source code, and instead extend the user's view of the subsystem's file/-object space. This may require static, dynamic and behavioural analyses of the source code, in order to extract and synthesise module resource interfaces and interconnections information that facilitates the automatic restructuring of the source code [48]. However, there is still opportunity for substantial improvement, through the introduction of new

§ For subsystems that use conventional terminal displays, this is trivial when using Xterm windows, whereas if the subsystem utilises a graphic display buffer, conversion to X Window protocols is necessary [45].

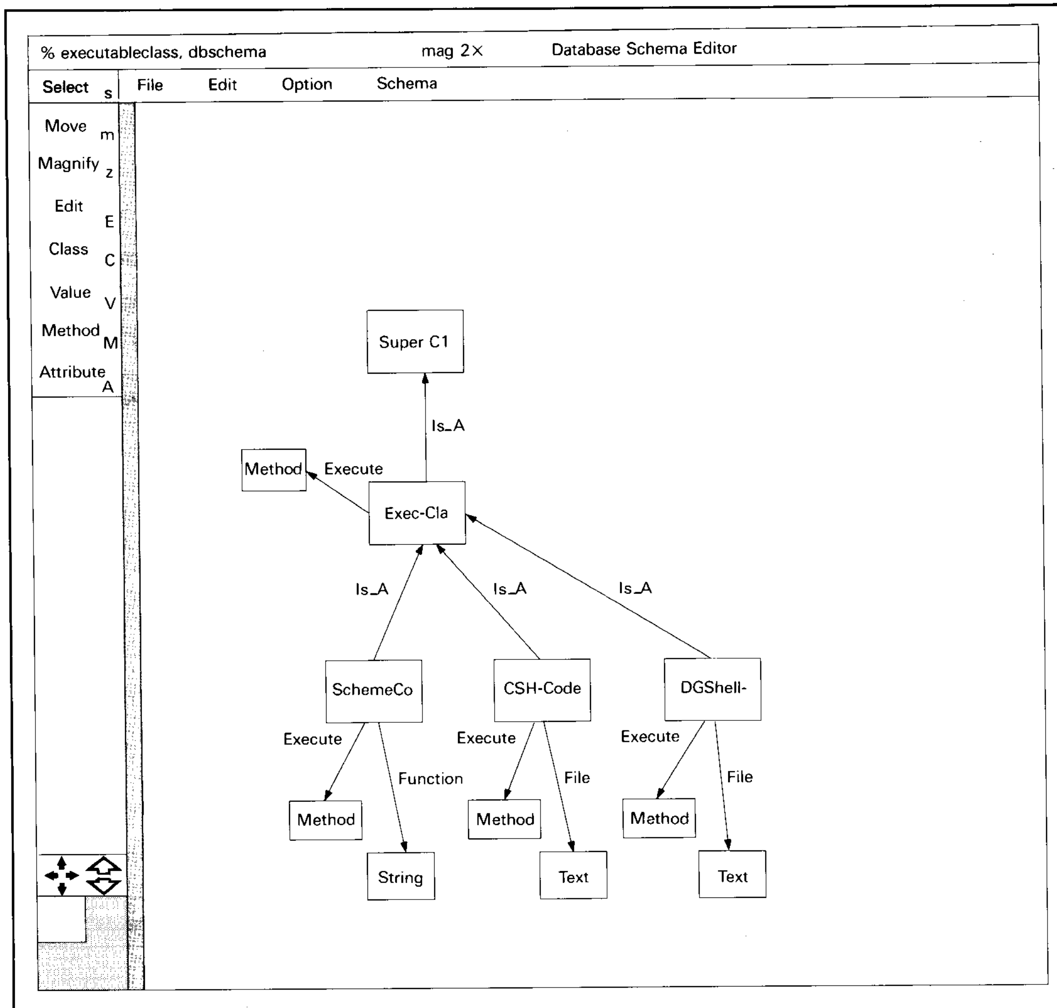

Fig. 2 A database schema editor

tools and methods that can more readily facilitate the rapid integration of externally developed, non-conforming programs. Nonetheless, this multi-stage process allows the utility and value of integrating an externally developed subsystem into the DSF infrastructure to be assessed in an incremental/stage-by-stage manner.

5.6 *Correctness-assuring engineering environments*

In our view, a software application system is *correct* if all of its life-cycle products are consistent, complete and traceable. Clearly, we mean to imply this is a restricted notion of correctness; one that simply indicates the degree of conformance to a set of quality assurance attributes and documentation integrity constraints. Nonetheless, we have demonstrated how to provide software life-cycle engineering environments that can assure this form of correctness, as well as incrementally track inconsistencies, incompleteness and traceability gaps [26]. Simply put, we have developed a software object model that provides all basic and composite software objects with *resource interfaces* reminiscent of module interconnections languages [25, 44, 48]. These resource interface attributes specify what types of entities can be imported or exported by each software product object. Such a scheme can be shown to be viable with any multi-stage software life-cycle methodology. Further, we show that it is relatively straightforward to construct software life-cycle engineering environments that process this object model, if certain classes of tools (e.g. language-directed editors), integrity constraint checking mechanisms (relational database predicates) and an object management server of the type already described can be provided [26]. Thus, at present, we are developing a new version of such an environment that incorporates collaborative multi-user language-directed editors, an enhanced integrity attribute set to handle multi-version objects and DOMS in order to make such a service environment accessible over a wide-area network.

5.7 *A software-process modelling framework*

Modelling the process of software engineering represents a

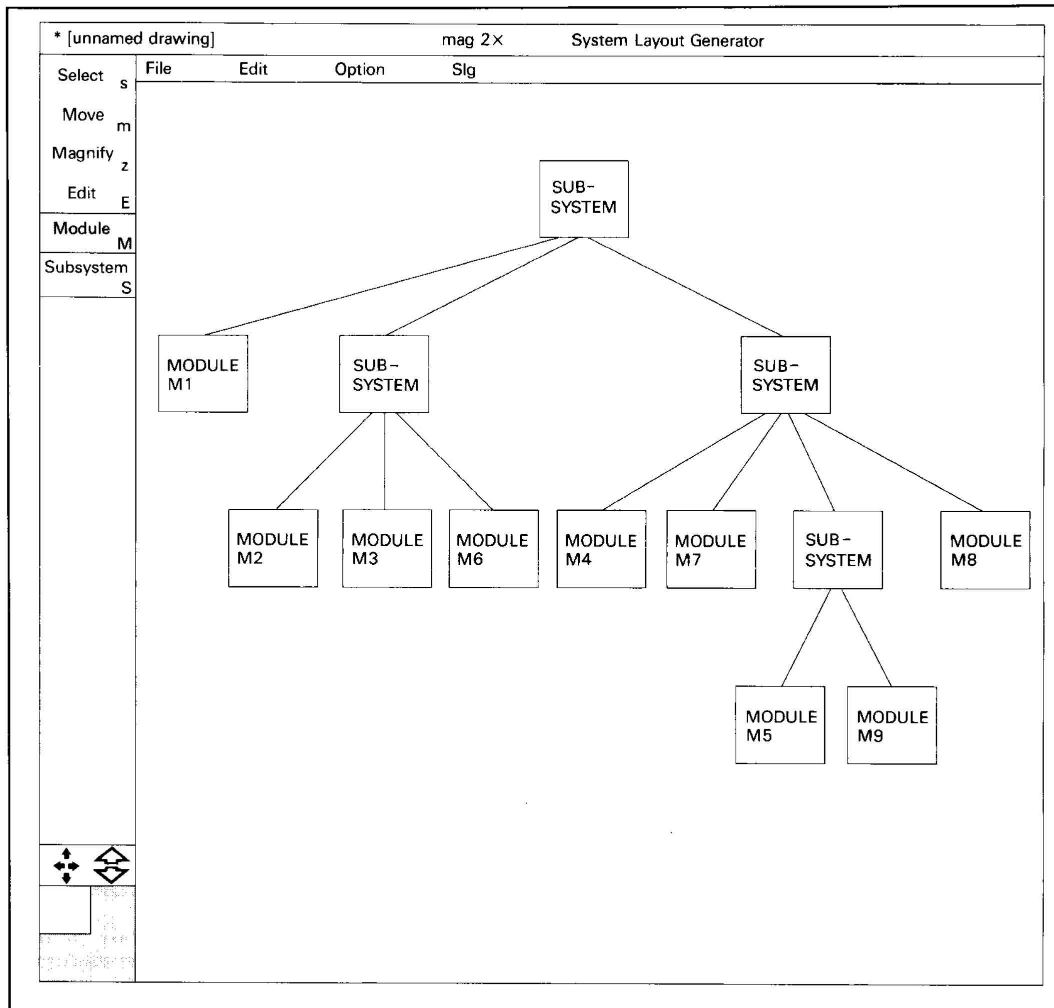

Fig. 3 A software system configuration editor

promising approach to understanding and supporting the development of large-scale software systems. Accordingly, a software process model is a prescriptive representation of software development tasks in terms of their possible orders of execution and resource utilisation. In turn, constructing such models requires a software process *meta-model* formalism, which provides the necessary constructs to create various types of software process models. We have developed such a meta-model, which utilises a knowledge representation language and supporting environment, as others have done [20, 49], to specify, query, simulate and analyse software processes of interest [18].

Our approach employs a software process meta-model, derived from empirical studies of computing and software development work in different organisational settings [4, 6, 8, 14, 21, 40, 42]. It directly incorporates the kinds of information structures, presented earlier, to model software engineering processes and settings. This encourages us to model large team-based development efforts as multi-agent software processes in an open systems manner [41, 50]. This ability to model and simulate software processes as open systems is essential, since it allows process conflict to emerge through multi-agent interaction which, in turn, must be resolved locally (e.g. through agent-agent negotiations [6, 8, 21, 51], rather than through some automated global control mechanism. This provides us with the ability to model the kinds of things that people do in enacting different software processes. It also allows us to construct both prescriptive software process models and their descriptive realisations as a way of maintaining a project-oriented process knowledge base. Further, this process modelling framework allows us to more directly model and simulate not only routine and broken software development processes [6, 8, 21], but also the empirically observed processes of software technology transfer and transition reported elsewhere [7]. Thus, our reliance on empirically grounded models of open multi-agent software processes is a departure from previous efforts, as well as, we believe, an indication of future directions for software process modelling.

We have constructed a prototype knowledge-based environment for modelling, querying and simulating soft-

ware engineering processes occurring in complex organisational settings. Overall, we cannot do any justice to presenting the details of this environment here and must instead refer interested readers to a companion paper [18]. However, it is our intention that such an environment is employed to model and simulate the emerging teamwork structures and organisational settings that participate in the DSF project, so that we might better be able to understand how the DSF project works, how things break down, how they are repaired or replanned, and how new software services migrate and evolve.

6 The DSF project: an opportunity for academic-industrial co-operation

Given the software infrastructure we have described, we anticipate that a DSF project can take on both academic and industrial participants. However, each poses different problems and opportunities.

We expect that it is possible for multiple academic organisations to form loosely coupled DSF projects. A DSF infrastructure might quickly be deployed and applied among a collegial group of academic researchers and their graduate students if it was made available on a little or no direct-cost basis. This is often the tradition for academic software research technologies. However, software technology transfer within and across academic organisations clearly involves more than this [7, 10].

Although we are optimistic about the possible outcomes of academic-only DSF projects, we are much more reserved with our outlook for industry-only DSF consortium projects. Rather than examining such matters here, we will instead recommend to those interested that they examine the dealings, workings and track records of various industry-only consortia that have been established in recent years [52].

Not surprisingly, an academic-industrial grouping is the best participant mix. Academic research groups may already be prepared to collaborate and participate in a DSF project. Similarly, industrial organisations may seek to join academic DSF projects as they become more viable. Further, industrial participation may also bring additional resources to further underwrite and stabilise the DSF infrastructure, in order to both ensure its continuing operation and to use this 'external knowledge factory' as an emerging industrial resource and testbed. Such movement may then become self-reinforcing as other academic and industrial groups seek to become participants. This reinforcement might then be rechannelled, so that DSF projects which become too large will be internally segmented into smaller, more specialised niche projects that nonetheless can share similar infrastructures. The emergence and growth of VLSI chip design and fabrication technology is a suggested analogy here.

7 Conclusion

Clearly, the DSF project and its software infrastructure are ambitious undertakings. However, our view is that such an effort is both technologically feasible and socially desirable. As such, we can, in a sense, initially fail in our effort but succeed nonetheless, since the DSF project is an experiment conceived to be participatory, open-ended, evolutionary and technologically interesting. A decade of experience in the USC System Factory project has convinced us that such an undertaking is possible and can be made to work in an incremental bootstrapping manner. As noted throughout this paper, it is not our intention to be in control of such a wide-area project. Instead, we see our technological and organisational efforts as a possible starting point or model for national and international projects with similar collaboration goals and technological objectives. Thus, if the Distributed System Factory concept is viable, it should be able to succeed with or without us. Obviously, our interest is nevertheless to participate, to encourage others to participate and to provide an emerging wide-area software infrastructure that can help make it succeed. As such, this paper outlines the software infrastructure that can support the DSF concept and provides references to related work that further describes some of the technologies we are developing to this end.

8 Acknowledgments

This work is supported as part of the System Factory Project at USC. Current sponsors include AT&T, Northrop, Office of Naval Technology through the Naval Ocean Systems Center and Pacific Bell.

The author would like to thank Salah Bendifallah, Joe Chen, Song C. Choi, Pankaj Garg, Abdulaziz Jazzar, Anthony Karrer, Peiwei Mi, K. Narayanaswamy and John Noll for their significant contributions to the ideas or system development work on which this paper is based; Peter Danzig and the referees who also contributed comments that helped clarify this paper; and the 600 or so USC CS graduate students who have participated in this effort since January 1981.

9 References

[1] LEDERBERG, J., and UNCAPHER, K.: 'Towards a national collaboratory'. Report of an Invitational Workshop at the Rockerfeller University New York, 1989

[2] COMPUTER SCIENCE AND TECHNOLOGY BOARD: 'Scaling up: a research agenda for software engineering', *Commun. ACM*, 1990, **33**, (3), pp. 281–293

[3] SCACCHI, W.: 'The USC System Factory project'. Proc. Software Symposium '88, Software Engineers Association, Tokyo, Japan, 1988, pp. 9–41 (see also *ACM Softw. Eng. Not.*, 1989, **14**, (1), pp. 43–65)

[4] SCACCHI, W., GASSER, L., and GERSON, E.: 'Problems and strategies for organising computer-aided design work'. Proc. IEEE Int. Conf. on Computer-Aided Design, Santa Clara, California, 1983, pp. 149–152

[5] SCACCHI, W.: 'Understanding software productivity: towards a knowledge-based approach', *J. Soft. Eng. Know. Eng.*, 1991, **1**, (3)

[6] BENDIFALLAH, S., and SCACCHI, W.: 'Understanding software maintenance work', *IEEE Trans.*, 1987, **SE-13**, (3), pp. 311–323

[7] SCACCHI, W., and BABCOCK, J.: 'Understanding software technology transfer'. MCC Technical Report STP 3010-87, Software Technology Program, Microelectronics and Computer Technology Corporation, Austin, Texas

[8] BENDIFALLAH, S., and SCACCHI, W.: 'Work structures and shifts: an empirical analysis of software specification teamwork'. Proc. 11th Int. Conf. on Software Engineering, ACM, Pittsburgh, Pennsylvania, 1989, pp. 260–270

[9] SCACCHI, W.: 'Developing software systems to facilitate social organization'. Work with Computers, September 1989, Vol. 12A, pp. 64–72, Advances in Human Factors and Ergonomics (Elsevier, New York)

[10] SCACCHI, W.: 'The System Factory approach to software engineering education' *in* FAIRLEY, R., and FREEMAN, P. (Eds.): 'Issues in software engineering education' (Springer-Verlag, 1989)

[11] BERNSTEIN, P.: 'Workplace demoncraticization' its internal dynamics' (Kent State University Press, 1976)

[12] BJERKNES, G., *et al.* (Eds.): 'Computers and democracy — a Scandinavian challenge' (Avebury, Oslo, 1987)

[13] EHN, P.: 'Work-oriented design of computer artifacts' (Almqvist and Wiksell International, Falkoping, 1988)

[14] CURTIS, B., KRASNER, H., and ISCOE, N.: 'A field study of the software design process for large systems', *Commun. ACM*, 1988, **31,** (11), pp. 1268–1287

[15] BABCOCK, J., BELADY, L., and GORE, N.: 'The evolution of technology transfer at MCC's software technology program' from didactic to dialectic'. Proc. 12th Int. Conf. on Software Engineering, ACM and IEEE, Nice, France, March 1990, pp. 290–299

[16] SCACCHI, W.: 'Managing software engineering projects' a social analysis', *IEEE Trans.*, 1984, **SE-10,** (1), pp. 49–59

[17] GARG, P.K., and SCACCHI, W.: 'On designing intelligent software hypertext systems', *IEEE Expert 4*, 1989, pp. 52–63

[18] MI, P., and SCACCHI, W.: 'A knowledge base environment for modeling and simulating software engineering processes', *IEEE Trans.*, 1990, **KDE-2,** (3), pp. 283–294

[19] BASILI, V.R., and ROMBACH, H.D.: 'Tailoring the software process to project goals and environments'. Proc. 9th Int. Conf. on Software Engineering, IEEE Computer Society, Monterey, California, 1987, pp. 345–357

[20] OULD, M.A., and ROBERTS, C.: 'Defining formal models of the software development process' *in* BRERERTON, P. (Ed.): 'Software engineering environments' (Ellis Horwood, Chichester, UK, 1988), pp. 13–26

[21] MI, P., and SCACCHI, W.: 'Modeling articulation work in software engineering processes'. Proc. 1st Int. Conf. on the Software Process, 21st–22nd October 1991, Redondo Beach, California

[22] BASILI, V.R., and ROMBACH, H.D.: 'The TAME project: towards improvement-oriented software environments', *IEEE Trans.*, 1988, **SE-14,** (6), pp. 759–773

[23] GARG, P.K., and SCACCHI, W.: 'A hypertext system to manage software life cycle documents', *IEEE Softw.*, 1990, **7,** (3), pp. 90–99

[24] SCACCHI, W.: 'On the power of domain-specific hypertext environments', *J. Amer. Soc. Inf. Sci.*, 1989, **40,** (3), pp. 183–191

[25] NARAYANASWAMY, K., and SCACCHI, W.: 'A database foundation to support software system evolution', *J. Syst. Softw.*, 1987, **7,** pp. 37–49

[26] CHOI, S., and SCACCHI, W.: 'Assuring the correctness of configured software descriptions'. Proc. 2nd Int. Workshop on Software Configuration Management, Princeton, New Jersey, 1989, ACM Software Engineering Notes, **17,** (7), pp. 67–76

[27] KEDZIERSKI, B.I.: 'Knowledge-based project management and communication support in a system development environment'. Proc. 4th Jerusalem Conf. on Information Technology, 1984, pp. 444–451

[28] MALONE, T., GRANT, K., LAI, K., RAO, R., and ROSENBLITT, D.: 'Semi-structured messages are surprisingly useful for computer-supported coordination', *ACM Trans. Office Inf. Syst.*, 1987, **5,** (2), pp. 115–131

[29] ELLIS, C.A., GIBBS, S.J., and REIN, G.L.: 'Groupware' the research and development issues', *Commun. ACM*, 1991, **35,** (1), pp. 38–58

[30] FRAKES, W., and NEJMEH, B.: 'Software reuse through information systems'. Proc. COMPCON '87, IEEE Computer Society, San Francisco, California, 1987, pp. 380–384

[31] WOOD, M., and SOMMERVILLE, I.: 'A knowledge-based software components catalogue' *in* BRERETON, P. (Ed.): 'Software engineering environments' (Ellis Horwood, Chichester, UK, 1988), pp. 116–131

[32] MALONE, T.M., YATES, J., and BENJAMIN, R.: 'Electronic markets and electronic hierarchies', *Commun. ACM*, 1987, **30,** (7), pp. 484–497

[33] ZAHN, L., *et al.*: 'Network computing architecture' (Prentice-Hall, Englewood Cliffs, New Jersey, 1990)

[34] CLEMM, G., and OSTERWEIL, L.: 'A mechanism for environment integration', *ACM Trans., Programm. Lang. Syst.*, 1990, **12,** (1), pp. 1–25

[35] NOLL, J., and SCACCHI, W.: 'Integrating diverse information repositories: a distributed Hypertext approach' 1991

[36] ONGKOWIDJOJO, M., COBANKAIT, M., and YEN, S.T.: 'Distributed graph shell' project documentation'. USC System Factory Documentation, Los Angeles, Spring 1990

[37] GARG, P.K., and SCACCHI, W.: 'Composition of Hypertext nodes'. Online Information '88 Proceedings, London, December 1988, Vol. 1, pp. 63–73

[38] DELISLE, N., and SCHWARTZ, M.: 'Contexts — a partitioning concept for Hypertext', *ACM Trans. Office Inf. Syst.*, 1987, **5,** (2), pp. 168–186

[39] HEWITT, C.: 'Offices are open systems', *ACM Trans. Office Inf. Syst.*, 1986, **4,** (3), pp. 271–285

[40] GASSER, L.: 'The integration of computing and routine work', *ACM Trans. Office Inf. Syst.*, 1986, **4,** (3), pp. 205–225

[41] GERSON, E.M., and STAR, S.L.: 'Analyzing due process in the workplace', *ACM Trans. Office Inf. Syst.*, 1986, **4,** (3), pp. 257–270

[42] KLING, R., and SCACCHI, W.: 'The web of computing: computer technology as social organization', *Adv. Computers,* 1982, **21,** pp. 1–90

[43] HUFF, K.E., and LESSER, V.: 'A plan-based intelligent assistant that supports the software development process'. Proc. Third Symp. on Software Development Environments, ACM SIGSOFT, Boston, Massachusetts, November 1988, pp. 97–106

[44] NARAYANASWAMY, K., and SCACCHI, W.: 'Maintaining configurations of evolving software systems', 1987, *IEEE Trans.*, 1987, **SE-13,** (3), pp. 324–334

[45] SCHEIFLER, R., and GETTYS, J.: 'The X Window system', *ACM Trans. Graphics*, 1986, **5,** (2), pp. 79–109

[46] LAI, K-Y., MALONE, T.W., and YU, K.: 'Object lens: a "spreadsheet" for cooperative work', *ACM Trans. Office Inf. Syst.*, 1988, **6,** (4), pp. 332–353

[47] KARRER, A., and SCACCHI, W.: 'An extensible object-oriented tree/graph editor'. ACM SIGGRAPH Symp. on User Interface and Software Technology, Snowbird, Utah, October 1990, pp. 84–91

[48] CHOI, S.C., and SCACCHI, W.: 'Extracting and restructuring the design of large systems', *IEEE Softw.*, 1990, **7,** (1), pp. 66–71

[49] MADHAVJI, N.H., GRUHN, V., DIETERS, W., and SHAFER, W.: 'Prism = methodology + process-oriented environment'. Proc. 12th Int. Conf. on Software Engineering, ACM and IEEE, Nice, France, March 1990, pp. 1–3, 277–289

[50] HEWITT, C.: 'Open information systems semantics', *Artif. Intell.*, 1991, **47,** pp. 47–106

[51] ROBINSON, W., 'Negotiation behavior during requirements specification'. Proc. 12th Int. Conf. on Software Engineering, ACM and IEEE, Nice, France, March 1990, pp. 268–276

[52] EVAN, W.M., and OLK, P.: 'R&D consortia: a new U.S. organizational form', *Sloan Manage. Rev.*, 1990, **31,** (3), pp. 37–46

The author is with the Computer Science Department, University of Southern California, Los Angeles, CA 90089-0782, USA.

The paper was first received on 18th June 1990 and in revised form on 3rd June 1991.