Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

T. Murer, D. Scherer

Organizational Integrity: Facing the Challenge of the Global Software Process

TIK-Report
No. 51, July 1998

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

T. Murer, D. Scherer
Organizational Integrity: Facing the Challenge of the Global Software Process
July 1998
TIK-Report No. 51

Computer Engineering and Networks Laboratory,
Swiss Federal Institute of Technology (ETH) Zurich

Institut für Technische Informatik und Kommunikationsnetze,
Eidgenössische Technische Hochschule Zürich

Gloriastrasse 35, ETH-Zentrum, CH-8092 Zürich, Switzerland

ABSTRACT
Emerging technologies influence the way software is produced and deployed. Software component developers build virtual organizations to produce properly interoperating components by connecting their software processes to a “global software process”. Its support challenges the traditional concepts of software engineering environments and requires organizational integrity to become the key design issue. This encompasses the common understanding of structure, state and evolution of three structures: product, process and organization forms, shared among all process participants: developers and tools. We propose a process modeling language that achieves organizational integrity through structural unity: by managing a unified simple model for the three structures.

Keywords
software process modeling language, integrated process-centered software engineering environments, organizational integrity, virtual organizations

1 INTRODUCTION
The goal of a software process is the development or maintenance of a software product. This process takes place in a complex, distributed and heterogeneous world of actors and resources that need to be integrated and coordinated, taking into account technical, organizational and social aspects. A PSEE (Process-Centered Software Engineering Environment) supports the definition and enactment of a software process and is essential for processes of a certain complexity and quality. Emerging technologies such as software components and the Internet challenge the traditional concepts of a PSEE and will change the way software is produced, marketed and deployed. Component developers need to cooperate tightly and share their software processes to produce properly interoperating components. This cooperation is based on a network of autonomous organizations (having various cultures, methods and strategies) recognized as one homogeneous virtual organization from an external viewpoint. As a vision, the software development processes of many individual component producers together may be regarded as one large *global software process* consisting of many linked processes. Therefore, the requirements for future PSEEs must emphasize more the support among organizations than within individual organizations, which should preserve their autonomy. To provide both a common understanding of organization and a base for communication, we believe that *organizational integrity* is the most important design issue for a PSEE. It should involve structure, state and evolution of product, process and organization forms, shared among all process participants. We achieve this through structural integrity, using a common simple model to represent the three forms, reflected in our proposed *process modeling language* which is the main focus of this paper, and is implemented in a PSEE framework prototype.

1.1 Paper Overview
After motivating for organizational integrity, we present our proposed software process modeling language reflecting organizational integrity, followed by support for virtual organizations and human resources; finally, further benefits and conclusions are discussed.

2 FACING THE CHALLENGE
We claim that developing high quality interoperating software components in a global context involving different autonomous organizations is as much an organizational problem as it is a technical one [16]. In other engineering disciplines (e.g. car manufacturing) where components play a key role and complex networks of enterprises cooperate tightly on a high level, organizations appear as one unitary organization from an external observer’s viewpoint and make up a so-called *virtual organization* [5]. In analogy, different software component producers need to share and link their development processes to result in tightly cooperating virtual organizations, and as a vision, in one large global software process consisting of many individual linked processes. While its management includes technical aspects, the more difficult challenge is posed by the organizational aspects such as decentralized process control and management, and the integration of various concepts, cultures, methods and strategies. A related challenge is global configuration management of software components. The decentralized management of products, processes and organization forms within a virtual organization, and the integration of a collection of heterogeneous, widely distributed and loosely coupled organizations provide a great *challenge* to future PSEEs.

2.1 Organizational Structures
Communication, and in consequence, organization, are the critical aspects in facing the challenge ([2], chapter 7), and not technical limitations. Therefore, different organizations within a virtual organization must share a common understanding of organization. We believe the organizational aspects of the product, the process and the organization form to be the three main organizational structures in a software project. The structure and evolution of the software product represents the project’s goal and includes product parts, dependencies, hierarchies, versions and configurations. The structure and evolution of the software process describes the route to the goal including process steps, dependencies, hierarchies and evolution. The structure of the organization form includes the functional tasks of the people involved and the hierarchic controlling and reporting relations among them.

2.2 Organizational Integrity
Conceptual integrity is the most important consideration in system design ([2], chapter 4). Similarly, organizational

integrity is of vital importance and critical to a software project, particularly if performed within a heterogeneous virtual organization. It covers on the one hand the common understanding of structure, state and evolution of product, process and organization form, shared among all project participants including people and tools, based on a common model. On the other hand, it answers the question "Which part of the product is the result of which process part performed by which person?", i.e. it covers the relation between elements of the three different structures in the past, the present and the future. Uncertainty and the requirement for multiple revisions and iterations are inherent in software projects [19]. Thus, providing powerful support to manage continuous change is inevitable, and it is also important for continuous project management estimation and assignment tasks. It requires a shared understanding of structure, state and evolution of the three organizational structures and the relationships of their parts.

2.3 From Structural Variety... To Structural Unity

The three organizational structures are often investigated by different research communities focusing on their specific aspect of organization, leading to specialized methods and structures, e.g. processes are modeled with petri-nets, configuration management uses version trees, organizational structures of a company or team use matrices to keep track of developers' skills, roles and interactions. Achieving organizational integrity is therefore not trivial. The difficulty is inherent in the problem of matching different structure types such as trees, graphs and matrices on each other. As in conceptual integrity in system design [2], where Brooks proposes a single architect, we propose to use similar structures to manage all three organizational forms [15]. Whereas structural unity achieves organizational integrity rather easily, it has limitations: each structure has to sacrifice some of its specific characteristics, resulting in a simple shared structure. We claim that gaining organizational integrity by sacrificing sophisticated but specific structures is the best choice in facing the challenge.

2.4 Revised Requirements for a PSEE

While the basic requirements for PSEEs have been studied extensively [10], we believe that the outlined challenge demands these requirements to be modified. In order to support virtual organizations consisting of a network of heterogeneous and fairly autonomous nodes, a PSEE should connect and integrate these nodes with an emphasis on the organizational aspects of support, serving primarily as a base for communication about organization. In addition to conventional requirements such as distributed process definition and enactment, support for process improvement, etc., it should emphasize simplicity and organizational integrity, providing consistent views of organization forms.

We achieve organizational integrity through structural unity. A goal was to find a modeling technique to enable the three main organizational aspects to be represented with a similar structure, suitable for application in a PSEE; this paper focuses on the process aspect of the resulting process modeling language, with [15] detailing other aspects.

2.5 Project GIPSY

The aim of our GIPSY (**G**enerating **I**ntegrated **P**rocess support **SY**stems) project is a PSEE framework supporting organizational integrity and allowing the distributed definition and enactment of concrete process models. The GIPSY project targets both support for component development in virtual organizations and support for developing reusable components. It is divided into two subprojects, each focusing on a different layer: The DIPS (**D**istributed **I**ntegrated **P**rocess **S**ervices) project focuses on the *organization layer* with the objective of achieving organizational integration; this paper covers its proposed process modeling language. Additionally, the CHIPS (**C**omponents for **H**ighly **I**ntegrated **P**rocess **S**upport) project focuses on an *artifact layer* which supports formal methods, since such methods are essential for component reuse (e.g. design by contract) [13]. While not the main focus of this paper, it allows specification and generation of highly integrated tool components for text-based formal languages and aims at high-level semantic data integration in PSEE tools, which is essential in preventing tools from misinterpreting data. Compiler-compiler technology is extended by the object-oriented paradigm to a tool specification technique using extensible attribute grammars to provide powerful decomposition, extension and import mechanisms. The artifact layer complements the organization layer by optionally allowing process semantics to be defined at a finer granularity where required and possible [14]. DIPS provides a process-invariant PSEE framework implementing our graphical process modeling language, whereas CHIPS provides process-specific tools. CHIPS-generated or external tools are plugged into the PSEE framework to provide a process-specific environment.

3 PROCESS MODELING LANGUAGE

3.1 Terminology

A *tool* consists of a data model and one or more views. High data integration requires tools to be strictly separated into models and views. A model holds all data semantics including all valid operations. A view only provides a user interface by representing the data model and its operations.

An *artifact* is an instance of the model part of a tool. Thus, artifacts are (instances of) abstract data types holding data and offering a set of valid operations. An artifact can have properties and can be related to other artifacts representing the artifact's context, and communicates with messages.

A *software product* consists of all data (contract, specification, implementation, documentation, code, test case, manual, etc.) which are produced during its development and maintenance, and these can be represented as a partially ordered set of artifacts. Every artifact carries the information of a part of the software product and typically depends on other artifacts, thus embodying both product and process [6]. These relations define the ordering of the set. Artifacts can exist in different versions and can be part of an enclosing artifact. The vision is that a whole software product can be understood as one compound artifact, similar to today's distributed compound document architectures [17], but extended by dependencies among the versioned parts; the product structure need not reflect the system structure. A software product contains all artifacts, not only those delivered to a customer.

A *process* is the dynamic view of an artifact and defines how an artifact is created, edited and confirmed. A process can be a part of another process, reflecting the hierarchic relationship of the corresponding artifacts. Thus, the all-enclosing process defines the life-cycle of the all-enclosing artifact representing the software product. This hierarchically top process, also termed whole process, reflects the conventional meaning of a software process [7]. We use the same term process for every hierarchic level of a process resp. every process fragment and compositions thereof to emphasize the fact that the same behavioral rules apply in every case.

3.2 Language Objectives

Our goal is to achieve organizational integrity based on the structural unity of the three main organizational structures. A technique to model structure and evolution of software product, process and organization forms is required, with simplicity as the dominant design aspect. This section focuses on the process aspects of organization and outlines the process modeling language, as implemented in our PSEE framework's process engine. In order to satisfy basic PSEE requirements such as process definition and enactment support, the language should allow definition of process steps as the elementary process components and their relations including partial ordering and hierarchy [4]. The language should also provide composition and enactment rules and enable process evolution.

A software process can be understood as a transformation of resources into a product, performed in a complex socio-technical system. The software product is the goal of a software process (note also the emphasis on goal in Feiler's/Humphrey's definition of a software process [7]). Thus, a *goal-oriented approach* is an obvious approach to design a process modeling language. Our understanding of a software product perceived as a compound artifact dominates the process language. When a product is (initially vaguely) planned, its structure, including its parts, their dependencies and hierarchies is defined (process definition). Thereafter, striving for the goal means filling up the planned product structure with concrete product parts, developed in distinct process steps (process enactment). The process is the *dynamic view* of a product, representing the planned structure of a product, i.e. the product structure implies the process structure; using the same structure for product and process illustrates goal-orientation. Permanent uncertainty about the final structure of the product [19] drives continuous revisions of product and process structure requiring support for process iteration and evolution.

3.3 Language Overview

We use a directed acyclic graph (DAG) to model processes. Its colored nodes represent process steps and their states, whereas arrows represent their dependencies, resulting in a 2-dimensional DAG structure (representing both dependencies and concurrent development). Process definition means structuring a process (creation and evolution), whereas process enactment means performing a process with a given structure; this is described first.

The process modeling language consists of process and dependency elements, and of definition and enactment rules. A process is always in one of two modes: in *enactable* mode, enactment operations may be performed, while in *definable* mode, definition operations may be performed. Operations are grouped in *transactions* consisting of one or more definition operations or enactment operations, and which define the states to be recorded.

3.4 Process State Space

One process is related to each part of the planned product. A process can be part of an enclosing compound process. This hierarchy relation reflects the structure of the planned product, understood as a compound artifact. However, atomic and compound processes will only be distinguished in the later section on process hierarchy. Meanwhile, only atomic processes representing the life-cycle of a single corresponding artifact are assumed.

3.4.1 Artifact Life-Cycle and Process States

For every process there is a defined state sequence reflecting the life-cycle of its corresponding artifact and the process' dynamic nature (fig. 1): planned, editable, confirmed. The transition from the initial state planned to the state editable includes the successful creation of the artifact, which is then assigned to the process. The process remains in state editable during development of the artifact and reaches its final state confirmed after a successful confirmation (detailed below). The artifact's development represents the creative work, whereas its creation and confirmation are performed by the process engine. E.g., for a source code artifact, most of the effort goes into its development, whereas for an object code artifact there is no additional development after its automatic creation (compilation).

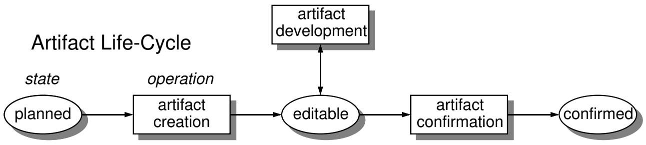

Fig. 1: Atomic process state transitions

Processes are represented by squares whose coloring indicates the process state (fig. 2); workflow is thus represented by progressing coloring of a process, while artifacts always remain assigned to the same process (unlike e.g. tokens flowing in a petri-net).

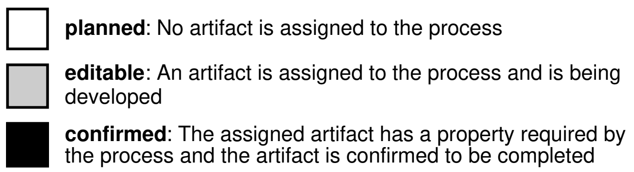

Fig. 2: Atomic process states

3.4.2 Process Context

Each process has a set of other (predecessor) processes it directly depends on: its process context. The process context is accessed during the creation, development and confirmation of an artifact. Each dependency between planned product parts is reflected by the corresponding dependency between processes (e.g. object code depends on the related source code, an implementation on the related specification). Dependencies are represented by arrows. All processes and their process contexts make up a directed graph. There is a distinction between the direct dependencies of a process to its process context and transitive dependencies defined by the transitivity of process contexts. The directed graph needs to be acyclic, thus its transitive hull defines a partial ordering among the processes.

A given set of colored processes and dependencies defines a finite state space (e.g. all processes in state planned represent the initial state and all processes in state confirmed represent the accepting state). A state transition results in a different coloring of the processes. The rules for state transitions are covered in the next section.

3.5 Process State Transitions

The initial state of a process is planned; no artifact is assigned. State transition rules and transitions of a process with respect to its process context are explained next.

3.5.1 Artifact Creation

The initial creation of an artifact is defined by a creating process and a creation function associated with each process.

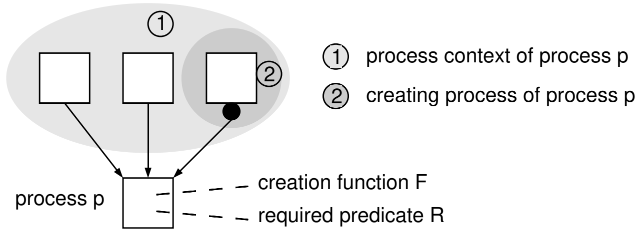

Fig. 3: Process elements

Referring to fig. 3, the creating process (2) of a process p is responsible for the initial creation of the artifact to be assigned to p. The creating process (2) of process p is an element of its process context (1); the respective dependency is marked by a filled circle. The creation function requires the creating process as an input and returns an artifact that is assigned to the process (p). E.g., the creation of an object code artifact may be defined by the creating process holding the related source code artifact and the creation function describing the compilation. The creating process may be absent, in which case a built-in creating process can be used (e.g. to create an empty template of a given artifact type).

The precondition for artifact creation at atomic process p is: (state(p) = planned) and (state(creatingprocess(p)) = confirmed). A successfully performed creation assigns the new artifact to process p and changes the state to editable.

3.5.2 Artifact Development

After its creation, an artifact is developed. In this highly creative phase, the process engine cannot provide much support, since it performs the non-creative formal tasks.

3.5.3 Artifact Confirmation

The transition of a process into its final state confirmed is defined by a required predicate and a confirmation signature that are associated with each process. The required predicate defines a property that the assigned artifact must hold to be valid for confirmation. The required predicate takes the process context as an input and decides if the assigned artifact holds the required property. The required predicate is implemented as a message understood and evaluated by the assigned artifact. After successful evaluation of the predicate, (electronic) signing by the responsible person completes the process' confirmation, a step representing human control which should not be automated or omitted. A required predicate is a powerful concept for artifacts based on formal languages. For other artifacts, where no formal predicate is defined, the confirmation signature is more significant. The responsible person confirms the required predicate with respect to the related process context.

The precondition for artifact confirmation at atomic process p is: (state(p) = editable) and (for all processes q in context(p): state(q) = confirmed). A successful confirmation triggers a state transition to state confirmed.

3.6 Process History

Due to the dynamic nature of software processes, its organization should be planned for change. The revision of confirmed processes or the redesign of the process structure are changes that occur frequently. Continuous process improvement and reuse is based on access to the process history. Thus, older process versions need to be recorded carefully. A powerful process engine must explicitly support changes and the recording of process versions. We distinguish two aspects of changes that are covered in the following sections: *process iteration* describes the invalidation and revision of confirmed processes and how these changes are recorded, whereas *process evolution* describes how the process structure can be modified and how these changes are recorded. While process iteration focuses on state transitions in an unchanged state space and occurs in enactable mode, process evolution deals with the evolution of the state space and occurs in definable mode.

To record process versions, the process language is extended to handle a third dimension: the history dimension (fig. 4). Every time a process is to be revised, it is copied, and the copy is reset to the initial state planned. The original version moves down the history dimension becoming an older process version, whereas changes are performed on the copy, which represents the *current* process version. Process history is the result of a sequence of structure modifying transactions. All states between such transactions are recorded in the history dimension.

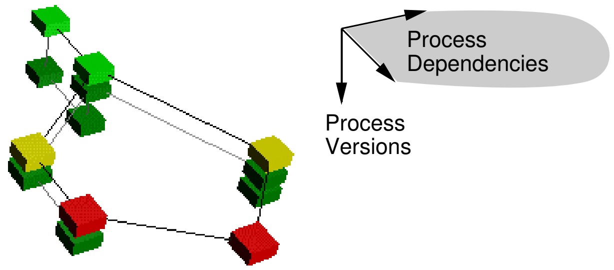

Fig. 4: Process dimensions

3.7 Process Iteration

A process revision triggering a new process iteration is called rollback. A rollback is activated on a set of processes. Precondition for a rollback on a set R of processes: for all p in R: current(p) and enactable(p) and (state(p) ≠ planned). Due to the transitivity of the dependency relation, the revision of a process has an effect on all transitively dependent processes: these need to be considered for revision (invalidation) as well. With the successor predicate succ(p, q) signifying a direct dependency of process q on process p (arrow from p to q), and $succ^+(p, q)$ signifying a transitive dependency, the processes concerned by a rollback on a set R of processes consists of the set of q: (for all p in R: (all q where ($succ^+(p, q)$ and state(q) ≠ planned) and (for all r in ($succ^+(p, r)$ and $succ^+(r, q)$) where state(r) ≠ planned))). Thus, the transitively concerned set of processes is a contiguous set of processes not in state planned.

In order to perform a rollback, the whole set of concerned processes including their dependencies are copied. The original process versions including their dependencies and their assigned artifacts move down the history dimension and become the second newest versions. The copies become the newest process versions and change to the initial state planned. A version relation between the older original process version and its corresponding copy is established. The new history layer contains all moved processes. Four time stamps are recorded for every process: initial creation, state change to editable, state change to confirmed, and when it becomes an older version.

3.8 Process Evolution

The previous sections covered process enactment including artifact creation, development and confirmation, as well as process rollbacks. All these operations are provided on a given state space and do not change the process (except for the history dimension). Additional operations are required to define and redefine processes and to record this evolution. After definition, processes can be enacted immediately (executed by interpretation), i.e. there is neither a separate process coding language, nor is compilation or explicit instantiation of process definitions required.

3.8.1 Process Components

The process modeling language defines process and dependency elements, and additionally requires creation function and required predicate elements (which are defined in the artifact layer). All these elements can be used to build larger processes by satisfying process definition rules. A creation function and a required predicate are assigned to a process to define its state transitions. A process needs to provide a defined process context in order to evaluate the associated function and predicate. Therefore, creation function and required predicate need to match with the process context (e.g. a compiler as a creation function requires source code in its process context). Thus, a process with its creation function, its required predicate and its process context make up a *process component* (fig. 5). A set of process components can be assembled to form a larger process. Every process can be an element of the process context of its succeeding processes. Thus, to fit together two process components, the process of one component is matched onto the process context of the other component. If valid, the resulting composition forms a larger process, which can in turn form part of another composition. The resulting process graph must be acyclic.

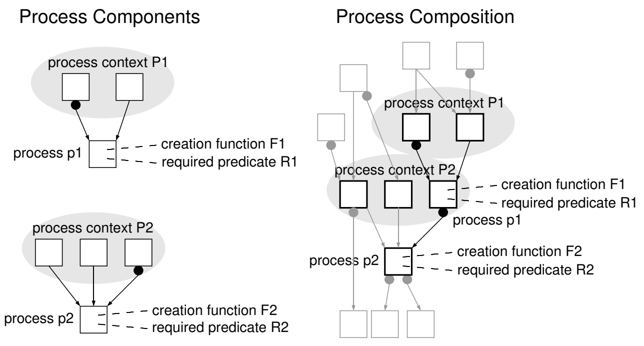

Fig. 5: Process composition

3.8.2 *Giving the Arrows a Meaning*

An arrow between two processes represents a dependency. The creation function associated with a process defines how the artifact is created with respect to the process context. The required predicate associated with a process defines the property that the artifact needs to hold, evaluated with respect to the process context. Thus, creation function and required predicate define the dependency and give the arrow specific semantics. While the organization layer defining the process modeling language is a framework which manages process dependencies, it does not define the semantics of the dependencies. The artifact layer aims at the definition and implementation of tool components used to define the semantics of messages which implement creation functions and required predicates. For some processes, this may be a complex task, but it is largely process-specific. E.g., for a real-time system, a formal required predicate based on tools we implemented for simple formal languages statically checks execution times of primitive-recursive functions against their maximum times formally specified in a preceding artifact. Dependencies with powerful semantics are limited to processes holding artifacts based on formal languages. In more informal parts of a process, dependencies' semantics are explained informally and validated through confirmation by the responsible person.

3.8.3 *Recording Process Evolution*

As described above, a process can be either in mode enactable, allowing for its state transition, or in mode definable, where state transitions are not permitted. A new process is initially in mode definable. A process can only change from definable to enactable if it satisfies the outlined composition rules. It is in state planned after changing to mode enactable. Processes can be redefined. To do so, a set of processes can be reserved for evolution. Every process within the reserved set must be in state planned (if not, a rollback first needs be performed which will reset (copies of) the processes to state planned). The reserved set is a part of the whole process and keeps its ordering: if processes p, q in reserved set and $succ^+(p, q)$, then for all r: if $succ^+(p, r)$ and $succ^+(r, q)$, then r in reserved set.

All processes within the reserved set including their dependencies are copied. The original process versions including their dependencies move down the history dimension, becoming the second newest versions. The copies become the newest process versions and are ready to be changed. A version relation between the older original process version and its corresponding copy is established. In the process history, iteration and evolution can be distinguished easily, since history of iteration never involves planned processes, whereas history of evolution exclusively involves planned processes. After evolution, all processes within the reserved set have to change to mode enactable together, and the reserved set is released for enactment, which requires all processes to satisfy the composition rules. Otherwise, the evolution is canceled and the state before the evolution started is reestablished.

Evolution is simpler where no existing process state space is affected: new processes can be appended to any process that is in state confirmed, whether it is the current version or not (while the appended processes become current).

3.9 Process Hierarchy

To cope with complexity in typical large processes, an hierarchical abstraction mechanism is essential. The process language is extended by an expressive hierarchy mechanism. A process can be part of one enclosing compound process. This tree-type transitive hierarchy relation reflects the structure of the planned product. Expressiveness is achieved since on the next higher level of hierarchy, the behavior of a compound process is similar to that of an atomic process. Instead of an artifact, a compound process holds a set of processes (atomic or compound).

A compound process can be in the same three states as an atomic process. Its state is defined by the state of its directly contained processes (fig. 6). A compound process is affected by a rollback and recorded with the affected atomic processes in the history dimension if the rollback changes its state back to a previous one. A compound process is affected by process evolution and recorded in the history dimension with the reserved atomic processes if it is within the process set reserved for evolution. An enactable compound process always contains another process.

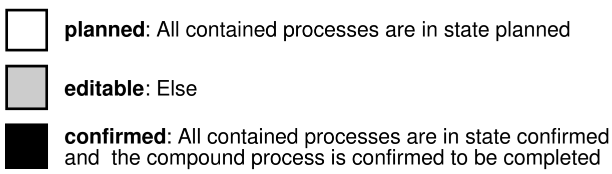

Fig. 6: Compound process states

The dependencies of a compound process are defined by the dependencies of the contained processes. If a directly contained process depends on a process that is not directly contained, the compound process also depends on this process. If a directly contained process is the dependent of a process that is not directly contained, the compound process is also a dependent of it.

3.10 Prototype

A distributed prototype of a PSEE framework containing a process engine that supports the presented process modeling language has been implemented [20], together with some tools for formal languages. It has been used successfully in student exercises of lectures at ETH Zurich, as in the example screenshot (fig. 7). It shows a 3D view (left) and 2D view (middle) of a small process consisting of 6 hierarchically top-level processes (2 of which are compound, represented by larger squares), each of which have up to 3 older versions (represented in the process history, 3rd dimension); the other 3D view (right) shows a zoom-in on one of the compound processes (clicking on processes reveals their contained processes resp. their artifacts). The whole process represents a student exercise: following an informal specification of a formal language, in 2 compound processes a corresponding grammar should be written and its compiler generated using a compiler-compiler tool (illustrating employment of formal methods to create an artifact out of a previous one), and in a concurrent atomic process test sentences are to be defined; the top process shows the split into two concurrent tracks after the specification, and the joining test process.

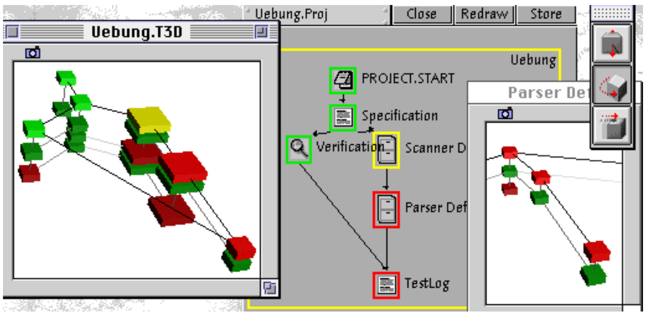

Fig. 7: Screenshot of a process showing 2D and 3D views

We have also experimentally applied the PSEE to manage its own process in a bootstrapping manner. This has provided us with valuable insights used to refine the process modeling language in an evolutionary manner, and so far indicates the validity of our concepts. While always keeping simplicity in mind, a number of language extensions are under consideration, including support for variants as the second major versioning concept besides revisions, and support for temporarily isolated developers' workspaces. In the interest of fast-prototyping, we have employed simple proprietary technology in constructing a limited-functionality prototype (e.g. simple tool integration protocols instead of comprehensive standards such as PCTE), since our focus is on organizational rather than technical issues; the prototype should serve as proof-of-concept. We are now considering facilitating platform-independent operation of the PSEE and supporting its deployment via WWW, by using Java, and CORBA distributed objects as the underlying technology.

4 SUPPORTING VIRTUAL ORGANIZATIONS

4.1 Interprocess Dependencies

So far, the process modeling language as proposed above and implemented is limited to support processes for individual components and within a single organization. Supporting a global software process shared among different organizations requires a language extension allowing processes to be connected that are managed and owned by different organizations or different nodes within a virtual organization, or that describe different components (as illustrated in fig. 8). E.g., for a graphical editor component built on top a component framework, the editor component's process depends significantly on parts of the framework's process, which is typically in a different organization, while the framework's process should remain independent of any process of a component using it.

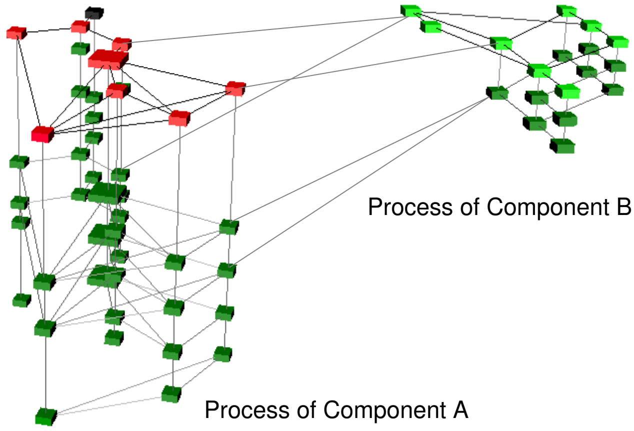

Fig. 8: Processes with interprocess dependencies

This requires interprocess dependencies, whose behavior differs from regular dependencies. Interprocess dependencies do not propagate process definition and enactment operations. In the example, while a rollback in the framework's process invalidates dependent processes within the framework's process, it should not affect processes of components built on the framework, i.e. they should continue to use the same version of a framework and remain unaffected by the creation of a new one, except for a notification. It is then up to the component's developer to establish new interprocess dependencies to newer versions of the framework where this is appropriate. Furthermore, with interprocess dependencies, dependent processes have no information about processes depending on them, e.g. a reusable framework's process should remain read-only for component developers and unaffected by the potentially thousands of component processes depending on it.

Another requirement for supporting virtual organizations is the publication of processes in a form enabling connection of dependent processes. We envisage providing representations of processes on the WWW, whereupon establishment of the interprocess dependencies occurs via PSEEs, but with more limited access privileges.

4.2 Metaprocess Dependencies

The software process itself [18] and the tools used are software too: the tools define the dependencies' semantics and are therefore constituent elements of the process, while they are also software products and therefore the results of their own software processes. These processes providing tools are called metaprocesses with respect to the process wherein they are used to develop a product.

A metaprocess dependency is a special type of interprocess dependency which describes a process' tool's dependency on its development process instead of a process' dependency on a predecessor process. In principle, every atomic process is related by metaprocess dependencies to two tools' processes implementing its creation function and required predicate and providing the tools used to work on its artifact (more precisely, the tools are described by specific configurations within their processes). Every version of an artifact is developed with specific versions of tools, and this information is retained by metaprocess dependencies, instead of automatically using newer versions of tools promising (but often not delivering) upward compatibility when they become available. Furthermore, tools again depend on other software in the global process (e.g. operating systems). Use of interprocess and metaprocess dependencies contributes to configuration management in the global software process by providing version, dependency and origin information.

5 HUMAN RESOURCES

Besides using a common model to represent structure and evolution of a software product and its process as outlined above, a similar model to represent an organization form needs to be established [15]. Today, processes are increasingly the dominant and strategic view of an organization, whereas static organization forms play a less important role. Since we see the software product as the goal of all activities, we do not desire it to be constrained by a static organization structure, as pointed out already in Conway's law ([2], chapter 10). Instead, the organization shall follow the product structure, which means that organization forms are allocated dynamically for ongoing processes, adapting to the structure and hierarchy of the corresponding process, which leads to the structural unity of product, process and organization form, and we omit a persistent or separate hierarchic organization form. Thus, the life-cycle of an organization form is constrained to the life-cycle of the corresponding process.

Additionally, a competence pool managing employees' skills and availabilities and serves as a resource which allows assignment of employees to roles in processes (e.g. planning, executing, confirming) by matching their competencies to the skills required for a role. While difficult to realize, some organizations already follow the pool approach by publishing project proposals on a blackboard and allowing developers to assemble appropriate teams by themselves. A related benefit is the possibility for (independent) information technology experts to be part of a virtual organization only during a specific project.

6 DISCUSSION AND FURTHER BENEFITS

PSEEs resp. process modeling languages have been studied extensively in recent years, collected e.g. in [9] and [8], and surveyed in [4], [1], [10] and [21]. Most PSEE projects are relatively large, focus mainly on process issues in software engineering and provide powerful yet complex and domain-specific process modeling languages; there is a great diversity in language paradigms. Perceiving the product structure including parts and versions is often a difficult task; many languages provide sophisticated possibilities to model processes but do not reflect a goal-oriented design or organizational integrity, whether rule- and trigger-based or graph-based approaches are used. E.g. with moving artifacts in petri-net-based languages, it seems difficult to support process and product evolution and record the corresponding history. In contrast, in our process modeling language, goal-orientation is central and is illustrated by structural unity, and we choose organizational integrity as a framework and do without a sophisticated language, intentionally delegating provision of more specific features to the artifact layer as required. Also compare the Internet, where the trade-off of functionality versus simple protocols and standards and an integrated user interface is enormously successful. Further benefits of organizational integrity achieved through structural unity are outlined below.

6.1 Configuration Management

Organizational integrity allows process and configuration management to be integrated. In our approach, a configuration consists not (only) of artifacts, but of the corresponding processes. Due to the 3-dimensional evolutionary structure of process models, persistence of all versions and dependencies of artifacts in a process is guaranteed. A configuration consists of a set of at most one version of each artifact developed in a process, and is defined by two projections on the whole process. The vertical projection is called filter projection and selects only the processes of concern (e.g. only object code processes). The horizontal projection defines which versions of the filtered processes contribute to the configuration and consists of a selection of maximal processes within the process model and all processes the maximal processes transitively depend on; dependencies can even lead across history layers.

As implemented, we use this process information for the assembly of valid configurations by assuring that dependency relations of corresponding processes are not violated. Furthermore, during the process we typically know which artifact versions are meant to fit together. Thus, only specific constellations of versions are valid, and of those, typically only a small subset make sense. This improved and comprehensible configuration management is only possible due to organizational integration, the lack of which can often be observed in existing systems that try to assemble configurations by following sophisticated rules but without possibility of accessing process information.

6.2 Software Deployment

While support for virtual organizations extends the locality scope of a process beyond individual processes, consequent pursuit of configuration concepts leads to the extension also of the duration scope of processes, beyond development time. Post-deployment configuration management [12] involves issues that can benefit greatly from using process-inherent information such as dependencies. In our vision, software components will be deployed via network in a process-aware manner, meaning that installation involves copying a configured section of a process to the customer, i.e. deliverables, documentation, etc. will retain complete origin, dependency and version information. A customer's installed base of software will consist of copies of many parts of linked processes from different suppliers, and can be regarded as a database of components and metainformation greatly exceeding traditional file and directory concepts. This calls for an extension of PSEE concepts to include process-integrated deployment support and even runtime systems. Loading a component for instantiation will not involve a search simply by name in the local filing system's directory along some arbitrary human-specified paths, but instead will involve a universally valid process address to a precise version, and may include downloading. While some problems are being addressed both in research [11] and in commerce, we are not aware of any integrated solution yet that directly employs process information for the complete life-cycle of software components, from development to deployment and runtime, with instantiation directly out of a process (copy). We believe these to be important future directions for software components.

6.3 Component Reuse Support

Component reuse is often limited due to insufficient information about component interoperability. Software interfaces typically define only syntax, with semantic specifications provided only in informal documents or in undisclosed source code. As a pragmatic alternative, we consider an intermediate information level between the syntactic interface level and the semantic level: the originator level [14]. It provides version, dependency and origin information of components directly out of the process, since a developer typically investigates such context information during a component's development. We propose to retain this useful information and publish it in an appropriate form (via WWW). We postulate that the approach to regard the development context represents an improvement over current practice in describing component interoperability, while bearing in mind that dealing with the semantic level is neither desirable for end-users nor at present usually feasible for developers.

6.4 Continuous Resource Management

Due to the dynamics of the organization of software product, process and organization forms, the estimation and assignment of resources, costs and responsibilities are difficult tasks which require continuous adjustment. Organizational integrity based on structural unity with simple relationships of the organizational structures' parts can make these more comprehensible and easier to manage.

6.5 Process Reuse, Process Patterns

A complementary issue to our project is the design of process models, i.e. how our proposed process modeling language should be employed to define concrete processes. Design is typically the result of experience and reuse of know-how, and we believe it should occur primarily in small concrete portions within an organization or a specific application domain. This could be inspired by work in the pattern community, e.g. patterns of organization and social interaction [3] could be applied. Process components or patterns for reuse in process design could be expressed in a pattern language extending our process modeling language.

7 CONCLUSIONS

Emerging technologies such as software components and the Internet will undoubtedly have a significant influence on the way software is produced, marketed and deployed. Software component developers will create virtual organizations to produce interoperating and interdependent components by connecting their software processes to a global software process. We believe that supporting organization and communication within such heterogeneous organizations is the main challenge of a PSEE. Therefore, the requirements for future PSEEs should emphasize more supporting links between organizations than within them, as they should preserve their autonomy. Thus, a PSEE should provide both a base for communication and organizational integrity through a common understanding of organization encompassing structure, state and evolution of product, process and organization forms shared among all process participants (people and tools). We propose to achieve organizational integrity through structural unity, as the trade-offs of applying a unified simple model to represent all organizational structures versus using specific sophisticated models are worth the gain in organizational integrity which is essential in facing the outlined challenge.

Since technical issues have been studied extensively elsewhere, our aim is to focus on complementary organizational issues. We propose a graphical process modeling language providing organizational integrity through structural unity, which manages processes with evolution and allows complete human control, while optional complementary textual formal methods are supported which are essential for developing reusable components. The main concepts of our process modeling language are rather intuitive, and our PSEE framework prototype demonstrates that they can be implemented and used in a rather straightforward way. While the transition from prototype to large-scale application obviously must include technical interoperability issues not addressed in our project, we believe that our proposed process modeling language could be the underlying concept of future PSEEs reflecting organizational integrity and supporting software processes within virtual organizations as well as distributed configuration management and software deployment. Adapting the ICSE'98 theme, "forging new links" is a key issue in organizing software development in a future global software process.

REFERENCES

[1] P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti, Software Process Representation Languages: Survey and Assessment, *Proc. 4th Conf. Software Engineering and Knowledge Engineering*, IEEE CS Press, 1992, 455-462.

[2] F.P. Brooks, *The Mythical Man-Month*, 20th anniv. ed. (ext'd), Addison-Wesley, 1975/1995.

[3] J.O. Coplien and D.C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995.

[4] B. Curtis, M.I. Kellner, and J. Over, Process Modeling, *Comm. of the ACM*, 35(9):75-90, Sept. 1992.

[5] W. Davidow and M. Malone, *The Virtual Organization: Structuring and Revitalizing the Corporation For the 21st Century*, Burlingame Books, 1992.

[6] M.J. Davis, Process and Product: Dichotomy or Duality?, *ACM SIGSOFT Software Engineering Notes*, 20(2):17-18, April 1995.

[7] P.H. Feiler and W.S. Humphrey, Software Process Development and Enactment: Concepts and Definitions, *Proc. 2nd Int'l Conf. Software Process*, IEEE CS Press, 1993, 28-40.

[8] A. Finkelstein, J. Kramer, and B. Nuseibeh (eds.), *Software Process Modelling and Technology*, Research Studies Press/Wiley, 1994.

[9] P.K. Garg and M. Jazayeri, *Process-Centered Software Engineering Environments*, IEEE CS Press, 1995.

[10] P.K. Garg and M. Jazayeri, Process-Centered Software Engineering Environments: A Grand Tour, in: A. Fuggetta and A. Wolf (eds.), *Software Process*, Wiley, 1996, chapter 2, 25-52.

[11] R.S. Hall, D. Heimbigner, A.v.d. Hoek, and A.L. Wolf, An Architecture for Post-Development Configuration Management in a Wide-Area Network, *Proc. 17th Int'l Conf. Distributed Computing Systems (ICDCS)*, Baltimore, MD, 1997, 269-278.

[12] D. Heimbigner and A.L. Wolf, Post-Deployment Configuration Management, *6th Int'l Workshop Software Configuration Management*, Springer LNCS 1167, 1996, 272-276.

[13] B. Meyer, The Next Software Breakthrough, *Computer*, 30(7):113-114, July 1997.

[14] T. Murer, D. Scherer, and A. Würtz, Improving Component Interoperability, *Workshop Component-Oriented Programming (WCOP)*, at 10th European Conf. Object-Oriented Programming (ECOOP), Linz, Austria, dpunkt Verlag, July 1996, 150-158.

[15] T. Murer and D. Scherer, Structural Unity of Product, Process and Organization Form in the GIPSY Process Support Framework, *Proc. 8th Conf. Software Engineering Environments (SEE)*, Cottbus, Germany, IEEE CS Press, April 1997, 93-100;
also: Project GIPSY homepage (with subprojects CHIPS and DIPS), <http://www.tik.ee.ethz.ch/~gipsy/>, 1997.

[16] T. Murer, The Challenge of the Global Software Process, *Workshop Component-Oriented Programming (WCOP)*, at 11th European Conf. Object-Oriented Programming (ECOOP), Jyväskylä, Finland, Springer LNCS (to be printed), June 1997.

[17] R. Orfali, D. Harkey, and J. Edwards, *The Essential Distributed Objects Survival Guide*, Wiley, 1996.

[18] L.J. Osterweil, Software Processes Are Software Too, Revisited: An Invited Talk on the Most Influental Paper of ICSE 9, *Proc. 19th Int'l Conf. Software Engineering (ICSE)*, Boston, 1997.

[19] D.L. Parnas and P.C. Clements, A Rational Design Process: How and Why to Fake It, *IEEE Trans. Software Eng.*, 12(2):251-257, Feb. 1986.

[20] D. Scherer, T. Murer, and A. Würtz, Designing the Distributed Architecture DIPS for Cooperative Software Engineering, *Proc. 30th Hawaii Int'l Conf. System Sciences (HICSS)*, IEEE CS Press, Jan. 1997, Vol. I, 150-158.

[21] S.M. Sutton, Jr., P.L. Tarr, and L.J. Osterweil, An Analysis of Process Languages, *CMPSCI Technical Report 95-78*, Computer Science Dept., Univ. of Massachusetts, Aug. 1995.

TIK-Reports

A Relational Data Base Design for an X.500 Directory System Agent

F. Perruchoud, C. Lanz, B. Plattner, July 1990, TIK-Report No. 1

Model and Functionality Definition for the Collaborative Editing Conferencing System MultimETH.

H. Lubich, July 1990, TIK-Report No. 2

X.400 Security Capabilities: Evaluation and Constructive Criticism

M. Müller, August 1990, TIK-Report No. 3

CPU Evaluation for ADaM

Schibli, M. Tadjan, Januar 1992, TIK-Report No. 4

Aspekte computergestützter Kooperation - Schriftliches Material eines Seminars an der ETH Zürich

Hannes Lubich, Januar 1993, TIK-Report No. 5

Extensible Attribute Grammars

R. Marti, T. Murer, December 1992, TIK-Report No. 6

GIPSY: A Generator for Incremental Programming Systems

R. Marti, T. Murer, June 1992, TIK-Report No. 7

Test Case Validation - TTCN Test Case Validation Against SDL Specifications

F. Kristoffersen, T. Walter, May 1994, TIK-Report No. 8

Conformance and Interoperability - A critical assessment

T. Walter, B. Plattner, September 1994, TIK-Report No. 9

OOP-Softwarearchitektur für Multimediakommunikation

Serge Hoffmann, 1995, TIK-Report No. 10

A Comparison of Selection Schemes used in Genetic Algorithms

Tobias Blickle, Lothar Thiele, April 1995, TIK-Report No. 11

Spezifizieren und Generieren von integrierten Umgebungen mit GIPSY

A. Helbling, T.Murer, Mai 1995, TIK-Report No. 12

Die Spezifizierungssprache GIPSY/L

A. Helbling, T.Murer, Mai 1995, TIK-Report No. 13

SCSM - Synchronous Composition of Sequential Machines

H. Fierz, Juni 1994, TIK-Report No. 14

Specification of GMS Access Protocol (GAP) Version 1.0

Erik Wilde, March 1996, TIK-Report No. 15

System-Level Synthesis Using Evolutionary Algorithms

Tobias Blickle, Jürgen Teich, Lothar Thiele, April 1996, TIK-Report No. 16

A Flow-Based Approach to Solving Resource-Constrained Scheduling Problems

Jürgen Teich, Lothar Thiele, July 1996, TIK-Report No. 17

Multi-User Multimedia Editing with the MultimETH System

Erik Wilde, February 1994, TIK-Report No. 18

Specification of GMS System Protocol (GSP) Version 1.0

Erik Wilde, September 1996, TIK-Report No. 19

A Proposal for a Real-Time Extension of TTCN

T. Walter, J. Grabowski, 1996, TIK-Report No. 20

OberonT -- eine Programmiersprache für sicherheitskritische Systeme

D. Schweizer, 1996, TIK-Report No. 21

GIPSY: Generating Integrated Process support SYstems - Project Overview

T. Murer, A. Würtz, D. Scherer, D. Schweizer, December 1996, TIK-Report No. 22

CHIPS Reference Manual

A. Würtz, T. Murer, October 1996, TIK-Report No. 23

Dynamic Min-Max Problems

L. Thiele, January 1997, TIK-Report No. 24

Da CaPo++ - Communication Support for Distributed Applications

B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Vogt, M. Waldvogel, January 1997, TIK-Report No. 25

NFS Performance Investigations Based on Classical IP over ATM

T.V. Prabhakar, D. Bauer, B. Stiller, June 1997, TIK-Report No. 26

Dynamic Semantics of the Oberon Programming Language

P.W. Kutter, December 1996, TIK-Report No. 27

Project Da CaPo++, Volume I: Architectural and Detailed Design

B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Vogt, and M. Waldvogel, July 1997, TIK-Report No. 28

Project Da CaPo++, Volume II: Implementation Documentation

B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, and M. Waldvogel, August 1997, TIK-Report No. 29

Traffic Shaping in an ATM Environment

Prabkahar T V, Burkhard Stiller, Thomas Walter, 1997, TIK-Report No. 30

Synchronisation Issues in Distributed Applications: Definitions, Problems, and Quality of Synchronization

Christina Class, 1997, TIK-Report No.31

Optimized Software Synthesis for Digital Signal Processing Algorithms - An Evolutionary Approach

Juergen Teich, Eckart Zitzler, Shuvra S. Bhattacharyya, 1997, TIK-Report No.32

SCF - State Machine Controlled Flow Diagrams

Lothar Thiele, Jürgen Teich, Martin Naedele, Karsten Strehl, and Dirk Ziegenbein, January 1998, TIK-Report No.33

A set macro language for ASMs

Philipp Kutter, January 1998, TIK-Report No.34

SCF - State Machine Controlled Flow Diagrams

M. Anlauff, P.W.Kutter, and A. Pierantonio, Januar 1998, TIK-Report No.35

Telepoly++ - Towards a Highly Distributed Synchronous and Interactive Teleteaching Environment

Thomas Walter, Hans Hänni, January 1998, TIK-Report No.36

Analysis of the Concord Algorithm and the Adaptive Synchronization Protocol Using QoSy

Christina Class, January 1998, TIK-Report No.37

HAM: Hardware Moved Molecules, Annual Report 1997

Martin Gerber, Thomas Gössi, February 1998, TIK-Report No.38

Introducing Design Patterns for Petri Nets

Jörn Janneck, Martin Naedele, February 1998, TIK-Report No.39

Symbolic Model Checking of Petri Nets Using Interval Diagram Techniques

Karsten Strehl, Lothar Thiele, February 1998, TIK-Report No.40

Efficient Security for Large and Dynamic Multicast Groups

Germano Caronni, Marcel Waldvogel, Dan Sun, Bernhard Plattner, April 1998,
TIK-Report No.41

Project Da CaPo++, Volume III: Performance Evaluation

B. Stiller, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Waldvogel, February 1998, TIK-Report No. 42

An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach

Eckart Zitzler, Lothar Thiele, April 1998, TIK-Report No.43

The WaveVideo System and Network Architecture: Design and Implementation

George Fankhauser, TIK-Report No.44

Integrating Domain Specific Language Design in the Software Live Cycle

P.W. Kutter, D. Schweizer, L. Thiele, May 1998, TIK-Report No.46

Object-based Abstract State Machines

P.W. Kutter, J. Janneck, May 1998, TIK-Report No.47

Swiss German Polyphone -- Schlussbericht

K. Huber, June 1998, TIK-Report No.48

Mapping Automata - Simple Abstract State Machines

Joern W. Janneck, Philipp W. Kutter, TIK-Report No.49

Object-based Mapping Automata - Reference Manual

Joern W. Janneck, TIK-Report No.50