
Configuring software, reconfiguring memories: the influence of integrated systems on the reproduction of knowledge and routines

Luciana D'Adderio

Recent advances in information and communication technologies have provided a substantial push towards the codification of organizational knowledge and practices. It is argued that codification, and the subsequent delegation of organizational memory to software, entails fundamental structural transformations to knowledge and routines as these are reconfigured and replicated in the form of new computer-embedded representations. The paper demonstrates that the process of embedding knowledge and routines in software holds fundamental implications for the ability of heterogeneous organizational groups, functions and communities to co-ordinate their efforts and share knowledge across function-, discipline- and task-specific boundaries.

1. Introduction

The recent diffusion of information and communication technologies (ICTs) has provided a strong push towards the articulation and codification of organizational knowledge and practices. Scholars, for example, talk of fundamental changes to the mechanisms of replication of organizational knowledge and routines being brought about by codification and the introduction of 'inscription technologies' (cf. Foray and Steinmueller, 2001).

As a result of such advances, increasing amounts of organizational knowledge or 'memory' are being embedded in software, or related computer-based media. Integrated information systems, for example, contain centralized data repositories that are specifically designed to store and co-ordinate all knowledge and activities that contribute to the definition, engineering, production, manufacturing and maintenance of an artefact across the extended organization and along its entire lifecycle. Such repositories are aimed at supporting the organization's ability to store and integrate distributed knowledge sources and to co-ordinate and synchronize dispersed processes and actions across function-, discipline- and task-specific boundaries.

This process of codification entails fundamental transformations to the knowledge and practices that are being codified. These transformations are of a structural nature as

they involve the reconfiguration of both knowledge and routines often according to the assumptions and the rationale that are embedded in software during its design, implementation and use. We argue that such radical reframing holds important implications for the organization's ability to create, retain, retrieve and reuse knowledge.

Specifically, the act of delegating organizational (i.e. product- or process-related) memory to software raises two fundamental issues. The first concerns the process of embedding knowledge and routines in software. For example, what kinds of knowledge can be embedded in software? How is knowledge changed (i.e. reconfigured, restructured) in the process of embedding in software? What is the role of software-embedded representations? How are they created? What is the influence of software, and software-based representations, on the organization's ability to retain, retrieve and reuse knowledge and to construct 'shared meanings' across functions?

The second issue concerns the reproduction of software-embedded knowledge and routines. For instance, once they are created, how are knowledge representations reproduced (i.e. circulated and adopted) across an organization? How are they enacted and incorporated in learning 'scripts'? In other words, how are such representations turned into actual expressions?¹ What is the influence of representations on the patterns of communication and knowledge sharing across heterogeneous organizational 'epistemic communities' and 'communities of practice'?²

We argue that the process of embedding memory in software has radical implications for the organization. The case studies show that it is not so much knowledge codification *per se*, but the way in which knowledge is restructured in the process of being codified that holds important implications for the organization's ability to acquire, retrieve and reuse knowledge, and therefore for its innovative potential. Specifically, the paper shows that the ways in which representations are generated and reproduced across an organization can fundamentally affect the organization's cognitive dynamics.

In analysing the introduction of a new software-based product and database structure, the first case study examines the influence of software on organizational *declarative* memory. The example shows that, while software aims to promote communication and the co-ordination of viewpoints and actions by imposing a single standardized product and process representation throughout the organization, it partially fails due to the persistence of local heterogeneities and idiosyncrasies, such as function-specific languages, cultures and knowledge bases. In contrast with much

¹For a theoretical characterization of routines 'as expressions' and 'as representations', see Cohen *et al.* (1996).

²'Epistemic communities' or 'epistemic cultures' can be defined as groups of agents working on a commonly acknowledged subset of knowledge issues and who at the very least accept a commonly understood procedural authority as essential to the success of their knowledge activities (Knorr-Cetina, 2000; Cohendet and Meyer-Krahmer, 2001). The notion of 'communities of practice' identifies groups of persons engaged in the same practice, communicating regularly with one another about their activities (Lave and Wenger, 1990; Cohendet and Meyer-Krahmer, 2001).

organizational literature, this example illustrates that organizational 'shared meanings' come not as a premise but as a result of learning activities aimed at establishing a sufficient level of coherence within the organization.

The second case study focuses on the influence of software on organizational *procedural* memory. It shows that the introduction of software restructures existing organizational processes and procedures, thus radically affecting the way formal and informal knowledge flow into the product design process. Specifically, it illustrates how the assumptions and parameters used by software to manage the workflow process can obstruct experimentation and hinder informal communication and knowledge sharing in the early stages of the design process. In contrast with some management literature's rather optimistic characterization of flexible software technologies, this example shows how their presumed benefits, such as the increased ability of functions and groups to communicate and collaborate, may only partially materialize in practice.

Our evidence draws principally upon a critical change event, namely the implementation of Product Data Manager (PDM) software at a leading automotive organization. In order to capture the mechanisms of coevolution of software-embedded and organizational cognitive dynamics in the context of actual practice, we have adopted a qualitative methodology based on participant observation.³ Ethnographic research, which looks at documenting the *actual* contents of routines, has been described by some authors as being ideally placed to generate '... rich and suggestive results as well as providing the essential grist for theory development' (Cohen *et al.*, 1996: 4). This in accordance with the remit of grounded theorizing, which, according to Eisenhardt (1989) and Glaser and Strauss (1967), can provide outstanding benefits in terms of improved understanding, empirical validity and frame-breaking theory building.

The participant observation was conducted over a one-and-a-half-year period with an average frequency of two days a week and was complemented by in-depth, semi-structured interviews. These activities were conducted across all organizational functions (industrial design, body engineering, product engineering, engineering release systems, production, manufacturing, marketing, etc.) and levels (from technicians/designers to project/programme managers and strategic management). To further validate the results and improve the robustness and generalizability of the findings, the evidence collected at the main case firm was compared with those obtained at two other world leading manufacturers in the aerospace (high product complexity) and consumer electronics (low product complexity) sectors; while having adopted the same software package, these organizations were at different stages in the software implementation lifecycle (the aerospace company being the most mature).⁴ The evidence collected has been compiled in the form of two detailed case studies focusing,

³Participant observation involves assisting with and, to different degrees, occasionally taking part in the organization's activities. To facilitate this, the author was provided with a contractor badge, which allowed her similar access to the premises and corporate events as a regular employee.

⁴A further set of one-day interviews was conducted across a range of other firms spanning from auto- and aero-engine manufacturers to Formula One. Interviews were also held at the software producer

respectively, on the implications of delegating declarative and procedural memory to software.

2. Software as a repository of declarative and procedural memory

Before we can proceed to analyse the influence of software on the processes of knowledge codification, storage, retention and reuse, we first need to identify exactly what types of knowledge and memory can be stored in software. A useful notion is Anderson's (1983) distinction between 'procedural' and 'declarative' memory. Procedural and declarative memory represent the *content*, or the 'what', of organizational memory (Walsh, 1995). They reside both in *social structures* and *practices*, such as shared information, group values or routines, and in *material structures* and *practices*, such as in blueprints, reports, procedures, etc. (Walsh and Ungson, 1991). Both social and material structures and practices evolve in relation to software structures and practices (and vice versa); however, because social structures and practices are more complex than the material counterpart, it is reasonable to expect that software cannot store or capture the former in their entirety (Walsh and Ungson, 1991). We therefore need to identify how both procedural and declarative types of memory are reshaped while being embedded in software and during the process of memory retrieval and reuse.

2.1 PDM as a repository of declarative memory

Declarative memory is 'memory for facts, events or propositions' (Anderson, 1983); it can be stored in written documents, databases, group records, individual knowledge bases, and in intranet systems that make declarative memory widely available within an organization (Moorman and Miner, 1998). The databases or 'vaults' of enterprise-wide software systems, for example, are designed to store increasingly vast amounts of product, process and organizational declarative memory.

PDM, for example, the object of our study, is designed to store, control and distribute the whole of the enterprise-wide information and (codified) knowledge required for product development and beyond. PDM is designed to support the storage, organization, management and access control to heterogeneous product definition data; for instance, its vaults store systematic records of data about component and assembly drawings and their set of attributes (size, weight, where used, etc.) and options. Documents relating to components and assemblies can be similarly classified, each document having its own set of attributes (part, number, author, date entered, etc.). Such software capability is intended to allow the organization's working stock of components to be organized in a clear, *hierarchical* network structure.

PDM is designed to store *structured* data, such as the relationships between parts, features, assemblies and systems that constitute a product or a technology. Within

organization. The names of all organizations and personnel interviewed cannot be disclosed to preserve confidentiality.

PDM, systems and parts are arranged into a multilevel hierarchy; by ‘exploding’ the structure of a bill of materials (BOM) for a specific product or item, for example, it is possible to visualize the hierarchy trees that identify the exact location of each part, feature, component, etc., within the overall product structure. This can allow an engineer to recall a complete BOM on screen, including documents and parts, either for the entire product or selected assemblies.⁵ Structuring the product into a hierarchy of parts that can be easily modified and moved within a product structure is meant to facilitate the process of introducing changes at every stage of product development. Engineers, for example, can ‘. . . change an assembly by adding parts, taking away parts, or copying whole sections of an assembly tree from one part of the assembly to another’ (Software Handbook, 1998).

This capability is supported by the software’s object-oriented architecture, which is designed to facilitate the process of the inscription of artefacts’ structures and organizational procedures in software while allowing their modification as they evolve over time. Specifically, the object-oriented software architecture is designed to ensure that: (i) both product and software development can be optimized through the reuse of previously designed and tested components—in this case, the memory stored in the software is intended to prevent duplication of efforts; and (ii) that there is integration with other software applications that also support configuration control and manufacturing resources management—the software-embedded memory, in this case, is meant to enable higher level co-ordination among software applications and organizational functions.

Due to its data storage and control capabilities, PDM can be used to support the management of product/process changes along the entire product lifecycle. For example, it helps the development team to keep track of the evolving product structure as well as of the evolution of the parts and models associated with it. The digital product configuration built into PDM is designed, managed and updated to reveal at each query by designers, engineers or managers the configuration of the product and the development status of each associated part at that precise stage in time.⁶

This capability is meant to procure important benefits, including: facilitating product/process maintenance throughout the product lifecycle; supporting the modification of configurations, from product definition to product release; helping to trace responsibility for failure in case of an accident; co-ordinating product development activities and stages; enabling product customization and the control and extension of product families. An example of this is the software’s ability to store product configurations. PDM can store the configuration of every item that makes up a product or a technology, including all its configurational variations. In the case of a vehicle, for example, our software will store all possible configurational combinations (left-hand drive, three-door hatchback, air conditioning, ABS, etc.) for each vehicle model. All

⁵Software producer’s web page.

⁶In practice, this is achievable only in advanced and successful implementation circumstances.

configurations are memorized in the software vault only once but can be retrieved many times, and used as input for many different BOMs.

The software-embedded product configurations continuously evolve, during product development and beyond (i.e. during production, manufacturing and maintenance). As the requirement for changes arises, the configurations are retrieved, changes are implemented and the revised configurations are again memorized in software. During product development, for example, PDM software can help to ensure that engineers are working with the most recent and correct configuration for each item (a part or a system); this facilitates the incorporation of feedback as well as preventing practitioners having to work on an outdated configuration. After product release, the software can help tracing faults in existing configurations as well as incorporating user feedback.⁷

Keeping track of past and evolving configurations can help *reconstructing* the 'whys' and 'hows' behind a part's design, testing or manufacturing, as well as providing a means to assign responsibilities for parts failures.⁸ Identifying problems, faults and responsibilities implies a process of reconstruction, as, due to the complex socio-technical nature of decision-making and problem-solving, only a small part of the knowledge can be stored in software. Reconstructing the rationale behind a decision and understanding the process by which such a decision is implemented requires tacit knowledge and experience to reapply meaning to the knowledge that has been codified and stored.⁹

A recent theory offers that, to the extent that software-stored information is *reinterpreted* and *recontextualized*, it can lead to retrieving the original meaning of the embedded memory (Bannon and Kuutti, 1996). Such a view, however, is problematic for two reasons. Firstly, in the process of embedding in software, the *meaning* of information is 'standardized': only the syntax (the data) can be stored, not the semantics (its meaning). For this reason, the reinterpretation of stored information is always required. Because such interpretation occurs in the light of new knowledge, the content of the information retrieved is *changed* every time this is being reinterpreted and adapted to a new context. Secondly, the data stored is 'historical' which implies that

⁷In the case of aircraft component failure, for example, an organization can trace the original configuration and modify it so that new aircraft model can incorporate the improved configuration: when a bolt fails, for instance, it is possible in principle to trace who designed it, who produced it, who tested it, who maintained it, when, and how often. This capability is especially important for artefacts that are built in batches (i.e. cars, aircrafts) (interview, N.G.).

⁸The 'why' is the rationale behind a certain decision; the 'how' instead represents the way in which a particular design solution was achieved or executed (i.e. the sequence of steps by which a part or system is designed).

⁹The notion of tacit knowledge refers to that form of knowledge that cannot be easily expressed (Polanyi, 1968; Vincenti, 1990; Cohendet and Meyer-Krahmer, 2001). For a deeper discussion about the nature of the boundaries between tacit and codified knowledge, see Cowan *et al.* (2000) and D'Adderio (2003).

those (technological and organizational) conditions, which underpinned a choice of a configurational solution in the past, may not be anymore retrievable or relevant.¹⁰ *Learning retrospectively*, or the incorporation of new knowledge into past configurations, involves the reinterpretation of stored information/knowledge in relation to current knowledge/patterns/capabilities. While some recent software applications are designed to store also a *reduced* and *stylized* version of 'know-how' (a codified version of a routine, a formal description of the design workflow, etc.), the process of reconstituting the 'hows' and 'whys' remains essentially problematic.

In emphasizing the software's inability to translate raw data into information or even knowledge, Kogut and Zander (1992) have emphasized that the rationale behind a design choice is always complex so that (i) it is very difficult to rebuild the meaning of an action or decision just by looking at a 'configurational history' stored in software; and (ii) it is largely unfeasible to embed the tacit side of routines and actions in software. These considerations raise an important issue: if only part of the individual's or organization's knowledge can be 'captured' by software, what is it exactly that software can store, and which are the consequences of embedding knowledge in software? While the 'interpretive' literature stream has given these issues consideration (cf. Weick, 1979), it has not sufficiently investigated the organizational conditions that are required to enable the capability of interpreting stored memory and learning from the past. Our first case study shows that, unless an organization has achieved a high level of integration and coherence, the interpretation of software-stored memory can prove highly problematic.¹¹

2.2 *Software as a repository of procedural memory*

While 'declarative' memory is related to facts, theories or episodes or 'know-that', procedural memory is centred on *skills*, or 'know-how' (Cohen *et al.*, 1996). Its definition as 'memory for how things are done' or for 'things you can do' relates procedural memory to skills or routines; in its long decay times, and greater difficulty of transfer and of verbalization, procedural memory often represents tacit knowledge for individuals and organizations (Winter, 1987; Cohen, 1991).

While computer systems and organizational learning literatures have principally explained the role of software in terms of storing 'declarative' data into computer databases, we argue that there is another way in which software can play a role with respect to organizational memory that goes beyond providing a source of declarative memory to the decision-maker. As the capabilities of software systems progressively expand to include the storage, management and control of organizational processes, these

¹⁰That is, the designer of the part may have left the firm; the part's supplier may be out of business; the organization may have lost the competences that sustained the design/production of that configuration.

¹¹In this case the evidence illustrates that the substitution of software for paper and CAD sketches creates a barrier to the interpretation of information for some functions.

are increasingly codified and embedded in software in the form of software-embedded 'routines'.¹² An important role is therefore created for software as a repository of organizational *procedural* memory.¹³

With the introduction of advanced software systems, an increasing number of organizational processes or 'routines' are embedded in software and managed through the enforcement of rules and constraints; these include the rules that ensure product/system integrity or reliability, or its coherence and compatibility with other products/systems; they also include constraints which can be of a legislative or regulatory nature or can be dictated by design requirements. Software-embedded rules regulate all changes in both the parts and model data, and their relationships in the hierarchy or product structure; acting as normative criteria they constrain changes and thus ensure their feasibility. For example, the software can help to ensure that any changes implemented are both valid and concurrent.

In embedding a set of rules and constraints, software behaves as 'dual enabler':¹⁴ it closes certain search spaces (i.e. technologically or organizationally unfeasible spaces) while expanding others (i.e. feasible spaces). This function is intended to diminish the risk of 'reinventing the wheel', or preventing the discovery of faults or incompatibilities downstream in development, where the risk and costs of failure would be much higher. A large amount of stored memory is not in fact necessarily inconsistent with innovation as it does not by itself lead to core rigidities (cf. Leonard-Barton, 1995). In delimiting territories of exploration to those technologically or organizationally feasible, the software-embedded constraints mostly imply that creativity or experimentation are only allowed within 'feasible' spaces. In constraining, therefore, software also enables.

There is another side, however, to embedding procedural memory in software. The process of embedding organizational routines in software is often preceded by substantial efforts to characterize and optimize all existing procedures. This operation often entails two steps: (i) the comparison of existing processes against a set of standardized procedures identified as industry 'best practice'; and (ii) the modification of existing processes to migrate towards best practice and therefore realize the full potential 'promised' by the software.

The extent of 'process re-engineering' brought about by software qualifies not simply as the straightforward substitution of existing for new codified procedures, as is often suggested in the BPR literature, but as a form of 'genetic re-engineering' (cf. Cohen and Sproull, 1991) that affects the interplay between formal and informal,

¹²Here the word 'routines' is intended in the narrowest of its definitions (cf. March and Simon, 1993).

¹³Advanced software systems, for example, can incorporate 'knowledge-based' applications, which automate small design routines, thus producing an opportunity for substantial time saving, avoiding duplication of effort, and supporting economy of action by reducing the amount of repetitive work required. These technologies capture the technological workflow allowing the software to record the procedure used by engineers in the design of an artefact (interview, R.M.).

¹⁴On the dual role of software, see also Orlikowski (1992).

declarative and procedural, codified and tacit knowledge.¹⁵ The main assumption here is that the way procedural knowledge is reshaped and reconfigured during the process of articulation, codification and inscription in software holds important implications for the organization's ability to retain, retrieve and reuse memory. How knowledge is embedded in software, for example, can influence the relative proportion of formal and informal, tacit and codified knowledge sources that become an input for the development process.

As mentioned earlier, only part of the designer knowledge can be effectively embedded in software. While the process of obtaining a solution can be codified, therefore, the process of codification is not able to fully capture *why* a particular design procedure or solution was chosen and *how* it was implemented (Wildawsky, 1983; Kogut and Zander, 1992; Bannon and Kuutti, 1996). According to the interpretive approach, since software tools are only able to capture part of the designer's knowledge (i.e. its codified, formal side), an engineer must every time reinterpret the software-embedded data or information in order restore its meaning in relation to a new context. In this sense, memory retrieval is a type of learning in itself (Spender, 1995).

The active interpretation of software-embedded memory (especially procedural), however, is not always possible or feasible. A key characteristic of procedural memory is that it becomes *automatic* or accessible unconsciously (Moorman and Miner, 1998). As a result of its automatic character, procedural memory can have contrasting effects on innovation: on one hand, by providing a rich vocabulary of action from which to choose, it can improve the likelihood that improvisation will produce coherent action (co-ordination); on the other, a high level of procedural memory can constrain novelty, due to the habituated or uninspired use of a vocabulary pattern (Cohen *et al.*, 1996). While this constraining effect, to some degree, may be true of all types of information stored in a memory (Leonard-Barton, 1992), its consequences are more problematic in the case of procedural memory, because this tends to be accessed automatically (Cohen, 1991).

A potential problem arises as software tends to substantially increase the extent of automatic retrieval of procedural memory. Software can behave as an 'invisible agent' automatically triggering routines, influencing the process of beliefs formation, and providing a 'path of least resistance' for action. Software-embedded memory can indeed both guide and constrain future behaviour. It can direct actions by embodying preferential rules and behaviours; and it can constrain action in a way that it is not always apparent to the decision-maker, as software-embedded rules and routines tend to sink in and reinforce automatic rather than deliberate behaviour.

¹⁵While some literatures, such as BPR, have implicitly examined the role of IT in storing process knowledge, these have failed to analyse the cognitive and motivational implications of delegating organizational procedural memory to software. Our approach aims to make up for BPR's lack of attention towards the *epistemic* content of routines; it also highlights the need to move beyond the abstractedness of those theoretical contributions that focus essentially on knowledge types, thus neglecting the importance of their link to organizational practices (cf. Nonaka, 1994).

To some extent, and in some circumstances, therefore, the automatic behaviour generated by the unquestioning following of software-embedded rules and routines can substitute itself to active interpretation of software-embedded information.¹⁶ Although automatic behaviour is always present in routines in some measure, this is emphasized when routines are embedded in software. During the process of inscription, for example, informal and tacit knowledge are abstracted and codified; it is the formalization of the tacit, 'interpretive' component of routines that reinforces a standardized and automatic behaviour.

Recalling Cohen *et al.*'s insightful characterization of routines 'as expressions' and 'as representations', we argue that only 'representations' can be embedded in software. The process of embedding in software in fact fundamentally alters the epistemic content of routines, reducing them from expressions to representations.¹⁷ In addition to this, as mentioned earlier, software embodies assumptions, and idealized, industry-standard processes that substantially reshape existing organizational processes as these are inscribed in software. The process of embedding routines in software has therefore important consequences over both the structure and content of organizational procedural knowledge.

Once we fully acknowledge the implications of delegating procedural memory to software, new issues arise that deserve further exploration. These include finding out to what extent and in which circumstances the software-embedded routines, rules and constraints may induce conservative behaviour and lock-in into existing patterns of action. Our second case study will show that it is not the *level* of procedural memory stored in software that can hinder innovation, but the *formats* according to which memory is stored. The structure of software-embedded memory, for example, can influence the extent to which new knowledge and actions can become inputs for the development process.

¹⁶In order fully to characterize the role of software as a repository of organizational memory, therefore, we need to shift our attention from the individual decision-maker (engineer or manager) to also account for the role of organizational routines and practices in deploying knowledge (Coombs and Hull, 1997). This theoretically entails complementing the insights provided by the 'organizations as interpretive systems' approach (Walsh and Ungson, 1991; Spender, 1995; Bannon and Kuutti, 1996) with those in the field of organizational knowledge and learning (Nelson and Winter, 1982; Levitt and March, 1988; Cohen, 1991; Cohen and Sproull, 1991). While the majority of contributions in the latter have been centrally concerned with the *stability* of routine patterns (cf. Nelson and Winter, 1982), however, we intend to concentrate on the *dynamic* evolution of routines, in characterising how routines evolve as a consequence of software implementation.

¹⁷Software provides an ideal ground to study the interplay of the two levels of routines: routines as expressions and SOPs. According to Dosi, the *relation between the two levels*, however defined, is a 'promising area of investigation' (in Cohen *et al.*, 1996: 18–19). The inscription of routines in software (cf. the workflow process, Section 3.2 in this paper) provides an excellent standpoint to observe such dynamics and therefore the 'reproduction of routines'.

3. Software as a repository of organizational memory: the evidence

3.1 PDM and the integration of heterogeneous technology and data structures

The first case study provides an empirical account of the difficulties encountered in the process of structuring, storing and reusing *declarative* product memory, and of constructing shared meanings within a leading automotive organization implementing PDM software. The introduction of PDM at the upstream (R&D) end of product development represents a fundamental step in the migration of technology and organization towards an integrated product development environment. The introduction of PDM at the engineering end of development, however, appears to have created a mismatch between two generations of technologies and two development (plus one administrative/control) functions that are the users of the technology.¹⁸

The coexistence of incompatible structures. PDM and Total Modular Statement (TMS) are a new and an old technology used, respectively, in the engineering and in the production departments of our organization. Both technologies are used to manage a complex document, the 'Engineering Parts List' (EPL), which is a structured list of all the parts that belong to the same development programme, including their relationships; the EPL is used to generate and maintain the evolving configuration of a product, plus all its variants, over time.

A vehicle's EPL is a hierarchically ordered list that contains all the items that compose a product's configuration (i.e. a passenger vehicle), including all of its component systems (i.e. car body), sub-systems (i.e. a door), features (i.e. door trim), assemblies (i.e. gearbox), sub-assemblies (i.e. gear stick), etc., down to the finest detail (Figure 1). An EPL is compiled for each and every vehicle being designed and produced; it contains all the product variants planned for a specific vehicle programme, and it is modified over time so to reflect the most updated version for a part or system configuration. Besides managing the product's parts, the EPL also memorizes the relationships between vehicle systems and their components, as well as links between each of the parts and the files containing that part's description, analysis files, CAD files, etc.

The EPL is used by both engineering and production,¹⁹ but with different objectives. Engineering compiles an EPL in order to facilitate the process of vehicle configuration extraction; this is done by extracting those parts that belong to an individual product configuration out of the total list of parts contained in an entire vehicle programme.

¹⁸While principally involving engineering, production and the engineering release systems departments, the issues discussed here involve several other functions including industrial design and manufacturing.

¹⁹Engineering is responsible for designing an artefact and releasing its configuration to production. The production function uses such configuration to build (partial and full) prototypes for testing purposes. Once testing is completed and modifications are approved, production releases the final configuration to manufacturing for build.

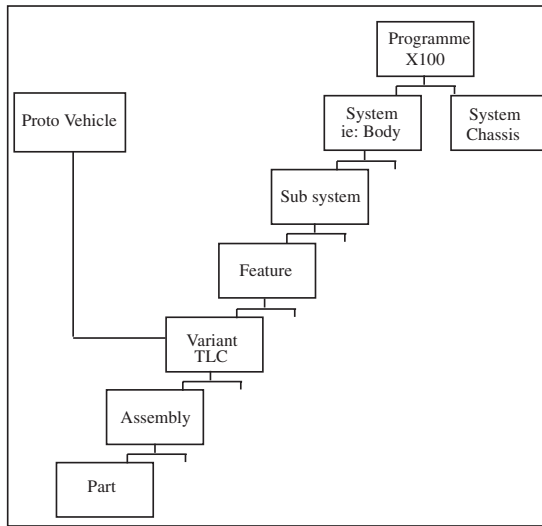


Figure 1 Schema of the EPL in Lotus Notes (interview, K.T.).

Production uses the EPL for ‘prototype verification’ of fully built production vehicles. Engineering release systems (ERS) is the function responsible for releasing the approved, or ‘frozen’, engineering list to production (Interview, K.T.).

While the engineering and production EPLs contain the *same data*, they vary substantially in the way they *structure* such data (i.e. the way they manage the relationships among the various parts composing a technology assembly). PDM and TMS technologies reflect this difference in the way they record data and their relationships (Figure 2). To generate an EPL, the PDM software orders product assembly data in a *hierarchical structure*, based on ‘parent–child’ type of relationship. ERS’s (production-released) EPL, instead, is characterized by a *flat file structure* where the relationships among different items and their exact position into product assemblies are captured by complicated, horizontal, Boolean statements. The more complicated is the product structure, and the higher is the number of configurational (vehicle) variants, the longer and more complicated are these algebraic statements (Figure 2).

In our organization, as a result of the introduction of PDM, PDM and TMS technologies co-exist side by side. As a consequence, the basic incompatibility between the two different modalities in which PDM and TMS technologies organize information and compose product structures is emphasized. PDM implementation has in fact highlighted a mismatch between the way data is acquired and interpreted by two different ends of product development: engineering and production.

Before the introduction of PDM technology, both the engineering and the production EPLs were manually compiled with the assistance of a number of basic tools including Excel spreadsheets, paper drawings and CAD sketches. The manual process of extracting a vehicle configuration from the EPL is very complicated, given the over-

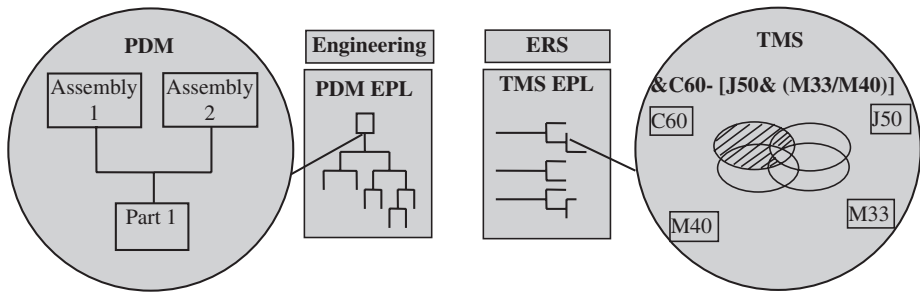


Figure 2 PDM and TMS EPL structures.

whelming amount of data and configurational combinations existing for each vehicle and for each one of the vehicle variants; as a consequence, the information extracted and assembled in configurations through a manual process is liable to contain numerous mistakes. The risks involved are high, as these mistakes go often undetected until they get to the production stage, where the configurations are tested on fully built prototype vehicles; at such a late stage, however, any major changes to parts or assemblies entail substantial disruption and significant time and profit losses, as modifying one part will have an impact on many other related parts and systems (interview, M.C.). The recent increase in product/technology complexity in the automotive industry, related to a higher level of product–market diversification and customization, has greatly enhanced this problem. For each vehicle now there are many configurational variations, corresponding to the various vehicle variants and customer options. The introduction of PDM software is partly related to the intention to address this problem and to support the management and control of product variant complexity by automating the EPL extraction process (interview, D.A.).

Another major motive behind the introduction of PDM and the implementation of a unique EPL across the whole organization is the attempt to standardize and unify the actions and viewpoints of different groups and departments; this is aimed at achieving a higher level of interfunctional co-ordination and integration in design and development. The potential benefits of implementing a single, PDM-managed EPL are, therefore, not simply related to the ability that the new technology provides to visualize and display assembly data; the act of embedding configurational data into PDM helps to ensure that the assembly data is at all times configured, controlled, synchronized and verified, and *can be shared* by different organizational functions throughout all stages of the development process, and beyond. In this sense it is intended to support the co-ordination and integration of knowledge and practices that is at the basis of the software implementation philosophy (internal company document, I.P.D.S.).

According to a senior engineering administrator, the introduction of PDM at the engineering end of PD has facilitated and possibly improved the management of the engineering List, facilitating the process of EPL extraction upstream in product

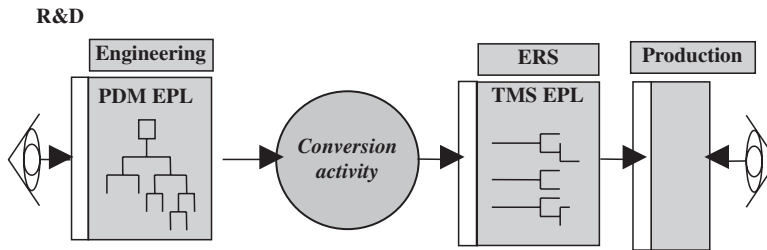


Figure 3 PDM and TMS engineering parts lists.

development (interview, M.C.). At the same time as improving the process at the engineering end, however, the introduction of a PDM-managed list has created a fracture with the production list; this fracture, as we have seen above, is due to the fact that TMS technology is structurally incompatible with PDM's parent-child database morphology. Because of their radical difference in structuring the data, it is not easy to shift between the two technologies. On one hand, production finds it difficult to understand PDM's EPL because they are not familiar with a parent-child structured list. On the other, only highly experienced practitioners are able to understand the complexities involved in structuring information according to TMS's logic. As an engineering administrator argued: 'Even with my experience I had to spend one week on this part of the configuration and found many mistakes' (interview, M.C.).

Before PDM implementation, basic techniques were in place that enabled each department independently to interpret the data in order to adapt it to its own specific information needs. The two different ways of structuring data were allowed to co-exist. In order to support the implementation of an integrated software strategy, however, practitioners in production are now expected to adopt the PDM-structured EPL, which is unfamiliar to them. The PDM-EPL configuration is information for engineering because they have the knowledge required to interpret it. Production instead comes from a different subdisciplinary background and is unable to make sense of the data contained in the engineering EPL.

Engineering, on one side, and production, on the other, are looking at the same data from two different angles, and they see two different structures (Figure 3): 'Both engineering and production look at their own EPL and think it is the correct one, and that it is the other that is wrong' (interview, K.T.). Clearly, none of the two EPLs is wrong. Simply each EPL is structured in a fashion that is unfamiliar to the other function, and therefore it is perceived by that function as being incorrect. What is interesting is that the two EPLs contain exactly the same data, although this is structured differently. Engineering and production basically speak two different configurational and database languages, corresponding to two different disciplinary and functional backgrounds.

The role of PDM in influencing meaning formation. Following the implementation of the integrated PD strategy, and the related introduction of PDM at the engineering end

of development, the parent–child typology of product configuration familiar to upstream functions is supported and becomes mainstream. This move follows the objective to introduce a single product definition that can (and indeed must) be shared by all functions. The common, standardized, product definition is implemented with the specific aim to co-ordinate and synchronize all actions and perceptions across the organization.²⁰

The PDM-embedded, parent–child-structured configuration, however, constitutes *information* only for the engineers who are able to interpret it and directly apply it to improve their processes; the same configuration is, in fact, simply *data* for production, where practitioners are unfamiliar with the parent–child structure, and are instead accustomed to using Boolean logic statements to identify product combinations. The introduction of PDM upstream has therefore emphasized a pre-existing *gap* in shared meanings formation, a divide in understandings and practice between engineering and production. The two departments have different subdisciplinary backgrounds (i.e. design versus production engineering), different subgoals (optimizing product design variants versus verifying physical vehicles' prototypes) and different incentives.

The introduction of PDM has provided an opportunity and a strong push for the integration of upstream and downstream development functions via the elimination of existing technological and organizational incompatibilities and inconsistencies. According to the software producers, the elimination of such 'bottlenecks' can be achieved by creating a common infrastructure that supports smooth communication and the sharing of seamless information flows across the enterprise (cf. software producer's White Paper); in their view, this would support the formation of common meanings among various groups within an organization, and various organizations within an enterprise.²¹ In order to achieve these potential benefits, the software-embedded rationality therefore dictates that the PDM-embedded configuration should be adopted across the entire organization; this includes downstream functions, whereby the new configuration should, in principle, replace any existing legacy technology such as TMS.²²

This is not what took place at our case organization, and indeed other organizations

²⁰Much of the IT and management literature have advocated the adoption of flexible manufacturing systems (such as PDM) as a means to reduce the trade-off between standardization and flexibility. In such literatures, flexible design and manufacturing systems (which are not just technology) are also reported to help the reallocation of resources from mundane tasks to more creative ones. The actual benefits of such technologies, however, have not been assessed in the context of practice. This work specifically shows how flexibility of the organization-software system is achieved (or not achieved) in actual practice; it demonstrates that, in certain implementation circumstances, these so-called 'flexible systems' can generate rigidities that prevent organizational innovation and adaptation.

²¹PDM represents in fact only one (though a fundamental) piece of the greater integrated software infrastructure.

²²While the issue of the incentives and costs associated with knowledge codification is important, this does not constitute the focus of this paper, which instead concentrates on the cognitive implications of software introduction for the organization.

interviewed were presented with similar issues. There, the decision taken was to maintain the two incompatible configurations and set up a complicated conversion procedure between PDM's parent-child configurations and TMS's Boolean algebra statements. Such a decision was due to a number of reasons, including financial considerations (i.e. the costs involved in simultaneously implementing the new technology across the entire organization); business considerations (i.e. the requirement to keep the production running while the rest of the enterprise caught up with PDM technology); and technology and departmental idiosyncrasies (i.e. production's familiarity with, and preference for, Boolean configurations, and the superior ability of TMS technology to convey a production engineer's knowledge into product definition).

According to the software producers, the seamless integration of data flows between PDM's and TMS's EPL is mandatory in order to achieve integration. Such integration would, in fact, 'close the [communication] loop' between engineering and production (software producer's web page). In terms of integration, the choice to set up a conversion activity rather than enforcing the implementation of a single structure across the two functions corresponds to the adoption of an 'interfacing' as opposed to the 'integrating' strategy supported by software producers. While the implications of the decision to retain both technologies is practically irrelevant from the point of view of exchanging information flows, this has important consequences for the formation of shared meanings.

Both integrating and interfacing in fact allow for information to be transferred from engineering to production and vice versa: configurational information from engineering gets translated into configurational information for production, via a process of conversion that is laborious, complicated and error-prone, but fundamentally feasible. However, while the conversion process ensures that the syntax is exchanged, this represents a missed opportunity for the (at least partial) integration of semantics; in other words, while interfacing is an effective means to exchange data, it does not support the formation of shared meanings across functions. As a result, the two functions cannot make sense of the other's EPL and are therefore unable to detect any mistakes generated by the automated conversion between PDM and TMS structures. This resulted in greater inefficiencies and inconsistencies than existed before software introduction.

It is important to emphasize that the integrated PD strategy has created the need to restructure product data according to a standard preferential format (parent-child). Rather than promoting interfunctional co-ordination and collaboration, the new configuration has emphasized a clash in function-specific languages, cultures and knowledge bases that, until that point, had been only latent; the attempt to impose a standardized representation across heterogeneous domains has emphasized existing incompatibilities attributable to idiosyncrasies of the development functions involved:

If you think about our organization, but, more importantly, about our Information Management Systems, it's nothing more than taking a picture. The problem you have got is: firstly, can everybody see the same picture?

The answer is, no way, we've all got different views. We might all say we are in the same landscape, but we all have a slightly different view of what that landscape is and what it should be. It might be the same subject, but it is a different picture; that's one problem . . . the second problem is that that picture representing our engineering information infrastructure is like a jigsaw puzzle, where *each piece must fit with the next piece*. The problem is, that we don't have the co-ordination required to create or cut out the pieces of the jigsaw. (interview, K.T., *emphasis added*)

An important issue is, therefore, that organizational co-ordination and integration (to an extent) must already be in place in order to support the integration of information systems. We can say that the two strategies mutually sustain one another. This does not imply that achieving superior integration of information systems should necessarily lead to unproblematic organizational integration. While some of the existing inconsistencies are likely to be eliminated as the implementation of the integrated PD environment progresses and a common configuration technology is introduced, others are likely to remain; this is due both to the fact that organizations tend to retain at least some of their specialized legacy applications, and to the persistence of departmental idiosyncrasies even in the most advanced and successful integration circumstances (D'Adderio, 2000).²³

The emergence of shared meanings and organizational coherence. The evidence above has illustrated how the introduction of PDM can radically reshape the process of organizational meaning formation; these results were obtained by emphasizing the difference between the integration of data flows and the integration of meaningful information. These findings unveil the limitations intrinsic in a substantial body of literature which chooses to deal with problems of interorganizational co-ordination by focusing principally on the exchange of information flows across departmental boundaries, rather than concentrating on learning (cf. Aoki, 1986; Sanchez and Mahoney, 1996). The need for co-ordinating heterogeneous specialized function, which derives from the increasing division of labour within firms and is central to contemporary

²³The persistence of department-specific specialized technologies, languages and methods is a likely scenario in many organizations (D'Adderio, 2001, 2003). This is attributable to several reasons: (i) legacies can be 'best in class' applications that display superior performance to the generic integrated system module that should replace them; (ii) local resistance to change; (iii) legacies are modified over time to embed new layers of knowledge and functionality so that eventually it is very difficult to know what kind of functions a legacy exactly performs, which makes re-engineering unfeasible; (iv) legacy applications sometimes represent the most efficient means to convey tacit and local, function-specific knowledge.

The persistence of clashes and inconsistencies is also attributable to the fact that often a new software implementation project begins (i.e. the implementation of a new package, or new version of same package) before an old one has been completed. This implies that at any one time, several overlapping and in some cases clashing technologies coexist in a firm. This is known in IT literature as 'the continuous building site' problem.

corporate innovative activities, however, '... cannot realistically be reduced to designing flows of codified information across functional boundaries. It also involves co-ordinated experimentation... and the interpretation of ambiguous or incomplete data, where tacit knowledge is essential' (Pavitt, 1998). In our case, production engineers are not able to use their knowledge to interpret and attribute meaning to the newly structured list. It follows that supporting the formation of shared meanings across the organization is not simply about ensuring smooth information and communication flows across functions, but also, and most importantly, about integrating meaning structures.

Another important observation stemming from our analysis is that the integration of organizational meanings does not inevitably follow the implementation of integrated software technologies. In the case of our organization, the introduction of integrated systems and the implementation of a new product and database configuration has instead emphasized existing clashes and incompatibilities between upstream and downstream development functions. These inconsistencies were not previously apparent, as before there was no need for the two functions to co-ordinate and synchronize their actions and perceptions. After the introduction of software, however, '... we have to operate in concert, towards co-ordinating our EPLs. Now we have to work together' (interview, K.T.). In enforcing collaboration and co-ordination between these functions, via the implementation of a common (standardized and standardizing) product structure representation, the integrated strategy has therefore highlighted existing incompatibilities, as '... people often cannot see what they take for granted until they encounter someone who does not take it for granted' (Bowker and Star, 1999: 44).

In our case, therefore, while resolving some inconsistencies as a number of economists have argued (cf. Cowan and Foray, 1997), software implementation and the subsequent codification and inscription of the product structure in software has also created *new* bottlenecks. These emerge as a consequence of rendering the existing manual procedures and informal languages obsolete, while forcing all functions to draw from the new, engineering-structured and management-enforced list. Codification, in our case, has involved the radical reordering of the knowledge relating to the product structure according to a specific format that is heavily biased towards the engineering language. Standardization, and the replication of such a format across the organization has proven more difficult than predicted.

Our example can be thus interpreted as a failed attempt to create a common (artificial) language by a means of standardization. The objective of integrated systems implementation was indeed to eliminate the inconsistencies among functions by introducing a common configuration that would allow different disciplines to communicate while maintaining their specific point of view. This strategy has encountered significant resistance, also due to the persistence of heterogeneous, local, discipline-specific languages, technologies and understandings across the various organizational functions. This has emphasized a trade-off between enforcing standardization and preserving heterogeneity, which is particularly significant in relation to the implementation of integrated

software technologies; as Bowker and Star (1999: 50) put it: 'a core problem in information systems design is how to preserve the integrity of information without *a priori* standardization and its often-attendant violence'. The issue of the persistence of heterogeneity (i.e. local knowledge 'pockets') across organizational domains has been also referred to as 'stickiness' of departmental contextuality of internally generated knowledge (von Hippel, 1994). This dictates that, even in ideal implementation circumstances, differences in interpretation of the same data are likely to persist as they correspond to the idiosyncrasies that characterize different groups within the same organization.

While a successfully implemented unique configuration could act as a standardizing device by helping to synchronize perceptions across heterogeneous domains, therefore, differences in interpretations are likely to persist. These differences in tacit knowledge, perceptions, interpretations, goals and incentives within and across an organization have not been emphasized by organization scholars, who have instead often taken the notion of shared meanings and organizational coherence as given (Brown and Duguid, 2000). The ability to achieve shared meanings comes not as a premise but as a result of learning activities aimed at establishing a sufficient level of coherence within the organization. In other words, rather than given, shared meanings and organizational coherence are 'emergent' systemic phenomena that must be explained, rather than assumed (cf. Cohen and Sproull, 1991; Holland, 1992). This emphasizes the need to unpack the concept of 'organizational culture' to study this as an emergent property that requires to be continually reconstituted.

Observing the processes by which our organization dealt with the escalating conflicts has helped us to characterize the circumstances in which shared meanings and organizational coherence can be achieved. We have shown that coherence, in our case, depended on the organization's ability to achieve a balance between forces driving the organization towards greater heterogeneity or greater standardization. Such tensions were exacerbated by the implementation of integrated software-based environments and the consequent push towards standardization. We have demonstrated that not so much the *increase* in codification but *the way* in which codified knowledge is stored and structured in the process of embedding in software, as well as the way in which knowledge was reproduced through standardization, constitutes a constraint that can influence significantly the ability of individual functions to interpret data and to draw meaningful information.

3.2 PDM and the management of the engineering workflow process

The second case study focuses on the implications of delegating *procedural* memory to software. It analyses the case of embedding the engineering 'workflow' process in software and explores the consequences of this for the organization. Specifically, it shows that while software-embedded procedural memory can introduce coherence, support economy of action and facilitate the synchronization of efforts, it can at the same time

obstruct informal actions and knowledge exchanges across design teams and organizational functions.

Workflow process definition and control. The management of the design workflow process is a complex task. During product development, many thousands of parts need to be designed; for each part, files need to be created, modified, viewed, checked and approved by many different people, many times over; different product parts call for different development techniques and require different types of data (solid models, circuit diagrams, analysis models, etc). Because work on any of these files has a potential impact on thousands other related files, there needs to be continuous cross-checking, modification, resubmission and rechecking. Due to the sheer amount of chain-induced changes, it is not uncommon for an engineer to be working on a design that has already been invalidated by the work someone else has done in another part of the programme (interview, M.C.).

Our automotive organization has implemented PDM software with the aim of bringing order into its workflow process; the software acts by disciplining and controlling the progress of each project by breaking down the workflow process into several different 'states' and by using predetermined 'triggers' and 'routings' devices in order to enforce these states.²⁴ Complex rules about the levels of 'authority to change' are embedded in the software, in such a way that only one department (or few people) at one time are allowed to implement changes to a configuration, while everyone else is only able to view it, by calling it up on their computer screen (interview, N.G.). This represents one way in which PDM can manage the design workflow process. Different PDM systems may differ in how much flexibility they permit within the framework discipline.²⁵ The most rigid systems, such as the one adopted by our case organization, utilize strict procedures to control the project's progress.²⁶ There, every individual or group is made to represent a state in the procedure: 'initiated', 'submitted', 'checked', 'approved', 'released' (internal document, W.I.P.). According to this procedure, a file record cannot move from one individual (or group) to the next without changing states. Every time, for example, an engineer working on a design wants to confer with colleagues as to the best way to approach a design, the system imposes that a change of state is triggered. Another way to express this is to say that, in this case, the 'authority to

²⁴Several sets of rules and assumptions are embedded in software at different stages of its design and deployment lifecycle. These include assumptions embedded by software designers about the industry and the sector where the software is to operate; rules and models to make the software work in a specific way (i.e. object-oriented programming philosophy); and it includes also management rules, which are imposed at the user level/adoption stage. All these assumptions and rules (software-embedded knowledge) interact with organization's knowledge and processes and influence the organization's cognitive dynamics as well as its behaviour.

²⁵Relative differences exist also between different revisions of the same PDM system.

²⁶The PDM package analysed here is a leading, state-of-the-art application solution adopted in most manufacturing sectors.

change' moves around with the file. This can constrain substantially the engineer's ability to communicate with others in the development team, as illustrated below.

The product engineering workflow. This is the process that regulates the handing over of a released configuration from design to engineering, ERS and production through many controlled 'release' (or 'freeze') stages. As part of the implementation of an integrated PD environment, the management of the workflow process is being delegated to PDM and CAD Data Manager (CDM) software.

Before being inscribed in software, the product engineering workflow is articulated and codified into a structured chart; the chart shows the workflow through the various changes of status of each part, from work-in-progress (WIP) parts through to full released (USE) status parts. The workflow chart, at our organization, reads as follows (Figure 4):

1. The CAD designer assigns the WIP status to a CAD model to generate a design solution. When design is complete, the CAD designer will change the status of the model from WIP to check in progress (CIP) and pass the model on to the surface engineer (SED) and product engineer (PED).
2. The SED and PED check the engineering content of the model; if this is not satisfactory, they reset the status to WIP and pass it back on to the CAD designer for modification; if it is satisfactory they revise the status to CHK (check) and forward the model to the engineering release system (ERS) group for release to production.
3. The ERS group can either release the model by changing the status to USE or return it to the SED/PED for any required changes while resetting the status to WIP.

This chart is then recorded into the CDM database, which is a relational database specifically designed to control the data and thus the workflow within the product engineering environment. CDM controls: (i) the evolution of a CAD model (status), which defines the position at which the part is considered to be at any given time, as it progresses to full released status; and (ii) who is the designer (or group) responsible for that model at any stage in time (ownership) (Software Handbook, 1998).

The CDM database stores the specific attribute prerequisites that control the change of status of a CAD model from the beginning through to release; this will ensure that, before a change of model status and ownership can be instigated, the system will perform a number of checks to ascertain that these changes are feasible and compatible

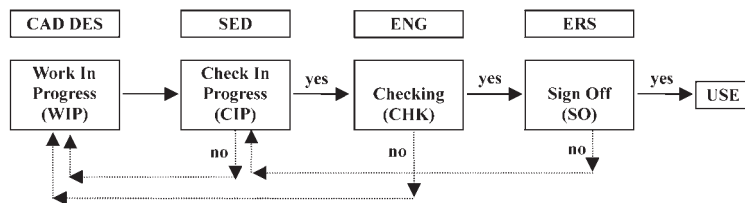


Figure 4 Workflow process.

(internal document, D.M.U.). In the CDM environment, CAD models are attached to the product structure tree where the relationships between parts and systems follows the parent–child structure mentioned in our earlier example; models of any type can be attached to the tree, as long as they are model types recognizable within the CAD application adopted (finite element analysis, illustration, kinematics, assembly, surface data, standard, part, package, etc.). Each CAD model is individually and temporarily ‘owned’ by a single user (engineer), who is the only person who has the privilege to update the model. Each user is able to relinquish ownership to a predetermined group of other users. A privileged user is also able to change the ownership of a model among a predetermined group (internal document, D.M.U.).

Co-ordinating such a complex workflow effectively requires the definition of the interdependence of the design tasks. PDM does this by creating a hierarchical relationship between files (i.e. the system can be instructed to prevent an engineer from signing off an assembly for release until all its individual parts have been released). A problem with this class of PDM systems is that the system can only handle data that have been *already released*, i.e. by an engineer or a group of engineers; the software cannot handle a *mix* of released and work-in-process data, which is typical of an evolving configuration. Within PDM, therefore, a model, file or record cannot move from an individual or group to the next without first changing status. The control exerted by the freeze/release procedure embedded in the software is such that the authority to change moves around with the file, and does not remain with the design originator; the file, therefore, cannot be shown to other people who have a different level of authority (i.e. people outside the particular design group or in other organizational functions).

There are two major implications. First, a problem may arise when the type of control exerted by PDM systems is applied at the early stages of development. Introducing a strong emphasis on administrative product control so early in the development of the product could decrease the innovative potential of the overall development process. For example, only released configurations (i.e. finished work) can be passed on to others, meaning that the designer cannot easily obtain feedback from other colleagues. The level of interaction in the product engineering environment, instead of improving, is likely to be hindered. In these circumstances, PDM may restrict the potential for co-operation across functions/disciplines, as authority to change is attached to status, and a file cannot be shifted around unless its status is changed.

Second, PDM does not facilitate the *exploration* of many technological and configurational alternatives, because it cannot handle mixed ‘development’ (WIP) and ‘released’ (USE) part data. This means that a designer may not be able to adopt a part (or file) taken from previous development programmes and use it in a new configuration, because the old parts hold a released status, while the parts they are working on are still WIP. This may also cause the premature freeze of a design, because it does not allow the designers sufficient scope for trying out different alternative configurations. This process inflexibility can, in principle, restrict the innovation potential of the development process significantly.

If the rules embedded in the software would allow for authority to be detached from the CAD file (i.e. if the software was able to handle a mix of released and development data), then the designer could potentially more easily interact and gain feedback from his/her colleagues, as is the case of some PDM systems. Other PDM systems, in fact, make it possible to give the task an identity of its own, separate from the people working on it. These systems use 'packets', which allow the engineer to manage and modify several different master documents simultaneously, as well as providing various supporting documents for reference. In this latter case, not only each packet's route through the system must be controlled, but also the relationship between packets (interview, M.C.).

By emulating paper-based processes, packets can facilitate the sharing of documents among team members. For example, although only one user can work on a 'master' design, colleagues working on the same project can be instantly notified that there is an updated master design and reference copies will be made available to them in their own packets. Packets also make it possible to move work around from department to department or from individual to individual, in logically organized 'bundles'. When an engineer requires the opinion of a colleague on a design, they can pass the entire job across to anyone else, as long as the master model and all the associated reference files are contained in and controlled by a packet. This operation will not trigger a change of state; besides, the formal workflow procedure is uncompromised by this informal rerouting because the authority to change the file's state does not move around with the packet, but remains with the designated individual (or 'owner').

This example demonstrates how software-embedded rules can potentially produce lock-in effects as well as obstructing flexible or innovative behaviour. While, on one hand, the software-stored memory prevents 'reinventing the wheel' as well as co-ordinating and economizing efforts and producing coherent behaviour, on the other it can introduce rigidities by preventing the exploration of numerous alternative technology configurations and by obstructing informal interaction among engineers.

Observations about procedural memory and software. The previous example has illustrated how software can influence the organization's ability to maintain flexible processes that allow the reuse of stored knowledge, the input of new knowledge, and substantial exploration and search for alternative design solutions. Stored procedural memory, however, does not necessarily induce conservative behaviour; it is rather the way in which such knowledge is structured (i.e. according to authority levels) that may be conducive to inertia. In the case illustrated it is the way in which procedures and rules are stored in software that can either help or hinder the possibility to 'capture' new knowledge as well as integrating informal knowledge in this formalized process.

By codifying procedures and specific sets of rules in software, certain actions are *constrained or impeded* (not necessarily implying that they cannot be bypassed, but that they can only be bypassed at great expenditure of resources, time or effort) while others are *supported and encouraged*: software performs as 'dual enabler'. The control exerted by software ensures that the actions taken by the various functions during PD are

coherent and synchronized. This can facilitate control and co-ordination as well as collaboration within and among development functions/groups. Thanks to such coherence, all departments can follow up the configuration as it evolves and timely deliver their input in the development process (concurrency, in other words). At the same time, however, the rules 'frozen' in software can hinder communication and informal exchange among designers and across functions.

Naturally, the interaction among engineers is rather more complex than exchanging files through wired connections. Designers can indeed share precious knowledge in very informal situations, and this remains a fundamental means for knowledge transfer, even after software implementation. The issue, however, lies in the fact that the behaviour 'embedded' in software often becomes invisible and/or unquestioned, and can therefore have greater influence over the final result than it may initially appear. The automatic behaviour embedded in software tends to become part of the status quo, 'the way we do things around here', and therefore its influence can be deep, pervasive and inconspicuous.²⁷

An engineer can avoid the rules and procedures enforced by the software; he or she has some degree of freedom in establishing when and how to follow a rule supported by software, and when instead not to comply and to 'work around' it instead. Workarounds are in fact an integral part of the design process (Gasser, 1986); they represent a recognized way to introduce flexibility, variation and tacit knowledge in the process. While practitioners can in principle work around computer-embedded rules and assumptions, however, this is not always feasible. First, often actions and process steps are triggered automatically; as they often become 'invisible' to the eyes of the engineers, processes embedded into the software are often followed without questioning. Second, software embodies controls and switches that cannot be easily bypassed. Scholars have named those properties of software that inhibit customization as the 'power of default' (Koch, 1998).²⁸

This case study has therefore shed some new light over the role of software in storing procedural memory, as well as over the process of embedding organizational routines and rules in software. The evidence presented has shown that the introduction of software restructures existing processes and procedures thus radically affecting the way formal and informal knowledge flow into the product design process. In contrast with the management literature's often optimistic characterization of flexible software technologies, this example has shown how their presumed benefits, such as increased flexibility and ease in communication, may only partially materialize in practice. The case study has also illustrated how routines are changed in the process of software

²⁷For work about how assumptions are embedded in technology, see also Pickering (1995).

²⁸Even visible rules are often not worked around or questioned as practitioners tend to follow the 'path of least resistance'. For example, an engineer would have to modify the software to circumvent the rule. Not many engineers, however, have the skills required to reprogram such a complex software tool, or indeed have the time or the motivation to do so.

implementation; we have seen that this does not simply involve straightforward codification of routines but re-engineering of the way knowledge streams flow into the process. We have seen that delegating procedural memory to software can induce automatic behaviour that can in turn prevent the exchange of informal knowledge among designers working on the same development programme. We have shown that the assumptions and rules to which software-embedded routines are subjected can induce conservative behaviour. This can hold important consequences for the extent of innovative potential of both the design workflow process and its outcomes.²⁹

4. Conclusions

This paper has shown that delegating memory to software has important implications for the organization. Our evidence has demonstrated that software fundamentally reconfigures the very mechanisms by which organizational knowledge is structured, stored, retrieved and reused. We have argued that, in the attempt to impose a new, common, 'language', software generates a push towards greater standardization and reduction of technological and organizational heterogeneity (i.e. local, idiosyncratic 'dialects'). We have shown how, while the newly introduced (standardizing) 'language' and routines were aimed at reducing the duplication of efforts and at improving co-ordination by eliminating inconsistencies of data and actions across the organization, they clashed with existing organizational heterogeneities; these took the form of various 'epistemic communities' and 'communities of practice', each having their own knowledge base, culture, objectives, incentives, and discipline- or activity-specific languages.

In our example of the two (incompatible) product structures, we have argued that the attempt to standardize across such heterogeneous organizational domains has paradoxically emphasized those existing inconsistencies and differences in knowledge bases and cognitive structures across functions. We have shown that, while standardization may eliminate some technological and organizational bottlenecks as some economists have argued (cf. Cowan and Foray, 1997), it also tends to create new ones. Our example can indeed be conceptualized as a failure to impose a common, artificial language, due to the persistence of local 'dialects' (i.e. existing database structures, technologies and routines). In other words, it can be argued that the software-embedded product and database structure have failed to perform as a 'boundary object' (Bowker and Star, 1999): the new structure does not in fact possess the interpretive flexibility required to support collaboration among heterogeneous functions; rather, its behaviour is more similar to that of an inflexible 'standardizing device'.

Achieving shared meanings in a heterogeneous organizational setting therefore requires more than the mere co-ordination of information flows advocated by some economists: it requires the integration of (often) incompatible meaning structures. The

²⁹D'Adderio (2003) provides additional examples of how software can play an important influence over an organization's flexibility, innovative and adaptive potential.

heterogeneous groups and functions that compose an organization significantly differ in their ability to learn, interpret, know and memorize. These inconsistencies are often heightened by the introduction of software, which tends to disintegrate existing organizational patterns while challenging the stability of existing routines.

This case study has shown therefore that it is necessary to 'unpack' the concept of *organizational culture* that characterizes much organization and innovation theory; organizational knowledge bases, language, culture, objectives and incentives, rather than being static and homogeneous, are instead the result of a *continuously emergent* equilibrium among conflicting and incompatible elements. A similar point was raised by Brown and Duguid, who advocated the need for studies that account for the divisions created by 'communities of practice' and 'epistemic communities', which produce noticeable 'local' variations in the organization's landscape in terms of knowledge, language and culture (cf. Brown and Duguid, 2000).³⁰ Similarly, organizational 'shared meanings' are not given but require instead to be *continually reconstituted*: they are not the premise but rather the result of (and a measure of) techno-organizational integration and coherence. They are emergent, rather than fixed, properties. This leads us to a fundamental paradox for the organization: the need to exploit heterogeneity (of knowledge and practices)—therefore fully exploiting the advantages of specialization—while at the same time reaping the advantages of software-induced standardization. Heterogeneity must (and indeed does) remain, but needs to be co-ordinated.

In the second case study we have examined the implications of delegating procedural memory to software. We have argued that software performs as a 'dual enabler', providing guidance and constraints that allow and support some actions while constraining or obstructing others. Further, due to the tendency of software-embedded rules and routines to sink in and become invisible to the eyes of the decision-maker, circumstances are created where active interpretation is replaced by passive, automatic behaviour. We have concluded that, while software introduction can support the synchronization of efforts and economy of actions, therefore, it can at the same time entail a risk of lock-in and reduced flexibility.

Our example has shown how rigid software-embedded rules (i.e. workflow status) and assumptions (i.e. definition of roles and authority to change) can obstruct informal actions and flexible behaviour. Specifically, we have observed how the strong emphasis on administrative process control introduced by software can reduce the scope for interaction among designers and the opportunity to obtain feedback from other functions. Inflexible software-stored parameters and procedures can also reduce the scope for exploration of alternative product configurations, causing an early configuration freeze.

³⁰This need for studies that characterize the nature and workings of such communities have been recently reflected in an upsurge of contributions in sociology, organization science, economics and management theory (cf. Lave and Wenger, 1990; Cowan *et al.*, 2000; Knorr-Cetina, 2000; Steinmueller, 2000; Cohendet and Meyer-Krahmer, 2001). This paper provides evidence that contributes to deepening this debate.

Similarly to our first example, this has led to the conclusion that the way knowledge is configured while being embedded in software (i.e. according to specific software-embedded assumptions) can, at different times, support conservative or promote innovative behaviour. Further rigidities can emerge as a consequence of the often unquestioned, repetitive behaviour that follows the conversion of routines as 'expressions' into computer-embedded 'representations' that are replicated throughout the organization.

These results have highlighted the need to add to the evolutionary economics' characterization of routines by exploring the *unstable elements* that challenge existing knowledge bases and alter the patterns of routines.³¹ Our evidence has illustrated an example of a failure to re-establish the informal patterns of communication broken up by software implementation and emphasized the need to support the emergence of new patterns by pursuing a new equilibrium between standardized, common, and flexible, function- or activity-specific knowledge and routines. Studying the evolution of routines that follows software introduction has therefore provided important new insights into the mechanics of knowledge creation and reproduction at the level of the organization.

Acknowledgements

A special thanks goes to all my colleagues at SPRU, and in particular to Keith Pavitt and Ed Steinmueller who provided invaluable comments on early drafts. I would also like to thank Patrick Cohendet at BETA (Strasbourg), the colleagues at Edinburgh University and the anonymous referees for their comments on later and final drafts. The usual disclaimers apply. I would like to acknowledge the ESRC Innovation Programme that sponsored the research throughout my time at SPRU (Sussex University). I am also grateful to the EPSRC 'Systems Patterns' and DIRC Programmes that supported the final stages. A different version of the paper will appear in L. D'Adderio, *Software Systems and the Creation of Knowledge in Organisations* (Edward Elgar: Cheltenham, forthcoming).

Address for correspondence

Research Centre for Social Sciences and Human Communications Research Centre,

³¹While providing invaluable insights into the nature and role of organizational routines and capabilities, evolutionary economics has so far essentially focused on elements that operate towards preserving stability, such as processes of knowledge accumulation and the persistence of continuity due to path dependency (cf. Cohen and Levinthal, 1990). So far relatively little attention has been paid to how routines change, at times break up, or even subside. These mechanisms lie at the basis of the formation of capabilities. The need to further our understanding of the dynamic aspects of organizational knowledge creation and reproduction and capabilities formation is highlighted in several contributions including Dosi *et al.* (2000) and Cohen *et al.* (1996).

The University of Edinburgh, 2 Buccleuch Place, Edinburgh EH8 9LW, UK. Email: l.d-adderio@ed.ac.uk.

References

- Anderson, J. R. (1983), *The Architecture of Cognition*. Harvard University Press: Cambridge, MA.
- Aoki, M. (1986), 'Horizontal vs. vertical information structure of the firm,' *American Economic Review*, **76**, 971–983.
- Bannon, L. J. and K. Kuutti (1996), 'Shifting perspectives on organizational memory: from storage to active remembering,' *Proc. IEEE HICSS 96*, Hawaii.
- Bowker, G. C. and S. L. Star (1999), *Sorting Things Out: Classification and its Consequences*. MIT Press: Cambridge, MA.
- Brown, J. S. and P. Duguid (2000), *The Social Life of Information*. Harvard Business School Press: Cambridge, MA.
- Cohen, M. D. (1991), 'Individual learning and organizational routines: Emerging connections,' *Organization Science*, **2**, 135–139.
- Cohen, W. M. and D. A. Levinthal (1990), 'Absorptive capacity: a new perspective on learning and innovation,' *Administrative Science Quarterly*, **35**, 128–152.
- Cohen, M. D. and L. Sproull (1991), *Organizational Learning*. Sage, CA: Thousand Oaks.
- Cohen, M. D., R. Burkhart, G. Dosi, M. Egidi, L. Marengo, M. Warglien and S. Winter (1996), 'Routines and other recurring patterns of organisations: contemporary research issues,' IIASA working paper, March.
- Cohendet, P. and F. Meyer-Krahmer (2001), 'The theoretical and policy implications of knowledge codification,' *Research Policy*, **30**, 1563–1591.
- Coombs, R. and R. Hull (1997), 'Knowledge management practices and path-dependency in innovation,' CRIC discussion paper 2, University of Manchester.
- Cowan, R. and D. Foray (1997), 'The economics of codification and the diffusion of knowledge,' *Industrial and Corporate Change*, **6**, 595–622.
- Cowan, R., P. A. David and D. Foray (2000), 'The explicit economics of knowledge codification and tacitness,' *Industrial and Corporate Change*, **9**, 212–253.
- D'Adderio, L. (2000), 'The diffusion of integrated software solutions—trends and challenges,' paper presented at the 3rd TSER/ESSY meeting, Berlin, 3–7 April.
- D'Adderio, L. (2001), 'Crafting the virtual prototype: how firms integrate knowledge and capabilities across organisational boundaries,' *Research Policy*, **30**, 1409–1424.
- D'Adderio, L. (2003). *Software Systems and the Creation of Knowledge in Organisations*. Edward Elgar: Cheltenham, forthcoming.
- Dosi, G., R. R. Nelson and S. G. Winter (eds) (2000), *The Nature and Dynamics of Organisational Capabilities*. Oxford University Press: New York.
- Eisenhardt, K. M. (1989), 'Building theories from case study research,' *Academy of Management Review*, **14**, 532–550.
- Foray, D. and W. E. Steinmueller (2001), 'Replication of routine, the domestication of tacit

- knowledge and the economics of inscription technology,' Conference in Honour of R. R. Nelson and S. G. Winter, Ålborg, Denmark, 12–15 June 2001.
- Gasser, L. (1986), 'The integration of computing and routine work,' *ACM Transactions on Office Information Systems*, **4**, 257–270.
- Glaser, B. and A. Strauss (1967), *The Discovery of Grounded Theory: Strategies of Qualitative Research*. Weidenfeld & Nicholson: London.
- Holland, J. H. (1992), *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press: Cambridge, MA.
- Knorr-Cetina, K. (1999), *Epistemic Cultures: How the Sciences Make Knowledge*. Harvard University Press: Cambridge, MA.
- Koch, C. (1998), 'SAP/R3—an IT plague or the answer to Taylor's dream?,' Institute for Technology and Social Science, Technical University of Denmark, Lyngby.
- Kogut, B. and U. Zander (1992), 'Knowledge of the firm, combinative capabilities, and the replication of technology,' *Organization Science*, **3**, 383–397.
- Lave, J. and E. Wenger (1990), *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press: Cambridge.
- Leonard-Barton, D. (1992), 'Core capabilities and core rigidities: a paradox in managing new product development,' *Strategic Management Journal*, **13**, 111–125.
- Leonard-Barton, D. (1995), *Wellsprings of Knowledge: Building and Sustaining the Sources of Innovation*. Harvard Business School Press: Cambridge, MA.
- Levitt, B. and J. G. March (1988), 'Organizational learning,' *Annual Review of Sociology*, **14**.
- March, J. G. and H. A. Simon (1958), *Organizations*. John Wiley: New York.
- Moorman, C. and A. S. Miner (1998), 'Organizational improvisation and organisational memory,' *Academy of Management Review*, **23**, 698–723.
- Nelson, R. R. and S. G. Winter (1982), *An Evolutionary Theory of Economic Change*. Belknap Press: Cambridge, MA.
- Nonaka, I. (1994), 'A dynamic theory of organizational knowledge creation,' *Organization Science*, **5**, 14–37.
- Orlikowski, W. J. (1992), 'The duality of technology: rethinking the concept of technology in organisations,' *Organization Science*, **3**, 398–427.
- Pavitt, K. (1998), 'Technologies, products and organisation in the innovating firm: what Adam Smith tells us and Joseph Schumpeter doesn't,' *Industrial and Corporate Change*, **7**, 433–451.
- Pickering, A. (1995), *The Mangle of Practice: Time, Agency and Science*. University of Chicago Press: Chicago, IL.
- Polanyi, M. (1968), *Personal Knowledge: Towards a Post-critical Philosophy*. University of Chicago Press: Chicago, IL.
- Sanchez, R. and J. Mahoney (1996), 'Modularity, flexibility, and knowledge management in product and organizational design,' *Strategic Management Journal*, **17**, 63–76.
- Spender, J. C. (1995), 'Organizational knowledge, learning, and memory: three concepts in search of a theory,' Rutgers University, Newark, NJ, revised version for *Organizational Change Management*.

- Steinmueller, W. E. (2000), 'Does information and communication technology facilitate "codification" of knowledge?', *Industrial and Corporate Change*, **3**, 47–64.
- Vincenti, W. G. (1990), *What Engineers Know and How They Know It—Analytical Studies from Aeronautical History*. The John Hopkins University Press: Baltimore, MD.
- von Hippel E. (1994), "'Sticky information" and the locus of problem solving: implications for innovation,' *Management Science*, **40**, 429–439.
- Walsh, J. P. (1995), 'Managerial and organizational cognition: notes from a trip down memory lane,' *Organization Science*, **6**, 280–321.
- Walsh, J. P. and G. R. Ungson (1991), 'Organisational memory,' *Academy of Management Review*, **16**, 57–91.
- Weick, K. (1979), *The Social Psychology of Organizing*, 2nd edn. Addison-Wesley: Reading, MA.
- Wildawsky, A. (1983), 'Information as an organizational problem,' *Journal of Management Studies*, **20**, 29–40.
- Winter S. G. (1987), 'Knowledge and competence as strategic assets,' in D. J. Teece (ed.), *The Competitive Challenge: Strategies for Industrial Innovation and Renewal*. Harper & Row: New York.

Internal company documents

An Integrated Product Development Strategy (IPDS).

'Digital Mock-up': Scope of Work (DMU).

The 'Workflow Process' (WIP).