Introduction

The purpose of this document is to provide a generic resource for any future researcher who may be interested in the domain of tool integration within software engineering environments.

This document provides an annotated bibliography of papers that have addressed this topic over the past decades. The author does not contend that this is a complete list of papers, merely, that these papers are those that have been discovered to be the most salient over the course of three years of research into this topic. The author does not intend to draw any conclusions within this document, merely to present a reviewed list of relevant literature.

This document consists of a single section, with two appendices. The main body is a list of all the identified papers within this subject in alphabetical order of first author, with the particular research effort described. The first appendix (page 59) provides a bibliography of references quoted in the main section of this document, as these references are not directly concerned with tool integration in software engineering environments. The second appendix (page 60) provides an index of all the authors that have papers included here, together with the years that each paper was written in. The intention is to give the reader alternative means to access this bibliography should the first authorship be deemed insufficient.
The IEEE has created a standard that provides a framework for CASE tool interconnections. These tool interconnections are defined in terms of the user, organisation, platform and other tools. This standard also describes the behaviour between users and other tools, and defines a semantic transfer language for the exchange of software descriptions between tools.


Abeti et al. describe an approach to the design of software agents that use Ontologies. Ontologies are used in the creation of semantic web services, where a semantic infrastructure is added to the static, existing web. Here a meta-layer is added, allowing the interoperability and sharing of tasks and information via the use of an ontology. An ontology is a set of terms and their descriptions, together with their relations, that build up to describe a specific domain of knowledge. This approach is intended to automate tasks and therefore ease the flow of work within an organisation.

The authors overview each of the relevant technologies being: UML Data Binding (UDB), which allows the translation of UML models to RDF and Java Classes; UML Based Ontology Toolset (UBOT), an architecture for managing DAML ontologies with UML GUIs; Agent Communication Languages (ACLs), and the considerations for their use with UML; and finally, an Ontology Driven Design (ODD) method, to create semantic web applications.

Finally the authors then describe an architecture using UDB and UBOT called UML for Knowledge Representation (UKR), based on style-sheet translation. This UKR method can then be used to extend the ODD approach to support the development of Agents, eventually arriving at an Ontology Drive method for Agent (ODA) methodology.
Altheide et al. describe a sustainable architecture for tool integration, by suggesting that users of an integrated development environment are looking for the ability to exchange data, maintain consistency, and support automatic process integration, as well as sufficient flexibility and extensibility. They suggest that a common interface for controlling tools is a better approach, and that it is inevitable that there will be a trade-off between the functionality of the integrated environment, against the sustainability of the environment over the long term. This suggested trade-off is illustrated in Figure 1. A single interface allows a more uniform mode of interoperation between tools, and makes use of an information backbone, to which tools are connected via a "Toaster-esque" adapter. This approach allows the federation of tools to be treated as a service provision mechanism, by segmenting functionality into reusable blocks within a framework. It also encourages the creator of the environment to adopt a "plug and play" mentality.

Figure 1 – A trade-off between Sustainability and Sophistication within Tool Integration, as suggest by M N Wicks
Eclipse [1] is an example of a contemporary and popular Integrated Development Environment (IDE) that has also been studied with respect to tool integration. Amsden suggests another tool integration continuum for Eclipse, consisting of five levels, starting at: None, where there is no requirement for the tools in Eclipse to interoperate; Invocation, where Eclipse simply invokes the corresponding tool for the selected resource type; Data Sharing, where tools in the Eclipse platform operate on the same set of data; an Application Programming Interface, where tools within the Eclipse platform provide an interface for consistent behavioural reuse of tools; and finally User Interface integration, where tools are invoked seamlessly within the platform without the user realising that multiple and separate tools are actually being used. Figure 2 illustrates this new continuum, showing again the popularity of categorising integration using multi-levelled continua, and indicating a surprising degree of correspondence with Rader et al.’s CASE Tool Adoption Continuum (see Figure 5).

![Figure 2 - Amsden's Tool Integration Continuum for the Eclipse IDE](image)

Arnold considers the links between Computer Supported Cooperative Work (CSCW) and tool integration, and suggests that there are 3 levels of integration, Individual, Group and Enterprise, making up a continuum of capabilities. When moving from the Individual level to the Group, Process and
Collaboration must be considered in the Integration equation, likewise moving from Group to Enterprise. Multiple groups bring their processes and culture to the new greater group, so environments must be able to adapt to situations of multiple simultaneous processes or collaborations.

Arnold continues by describing a mediator system that integrates process centred environments with collaboration environments. He defines process support to be when a single user manipulates artefacts, and collaboration support to be when multiple users manipulate artefacts at the same time. The resulting system uses control integration and “black-box” applications to provide a message bus, however admitting that data integration is not achieved.


Aversano et al. describe the workflow management system within Genesis, consisting of a process definition tool, a project management tool, a workflow engine and a work-list handler.


Baentsch et al describe a mechanism to provide a compile facility via a CGI program activated via a web interface. The user of the facility has to create a set of documents that describe the system being built, and to store these on the web server that hosts the Webmake tool. Context sensitive links are incorporated into the system that decide where to schedule jobs to execute the system build that has been requested. The system can either request the code to be compiled with the object code returned, or if the requested systems are busy, then the source on its own is returned. The system then decides itself to delegate the compilation to another system. Further detail is included in the document itself, but problems are discussed, mainly centred around the problem of achieving asynchronous data communications using a synchronous protocol. Security remains a problem, and performance issues remain due to the use of CGI processes instead of threads.

Baik et al. discuss the COCOMO cost model for software development and suggests that a single "use of tools" parameter is now insufficient for use within this model. A more sophisticated parameter should now be used, and the authors discuss the degree of tool integration on a 6 point scale, based on Wasserman's (1989) five dimensions of tool integration. Also considered in this new model, is the range of coverage provided by the tool, as well as the maturity of the tools being used.


Baldamus et al. suggest that the integration of Formal Verification Toolkits (FVTs) should be mediated via web services, because service integration is at the core of this paradigm, and that the promotion of distributed and open integration matches the philosophy of the Formal Verification community.


Ballarini et al. discuss Genesis in more detail, describing two principles that have driven the design of this system, being: a design based firmly on defined user requirements for cooperation, collaboration and coordination; and an integrated conceptual approach, considering process management, artefact management, resource management, and event management all together.


Bandinelli et al. describe the SPADE-1 Process Centred Software Engineering Environment (PCSEE), being an environment for software process analysis, design and enactment. Tool integration is discussed with respect to Data Integration, with the authors suggesting that object-oriented
database management systems (OODBMSs) are ideally suited for this application.


Bao and Horowitz present a novel approach to integration with the use of Commercial-Off-The-Shelf (COTS) products. They believe that existing integration frameworks provide either "White", "Grey" or "Black Box" approaches to integration only, and that this must be updated to incorporate support for COTS products. They further suggest that a dynamic framework is required which consists of four parts: a dynamic integration model which separates the relationships from the tools themselves and provides a dynamic binding mechanism; a method that uses non-traditional interfaces for tool integration such as graphical user interfaces and system interfaces; a Tool Integration Language (TIL) that can access a tool through its interface; and finally, a Tool Integration Server System (TISS). The resulting framework is based on imitating a user interacting with a tool through a GUI. The overall effect of this approach is not to integrate the tool into the framework, but rather to integrate the framework into the environment. To support this architecture, Bao and Horowitz have created a dynamic integration model which separates out relationships from the tools, with the relationship treated as a separate entity having semantic meaning as well as physical identity, with the result that tools are invoked indirectly. Control integration is achieved by monitoring the GUI system itself. Data integration is achieved by using a repository. Bao and Horowitz conclude that this framework has a number of advantages, including no requirement for special interfaces, together with the modelling of relationships between tools decouples the integration policy from the mechanisms.


Barthelmess provides a comprehensive survey of the current state of the Process Centered Software Development Environment (PCSDE) area, with a wealth of references. This paper provides an ideal jumping-off point for any further research into tools and PSEEs. Thirteen tools are discussed in the paper: Adele/Tempo; ALF; APPL/A; EPOS; MARVEL; Merlin; OIKOS; Oz; Process Weaver; Prosyt; Provence Shamus; and finally SPADE. The author concludes by suggesting that there is a trade off between the benefits of adopting PCSDE solutions, and the cost and complexity of building a sufficiently rich supporting process model.

Belkhatir and Ahmed-Nacer provide a historical setting to the progression from configuration management systems, software engineering databases, towards the software process. The authors identify three separate paradigms for the software process: Rule; Process Programming; and Graph/Net. The authors then suggest that product and process modelling can be integrated to improve the level of automation provided by an environment. The resulting problem of consistency is identified. This problem is only addressed within a Process Software Engineering Environment (PSEE), an architecture for such is described in the article. The main conclusion from this work is that current data models are insufficient for all the variety and complexity needed within a PSEE, to increase the richness of available semantics.


Belkhatir and Estublier discuss the Adele database of programs, which was built to explore issues surrounding large software developments. The authors contend that only the physical sharing of objects is a solved problem, whereas logical sharing, access rights and structuring of teams and products, are not solved issues. These issues are explored through the development of Adele, which are briefly described in this paper.


Bernas describes the PALAS-X Software Development Environment, which is a configuration management tool with tool integration capabilities, and is built on top of Adele 2, whose primary focus is the support of the development of safety critical systems. Within PALAS, a module is represented by a set of interfaces, interface bodies represent diverging implementations, with the branches supporting development increments within a revision.

Bernas contends that the goal of integration is to provide the most homogeneous access to the most heterogeneous set of tools, and propose the use of process management techniques that address the problem from three angles, interface, tool processing data, and long term data. He decided to use a standard interface and to dismiss the direct interaction of tools, instead to concentrate on repositories, in order to manage data consistency. Bernas then considers control integration against data integration, and
decides that data integration is more important for safety critical systems, and
then finishes by defining a policy to enforce this (all access is via the PALAS
interface; full data integration for locally stored information in a central
repository; and central storage of local models).

W R Bischofberger, T Koflera, K-U Matzel, and B Schaffer. Computer
supported cooperative software engineering with beyond-sniff. In

Bischofberger et al. suggest a new genre of software engineering, called
Cooperative Software Engineering (CSE), with enhanced cooperation, not
merely using simple coordination of tasks. They contend that cooperation can
either be policy driven via a formalised exchange and handling of structured
documents and artefacts, or informal, where exchanges are unrestricted.
They suggest that the software process is a mixture of both these cooperation
activities, thus "The challenge of Project Management is to work towards an
optimal relation between them (Policy Driven and Informal cooperation
activities), depending on the specific characteristics of a Project".

Bischofberger et al. continue by discussing how Process-centred Software
Engineering Environments (PCSEs) and Computer Supported Cooperative
Work (CSCW) systems do not fully meet the requirements of Cooperative
Software Engineering (CSE), by explaining how their CSE tool meets key
requirements for tool integration. These requirements satisfy both control and
data integration through the use of point to point communication for request
processing, combined with multicasting for notification, resulting in scalability.
Also these requirements satisfy data integration solely, through the use of a
federated approach to data management. A federated approach to data
management does not specify the creation of an overall data model, instead,
each service is responsible for the consistency of their own data.

Cornelia Boldyreff, David Nutter, Stephen Rank, Mike Smith, Pauline Wilcox,
Rick Dewar, Dawid Weiss, and Pierluigi Ritrovato. Environments to support
collaborative software engineering. In A Cimitile, A De Lucia, and H Gall,
editors, Cooperative Methods and Tools for Distributed Software Processes,
volume 380.222 of RCOST / Software Technology Series. Franco Angeli, 1

Boldyreff et al. suggest that globalisation is having a significant impact on
software development and this, combined with the increasing longevity of
software products, has increased the need for collaborative software
engineering approaches. Two approaches are explored in this paper, being
Ophelia (Wilcox et al. 2003, Weiss et al. 2003) and Genesis (Aversano et al
2003, Ballarini et al 2003, Nutter et al. 2003). Genesis is an open-source
platform to support cooperation and communication combined with an artefact
management module, whereas Ophelia has a similar aim, but concentrates on
products, by providing a set of core interfaces to support interoperability
between different tool categories.

Bosua and Brinkkemper discuss how to integrate the development of both procedural and object oriented systems, by suggesting meta-modelling as a solution. They draw a distinction between tools that are heterogeneous, supporting different paradigms, and homogeneous tools that support a single paradigm, and continue by arguing that integration must instead, be performed at a conceptual level to maximise the adaptability and flexibility of any solution. Bosua and Brinkkemper contend that merely integrating a set of tools is not sufficient, as the tools may overlap or overlook steps within particular activities of the intended development process. They consider the relationships between the degree of conformity between method and tool, the conceptual model of a particular method, and the method data and process models. These relationships are then used to link tools, a process that Bosua and Brinkkemper state has not yet been practically realised.


Brown considers control integration, and suggests that an integration strategy must be both flexible and adaptable to suit differing users, whilst remaining efficient and simple. Brown suggests that integration techniques up to 1993 have focused solely on data integration, whereas a control integration strategy based on message passing would be a more appropriate solution. Brown identifies three avenues of work regarding integration: the definition of new mechanisms for integration; integration semantics; and the relationship between integration and process. The general conclusion of all of these strands of research is that integration is a more subtle and pervasive characteristic of a software development environment, than just simple tool to tool communication. This proposition then undermines the prevailing belief that integration should be achieved using a central repository with standardised data structures. In such a solution, a large software system is required to centrally manage all data, resulting in a reduction in usability of the overall software development environment itself as it soon becomes too cumbersome to manage and use. Instead Brown suggests that research into control integration provides solutions that are simpler and easier to manage, that would be typically enacted using a broadcast or multicast message server, or via a point-to-point communications facility.

Brown discusses three systems that use message passing to achieve control integration; FIELD, SoftBench, and ToolTalk, and then continues by introducing a conceptual framework to classify the level to which a tool is able to be integrated (see Figure 3). This framework has five levels: Carrier, where
a single file format is required; Lexical, where a common understanding between tools of shared structures is supplied; Syntactic, that provides an agreed set of data structures and how to manage them; Semantic, describing a common understanding of the shared structures; and finally Method, where a common model of the software development process is supplied. Brown suggests that the FIELD system has reached the Carrier level, SoftBench rises to the Syntactic level, and finally ToolTalk has reached the Semantic level.

Increasing Tool Integration Capability

Method
Semantic
Syntactic
Lexical
Carrier

Figure 3 - Brown's 5 Steps to classify the degree of Integration Capability for a Tool

Brown concludes by suggesting that for tools to reach the Method level, policy or process information must be encoded in the message passing mechanisms. He finishes by addressing practical issues that could be incorporated into tools to aid integration. Tools must be extensible and tools must be able to be "encapsulated" to allow third party tools to be integrated. This requires that the message passing paradigm must be extended to other forms of integration and that a standard messaging protocol must be used, for example CORBA.


In Budgen et al., an attempt is made to discover whether CASE tools integrate with industrial practices. They believe that good design practices must incorporate an element of opportunism, such that the development of software cannot be fully defined using a rigid and prescriptive method. They suggest that it is unrealistic to assume that all development follows a "waterfall" pattern, and that not all development proceeds "top-down". They continue by
suggesting that the design of CASE tools reflects how the methodologies should be used, rather than how designers actually decide to use the methodologies. As a result of this belief, Budgen et al. set out consequent implications for the design of more useable CASE tools, with the recommendation that a successful tool must only represent designs that are capable of being written with a pen on a single piece of paper. Also they suggest that tools must provide an interface that reflects the way the designer works, and not how the data is stored. They conclude by suggesting that Wasserman's (1989) five original dimensions for CASE tool integration should be extended beyond the need to provide simple presentational integration, and to provide a sixth, designer-centred dimension, termed "Cognitive Integration".


Budgen and Thomson review the various techniques that can be used to evaluate a prototype CASE tool. The authors design a set of experimental studies to extract data, in order to assess the degree to which the tool meets its design goals. The paper outlines various design strategies, by contrasting the possible approaches to computing research. These approaches are: Scientific; Engineering; Empirical; and Analytic. However, all of these approaches involve an Evaluative element. The paper continues with background material on evaluation methods.


Burmester et al. describe the Fujaba real-time tool suite, not to be confused with Fujaba tool modelling facility, as described by Geiger and Zündorf (2006). The Fujaba real-time tool suite is an attempt to provide an answer to ever more complex functionality in real-time systems. This suite supports model drive development, by using a restricted high-level version of UML, which provides incremental model checking. This approach uses compositional reasoning to solve the problem of design verification.


Cao et al. describe Pounamu, a meta-CASE tool that uses XMI-based data exchange as the integration mechanism.

Chen and Chou describe a monolithic approach to the problem of consistency checking throughout the development process, by creating a new language to define the inter-relationships between artefacts that trigger actions. Chen and Chou describe an environment for such a system, based on the client server model, designed around the wrapping of Java classes.


Chen and Norman describe the state of the art for CASE technologies at the start of the 1990s, and they contend that integration is a key factor in limiting the take up of tools. The technical framework for integrated CASE is discussed, covering data (Direct, File based, Communication based and Repository based), control and presentation integration, but the authors suggest that tool integration on its own is not enough, and must be placed within a context of an organisational framework. The authors note that not even the ECMA or "Toaster" reference model (see Earl 1989) is not sufficient, as the integration it offers, both vertical (full life cycle) and horizontal (methodological), is lacking.

Chen and Norman suggest that there are three levels of activity that impact on systems development, being: Infrastructure planning; project management; and software development levels. Successful integration can only occur when these levels have been addressed. They go on to conclude that integration must be balanced with flexibility usually tight integration means less openness.


Chou and Chen provide a thorough exposition on the subject of Process Centred Software Engineering Environments (PSEEs), and in particular, a PSEE called CSPL (Concurrent Software Process Language). The authors identify three problems with current PSEEs: difficulties in planning activities; process change during enactment; and the length of time large objects take to build. The paper suggests that the first two problems could be tackled by adopting process evolution, with the third problem solved by allowing object decomposition. The paper continues by describing the CSPL itself, and then describing process evolution in CSPL. Also of interest in this paper are the supporting references to others working in the field, and the authors have included a discussion of the various techniques that they have adopted to tackle similar problems.
Chou introduces an agent based approach to solve problems with PSEEs, with a decentralised process engine called agent-based decentralised process engine (ADPE). This decentralised approach solves the problem of when a central server is overloaded and becomes a bottleneck, and also when the central server becomes unavailable. ADPE consists of a developer interface a language translator, a product manager, a product base, a token manager and a developer agent. The developer agent enacts process programs, handles exceptions, manages products and relationships, and finally handles product change.

Chou contends that heterogeneous Process-centred Software Engineering Environments (PSEEs) need to be coordinated, therefore DPEM and DPEL have been created (Decentralised Process Enactment Model and Decentralised Process Enactment Language). The paper concludes with a detailed description of both DPEM and DPEL.

Chou et al. describe an engine to decentralise a Process-centred Software Engineering Environment. The authors survey many existing PSEEs as well as Workflow Management Systems (WFMSs), and continue by describing their DPE/PAC model.

Chung et al. present a method to integrate tools using knowledge engineering techniques. They start by describing three ways to integrate tools in a software engineering environment, "Brute force" involving translation of data between tools, "Vendor dependant" involving buying all your tools from one vendor, and "Vendor independent" allowing tools from different vendors to be used together. Chung et al. suggest that no current tool integration product covers all these levels of integration, hence their creation of an architecture called the Tool Integration Platform (TIP). TIP consists of five elements, including a message server, an interface, an integration inference engine, and inference rules stored in a repository.
Corradini et al. present an agent-based approach to tool integration, with particular emphasis on the bioinformatics domain. Each tool is wrapped using a wrapper agent, with the result that the tool is presented to other tools via a common interoperability mechanism. A workflow is then constructed using a set of agents to produce a user application. This pool of agents, or Workflow Executors, can then enact the workflow itself, via the agent platform. The authors suggest that users cannot or should not be bothered with detail on how to set up the technology, so a three layer architecture was created, each layer split into two conceptual layers: the User layer defines workflows, by extending UML Activity Diagrams to include UML notes, specifying input and output parameters, configurations and global variables, known as configuration notes; the System layer houses the Workflow Executors, and is mapped to the User layer via a User-Level Activity Diagram (ULAD); finally the Run Time layer provides tools to support deployment and agent execution.

Costagliola et al. describe a meta-CASE workbench which leverages the graph exchange language (GXL) to achieve the resulting possibilities within the CASE environment. The tool consists of a modelling environment generator (MEG) and a workbench generator (WoG). GXL is also used to achieve data integration with any tools required in the environment.

Cybulski and Reed describe an attempt to provide hypertext mechanisms for a CASE repository, called HyperCASE. The authors suggest that any CASE system should have mechanisms for not only creating software artefacts, but to create navigable links amongst them, and to allow this links to browse the repository itself. The paper refers to the work of Mi and Scacchi (1992) as an example system that uses elements that they propose here.

The key integration feature of the paper is the ability of HyperCASE to integrate tools by combining a hypertext based user-interface, with a common knowledge-based document repository. The tools referred to here include
HyperEdit, HyperBase and HyperDict that combine to make up HyperCASE. The rest of the paper describes these three tools in detail.


D’Ambrogio and Iazeolla discuss the derivation and evaluation of software performance models, and describes an environment to exchange models via metadata exchange, typified by MOF/XMI. MOF/XMI eases the integration of UML-based CASE tools.

There are now more facets to the interoperability question, including now metadata-based model interoperability. This use of standardised meta-models allows metadata to drive environment tools via introspection, reflection in some programming languages, as well as facilitating model interchange, and the linking of UML-based software development tools.

In this case tool integration is achieved by wrapping tools within Web Services. MOF-Based interoperability is achieved via the use of an XMI-bus, connected to an XMI repository. This repository consists of 5 components to manage the resulting XML documents: a Transformer; a Merger; a Producer; a Creator/Evaluator; and a Wrapper.


Damm et al. explore integration issues with XMI and component technology (for example COM), by building an electronic whiteboard system. They were able to explore issues of asynchronous communication with shared data, and synchronous communication with separate data, using a multitude of architectural styles and separate components. Damm et al. were able to carry out experiments to evaluate different possible architectures, and were able to suggest that a standard interchange format was required, with the conclusion that the UML meta-model implemented using XMI was not stable enough for consistent data exchange as data could be lost. Not only this, but they highlighted the lack of a standard programming interface required to achieve both synchronous and asynchronous communication and conclude that standards must be implemented, but that this implementation must be both extensible and flexible.

Desikan and Bulusu describe another technique for data exchange that uses a framework for developing dynamic model driven translations with transformation tools. These tools are combined with a translation modelling language within a framework to provide tools and APIs, that produce dynamic translations. The hypothesis behind the Sankhya Translation Framework (STF) is to tackle the problem of data conversion between formats (Sankhya is the name of the company for whom Desikan and Bulusu work). Sankhya Translation Modelling Language (STML) is used to model language grammars, document schemas, and translation and transformation rules, and can be used to establish equivalence between representations of the same information held in different models. The framework has been compared by Desikan and Bulusu to XML, XLST, and grammars, and it is suitable for use in the development of data integration tools, model-driven program translation tools, natural language translators, and finally model-driven document servers.

Earl introduced the concept of the “toaster” or reference model for computer aided software engineering environments (CASEEs), see figure 4.

![Figure 4 - The ECMA “Toaster” Reference Model for CASE Environments](image)

1. User Interface Services
2. Task Management Services
3. Data Integration Services
4. Data Repository Services
5. Operating System Services
6. Message Service Network
7. Tool Slots

Earl intended this reference model for others to use to develop software engineering environments to a common standard. Earl describes the requirements of such a reference model being: usable to describe proposed environments; focus on interoperability of tools; support a wide range of configurations, thus supporting flexibility; be a tool for educating software engineers; be a framework for future standards; be concise; and be complete.
Earl continues by describing all the components of the reference model being: operating system services; data repository services; tools; task management services; a message server network; and a user interface (as reflected in figure 4). This reference model was subsequently used by the European Computer Manufacturers Association (ECMA), to create PCTE-like software products, thereby providing a means to integrate software engineering tools.


Eikemeier et al. present a peer-to-peer collaboration application targeted at large teams working together to support software engineering tasks. They believe that instant messaging is not sufficient for this type of work, and that distributed applications working in dynamic physical environments discourage the take-up of centralised process-oriented platforms. Eikemeier et al. suggest light-weight tools to support dynamic ad-hoc collaboration, and describe such a tool built using JXTA. They describe how messaging technology is not yet sophisticated enough to support proper knowledge management, in both elicitation and sharing, and continue by describing various peer-to-peer technologies. Eikemeier et al. identify three principles for any potential system: resource sharing; servers that can act as clients; and a topology that can be dynamic.


El-khoury et al. discuss the need for tool integration for multi-disciplinary development that uses standard practices from both computer and mechanical engineering. They contend that whilst it is desirable to have a single tool for system development, this will never practicably be realised, so instead they suggest that a platform must be made available to allow the integration of required tools by: supporting domain specific tools and languages; share data and view integration; and provide the ability to manage models. Their suggested architecture comprises three components: a data repository; meta-model mappings; and an adaption layer between the previous two components. The authors conclude by suggesting that they share similar aims to the work of Freude and Königs (2003), as well as Burmester et al. (2005).
Estublier et al. describe the process by which they have designed a PSEE, called Adele, in response to the problems with existing CASE technology. As such the paper provides a useful historical background, detailing the movement from file system-based approaches, to database approaches, suggesting that the way forward is for a PSEE to incorporate a versioned repository, combined with executable formalisms and a facility for managing long transactions. The authors suggest that there are two ways of process modelling: the rule-based paradigm, typified by the use of artificial intelligence techniques; process programming, in which the whole software process is described as a meta-program (see [2]); and finally active databases, where a database stores Event-Condition-Action rules, into this category Adele fits.

Estublier et al. conclude that PSEEs must address four issues: machine and operating system independence; provide tool and application writer support; provide tool interoperability; and support CASE builders. They suggest that PCTE systems address the first two points, Softbench addresses points two and three, with Adele addressing points three and four. By providing a PCTE interface, the authors suggest that Adele will answer the first two issues, but suggest that further work for successful PSEEs must address cooperating heterogeneous distributed repositories, CASE cooperation, and process and policy programming.

Estublier and Garcia discuss the problems associated with concurrent access to software artefacts, as most software configuration management (SCM) systems provide poor support for maintaining data consistency. They contend that making developers aware of the problem is key to reducing the potential for inconsistency, but that this awareness issue must take into account the cooperative process and system model being used. The authors use their tool called Celine, to explore this phenomena.

Flatscher describes in great detail the CASE Data Interchange Format (CDIF) standards, its history and origins semantics and syntax, and its relationship to MOF is noted.

Freude and Königs describe an integration framework to provide both control and data integration, that leaves the management of the artefact data to the tool itself, and concentrates on specifying functional dependencies. XML is used to exchange the tools' data, with reference objects used to represent each artefact managed by a tool, and consistency relations used to navigate between objects. The overall approach is realised by implementing an API that is similar to the Java Metadata Interface. The repository is then visualised to allow the user to understand the structure of the project, together with its dependencies.


Fromme and Walker describe the SoftBench framework, based on the ECMA "Toaster" model (see Figure 4) for creating software development environments. SoftBench allows tools to the "plugged" in, depending on user requirements, and makes use of a broadcast message server (BMS) to achieve this plug-ability. The BMS acts as a central point of communication, built directly on top of the sockets interface. Integration within SoftBench is achieved via an API called the SoftBench Encapsulator, through which tools are wrapped as objects that are then used in the environment and controlled by the BMS. Therefore using SoftBench allows a customised environment to be built out of tools, which provides a degree of flexibility.


Fuggetta describes the state of the art within the PSEE domain, and sets out the challenges and opportunities that need to be addressed. A framework for CASE technology is set out, together with a description of all the different possible paradigms of process model that must be selected within a PSEE. The architecture of PSEEs is discussed, together with a list of areas that need to be addressed, such as flexibility within the process model, as well as support for distributed and heterogeneous development. Also mentioned is the problem of Tool Integration, and various projects that have attempted to address this issue. Bandinelli et all (1994) is discussed as an example of one of these integration attempts.
Garlan introduces the concept of using Views to integrate tools within a software development environment. The idea behind a “view” addresses the problem of different tools needing different data models to refer to the data, which then cannot easily be interchanged between the tools. A “view” is a description of the data required by a tool, yet stored within a common set of objects. Each tool accesses the object it requires through the view that it defines, resulting in a common repository of software artefacts across multiple tools. Views encourage modularisation, sharing, and reusability, however the problem of type compatibility is introduced, which is the problem of mapping operations from one type to another. Garlan continues with a detailed description of the implementation of Views, and concludes by suggesting the advantages of views significantly outweigh the disadvantages.

Gautier et al. discuss their experiences of service creation environments in the BOOST project, which considered tool integration both by encapsulation as well as direct tool communication. They start by discussing the Wasserman themes of integration, but suggest that process integration should be more deeply defined at the carrier, lexical, syntax, semantic and method levels, as discussed by Brown (1993). Gautier et al. relate their experiences with encapsulation by suggesting that success depends on factors such as the presence of an API, consistent user interface, the complexity of a tool inputs and outputs, and suggest that the Tool Control Language (TCL) realises this goal, by encapsulating any required tools.

Geiger and Zündorf present a small tutorial on the Fujaba system, that models tools by using a graph grammar approach. They provide a detailed explanation of the Fujaba’s abstract and concrete syntax, static and operational semantics together with model transformations, consistency checking and persistency support features. Also mentioned in passing, is Fujaba’s ability to exchange models by using XMI.

Gerber and Raymond describe the commonality between MOF (Meta Object Facility) and EMF (Eclipse Modelling Framework), and suggest that a translation facility between the two would be useful. Hence they have produced a tool called E-MORF, which is built in XLST, combined with the use of JavaML (an XML description of Java Classes) in the transformation.


Gisi and Kaiser describe a software engineering environment called Marvel, which uses a rule-base approach to model software processes. Marvel uses tools to execute required software processes, with these tools being integrated into the whole through the use of envelopes that wrap the individual tools. Each step of the process is called an activity, and typically these are enacted by commercial-off-the-shelf (COTS) tools. These activities are viewed as “black boxes”, with Marvel itself having not knowledge of the internal workings of the activities, instead only knowing the input and output parameters. An envelope represents the implementation of an activity, and the authors describe how they have moved away from using the Unix shell language to instead use an extended shell language. This new Shell Envelope Language (SEL) requires full declaration of attributes and associated types. The new language can return more complex information instead of a binary success or failure marker regarding a certain process, in fact, an arbitrary number of value can be returned as desired. The authors conclude by suggesting that these improvements have made tool integration easier.


Grundy et al. describe the history of a number of attempts to combine CSCW systems with Process-centred Environments (PCEs). The authors note that CSCW systems support shared editing, but lack coordination support; and that PCE systems have a limited range of coordination strategies and are hard to integrate with third party tools. CSCW systems and PCEs can be integrated, but the result lacks flexibility for specifying coordination and tool integration. The authors describe the systems that they have created over time to explore these problems.
Starting with C-SPE, which is a distributed software development environment allows semi-synchronous and asynchronous editing, was found to lack facilities to determine why artefacts were changed, as well as to determine what other artefacts were changed at the same time. Next came SPE-Serendipity, as process modelling and enactment environment, which was found to not easily integrate tools, as well as performing poorly. Serendipity was then "rearchitected" using Java Beans component technologies, enabling an incrementally extendable environment, which is more easily controlled by a process modelling environment.


Grundy et al. describe the problems encountered when they tried to port a suite of software engineering tools from Prolog, to C++ and Java. The resulting environment included four tools: BuildByWire, a constraint based visual notation construction kit that allows a user to visually create element by direct manipulation; JViews, a modelling repository; BBW Composer, a tool that uses BuildByWire and JViews to create components; and finally, JComposer, a higher level design tool).

Grundy et al. discuss the traditional tool integration issues (data, control, presentation and process integration), with examples of the various flavours of system and where their strengths and weaknesses lie. The authors contend that their tool described here solves the four key integration issues.


Grundy et al. discuss a component-based approach to tool integration, in many domains including Software Engineering. A component-based approach is a "plug-and-play" answer to "monolithic" applications, and offers the prospect of increased reuse. This solution, called aspect-oriented software development, as coined by the authors, is achieved by an enhanced JavaBeans component API. The authors contend that out of the tool integration approaches of federation, enveloping, middleware architectures, database and file integration, message passing, none provides an ideal solution. Component-ware is similar to a combination of enveloping, middleware and message passing approaches.
Grundy et al. describe a component-based software development environment, and suggest that component-based architectures are highly reusable, externally controllable, and are easily extended and integrated into other tools. They also believe that component-based tools show a correspondence to canonical and message-based architectures, and that this allows more effective data and control integration. Grundy et al. continue by suggesting that a number of key design choices must be addressed if this architecture is to be selected, including the appropriate selection of components, design of common components, design of external facing components, tool provision, and integration and deployment support. They conclude by suggesting that whilst this approach is easier to build and integrate, the resulting systems are let down by user interface problems and a lack of inter-component event and operation mappings.

Guo et al. contend that CASE environments are either: Monolithic, one tool does more than one thing; Layered, common front or back ends are presented to the user; or have a Tool-bus architecture, where some form of “plug’n’play” interoperability is offered. The structure of such a tool-bus consists of: message passing; data exchange; bus management; and tool management. The authors agree that there is a degree of commonality between tool-buses and middleware, with tool-buses typically being implemented in three ways, being: the use of proprietary middleware based on TCP/IP protocol stacks; the use of Message Oriented Middleware (MOM); or the use of ORB-styled middleware, such as CORBA, COM or even RMI. They have decided to use an ORB-based approach for their Tool-bus.

Hansen presents a tool integration strategy based around activities, rather than concerns. "Activities" are taken to be those actions and situations that take place in system development, typically realised by the application of techniques, for example, unit testing. This contrasts with "Concerns" being objectives of system development, for example, analysis or design. Hansen believes that one activity may contribute to more than one concern, and therefore the provision of tool support for concerns is problematic, and that tool integration should instead concentrate on integrating tools to support
specific activities. He concludes that a type-based "publish and subscribe" mechanism should be used to realise tool integration, with the result that the publisher is decoupled from the subscriber in the dimensions of time, space, flow, and data.


Harrison et al present an object-oriented approach to tool integration. This approach concentrates on building systems from fine grained components, instead of dealing with coarse grained sets of tools, and provides a solution for both control and data integration. The authors suggest that a fine grained approach can lead to increased reusability and extensibility. Fine grained integration uses smaller units of information, from which larger grained objects are created, which must be managed in an object oriented database. Fine grained tools can be viewed as packages for methods and classes, which expose operations for invocation by other tools, thus the fine grained control integration mechanism. Fine grained integration requires more frequent operations to be executed on smaller entities.

Tool Integration is achieved by supporting the Portable Common Tool Environment (PCTE) and extending this to handle objects, thus bridging the gap between coarse and fine tools. A full data model is explained by the authors in the paper. Tools can then be built from these fine and coarse components, to produce flat tools using only internal components, or nested tools that use both internal and external components. Tools can then be composed in a number of styles, directly, by extension, or by merging.


In this position paper, Harrison et al. describe the then current "state-of-the-practice" within software engineering tools and environments. They review the history of tool usage in software engineering, and suggest some future research goals and directions. Harrison et al. start by suggesting that the use of tools to perform software engineering is pervasive, and development cannot be successful without them. They trace the history of tool usage from standalone tools that were able to be combined within the UNIX environment, through to Programming Support Environments (PSEs) for coding activities, and on to Software Engineering Environments (SEE) that support a particular lifecycle, including features such as traceability, interaction, and a repository. Harrison et al. note a trend towards the support for multiple views within software environments, and ultimately to the development of Process-centred Software Engineering Environments (PcSEEs), developed in the light of Osterweil's seminal paper [2] on the Software Process.
Harrison et al. identify key challenges for software engineering environments, for example the issue of integration of tools, processes, artefacts and views, and they continue by stating that the current state-of-the-practice within software engineering falls some way short of the possible state-of-the-art, chiefly because environments are still tied to a specific context. This contextual problem implies that tools implement concepts that are transferable to other contexts but the tools themselves are not transferable. For example, a tool might require the presence of supporting facilities that are not used directly by the user, but are required to ensure the correct tool works as specified. Harrison et al. suggest that this is the key challenge for integration, being the search for ways to integrate tools such that they can then be used in new contexts. They contend that adaptation and integration will be more appropriate to this challenge, and not standardisation. This requirement to adapt and evolve is not currently achievable in present day software systems, as anticipation and pre-planning are hard to justify and to implement, particularly in agile approaches, where large scale re-engineering is error prone and costly. Also software products must be delivered as fast as possible in order to capture new markets, often with a poorer quality product, rather than waiting for an all encompassing and complete product.

Harrison et al. have coined the term, "morphogenic software", which is software that is malleable during its life, like the soft metal gold, instead of being initially malleable and then brittle in the long term, like clay. They identify a requirement not only to create software swiftly, but to allow later adaptation into newer contexts, without any prior knowledge of what those contexts might be. Harrison et al. continue by identifying a key barrier to the development of malleable software; namely the inadequate separation of concerns [3]. This is the problem of developing software that is itself dependant on the presence of some other functionality. For example, two relational databases may enact a row numbering scheme in slightly different ways that may require significant re-engineering for any application software that uses the database. Current software products are locked-in to particular contexts, hence the phrase the "tyranny of the dominant decomposition" was coined [3]. For instance, functional programs can only be broken into functions, and object-oriented programs into classes. This is the underpinning principle of multi-viewpoints in a software engineering environment, which attempts to eliminate an over-riding compositional "tyrant". Harrison et al. continue by identifying some recent developments and technologies, and how these will address some of the identified issues. These issues include the deployment of commercial technologies, the management of concerns and an environment for "morphogenic" software. Harrison et al. see data integration using XML as the first step to allowing software to achieve some independence from its context. XML also allows the reshaping of content from one ontology to another, for example, from the details of the specific tool used to produce the data and the problem of then adapting the data to another context.

Harrison et al. believe that when a common language for sharing data between tools is created, a need for access to particular dialects is therefore required in order to transform data between formats. They suggest that XML
is appropriate for loosely coupled, low bandwidth integration, whereas tightly-coupled, high bandwidth integration requires the use of a repository. To this end data integration using Enterprise Java Bean (EJB) technologies, has been designed to allow the deployment of software that have differing internal models to be used in a common repository. In an EJB, data and its access methods are separated from the business logic, giving rise to a form of "Stem-software", where software is then deployed in order to realise its implementation in a specific context. The next technology that can provide control integration is Java Messaging Services, used to provide a software "back-bone". However, Harrison et al. suggest that this approach is not rich enough to achieve sufficient separation of concerns, and instead, propose that message brokering would be more applicable. Message brokering examines properties of the message to determine the recipients, dependant on some rules. Harrison et al. discuss the management of concerns, and suggest that research should be undertaken into the best methodology and processes to achieve this, to address issues of identification, encapsulation and integration. Finally Harrison et al. consider the application of non-traditional software engineering techniques. These include Pervasive Computing (software engineering for constrained devices) and e-Commerce (typified by Commercial Off-The-Shelf (COTS) software and reuse, to build decentralised systems from a multiplicity of providers).


In these two papers, Harvey and Marlin describe a method for comparing integration frameworks that use control integration techniques to achieve integration in order to assess the expressiveness of the integration on offer. Harvey and Marlin start from the basis that a semantic model could be used to assess the integration available. This model has five layers, from the information structures, the communications substrate, primitive operations, higher-order operations, and finally the descriptions of the integration devices themselves. They then use this model to build an algorithmic description of the events corresponding to each inter-tool communication feature, which describes the semantics of the inter-tool communications themselves. A mapping was defined for both the SoftBench and FIELD frameworks, in order to compare the two. One of the conclusions reached was that SoftBench is intended for purely user driven integration, with all operations of the integrated tools visible to the user. This contrasts with FIELD where semi-autonomous communication is also possible due to its tool-centric design, thereby allowing the integrated tools to operate independently if required.

Huck et al. describe an integration system for the sharing of performance data management technologies, by implementing the performance data management framework (PerfDMF). The objective of this framework are: to transfer data between parallel profiling tools; handle large amounts of data and experiments; provide a robust and portable data management system; provide a programming interface to this management system; and finally to provide customisation facilities. The authors continue by describing their implementation and mention XML as an exchange facility.


Iyengar describes efforts to adopt a standard universal object repository. They note a paradox between emerging technologies such as MOF and XMI that could be used to leverage an object repository, yet at the same time, and yet the infrastructure technologies such as COM and CORBA are diverging. Also of interest is the view that MOF is the key to MDA.


Jiang and Systä contend that XMI is now the de-facto standard for tool data exchange, however, the universal adoption is thwarted by vendor extensions and modifications to XMI creating tool-specific dialects, thereby sacrificing interoperability. Therefore the authors propose a DTD comparer, the success of which is documented in three case studies. This comparer, the authors contend, will increase awareness of potential and actual losses in both data and metadata.


Jin and Cordy attempt to provide a means to integrate program comprehension tools, as they believe that no single tool exists that provides all the functionality needed by the software maintenance community. They hypothesize that previous attempts at integration in this field have concentrated on a data-centric, and therefore a necessarily prescriptive,
approach, and that it would be better to share services. This non-prescriptive service sharing approach would take advantage of commonalities between tools, instead of exchanging data in a prescriptive fashion. Jin and Cordy suggest a 3-tier architecture consisting of a fact extractor, a repository together with an analyser and a visualizer module. They decided to use a domain ontology approach to provide a common vocabulary across tools, and conclude that work has started towards the creation of a tentative ontology.


Jin et al. attempt to classify every integration technology for exchange of information in order to back up the case for a Standard Exchange Format (SEF). The authors identify schema definition (implicit and explicit) and locality (internal and external) as defining characteristics of the data models used in the various tools that they have identified. They then go on to describe the various type of exchange pattern that can occur, as well as the benefits and disadvantages of each definition and locality. Thirteen patterns of exchange (Transparency, Scalability, Simplicity, Neutrality, Formality, Flexibility, Evolvability, Popularity, Completeness, Metamodel Identity, Solution Reuse, Legibility and Integrity) are identified, considered, and given a weighting for each definition and locality of exchange. The authors conclude that explicit and external schemas offer the best solution to most problems, and suggest that Graph Exchange language (GXL) should be adopted as the standard format for exchange in this domain.


Jin et al discuss tool integration in the reverse engineering domain. Specifically the authors concentrate on tool interoperation by means of the sharing of tool services, and identify that the prime barrier to interoperation is the differences in representation of knowledge that various tools use. This knowledge can be both syntactic and semantic, and the goal of achieving tool interoperation to mediate these two aspects of knowledge. The authors identify an "integration utopia" of full interoperability, however this would require n(n-1) converter programs to fully share data. Data integration in this domain has concentrated on the negotiation of information transfer between schema and instance (fact-base) levels. Approaches so far have failed in dealing with the differences between data structures, semantics and models that are used by different tools.

Jin et al. introduce a transaction adapter that operates at the conceptual level, based on cooperation. This adapter uses a domain ontology to compile all the conceptual transactions, that the adapter can then use to translate into
actual queries to the selected fact-base. Constructing this ontology is very
difficult, if the inner working of tools are not available, however, using tools
that support an agreed export format may alleviate this problem.

Dean Jin and James R. Cordy. Ontology-based software analysis and
reengineering tool integration: The OASIS service-sharing methodology.

Jin et al. describe an alternative means to integrate tools, base on the use of
ontologies and service sharing. They briefly report on their system called the
ontological adaptive service sharing integration system (OASIS), which is
promoted as being non-prescriptive, by combining shared services and a
domain ontology to achieve integration. They believe that previous
approaches to integration were prescriptive and too data centric to achieve
success. They continue by describing their architecture, and conclude by
suggesting that this approach achieves independence between integration
participants; changeability is achieved without side effects; and finally
localised participation reduces maintenance of the integration.

Dean Jin and James R. Cordy. Factbase filtering issues in an ontology
based reverse engineering tool integration system. Electr. Notes

Jin and Cordy continue their research on OASIS, and in this instance describe
issues surrounding factbase filtering within the environment. Factbase
filtering concerns the mapping of instances with the tool factbase to the
conceptual space of knowledge about tools. They highlight three issues
concerning this filtering process: representational correspondence; loss of
precision; and information dilution. Representation correspondence concerns
the problems that may arise when re-mapping transformations when more
than one possible solution may present itself. Loss of precision is the same
as representational except in the opposite direction. Finally information
dilution occurs when user expectation are not satisfied with the results
provided by the selected tool, leading to subtle representational differences.
These differences affect the factbase that in turn impact the conceptual
space, and ultimately lead to unexpected results. This highlights some issues
with an ontological approach being the meaning and representation of
knowledge, which can have different impacts for different users.

Database support for knowledge-based engineering environments. IEEE

Kaiser et al. describe in great detail, their system called Marvel, which is
intended to implement knowledge-based engineering environments, which
may include software engineering. Marvel is intended to use expert system
technology to automate many engineering processes, by using a knowledge
representation language, a translator and a database manager. The Marvel strategy language (MSL) can traces its origins back to Carnegie Mellon University which developed Smile and in turn Gandalf, programming environments. Through the use of worked example the structure and behaviour of Marvel is explored, with particular focus on the system architecture, database structure and behaviour, and moving on to discuss the wrapping of engineering tools through the use of envelopes, and finally strategies that give differing views of an overall process. The authors conclude by suggesting that the resulting environment behaves like an “intelligent assistant”, and reduces mundane tasks and effort from the environment user.


Kaplan et al introduce the concept of the “software backplane” or “software bus”, which is a common communications architecture across which software tools can be integrated. This concept is encapsulated in the coined term, “open systems architecture” (OSA). The authors conclude by suggesting that additionally, a large distributed object oriented database is mandatory in any software development environment.


Karsai takes a model-based viewpoint on tool integration. In this approach models of each set of data held in each tool are represented in terms of the semantics and behaviours of the tools and the transformations between them. The aim is to extract tool characteristics that can then be used to generate software components which solve the integration issue. Karsai summarises previous integration efforts by saying that any integration solution must cost less than manual translation, must be scalable, and requires deep semantic understanding of the tool itself. In effect, Karsai has created an approach that addresses the issue of semantic interoperability to solve the mapping problem between different data stores, by creating an integrated data model and mapping each set of data to this model. The architecture consists of a semantic translator providing semantic mappings, with a tool adapter providing syntactic mappings, with the model transported from tool to database via CORBA middleware. He has then used design patterns to generate automatic facilities to create new adapters and translators for new
tools. Karsai contends that this system has successfully addressed data integration issues, but not event-based integration.


Karsai et al. follow up previous work (2000), by describing a pattern-based approach to tool integration. They suggest that tool integration is difficult to achieve within the domain of embedded systems, due to the multiplicity of tools that are used, and remains a recurring issue. They attempt to solve this problem by the use of a meta-model based approach to allow for model transformations. Meta-models are created for the input and output models, in order to abstract the syntax and to ensure well-formed models. A model is then created for the semantic mapping between the input and output domains, which is incorporated into a semantic translator. The methodology is used in a reusable framework, either by integration based on integrated models, or integration based on process flows. Karsai et al. suggest that change propagation is much more appropriate in the process flow approach, as the integrated data model for full integration requires as many bi-directional translators as there are tools to be integrated. The process flow approach uses pair-wise tool integration, and requires fewer uni-directional translators as there are tools. The integrated data model is suitable for integrating no more than three tools, whereas the process flows approach can integrate successfully up to six tools, as change is always localised to those two tools that share a model, instead of all tools that share a common model.


Kelly presents a practical experience of tool integration, noting that standards for tool integration have not been successful (as we have noted previously), and that CASE tools in general have not provided the expected panacea for software development. Kelly also suggests that the adoption of UML as the de-facto design standard has tied the design process into a particular solution space that is too tightly related to the implementation technologies of the design. Instead Kelly suggests that development should be domain specific, using a unique language developed for the particular domain of interest. Domain-Specific Modelling (DSM) attempts to provide 100 percent code generation, making low-level design and coding redundant. Kelly describes a meta-CASE tool, called MetaEdit, which can support any modelling language. MetaEdit allows control integration by command line argument execution of other tools, and data integration via a SOAP API, as well as allowing the use of XML for data exchange. Kelly concludes by stating that the decision to add integration functionality to a tool should by determined by the amount that the
end users need to use the facility. This problem is compounded by the lack of initial requirements from users until they have seen what is actually possible to do with a tool. These suggestions add to the previous hypothesis regarding the net value accruing from increasing the integration level, by suggesting that there is a relationship between integration level and the usage level of any facility that may be constructed in an integration effort.


Kempkens et al. describe another modification to a Process-centred Software Engineering Environment (PSEE), called Spearmint. This is a tool that allows multiple views of the same process model, so the process can be represented differently depending on the person viewing the model (manager, customer, developer etc). The tool allows for multiple perspectives of a single software process, and multiple view of each of the perspectives, as well as the ability to use different notations within each view. This multiple view based model, is built on a extension of the classic Model-View-Controller architecture, familiar within Java AWT development, called Multi-Layer Multi-View (MLMV). MLMV is a 4 layer architecture, and was found to aid communication and understanding between developers, is loosely coupled, uses persistent storage, but does come with a performance overhead.


Khare et al. suggest how XML should be used in architecture-centric integrated development environments, in order to aid to tool integration. They suggest that environments have progressed from repository-based approaches, through process-based approaches to the current architecture-based approach. In an architecture based approach, the evolution of software is controlled throughout the lifecycle by using architecture descriptions as "its primary unit of discourse". In this approach, repositories and automated processes are assumed to already be present, so the focus moves to the design, evaluation, instantiation and editing of architectures. Khare et al. suggest that tool integration should now be achieved by using an open hypertext web to represent the entire product. To this end the authors have developed various flavours of XML, and suggest that their environment is a technological update to FIELD, and conclude by suggesting that the use of XML encourages reuse, allows the easy incorporation of multiple architecture descriptions, and allows hyper-linking.

Kienle presents, in the form of an annotated bibliography, a summary of data exchange formats, both text and binary formats. These include: GXL, RDF, XML, ATerms, GEL, CDIF, ASN.1, XDR and ASDL, plus various Interface Definition Languages (IDL), used by technologies such as CORBA, etc...

Kienle describes a number of these formats as being intermediate, with CASE Data Interchange Format (CDIF) for example, which is analogous to UML's MOF. Also Portable Common Tool Environment (PCTE) uses graphs to store data within its repository.


Kim and Park present a flexible data model called FORM, which is described in detail, together with its syntax. Within this paper all of the models used in process-centred software engineering environments (PSEE) are identified, and the authors suggest that FORM answers the problems associated with these other models, however, few examples of how this new model is used or implemented are given. A prototype implementation is referred to.


Königs and Schürr present an approach to tool integration which uses triple graph grammars. They suggest that keeping the relationships consistent between repositories of different tools is a nearly impossible task, and that an alternative strategy is required. They recommend a rule-based approach that allows the specification of data integration links, by using triple graph grammars that are MOF complaint, which are influenced by the model driven development (MDA) approach to software development. The authors suggest that message servers such as Softbench or Tooltalk require a common database, and cannot easily represent document dependencies. The rest of the paper describes the triple graph approach, including how rules are specified using the object constraint language (OCL).


Lundell and Lings review the history of the development and acceptance of CASE tools, and gives many examples from studies from the previous decades of good and bad points associated with them. The authors describe the CASE Data Interchange Format (CDIF) interchange efforts, and note that
technologies are not yet mature enough, and advocate the development and adoption of interchange standards to tackle this thorny issue. Also noted is the upsurge in interest in UML and its associated interchange standards (MOF/XMI), however, a warning is sounded as UML is a “moving target” so tools and interchange standards will certainly lag behind.


Maguire et al. describe the various options that are available to designers and builders of embedded systems that need to use an integrated software development environment, a specialised field with few suppliers. They survey the requirements for such a development environment, and conclude that no commercial tool is available that satisfies all possible requirements. Maguire et al. then continue by discussing three architectural possibilities for a development environment, being "Closed", "Open" and "Distributed", and discuss advantages and disadvantages of each. They decide that a distributed object approach is suitable for embedded systems design, and is best implemented on a Windows platform, using COM as the messaging medium for dynamic communication, with standards-based methods for static communication.


Margaria and Wubben describe an Internet based approach to tool integration, via the Electronic Tool Integration (ETI) platform. ETI allows users to find out about tools, execute them discretely, and to combine tool functionality to generate scripts that can then be run sequentially. ETI also allows tool users to sample the variety of available tools, and providers to promote their wares. Therefore, ETI provides a tool repository, which can be modified to take advantage of newer technologies. Margaria and Wubben suggest that current integration architectures are not sufficient as the burden of integration falls on the tool providers. However, tool providers believe that it is not their job to integrate tools, instead believing that infrastructure providers should build integration mechanisms. Therefore Margaria and Wubben intend to provide tool registration mechanisms, via an XML-based tool description language, with the architecture executable via SOAP.

Margaria et al. describe the latest enhancement to the ETI platform called jETI, simply being ETI rewritten in Java and combining the latest enhancements to the ETI platform as a result of its use over time. The authors suggest that the most import enhancement was to reduce the effort require to integrate tools. jETI provides a web services and Eclipse based approach to tool integration, by providing an Application Building Center (ABC) for the developer integrate tools. ABC provides lightweight remote tool integration, with distributed component libraries within a graphical environment in which tools are coordinated, combined with a distributed execution mechanisms. The paper describes jETI in some detail.


In Marlin et al., a mechanism is described that integrates a process-centred software development environment, Merlin, with an integrated software development environment, MultiView. It was suggested that tool integration research in 1993 is focused on two areas, these being the provision of a highly integrated set of tools for a particular life-cycle phase, or the loose integration of tools to provide support over a range of phases. Marlin et al. propose that MultiView is an example of the former, with Merlin an example of the latter. They contend that an attempt to integrate both of these provides a worthwhile experiment to use the strengths of each tool, whilst negating their individual weaknesses. They also describe an integration architecture that makes use of an adaptor to provide 2-way communication between the tools via customised control protocols. Marlin et al. conclude that the experience of creating an integrated environment that addresses weaknesses in each of the tools, could be used to define a general model for tool integration mechanisms.


Maryama and Yamato describe how Java Source code is translated into XML, and this representation is used to share artefacts across tools.

Mehra et al. describe an attempt to provide a tool for comparing diagrams, which is realised as a plug-in for the Pounamu meta-case tool, as described by Cao et al. (2004), Stoeckle et al. (2003) and Grundy et al. (2000). Mehra et al. describe their approach to building this tool, and discuss its architecture, design and implementation issues, and attempt to critically evaluate its effectiveness.


Melo et al. describe enhancements to a Process-centred Software Engineering Environment (PSEE), with particular attention paid to consistency of the artefacts held in the environment. The PSEE is made up of two components, a Resource Manager and a Process Manager, which combine to provide both the low level functions required, and the formalisms required to enact any defined software process. These core functions are based on triggers that enact the required functionality of the environment. The complete software process is then describable via a series of "event-condition-action" rules, together with an active database.


Mens et al. ran a workshop on graph-based tools. These tools can exchange data and hence be integrated through the exchange of graphs by using Graph eXchange Language (GXL). GraBaTs is now a regular annual event.


Mi and Scacchi discuss process integration with respect to CASE environments, in which a process model is used to guide development of software artefacts. The authors describe a "conceptual tool-invocation chain" and "conceptual resource-transformation chain" to describe a process model. Each of these chains describes a set of tasks to be executed. The authors then suggest that the purpose of process integration is to make the conceptual task-execution chain explicit, flexible and reusable, and then go on
to contend that a software process model can then achieve a more integrated CASE environment.

The authors continue by describing the process model as an activity hierarchy and resource requirements. The activities are then composed of sub-activities and finally atomic tasks and actions. A process driver is required to interpret and execute the software process model (SPM) according to the hierarchy. The SPM can then be presented to the developer for action via the developers' interface. The management interface allows the SPM to be defined, monitored and controlled, allowing managed concurrent development. The authors put this theory into action within the development of the Softman Project.


Mi and Scacchi present a knowledge-based meta-model for the various objects that appear in Software Development Models (SDMs), including tools, known collectively as a Unified Resource Model (URM). Mi and Scacchi suggest an ontology for a URM that can then be used to integrate the components into a defined resource.


Michaels describes the state of the art with respect to tool integration and in particular, the work of the ANSI CASE Tool Integration Models Committee. This committee recommended that efforts should be focussed towards producing solutions to implement control integration. Michaels suggests that integrating tools using "Plug and Play" has not been fully realised, and continues by reviewing the various industry attempts to define common standards to realise this goal. Michaels describes a proposed architecture, based on message-passing that uses requests and notifications, conforming to a standardised "servicegram" which makes use of a defined structure and meaning. Michaels concludes by assessing the level of conformance to this proposed model by existing tools, and suggests that integration in future would be better achieved if all tools were designed and built using the same abstract architecture.


Morel and Faget describe a project that has developed process models that incorporate reuse in the development process. Tools are mentioned in passing, but the greater emphasis is on the process model itself. The process
model is implemented by the use of a set of tools within an environment. However integration is achieved by the use of a central database repository, which is built on top of the Adele DBMS.


Mota et al. discuss the specific integration of Formal Verification Tools (FVTs) and CASE Tools. The authors have created a “Protocol Interface” that enables translation to occur between FVTs and CASE tools, thereby contending that a hitherto missing focus on reliability issues can be brought into the CASE tool domain. This results the possibility that safety critical systems can now be designed using CASE tools. It is suggested that tools need to display the following characteristics: Composition; Abstraction; Reuse of Models and Theories.

XMI is the key integration enabler here, as FVTs typically manipulate First Order Logic (FOL) which can be represented as an XMI format; via First Order Specification Models (FOSMs); First Order Verification Models (FOVMs); and finally First Order Verification Results (FOVRs). The key reason that XMI is appropriate is the fact that all XML documents can be described by an associated Document Object Model (DOM). The associated DOM can provide an easy mapping to First Order expressions. As a result all tools that support XMI can be integrated with FVTs.

As a result this paper provides an interesting proposition that Formal Methods, techniques, procedures and philosophy, can be brought into the mainstream via this integration of FVTs and CASE tools.


Mueller et al. focus on CAD tool integration issues, and use Simple Network Mail Protocol (SNMP) as the basis for their integration. They counsel against the use of XML/SOAP methods due to unsolved security issues.


Muller-Glaser et al. describe a CASE tool integration platform, called GeneralStore, that uses XMI as the data exchange mechanism. However,
XMI used as a Meta-model for system description lacks a standardised graphical representation for class and object diagrams (as requested for UML 2.0).


Nitto et al. describe a technique for converting UML diagrams, that are not designed to be executable by definition, into process modelling languages (PMLs) for use by process centred software engineering environments (PSEEs). In this instance UML diagrams are generic and non-executable, whereas PMLs are specific and executable within a suitable supporting domain.

The authors also note that PSEEs have been largely ignored by industry. Workflow Management Systems (WFMSs) are characterised by simple and repetitive tasks that are more suitable for reification by process modelling languages. The authors are exploring the possibility that UML could be used as a PML. An PSEE called Orchestra Process Support System (OPSS), is a workflow-based system that uses a middleware stack called JEDI. CASE tools are used to produce UML process models that are then processed from XMI files via XML/XSLT into Java Classes, for enactment in the target PSEE (OPSS).


Norrie and Wunderli describe an agent-based architecture for coordinating tools. The authors favour control integration over data integration, by the use of a central repository to coordinate work, and local repositories to handle and manage the tools themselves. The architecture is based around two types of agent: tool, concerned with tools themselves; and coordination, responsible for inter-system dependencies. Tool agents then talk to coordination agents, with coordination agents talking to each other. The authors describe this approach as being data-driven.


Nuseibeh and Finkelstein describe an object based framework for the development of heterogeneous systems. The idea behind this is to recognise the fact that developers have different notations and methods for achieving similar or identical goals. ViewPoints is a framework that recognises this fact,
and instead allows developers to work in their own way, and yet coordinate their activities for a common goal or end.

A ViewPoint is a loosely coupled object that encapsulates representation, development and specification knowledge for a given problem area. Each ViewPoint has five slots: style; work plan; domain; specification; and work record. Each ViewPoint has an owner, and can reuse an existing template. The idea then progresses to define a software engineering method as a collection of templates, that are bound together to form a set of Inter-ViewPoint Rules, that together constitute the integration mechanism. Hence the term ViewPoint-Oriented Systems Engineering (VOSE).

VOSE is enacted using a Support Environment called Viewer, which is described in the paper. The framework the provides three forms of Integration: method; ViewPoint ;and tool. Method Integration is the set of templates and their inter-dependencies. ViewPoint Integration is the configuration of the ViewPoints, via their relationships. Finally, tool Integration is provided by constructing tools that conform to the templates provided in the framework. At present this framework is completely dependant on Smalltalk, as it is used as the basis for the framework itself.


Nutter et al. present another paper on the Genesis project, concentrating on the Open Source Component Artefact Repository (OSCAR) that interoperates with the workflow management system. OSCAR provides extra capabilities to standard workflow systems, by providing a unified namespace for all artefacts and treating all data controlled by OSCAR as an artefact. All functions within OSCAR are provided by existing third party open source applications, with meta-data stored together with the artefact itself using XML.


Osterweil preceded his seminal paper [2] on the software process, with this rehearsal of many of the ideas that were to be published a year or so later. Once more his hypothesis is that software development is just another software process, and that this can be treated as yet another software product, with the understanding of this remaining at a simplistic level. He suggests that any paradigms and frameworks to address this issue should
remain generalised. He concludes by suggesting that such a view leads to an architectural view of software development that is flexible and extensible.


In Rader et al. an in-depth examination of the state-of-the-practice with use of CASE tools is described. They suggest that the state of research into CASE tool integration in 1993, is far in advance of the actual take up and use of CASE tools in the workplace, and contend that any future research that does not take this fact into account will not be soundly based. Rader et al. start by precisely defining terms, as they have discovered that a common terminology is not used and understood. For instance they differentiate between integration, the coalition of more than one component, and tool integration, the coalition of complex CASE tools via the use of third-party frameworks. Rader et al. have identified that CASE tools are typically used either as a documentation aid, or as a code generator, or as a mechanism to enact a wider software cycle. They characterise existing tools as not totally reliable, requiring tailoring for each customer, and not offering sufficient support for "programming-in-the-large". Also noted was a significant gap between the claims made for the alleged ease of integration, against the practical realisation of intended integration, with significant customisations usually required to effect this requirement. It was also found that no standards in 1993 were being used for integration instead data exchange is usually achieved using proprietary vendor supplied mechanisms. In fact it was a further decade before the IEEE proposed a standard (2003). In addition: no instances of "plug-and-play" were observed; users of CASE tools were forced to assess the organisational impact of these tools, to the extent that new structures and groups were being created to support them and their processes; and tool vendors were found to be difficult to work with, with upgrades a particular problem.

In conclusion, Rader et al. suggest that most organisations can be placed within a five level spectrum (see Figure 5) that describes the level of CASE tool adoption, ranging from: isolated CASE tools; clusters of CASE tools; framework-based integration; loose integration of CASE clusters; and finally, a complete integrated CASE environment. Rader et al. place most organisations at either the first or the second levels, with some taking the first steps towards achieving the third and fourth levels, whilst the fifth level is viewed as a distant future target. Rader et al. suggest that whilst most current research efforts are focused towards achieving this goal, this effort may be misguided. Instead, they suggest that organisations must develop a CASE "pedigree", typically with years of incremental experience in the use of such tools and associated frameworks. Rader et al. finish by suggesting that better metrics for this domain need to be defined and then collected to enable credible tool comparisons to be made, thereby allowing the best fit between a tool and a user's requirements to be achieved.
It is worth noting the similarities between this five level classification scheme for CASE tool adoption, with Brown's (1993) five level classification scheme (see Figure 3) for the degree of tool integration, as discussed previously. Rader et al. discuss operational use of CASE tools, whereas Brown discusses how the usage of CASE tools is achieved. Another opportunity for research in the field becomes apparent with these classification schemes, in order to determine what the relationship is between these schemes. Also of note is Yang and Han's (1996) continua (see Figure 6) which provides only four levels for both data and control integration, which contrast with the five level schemes as proposed by Rader et al. (1993) and Brown (1993).


Reichmann et al. include greater detail on the meta-model employed within GeneralStore, noting that XMI was influenced by CASE Data Interchange Format (CDIF).
Reiss describes the Desert software development environment, an enhancement of an older system called FIELD (previously discussed by Brown, 1993) that attempts to explore broadcast messaging. Reiss believes that messaging typifies control integration and that messaging gives the impression to the user of a seamless environment. This contrasts with the actuality of such a system, as it has no notion of history or context, and therefore lacks extensibility. The Desert environment also attempts to address problems of data integration via the provision of hypertext links to connect artefacts across the software engineering process. This provides a single view of the whole system via a single dynamic document, and as a result, presents to the user an open environment. Reiss suggests that the implementation of control integration is not enough to provide complete integration on its own, due to the lack of any temporal dimension, and instead enforces the requirement for all tools to operate within a common framework. Reiss suggests that the implementation of data integration is achieved by the use of a common data-store, however, this is difficult to manage due to the amount of data needed to be stored, and the lack of a common representational format. Reiss concludes that the trend within software development environments is for the use of control integration, combined with a specialised data repository.

Reiss contends that software development is best achieved by letting the developer manipulate conceptualisations of the system they are trying to create, typically by graphical representations. Reiss goes on to describe a system that supports graphical programming, called GARDEN, together with all its associated tools. Of interest is the fact that GARDEN superseded the Pecan system which used views tied to syntax which proved ineffective.

Robbins and Redmiles mention XMI as a means for sharing UML data. Some CASE software is often described as being “shelfware”, due to lack of actual regular use, with later case tools failing to address the cognitive challenges posed within the process of software development. The authors’ motivation is the notion that design tools support decision making, and hence improve...
productivity and quality. Argo/UML is described in great detail together with its particular design features that address all the cognitive challenges previously identified in the software development process. Some discussion is included on a cognitive study carried out on a discrete aspect of the Argo/UML tool.


Rover describes a roundtable meeting of parties that are interesting in integration of tools primarily from the performance evaluation of parallel and distributed systems. The views of the parties are stated, and that forms the whole of the paper itself.


Sen, in the first part of this paper, provides a very thorough and in-depth description of the history of metadata, stating three research goals in the use of metadata in the 1990s: code reusability; asset repository; and data warehousing. Sen concludes by describing how a metadata warehouse is needed in order to properly create a meaningful and useful data warehouse, and that no such product exists that fully addresses the core problems of metadata.

However, Sen discusses metadata in relation to software development, suggesting that the objective of metadata is to support the selection of tools used in a software development environment (SDE). There are three groups of SDE in existence, Programming Environments (PEs), CASE, and Software Engineering Environments (SEE). A PE is described, called REBOOT (Morel and Faget, 1993), that uses facet-based classification to create the metadata of the component, the facets being: abstraction; operations; operates-on; and dependencies. These four facets allow interdependent artefacts to be changed if corresponding entries are changed.


Silaghi describes an Eclipse-based tool that reifies the “Enterprise Fondue” software development method. This comprises four facets into a single method: Component-based software engineering; Separation of Concerns; Model driven architecture; and Aspect-oriented programming. The resulting tool is called Parallax, and uses XMI as the means by which data is imported from other tools.
Sim describes the need for a standard exchange format with respect to the reverse engineering domain, with the requirement for an API built into tools from this area. Sim suggests that encapsulating a tool like a library, or using middleware such as CORBA, as a communications medium, or even a Hybrid approach using a framework, would enable dynamic tool interoperability to be more easily achieved.

Sim and Koschke ran a workshop on a Standard Exchange Format (SEF), which concluded by agreeing to refine graph eXchange Language (GXL) with the aim of establishing it as the standard. Again this is acknowledged to be work in progress.

Singh describes a working party that was set up to bring together tool providers, framework providers and end users. The paper briefly describes the integration problem in terms of three approaches: piece-meal; single-vendor; or best-in-class. In the case of best-in-class, two variations are possible: point-to-point interoperability via tool or vendor-specific approaches; and frameworks that classify tools and attempt to provide vendor-neutral solutions. The author summarises by stating that the piece-meal approach lacks consistency, single-vendor risks lock-in, and the point-to-point variation sacrifices tool choice for the sake of integration. Singh concludes by suggesting that the framework-based approach is the ideal solution that can only be implemented with standard tool interfaces, using standard mechanisms to achieve seamless integration. The intention is for the working party to provide best practice advice from this area, to then give a stronger focus for further research and development.

Smith et al. describe an exchange facility for software performance tools, based on two XML formats, software performance model interchange format (S-PMIF) and the performance model interchange format (PMIF). These
formats are used to translate a UML architecture description into system and software performance models that are required to study the performance characteristics of the system architecture. The authors note the relationship between PMIF and CDIF interchange formats, and suggest that PMIF which is an XML-based format is more appropriate and modern format to use, due to the predominance of XML as the current tool interchange format of choice.


Smith et al. concentrate on the traceability layer with Ophelia that enables users to manage the change process using notification mechanisms for artefacts. The traceability layer uses CORBA objects to represent artefacts in the software engineering process. Ophelia also manages the relationships between the objects by using graph-based mappings on the metadata of the objects.


Snodgrass and Shannon suggest how the interface description Language (IDL) can be used and extended to leverage tool integration solutions. They start by suggesting that tool integration is either monolithic, where one program does everything, such as in Gandalf and Pecan, contrasts with the toolbox approach, typified by the vast array of Unix tools that can be aggregated fairly informally. The contend that there are advantages and disadvantages of each approach, but that the way forward is to utilise the best of both worlds, by separately developing tools and then organising them in a coherent manner. The rest of the paper describes their approach using an extension to IDL


St. Denis et al. provide strategies to address model interchange issues, by defining thirteen requirements for any candidate model interchange format being: transparency; scalability; simplicity; neutrality; formality; flexibility;
evolvability; popularity; completeness; metamodel identity; solution reuse; legibility; and finally integrity.

The authors contend that there are five possible optimal formats in existence: RSF; TA; RDF; XIF and XMI. They conclude that XMI is the most promising as this describes the alignment of UML with MOF, making the resulting language very flexible and evolvable, with the widest support from industry.


Stoeckle et al. concentrate on the exchange of data as a requirement of tool integration, by addressing the conversion to and from formats. The authors describe experiments using a tool called Pounamu, to explore the problems of data conversion from the format used by the tool itself, to generic formats such as Graph eXchange Language (GXL), to other tool specific formats such as SoftArch, and also to other rendering formats such as SVG and VRML. The rationale behind this tool is to exchange software visualisations between environments, and to enable users to interact with different viewing and editing technologies.


Stoeckle et al. suggest that support for information exchange of visual notations is lacking. Therefore they contend that a unified converter generator framework should be developed for notation exchange. Firstly the authors survey the various integration approaches that exist at present, being: data-oriented; control-oriented; user-interface-oriented; and process-oriented approaches.

Data-oriented approaches are typified by attempts at data exchange via common exchange formats. However these lack dynamic integration support, and are by definition limited to the exchange of data. User Interface-oriented approaches require all tools to present a common user interface metaphor, however, these typically lack back-end integration support. Control-oriented approaches typically use API and object-based approaches, such as CORBA and COM to achieve integration. These approaches are powerful, however, they lack adequate user interface and process integration support across the integrated toolset. Finally Process-oriented approaches require either or both data and control strategies to be present first.

Stoeckle et al.’s approach to this problem is a two part solution: firstly, a Converter Generator tool is created and then; combined this with tools that
draw their own independent visualisations of the data on which they operate. They contend that similarities exist between such converters, and that therefore a general approach should be developed for modelling the conversion between notations. To this end they have designed a Unified Notation Mapping language (UNM), and an Automatic Language MApper TOol (ALAMATO) to translate between UNM inter-notation mapping specifications and notation to notation converter implementations. They finish with an interesting tabular summarisation of the main commercial and academic notation integration approaches, including their own (UNM + ALAMATO).


Suzuki and Yamamoto describe a distributed platform share UML models over the internet. The authors have created a UML Exchange Format (UXF) which enables developers to improve the interoperability of data between development tools, and communication between developers. The system, SoftDock, uses CORBA to provide the middleware glue to encapsulate the distribution of the models. Web servers also provide access to the models as well. The system provides 4 facilities: model metadata management; local caching of remote UXF descriptions; both Synchronous and Asynchronous editing; and finally change notification to ensure consistency.

This paper refers to the following paper that more fully describes the UXF model.


Suzuki and Yamamoto describe more fully describes the UXF format, as hinted at in the previous paper, a format for describing UML models using XML, as well as the software architecture developed to support this infrastructure, which implements a DOM interface on top of CORBA. The references provide a very thorough resource for UML, Web, XML and SAX/DOM descriptions.


Thomas and Nejmeh build on Wasserman's (1989) work by emphasising and extending the relationships between the tools to be integrated, and the degree to which those relationships are recognised and expressed in the tools themselves. Thomas and Nejmeh discard platform integration, and instead
expand Wasserman's original base dimensions of presentation, data, control and process integration, by identifying characteristics for each. For presentation integration, they identify properties of appearance and behaviour, and the interaction paradigm used, thereby allowing the user to switch easily between tools that have a similar "look and feel", as well as using similar presentation metaphors. For data integration, properties of interoperability, non-redundancy, data consistency, data exchange and synchronisation are identified, thereby allowing a user to easily share data between tools with: recognised levels of redundant data; the data from one tool not breaking any rules required by another tool; and the provision of means to convert data between tools. For control integration, properties of appropriate "provision and use" are identified for tools, where the ability of a tool to provide common services is gauged, as well as the ability of a tool to use common services. Finally, for process integration, Thomas and Nejmeh identify properties of a process step, a process event, and a process constraint, thereby allowing an environment to support a unit of work, trigger events from a particular unit of work, and to provide constraints on a unit of work.


Tichelaar et al. describe FAMIX, which is a language independent, extensible model for modelling object-oriented systems, that uses CASE Data Interchange Format (CDIF) as its exchange mechanism. CDIF is now defunct in the authors' view, so they have now built an XMI importer/exporter for FAMIX based on IBM's Alphaworks.


Urwiler et al. expand on the reasons why CASE tools are so poorly adopted, despite the benefits of productivity and quality improvements. The paper details a survey of software practitioners and organisations. Some interesting background statistics and references are included: for example, only 25% of purchased CASE tools continued to be used in organisations; and 75% of organisations were still at CMM level 1, with an ad-hoc and crisis driven development strategy. Also noted was the fact that CASE tool adoption was decreasing due to their perceived failure within the adopting organisation.

The findings of this paper suggest that the enforcement of a development methodology, together with the adoption of metrics contribute to the perceived improvement in software quality when using CASE. The use of metrics, consultants and training add to the developer productivity when using CASE.

Valetto and Kaiser describe a wrapping strategy for the incorporation of CASE tools into a process-centred CASE environment, called Oz. They describe three strategies for wrapping: "White Box", where a custom tool is implemented; "Grey Box", where a tool provides an API as a controlling interface; and "Black Box", where control of the tool is achieved via an "envelope" such as a protocol for control. Valetto and Kaiser address "Black Box" wrapping and describe the Shell Envelope Language (SEL), a scripting language to control tools, and conclude that their model is best suited to asynchronous groupware.


Wang et al. describe their attempts to create a software development environment using multiple views. The authors decided from the outset to use a fine-grained, tightly integrated environment, based on the Portable Common Tool Environment (PCTE) architecture. They discuss consistency, concurrency, integrity, extensibility and efficiency issues with regard to integration, and state why a fine grained database is more desirable than a course grained solution. The authors present and explain their schema in great detail, discuss their implementation and the direction of their future work.


Wasserman identifies five types of integration being platform, presentation, data, control and process, and proposes an open architecture for tools.

Each of these dimensions is described as follows:

- **Platform** integration concerns common framework services used by tools;
- **Presentation** integration is concerned with user interaction;
- **Data** integration is concerned with the interchange of data between tools;
- **Control** integration is concerned with the interoperability of tools;
- **Process** integration is concerned with the role of tools within a whole software process.
In his seminal work, Wasserman suggests that there is also a need for process management tools, however, he states that there is no agreement on the best way to represent process information. Wasserman believes that the key issue within tool integration is the ability to determine the appropriate level of integration, and to then identify a set of conforming tools for the chosen level. This issue is, in his opinion, hampered by a lack of agreement between the tool manufacturers on suitable mechanisms, levels of integration and standards. Wasserman sums up the situation at the time of his writing, by stating that data integration is typically achieved using unidirectional files between two tools, presentation Integration is achieved using a "windowing system", and that control Integration is completely absent from all integration situations. Also integration is usually achieved by using custom pair-wise facilities. Wasserman concludes by suggesting that integration must be achieved through the adoption of standards by all tool manufacturers, and that all tools should be built using a layered approach, to facilitate integration.


Wilcox et al. describe in more detail, the Ophelia project that aims to provide support for integrating heterogeneous tools by defining a set of standard interfaces to access and operate on the artefacts that tools produce. The Ophelia architecture consists of four layers, and provides a trace facility, called "Traceplough". Wilcox et al. do recognise that traceability has not been studied in practice, however they describe the management of event driven metrics, where an artefact changing causes the creation of new project metrics.


Wuyts and Ducasse describe a classification model that allows tools that were not designed to interoperate to be integrated. This is achieved by the Tool Integrator, not provided by the Tool Builder. The authors contend that tool integration occurs when the output from one tool is used as the input to another tool. The authors also state that if the tool integration architecture is supplied or exists in the tool currently, then this is anticipated tool integration. If no integration mechanisms are supplied or they do not comply to a specified framework, then this can be considered as an unanticipated integration situation. This idea is based on Design Patterns work, and is reified in terms of a 4 component model consisting of: Items; Classifications; Services; and Service Configurations. The Model is customised for a new integration situation, by supporting new Items and Services, typified by the use of a
Visitor Design Pattern. Tool outputs are translated into the customised standard model. Examples are given using the Smalltalk StarBrowser IDE, and Advance and SmallBrother tools.

The authors suggest that Eclipse supports Unanticipated Integration, as Eclipse Extension Points are analogous to instances of the Observer Design Pattern. Plug-ins to Eclipse use a synchronous Message Passing Model, and that extending this model is easier than writing a Plug-in.


In Y Yang et al., the use of multiple tool integration paradigms is considered instead of the adoption of a single paradigm for all tools. Here they have identified that tool integration may be achieved with tightly coupled, loosely coupled or uncoupled mechanisms, and that no one method is suitable for all development scenarios. They relate their experiences with the development of a single generic interface to allow the tool integrator to select the most appropriate mechanism for a particular situation. Each coupling paradigm is evaluated in a practical integration situation, with the conclusion that the loosely coupled paradigm should be favoured in general. However, they do recognise that other paradigms may be more applicable in other situations hence they suggest that the provision of a generic interface would facilitate this requirement. This work is continued by Yang (1994) on his own, by explaining in more detail each of these coupling paradigms. He describes four techniques for data integration: intermediate files; a repository; message passing; and a canonical representation. He also describes four techniques for control integration: indirect; triggers; a message server; and procedure calls. Yang finishes by suggesting which of the three coupling paradigms is best suited for each technique of control and data integration. For the uncoupled paradigm, he suggests that all communication is carried out via operating system facilities, with data stored separately by each tool. Thus, control integration is indirect, and data integration is via intermediate files. He goes on to suggest that control integration can be enhanced by the use of a message server. For the tightly coupled paradigm, all tools are completely integrated into a single executable system, with all tools manipulating the same data. Control integration uses procedure calls, and data integration implemented using a canonical representation for the data format. For the loosely coupled paradigm, Yang suggests that this paradigm is a compromise, and suggests that tools are likely to maintain their own separate data, but use fine-grained control and data mechanisms, typically by using combinations of a message passing system and a central repository. Experiments were carried out to determine the applicability of each of the models, based on four
criteria of feedback behaviour, response time, integration cost, and component independence. In summary, Yang found that the loosely coupled paradigm was the most effective means to integrate tools in a software engineering environment, which maps somewhat onto Figure 5.


Yang discusses traditional integration issues within a web-based development environment, and illustrates how various flavours of control and data integration are enacted within a web-based development environment. In such an environment, tools are invoked in directly using applets, with the supporting system built around an “N-tier” architecture. In summary, control integration is achieved by direct tool invocation, or indirectly via an applet embedded in a web page. Data integration is achieved by using common web standards, although this may not simplify the process. In conclusion, Yang suggests that a distributed heterogeneous environment is a natural environment for tool integration, that tools can be easily distributed via the web, removing upgrade issues, and that direct and immediate access is possible to the tools, with the data stored in a flexible and distributed manner. However, quality of service issues remain, as well as the restricted interface offered by HTML forms.


Yang discusses how to interface tools at the organisational level as well as at the code level, by contending that the development of programming tools and environments at each of these levels, has become disconnected, and must be rejoined to improve productivity. Yang contends that XML is the ideal technology for effective data integration, and that component based software is appropriate for effective tool integration. Data integration must be supported by a suitable RDF Ontology. The author also contends that the traditional client server architecture is not necessarily the best, and that standardised interchange formats, full integrated workflow and a process service, together with the internet, point the way to a light-weight dynamic architecture to reduce costs and increase flexibility. A centralised process engine is not necessarily required, and the specification of dynamic and yet incomplete processes must be supported.
Yang and Han start from the premise that the integration of tools is done to achieve productivity improvements, and that the traditional dimensions of integration are not sufficient to measure this goal. Yang and Han classify integration based on productivity goals, such as the elimination of user-generated delay, the elimination of tool-generated delay and the introduction of tool construction. Yang and Han continue by specifying a range (see Figure 6) for both control integration (indirect, triggers, messaging and procedures), and a range (see Figure 6) for data integration (intermediate files, database, message passing and canonical representations). Note the similarity between Yang and Han's continua, and Brown's (1993) classification steps (see Figure 3). Yang and Han experimented with these classifications and ranges, in order to determine which combinations of control and data integration are relevant to each class of productivity, with the result that the productivity classification is dependant on the level of coupling between the tools being integrated. For example, the tightly coupled paradigm corresponds to the use of procedure calls with a "canonical", shared data representation, for control and data integration respectively.

![Yang and Han's Control and Data Integration Continua](image-url)

**Figure 6 - Yang and Han's Control and Data Integration Continua**

Yap et al. describe an approach to building software engineering environments through the use of web services. Web services allow components to support the dynamic discovery, integration and invocation of remote software tool facilities, in this instance for an open source integrate development environment (IDE) called JEdit. This approach allows the building of flexible and more readily customisable environments to address the identified issues with the use of existing IDEs, being: diversity, where developers have a wide range of experience using many and various tools; different usage practices, where some tools are specified as mandatory by an organisation, as against ad-hoc infrequent use of tools for specific purposes; process issues, where an organisation’s software development process requires the use of a particular tool; and finally licensing issues, often when new tools are used on a trial basis, and do not fit with existing arrangements.

Yap et al. continue by describing the web service approach in terms of the usual web service description language (WSDL) reflection mechanisms, together with registry system for locating services, (universal discovery, description and integration, UDDI). These enable the web service enabled IDE, JEdit, to bring in services as an when required. The authors describe an example, together with a mini evaluation of the approach which they believe underlines its viability.


Zelkowitz and Cuthill discuss the Information Technology Engineering and Measurement (ITEM) model, which describes the information processing activities of any organisation. The model attempts to map out the actions of any enterprise, together with its software development process. Like the CMM, ITEM grades the organisation in terms of the degree of process automation, and recognises four major stimuli on this organisation: market forces; technological change; models; and measurement. With regard to the software process ITEM addresses the degree to which automation is supported, and the processes that have been automated, and recognises that the paramount problem for increased automation is the stumbling block of the poor degree of tool integration. The paper recognises this problem, and identifies that distribution and heterogeneity, as well as the lack of development of a software environment supporting enterprise level process, process management and decision making, remain a problem. Some discussion is included about the balance between abstraction, automation and process complexity, by concluding that a metric to describe program complexity.
Zelkowitz discusses the creation and construction of integrated software engineering environments, and attempts to define a set of formal criteria that can be used to measure and compare different candidate tools for inclusion in any proposed environment. Zelkowitz identifies the need for not only a protocol governing the movement of data through interfaces, but a set of semantics as well. Zelkowitz contrasts integration with openness, by describing openness as a property of the interfaces between two tools, whereas integration is a property between two tools themselves. Zelkowitz defines integration to mean the ability for two or more tools to interoperate, realised as the ability to pass information and control amongst themselves, and that integration also implies that the behaviour of the tools must occur in a consistent manner.

Zhao et al. discuss their proposal for an agent-based approach for process-centred software engineering environments (PSEEs). They are convinced that an agent-based approach is an appropriate method for building PSEEs, due to the fact that agent-based computing provides the means for autonomous software entities that interact to produce the desired goals. They suggest that this is useful for the software development process itself as: this autonomous nature reflects the nature of software development; the flexibility that agents possess make this a suitable match for the dynamism of software development; and agents reflect the way that humans go about achieving goals. The authors continue by describing the components of their agent-based software, which consists of process modelling, process enactment and process application layers, with the rest of the paper covering each of these aspects in more detail.
Appendix A - Further Reading


### Appendix B - Index

<table>
<thead>
<tr>
<th>Year</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986</td>
<td>8, 22, 32, 42, 45, 48</td>
</tr>
<tr>
<td>1988</td>
<td>31</td>
</tr>
<tr>
<td>1989</td>
<td>18, 52</td>
</tr>
<tr>
<td>1991</td>
<td>2, 23</td>
</tr>
<tr>
<td>1992</td>
<td>13, 15, 20, 26, 38, 41, 50</td>
</tr>
<tr>
<td>1993</td>
<td>10, 11, 21, 37, 39, 43, 47, 52, 54</td>
</tr>
<tr>
<td>1994</td>
<td>4, 6, 46, 54</td>
</tr>
<tr>
<td>1995</td>
<td>5, 8, 9, 10, 22, 38, 51, 52</td>
</tr>
<tr>
<td>1996</td>
<td>7, 21, 28, 39, 45, 56, 58</td>
</tr>
<tr>
<td>1997</td>
<td>28, 35, 41, 57</td>
</tr>
<tr>
<td>1998</td>
<td>14, 23, 24, 29</td>
</tr>
<tr>
<td>1999</td>
<td>13, 36, 50, 59</td>
</tr>
<tr>
<td>2000</td>
<td>16, 25, 26, 32, 34, 45, 47, 48, 51, 55</td>
</tr>
<tr>
<td>2001</td>
<td>4, 34, 35, 47</td>
</tr>
<tr>
<td>2002</td>
<td>2, 6, 20, 30, 38, 41, 55</td>
</tr>
<tr>
<td>2003</td>
<td>2, 3, 5, 6, 7, 9, 12, 15, 17, 19, 21, 23, 25, 29, 30, 33, 36, 40, 42, 48, 49, 53, 59</td>
</tr>
<tr>
<td>2004</td>
<td>6, 12, 25, 35, 37, 40, 44, 46, 53</td>
</tr>
<tr>
<td>2005</td>
<td>12, 14, 16, 19, 20, 29, 31, 37, 38, 47, 49, 57, 58</td>
</tr>
<tr>
<td>2006</td>
<td>1, 15, 22, 35</td>
</tr>
<tr>
<td>Abeti</td>
<td>2</td>
</tr>
<tr>
<td>Ahmed-Nacer</td>
<td>8, 20</td>
</tr>
<tr>
<td>Allison</td>
<td>54</td>
</tr>
<tr>
<td>Altheide</td>
<td>3</td>
</tr>
<tr>
<td>Amsden</td>
<td>4</td>
</tr>
<tr>
<td>Apperley</td>
<td>24</td>
</tr>
<tr>
<td>Arnold</td>
<td>4</td>
</tr>
<tr>
<td>Aversano</td>
<td>5</td>
</tr>
<tr>
<td>Baetsch</td>
<td>5</td>
</tr>
<tr>
<td>Baik</td>
<td>6</td>
</tr>
<tr>
<td>Baldamus</td>
<td>6</td>
</tr>
<tr>
<td>Ballarini</td>
<td>6</td>
</tr>
<tr>
<td>Bandinelli</td>
<td>6</td>
</tr>
<tr>
<td>Bao</td>
<td>7</td>
</tr>
<tr>
<td>Barghouti</td>
<td>31</td>
</tr>
<tr>
<td>Barthelmess</td>
<td>7</td>
</tr>
<tr>
<td>Belkhatir</td>
<td>8, 20, 38</td>
</tr>
<tr>
<td>Bell</td>
<td>29</td>
</tr>
<tr>
<td>Bengtson</td>
<td>6</td>
</tr>
<tr>
<td>Bernas</td>
<td>8</td>
</tr>
<tr>
<td>Berrigan</td>
<td>57</td>
</tr>
<tr>
<td>Beyerlehn</td>
<td>19</td>
</tr>
<tr>
<td>Bischofberger</td>
<td>9</td>
</tr>
<tr>
<td>Boehm</td>
<td>6</td>
</tr>
<tr>
<td>Boldrefff</td>
<td>9, 42</td>
</tr>
<tr>
<td>Bosua</td>
<td>10</td>
</tr>
<tr>
<td>Brinkemper</td>
<td>10</td>
</tr>
<tr>
<td>Brown</td>
<td>10, 43</td>
</tr>
<tr>
<td>Budgen</td>
<td>11, 12</td>
</tr>
<tr>
<td>Bulusu</td>
<td>17</td>
</tr>
<tr>
<td>Burmester</td>
<td>12</td>
</tr>
<tr>
<td>Cadoli</td>
<td>6</td>
</tr>
<tr>
<td>Campbon</td>
<td>32</td>
</tr>
<tr>
<td>Cao</td>
<td>12</td>
</tr>
<tr>
<td>Chart</td>
<td>58</td>
</tr>
<tr>
<td>Chen</td>
<td>13, 14</td>
</tr>
<tr>
<td>Chiong</td>
<td>57</td>
</tr>
<tr>
<td>Chiongs</td>
<td>13, 14</td>
</tr>
<tr>
<td>Chow</td>
<td>14</td>
</tr>
<tr>
<td>Chung</td>
<td>14</td>
</tr>
<tr>
<td>Ciancarini</td>
<td>2</td>
</tr>
<tr>
<td>Cimitile</td>
<td>5, 6, 9, 19, 42, 48</td>
</tr>
<tr>
<td>Clarke</td>
<td>40</td>
</tr>
<tr>
<td>Cordy</td>
<td>29, 30, 31</td>
</tr>
<tr>
<td>Corradini</td>
<td>15</td>
</tr>
<tr>
<td>Cortellessa</td>
<td>47</td>
</tr>
<tr>
<td>Costagliola</td>
<td>15</td>
</tr>
<tr>
<td>Cuthill</td>
<td>57</td>
</tr>
<tr>
<td>Cybulski</td>
<td>15</td>
</tr>
<tr>
<td>D’Ambrogio</td>
<td>16</td>
</tr>
<tr>
<td>Damm</td>
<td>16</td>
</tr>
<tr>
<td>De Lucia</td>
<td>5, 6, 9, 19, 42, 48</td>
</tr>
<tr>
<td>Dean</td>
<td>30, 31</td>
</tr>
<tr>
<td>Desikan</td>
<td>17</td>
</tr>
<tr>
<td>Deufemia</td>
<td>15</td>
</tr>
<tr>
<td>Dewar</td>
<td>9, 48, 53</td>
</tr>
<tr>
<td>Di Marco</td>
<td>47</td>
</tr>
<tr>
<td>Di Nittos</td>
<td>41</td>
</tr>
<tr>
<td>Dörfel</td>
<td>3</td>
</tr>
<tr>
<td>Dörr</td>
<td>3</td>
</tr>
<tr>
<td>Ducasde</td>
<td>51, 53</td>
</tr>
<tr>
<td>Earl</td>
<td>18</td>
</tr>
<tr>
<td>Eikemeier</td>
<td>19</td>
</tr>
<tr>
<td>Eikering</td>
<td>40</td>
</tr>
<tr>
<td>El-hour</td>
<td>19</td>
</tr>
<tr>
<td>Estublie</td>
<td>8, 20, 38</td>
</tr>
<tr>
<td>Fagel</td>
<td>39</td>
</tr>
<tr>
<td>Falcao</td>
<td>40</td>
</tr>
<tr>
<td>Feller</td>
<td>31</td>
</tr>
</tbody>
</table>
Königs..............................................21, 35
Koschke..............................................47
Kühl...............................................40, 44
Lang..................................................33
Lavazza...............................................6, 41
Lechner..................................................19
Leung...................................................52
Li.........................................................58
Lin........................................................14
Lings....................................................35
Liu........................................................32
Lladó......................................................47
Lo..........................................................14
Lodde.....................................................19
Loftus.....................................................22
Long.....................................................18, 52
Lundell...................................................35
MacKinnon..............................................53
Maguire..................................................36
Maligny....................................................29
Mancini.....................................................6
Marashi.....................................................11
Margaria....................................................36
Mariani.....................................................15
Marlin.....................................................28, 37
Maruyama..................................................37
Matzel.......................................................9
McCarthy....................................................37
McDaid.....................................................36
McGinnity..................................................36
Mecella.....................................................6
Medvidovic...............................................34
Mehra.....................................................38
Melo.......................................................20, 38
Mens.........................................................38
Merelli.....................................................15
Mi..........................................................38, 39
Michaels...................................................39
Molter......................................................5
Morel.........................................................39
Morris.....................................................29, 43
Mota.........................................................40
Mueller.....................................................40
Mugridge...............................................23, 24, 25
Muller-Glaser..........................................40, 44
Nagel.......................................................37
Neema.......................................................33
Nejmeh.....................................................50
Norman.....................................................13
Norrie......................................................41