# Autonomies in a Software Process Landscape

Volker Gruhn, Ursula Wellen
University of Dortmund
{gruhn, wellen@ls10.cs.uni-dortmund.de}

**Abstract**

Until now autonomy properties have been mainly discussed in the research area of database management systems, agents and robotic systems. We address these properties to software process models, and distinguish between data autonomy, operation autonomy and communication autonomy. In this paper we develop a classification framework of different granularity levels and different degrees for each of the autonomy types. We analyze the autonomy of an example software process landscape modelled with a Petri based net notation. The example process landscape represents software process models for the development of multimedia applications. Detailed analyses of the example check the classification framework, and consider the impact of autonomy properties on software processes and on software process management.

**Keywords**
Autonomy, distributed processes, software process landscape, software process model, process modelling language, Petri nets, Process Landscaping

## 1 Introduction

In the research area of database management systems (dbms) the term "federated dbms" is an established description for a "collection of cooperating but autonomous component database systems" [SL90]. The participating databases, possibly heterogeneous, are distributed and integrated to various degrees. We apply the term "federated" to a software process landscape: it consists of a set of (distributed) process models describing processes necessary for the development of software applications. They cooperate with each other via interfaces in order to produce a certain software system. The modelled processes are often also distributed, heterogeneous and integrated to various degrees. These are all properties well-known in software process modelling research: for example in [ALO96], cooperative (and distributed) software processes are discussed, especially requirements for cooperation support through process centered environments. Greenwood considers coordination as a fundamental feature of software process technology [Gre95]. He criticizes that coordination in software engineering is only achieved by low level details and suggests research results from coordination theory to improve coordination in the area of software process technology. Obbink argues about differentiation and integration of key development processes [Obb95]. Similar to data base management systems he demands a federated organization of different processes like hardware and software engineering (co-engineering) and management activities in order to fulfil market requirements like low costs, short development time and high quality. Cugola et al and Graw et al focus on distribution aspects [CDF98, GG95]. Whereas the first authors present a Java event-based distributed infrastructure (JEDI) to support the implementation of reconfigurable distributed software components, Graw et al discuss coordination issues of distributed modelling and an architecture to support distributed process enaction.

The different processes represented as a software process landscape are also autonomous to various degrees. Until now, this property has been analyzed for agents [DF98], robotic systems [CW90] and of course for federated [Ham94] and distributed [CMW94] database systems, but not yet explicitly for cooperating software processes. In database systems research, Veijalainen and Popescu-Zeletin distinguish between three types of autonomy [VP86], namely design autonomy, communication autonomy and execution autonomy. Subsets of these types also have been identified. Alloui et al discuss these and additional types in [ALO96]. Other authors classify autonomy in dependency of their relevance to operating systems and transaction management issues [GK88].

The approach of Birk to discuss autonomous systems from a more general point of view, also does not consider software processes [Bir99]. He summarizes the research issues in autonomous systems as physical devices, distributed, embedded and cooperating autonomously. Autonomy has not been discussed as property of software processes. We conceive that the increasing complexity of software development activities and the increasing difficulty of its management requires discussion of the property of autonomy for software processes. A certain

degree of process autonomy supports the management of software projects and can improve the quality of software processes' cooperation. In this paper we define different types of software process autonomies and discuss their semantics at different levels of degree and granularity.

First of all we have a look at the structure of a process landscape in order to identify types and levels of process autonomy. Processes within a software process landscape are hierarchically structured and related via interfaces. The hierarchical structure indicates that process models are arranged as activities within more abstract process models. We use the term activity analogously to Chroust who defines it as the smallest unit of a job at a considered abstraction level [Chr92]. Depending on the concrete level of abstraction an activity can be divided into several subactivities, arranged within a separate process model. In the following we restrict ourselves to the term activity instead of talking about activities and process models. This allows us to discuss a process landscape at different levels of abstraction without specifying the concrete level previously.

The hierarchical structure of a process landscape with activities as first class entities allows the modelling at different levels of detail which means adding further information about process landscape elements only where it is useful. Analysis of process landscape properties is therefore possible in different granularity. It can be done either for single activities or for a set of activities, describing a whole abstraction level of a process landscape [GW00b].

Interfaces also serve as first class entities of a process landscape which means that they are already modelled at the more abstract levels [GW00a]. Properties of interfaces and associated information objects to be exchanged are defined and controlled by activities using these interfaces and exchanging information objects with each other. The fewer activities that participate in the definition of a communication infrastructure needed for those interfaces and information objects, the higher is their degree of autonomy.

For a software process landscape, we relate different types of autonomy to activities at different levels of detail. We distinguish between operation autonomy, data autonomy and communication autonomy. Operation autonomy concerns the way how an activity produces a certain result. Data autonomy describes the degree of responsibility for a set of information objects concerning access and maintenance. If an activity decides about communication infrastructure for information exchange with other activities, we talk about communication autonomy. In the following sections we define in more detail what these types of autonomy mean for software processes. We discuss their importance within a software process landscape and analyze an example representing the software development of multimedia applications.

In order to analyze the autonomy within a process landscape we have to define precisely to what the term autonomy is related. Section 2 shows a classification for process autonomies and discusses its relevance in the area of software process management. We developed a formal process modelling language allowing us to model the key elements of a process landscape and their (autonomy) properties quickly and easily. This Process Landscaping Language (PLL) is introduced in section 3 together with a suitable graphical representation. Section 4 discusses autonomy issues of a concrete software process landscape by analyzing the related properties and discussing the resulted benefits. Section 5 sums up our experience applying the term autonomy to a reference software process landscape and gives an overview of our future research directions.

## 2 Autonomy – Types, Degrees and Granularities

In this section we introduce different types of autonomy. We discuss different levels of granularity and define different degrees for each autonomy type.

### 2.1 Types of Autonomy

We distinguish three types of autonomy relevant for activities of a software process landscape:

- If an activity retains the control for a certain set of data, it is responsible for updates and other maintenance tasks. But it is also the only one entitled to make changes and to inform other activities about these changes. If this activity also keeps the data persistent, we talk about data autonomy. Persistency means, that an activity stores locally the data it receives or sends. The quality management (as a complex activity) of a software development project for example is called *data autonomous* with respect to programming guidelines, if it is the only activity allowed to change these guidelines. It updates all affiliated document copies, keeps the update of the original persistent and informs other activities using the guidelines (i.e.

software development activities) about changes. This property of data autonomy can be extended to further data within a process landscape. In the case of quality management, this could be the extension to all guideline and recommendation documents.

- We call an activity *operation autonomous*, if it decides on its own how to produce a requested result. Furthermore, such an activity does not need to inform others about the way it creates certain results. Therefore, activities with operational autonomy can be seen as black boxes, where only interfaces and incoming and outcoming information objects are known. One software process example for operation autonomy is an activity which tries to find out how to develop a software component by prototyping. Incoming information objects are some of the application requirements, and the outcoming information object is the prototype itself.
- *Communication autonomy* concerns the decision making of an activity about the communication infrastructure needed for information exchange with other activities. Often all activities taking part in a communication decide together about how to exchange information, coded or not, synchronously or asynchronously, etc.. If one of the participating activities is communication autonomous, it solely decides how to communicate with other activities. We apply this type of autonomy to a single information exchange of one document between two activities. But we can also extend it to the whole information exchange of one activity to all related activities sending or receiving data to/from the communication autonomous activity. In software development this property is interesting, especially for processes distributed among different locations. If, for example the project management is communication autonomous and takes place at a separate location, all related activities have to adapt their communication infrastructure to the requirements of the project management activity.

### 2.2 Granularity of Autonomy Analysis

The granularity concerns the set of activities forming parts of process models to which autonomy attributes are applied. If this set consists of only one activity, the granularity of the set of activities to be analyzed is very fine. Analyzing a set of several activities can be useful, e.g. when this set takes place at the same location. In this case we extend the autonomy property from single activities to locations. It is also useful to analyze the autonomy of activities belonging to the same abstraction level of a process landscape. If all activities with similar autonomy properties are parts of the same activity at an upper level of the process landscape, we can generalize these properties for the upper abstraction level. A further suggestive set of activities to be analyzed is that of all activities related via interfaces between two locations. This is interesting especially for data and communication autonomy. These two types of autonomy may create inconsistencies when different activities are defined as autonomous and exchange information objects. Summing up, we distinguish four levels of granularity for each type of autonomy: a single activity, a set of activities which take place at the same location, a set of activities belonging to the same abstraction level, and a set of activities communicating between two locations.

### 2.3 Autonomy degrees

Autonomy degrees describe the quantity of autonomy an activity or a set of activities may have. Activities belonging to the same granularity level may be described as

- non-autonomous,
- weakly autonomous,
- semi-autonomous,
- strongly autonomous or
- (completely) autonomous.

The concrete degree of autonomy depends on the proportion of autonomous to non-autonomous activities at one granularity level. For our example process landscape discussed later on, non-autonomous means that there are less then 10% autonomous activities at a certain granularity level. Weakly autonomous means that up to 45% of all activities belonging to a certain granularity level are autonomous. Semi-autonomous means autonomy for up to 55% of the related activities and strongly autonomous means autonomy for up to 90%. If more than 90% of autonomous activities are at the same granularity level, we talk about complete autonomy.

The concrete percentage for different autonomy degrees should be defined individually for each process landscape. The advantage of assigning percentage numbers to informally described autonomy degrees individually for each modelling project is that it allows the comparison of different process landscapes independent from specific landscape characteristics.

### 2.4 Interrelation of types, granularity and degrees

Table 1 gives an overview of the interrelation of different autonomy types, their assignment to granularity levels and their possible degrees. The types of autonomy (table columns) are specified for different sets of activities, representing different granularities of autonomy (table lines). Each table field indicates a certain ratio of granularity to degree for a certain type of autonomy. For data autonomy the real instance of degree is dependent on the ratio of the autonomous activity to the set of used and/or produced data. In the case of operation autonomy it is dependent on the ratio of autonomously executed steps within an activity to the whole set of steps necessary to carry out a certain task.

| type of autonomy / granularity of autonomy | data autonomy | operation autonomy | communication autonomy |
|---|---|---|---|
| activity | % | % | % |
| activities at the same location | % | % | %, ...,% |
| activities at the same abstraction level | % | % | % |
| activities communicating between two locations | % | % | % |

degree of autonomy

Table1: Classification of types, degrees and granularities of autonomy

The degree for communication autonomy depends on the ratio of the set of activities defining autonomously how to communicate with a second activity to the set of those activities which are not allowed to define the way of communication on their own. Calculating the degree of communication autonomy as one percentage number for the granularity level of one location does not make sense: if one activity determines the communication infrastructure for an information exchange, the second participating activity does not. Therefore, one location is always semi-autonomous with respect to communication. That is why we restrict ourselves to the consideration of the ratio between the single activity autonomies.

In order to examine (autonomy) properties of a concrete software process landscape we need a formal basis allowing us to model a complex process landscape at different levels of abstraction. In this paper we discuss a reference software process landscape as result of applying the method of Process Landscaping. We do not discuss the method itself. It is sufficient to know that Process Landscaping is a suitable approach for modelling complex sets of (distributed) processes [GW99]. The underlying process modelling language PLL (Process Landscaping Language) supports the method of Process Landscaping by allowing the modelling of processes and its interfaces at different levels of detail and provides the analysis of the resulting process landscape. PLL can easily be extended by user-defined attributes describing properties which are considered important. In the following we introduce the notation of PLL and extensions relevant for the analysis of autonomy properties.

## 3 Formalization of a Software Process Landscape

In PLL, activities at different levels of detail, information objects to be used and/or produced and access relations between activities and information objects are considered as first class entities. In order to focus on static process landscape properties like autonomy we abstract from the sequence of the activities' order and different types of information objects. In section 3.1 we explain the formal basis consisting of a small language core and functions enabling us to do some autonomy analysis. Section 3.2 discusses the graphical representation of PLL and introduces an example process landscape modelled with PLL which is analyzed in section 4.

### 3.1 PLL Notation

PLL is an abstract Petri net notation [Pet81, Rei86] without control flow, where a word $\omega \in$ PLL represents a process landscape. $\omega \in$ PLL is defined as a triple (V, D, Z) where V is a set of activities, D is a set of document types, and Z is a set of relations between activities and document types. Document types are used as an abstraction for (different types of) information objects. Relations out of Z represent either write or read access of an activity to a document type. $(v,d) \in Z$ represents write access and (d,v) represents read access.

With relation AB $\subseteq$ V×V we describe the hierarchical composition of activities as a tree, more formally: $(v_1,v_2) \in$ AB means that $v_2$ is refining $v_1$. We call AB an activity tree. The root r of this activity tree does not denote an activity, but the process landscape itself.

An interface between two activities $v_1, v_2 \in$ V is defined as follows:

*interface*: V×V $\rightarrow$ *P*(D) with
*interface* $((v_1,v_2)) := \{d \in D \mid ((v_1,d) \in Z \wedge (d,v_2) \in Z) \vee ((v_2,d) \in Z \wedge (d,v_1) \in Z)\}$

This definition relates two activities with document types, one reading and the other writing the document types. Elements out of *interface* $((v_1,v_2))$ are called interface document types.

With function *loc*: V $\rightarrow$ L, where L is a set of locations, we assign a location l $\in$ L to each activity v $\in$ V. With this activity attribute it is possible to analyze a set of activities at the granularity of locations (see table 1).

With the formalization of activities, document types as abstract information objects, interfaces, and AB as relation structuring activities at different levels of abstraction it is possible to create a process landscape. In order to analyze properties of a given process landscape, we extend PLL by functions assigning attributes to activities, document types and relations. Extensions relevant for autonomy analysis are discussed in the following.

### 3.1.1 Definition of Data Autonomy

- ***per*: Z $\rightarrow$ {0, 1, undefined}** is a function assigning either zero, one or undefined to each relation z $\in$ Z.
  *per* $((v_1,d_1)) = 1 \vee$ *per* $((d_1,v_1)) = 1$ means that activity $v_1$ stores a document of type $d_1$ locally.
  *per* $((v_1,d_1)) = 0 \vee$ *per* $((d_1,v_1)) = 0$ means that activity $v_1$ does not store a document of type $d_1$ locally.
  *per* $((v_1,d_1))$ = undefined $\vee$ *per* $((d_1,v_1))$ = undefined means that it is not yet defined, if activity $v_1$ stores a document of type $d_1$ locally.

  This attribute is important for the analysis of data autonomy (see section 2). But it is also useful for the analysis of the effort for updates of different databases: if $\lvert per\ (z) = 1 \rvert$ with $z = (v_1,d_1)$ or $z = (d_1,v_1)$ is "high" for a specific document of type d, one should consider about a central database where *per* (z) = 0 for the affiliated relations. *per* (z) = 1 should retain only for the data autonomous activity.

- Let $Z\mid_{(v,d)} := \{(v,d) \mid (v,d) \in Z\} \subset Z$. ***d-aut*: $Z\mid_{(v,d)}$ $\rightarrow$ {0, 1, undefined}** is a function assigning either zero, one or undefined to each relation z $\in$ Z.
  *d-aut* $((v_1,d_1)) = 1$ means that only activity $v_1$ is allowed to change a document of type $d_1$ and
  *d-aut* $((v_1,d_1)) = 0$ means that activity $v_1$ is not allowed. *d-aut* $((v_1,d_1))$ = undefined means that it is not yet defined, if activity $v_1$ is allowed to change a document of type $d_1$.

  Function *d-aut* is restricted to relations representing write access because it does not make any sense to define an activity as data autonomous concerning a specific document when it only has read access to it. The following condition has to hold:

  *d-aut* $((v_1,d_1)) = 1 \Rightarrow \forall w \in V, w \neq v_1$: *d-aut* $((w,d_1)) = 0$

  We check this consistency condition, if we want to prove the data autonomy of an activity.
  Activity $v_1 \in$ V is called (completely) data autonomous if

  1. *d-aut* $((v_1,d)) = 1 \ \forall d \in D$ with $(v_1,d) \in Z$
  2. *per* $((v_1,d)) = 1 \ \forall d \in D$ with $(v_1,d) \in Z$

### 3.1.2 Definition of Operation Autonomy

- ***op-aut*: Z $\rightarrow$ {0, 1, undefined}** is a function assigning either zero, one or undefined to each relation z $\in$ Z.
  *op-aut* $((v_1,d_1)) = 1 \vee$ *op-aut* $((d_1,v_1)) = 1$ means that only activity $v_1$ defines guidelines how it creates, changes or uses a document of type $d_1$ and therefore does not have to follow guidelines from other activities.
  *op-aut* $((v_1,d_1)) = 0 \vee$ *op-aut* $((d_1,v_1)) = 0$ means that activity $v_1$ may has to follow guidelines when it creates, changes or uses a document of type $d_1$.

*op-aut* $((v_1,d_1))$ = undefined $\vee$ *op-aut* $((d_1,v_1))$ = undefined means that it is not yet defined, if activity $v_1$ has to follow guidelines when creating or changing a document of type $d_1$ or using its content.

Function *op-aut* defines the operation autonomy of an activity with respect to a single document of type d. If *op-aut* = 1 for all documents of type $d \in \{(v_1,d), (d,v_1)\}$, activity $v_1$ is called operation autonomous.

### 3.1.3 Definition of Communication Autonomy

In order to specify the property of communication autonomy, we first have a closer look at a set of attributes forming a communication infrastructure. Those are synchronity, changeability, coding, privacy, and persistency. For a communication in working order, these attributes have to fit together for relations affiliated to communication between two or more activities. In other words, if e.g. a sending activity requires synchronous communication for a document type to be exchanged, the receiving activity also has to define its relation to this document type as synchronously. The same has to hold for all other attributes. The following functions define more formally, what we mean by synchronity, privacy, coding and changeability. They form the basis for the analysis of communication autonomy.

- ***synch*: $Z \rightarrow$ {0, 1, undefined}** is a function assigning either zero, one or undefined to each relation $z \in Z$.
  *synch* $((v_1,d_1)) = 1 \vee$ *synch* $((d_1,v_1)) = 1$ means that activity $v_1$ expects synchronous data interchange when sending or receiving a document of type $d_1$ to/from other activities.
  *synch* $((v_1,d_1)) = 0 \vee$ *synch* $((d_1,v_1)) = 0$ means that activity $v_1$ expects asynchronous data interchange when sending or receiving a document of type $d_1$ to/from other activities.
  *synch* $((v_1,d_1))$ = undefined $\vee$ *synch* $((d_1,v_1))$ = undefined means that it is not yet defined, if activity $v_1$ expects synchronous data interchange when sending or receiving a document of type $d_1$ to/from other activities.

  The attribute synchronous has impact on the communication infrastructure between activities: communication via letter post for example always has to be defined as asynchronous, whereas calling per telephone has to be defined as synchronous communication.

- ***priv*: $Z \rightarrow$ {0, 1, undefined}** is a function assigning either zero, one or undefined to each relation $z \in Z$.
  *priv* $((v_1,d_1)) = 1 \vee$ *priv* $((d_1,v_1)) = 1$ means that activity $v_1$ sends/receives a document of type $d_1$ to/from exactly one other activity.
  *priv* $((v_1,d_1)) = 0$ means that activity $v_1$ sends a document of type $d_1$ to more than one other activity.
  *priv* $((d_1,v_1)) = 0$ means that not only activity $v_1$ receives a document of type $d_1$ but also others.
  *priv* $((v_1,d_1))$ = undefined means that it is not yet defined, if activity $v_1$ sends a document of type $d_1$ only to one or to several other activities.
  *priv* $((d_1,v_1))$ = undefined means that it is not yet defined, if only activity $v_1$ receives a document of type $d_1$.

  Function *priv* defines whether information exchange between activities is private (*priv* (z) = 1) or not (*priv* (z) = 0). This relation attribute has impact on the way how documents can be distributed between several locations. If an activity wants to send information to others e.g. via broadcasting, *priv* (z) has to be zero.

- ***coded*: $Z \rightarrow$ {0, 1, undefined}** is a function assigning either zero, one or undefined to each relation $z \in Z$.
  *coded* $((v_1,d_1)) = 1 \vee$ *coded* $((d_1,v_1)) = 1$ means that activity $v_1$ sends/receives a documents of type $d_1$ encoded.
  *coded* $((v_1,d_1)) = 0 \vee$ *coded* $((d_1,v_1)) = 0$ means that activity $v_1$ sends/receives a document of type $d_1$ not encoded.
  *coded* $((v_1,d_1))$ = undefined $\vee$ *coded* $((v_1,d_1))$ = undefined means that it is not yet defined, if activity $v_1$ sends or receives a document of type $d_1$ encoded.

  Encoding documents before sending them indicates that no other but the recipient(s) should read the content. It also requires that decoding mechanisms are available at the recipient's side.

- ***change*: $Z \rightarrow$ {0, 1, undefined}** is a function assigning either zero, one or undefined to each relation $z \in Z$.
  *change* $((v_1,d_1)) = 1 \vee$ *change* $((d_1,v_1)) = 1$ means that activity $v_1$ sends or receives a document of type $d_1$ in a way that the recipient is able to change the content. Sending a document type e.g. as MS-Word file leaves the file changeable, sending it as pdf file or in paper format does not.
  *change* $((v_1,d_1)) = 0 \vee$ *change* $((d_1,v_1)) = 0$ means that activity $v_1$ sends or receives a document of type $d_1$ in

a way that the recipient is not able to change the content.
*change* $((v_1,d_1))$ = undefined $\vee$ *change* $((d_1,v_1))$ = undefined means that it is not yet defined, if activity $v_1$ sends or receives a changeable document of type $d_1$.

This attribute corresponds with the data autonomy of an activity: if an activity wants to retain the control with respect to certain documents, not only *change* (z) should be one but also the corresponding function *d-aut* (z).

Communication infrastructure can be described by communication channels associated to interfaces. Communication channels describe how an activity sends or receives information objects, and whether it determines the communication infrastructure. This is specified by attribute values of relations belonging to an interface, namely the values of functions *per*, *synch*, *priv*, *coded* and *change*. The attribute values have to fit together to form a communication channel able to work. The communication infrastructure can be based on:

- electronic data interchange like email or sms,
- synchronous communication infrastructure like telephone for oral information exchange or
- real document interchange like letter post.

Let CC be a set of communication channels. ***c-channel*: $(Z\times Z)' \rightarrow CC$** is a function assigning a communication channel $c \in$ CC to each tuple $(z_1, z_2) \in (Z\times Z)'$. $(Z\times Z)'$ is a subset of $Z\times Z$, where $z_1$ and $z_2$ are relations belonging to the same document type and at least $z_1$ or $z_2$ is defined as a write access. More formally: $(Z\times Z)' \subset Z\times Z$ and $\forall (z_1, z_2) \in (Z\times Z)'$: $(z_1 = (v_1,d_1) \Rightarrow z_2 = (d_1,v_2)) \wedge (z_1 = (d_1,v_1) \Rightarrow z_2 = (v_2,d_1))$.

A communication channel is called *congruent*, if each attribute of relations $z_1$ and $z_2$ has the same value. If for example persistency of $z_1$ is 1, it also has to be 1 for $z_2$. Communication channels which are not congruent can be identified by consistency checks. Each congruent communication channel defines how an activity $v_1$ receives or sends a document of type $d_1$ via an interface to another activity $v_2$.

With the functions introduced above we are now able to define the property of communication autonomy for an activity of a process landscape:

- ***com-aut*: $Z' \rightarrow$ {0, 1, undefined}** is a function assigning either zero, one or undefined to each relation $z \in Z'$, where $Z' = \{z \in Z \mid \exists\ c_i \in CC, d_i \in D$ where $\forall\ v_a, v_b \in V, v_a \neq v_b$: (c-channel $((v_a,d_i) \times (d_i,v_b)) = c_i) \vee$ (c-channel $((v_b,d_i) \times (d_i,v_a)) = c_i)\}$.
  *com-aut* $((v_1,d_1)) = 1 \vee$ *com-aut* $((d_1,v_1)) = 1$ means that only activity $v_1$ determines the values of relation attributes corresponding to a communication channel for exchanging a document of type $d_1$.
  *com-aut* $((v_1,d_1)) = 0 \vee$ *com-aut* $((d_1,v_1)) = 0$ means that activity $v_1$ does not determine the values of relation attributes corresponding to a communication channel for exchanging a document of type $d_1$.
  *com-aut* $((v_1,d_1))$ = undefined $\vee$ *com-aut* $((d_1,v_1))$ = undefined means that it is not yet defined, if activity $v_1$ determines the values of relation attributes corresponding to a communication channel for exchanging a document of type $d_1$.

  The related communication channel has to be either congruent or is adapted by the second activity $v_2$ affiliated with this communication channel to be able to work. In other words, the values of functions *synch*, *change*, *coded* and *priv* have to match. We do not need to adapt persistency values because it has no impact on the operativeness of a communication channel, if a sending activity keeps a document type persistent whereas the receiving activity does not. Additionally, the following consistency conditions have to hold: *com-aut* $((v_1,d_1)) = 1 \Rightarrow$ *com-aut* $((d_1,v_2)) = 0$ and *com-aut* $((d_1,v_1)) = 1 \Rightarrow$ *com-aut* $((v_2,d_1)) = 0$. These conditions ensure that always only one of two activities belonging to a communication channel can be defined as communication autonomous.

  Function *com-aut* defines the communication autonomy of an activity $v_1$ corresponding to one communication channel and with respect to a single document of type $d_1$. Activity $v_1$ is called (completely) communication autonomous if

  1. *com-aut* $((v_1,d)) = 1 \wedge$ *com-aut* $((d,v_1)) = 1\ \forall\ d \in D$ with $(v_1,d) \in Z' \vee (d, v_1) \in Z'$
  2. $\forall\ c \in CC$ corresponding to activity $v_1$: c is either congruent or is adapted by the second activity corresponding to this communication channel to be able to work.

With this set of functions assigning attributes to activities and relations we are now able to analyze communication and autonomy properties of a process landscape. In order to do so, we first explain how process

landscapes are graphically represented. Those representations facilitate the understanding of a landscape and support the processing of analysis.

### 3.2 Graphical Representation of a Software Process Landscape

In this section we introduce a graphical representation of a software process landscape. For this purpose we model a process landscape which covers parts of the development for multimedia applications. Figure 1 shows the activity tree of this software process landscape which already indicates some key features of multimedia software development [Sof99, Hän97]. Activity "Media Management" for example is an activity at the same level like "Project Management" or "Software Development". Other activities typical for this kind of software development are refined to more concrete levels. Examples are "SD_Media Production" and "SD_Storyboard Production" within the design phase and "SD_Media Integration" within the implementation phase, both as parts of activity "Software Development". Activities modelled at a more detailed level start their name with an abbreviation indicating their origin. For example, activity "SD_Documentation User Manual" is part of activity "Software Development".

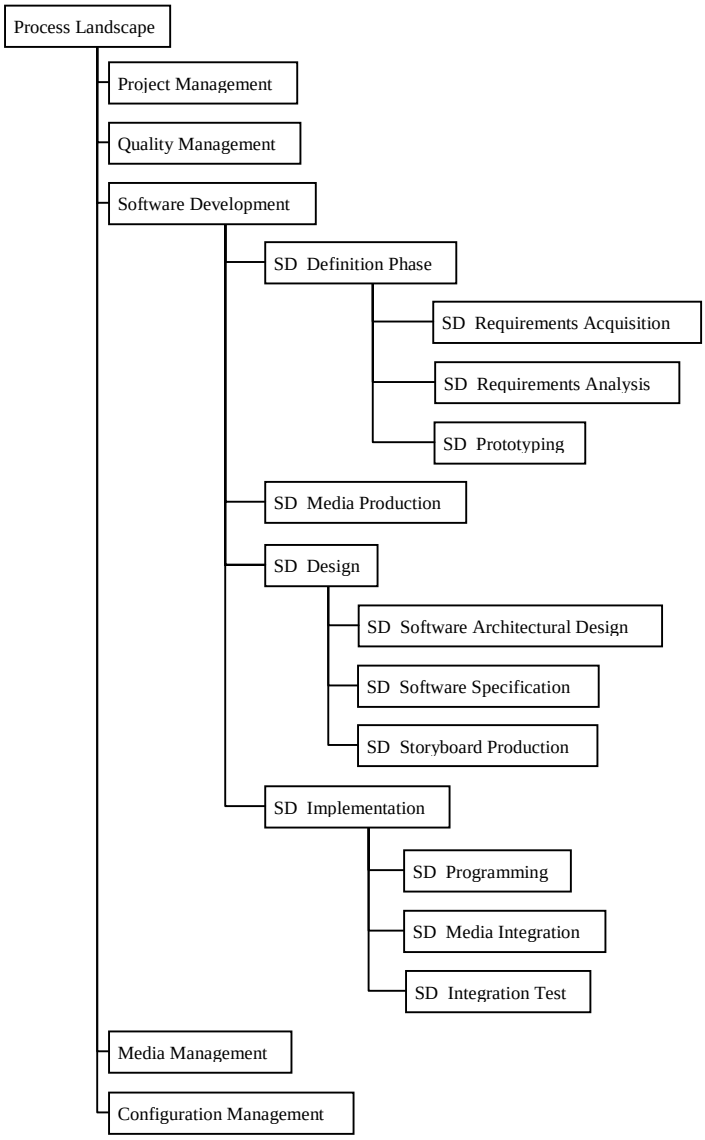


Figure 1: Activity tree of a software process landscape for multimedia application development

As already mentioned, activities can be refined in terms of process landscapes (see section 1). Figure 2 shows such a refinement for activity "Software Development". It represents four activities and indicates interfaces between them by bidirectional arrows. The arrows indicate the existence of interface document types exchanged between two activities. They do not show the concrete document or relation types. The activities are still modelled coarse-grained at this level of refinement.

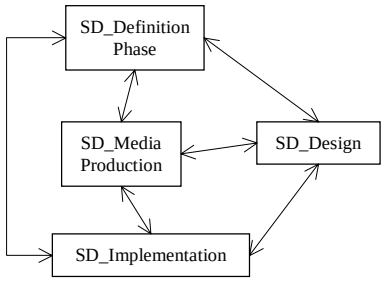


Figure 2: Refinement of activity "Software Development" with indicated interfaces

Figure 3 shows another part of the software process landscape, where activities and their relations to concrete types of documents are depicted. Document types are represented as circles, their names are listed in the legend on the right side of figure 3. A document type related as interface document type to an activity which is not at the same refinement level is indicated by an arrow pointing out of this level or pointing into it. In figure 3, "programming guidelines", "software specification", "software code", "multimedia types" and "multimedia application" are examples of interface document types.

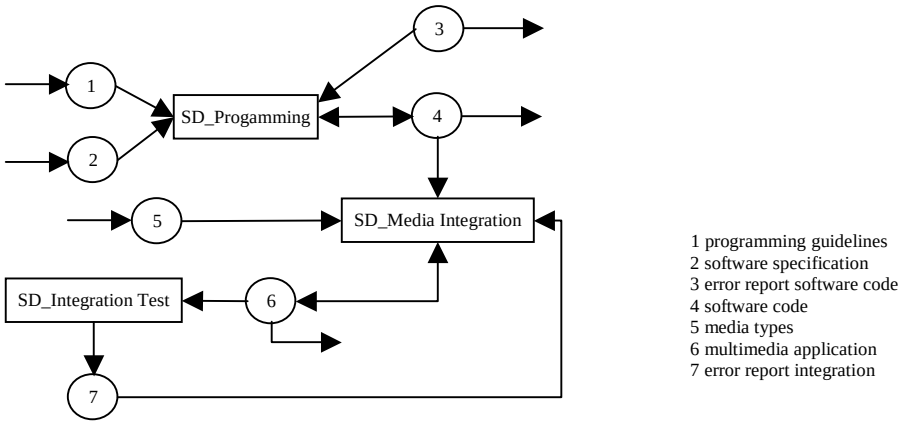


Figure 3: Refinement of activity "SD_Implementation"

The document view of a software process landscape is presented in figure 4. It shows a document of type "software code" affiliated to activities by different access relations. This view supports the analysis of access relations from activities to documents of a certain type. The naming conventions – starting the name with the abbreviation of the origin activity at the top level of the software process landscape – help to identify the position of the activities in the activity tree.

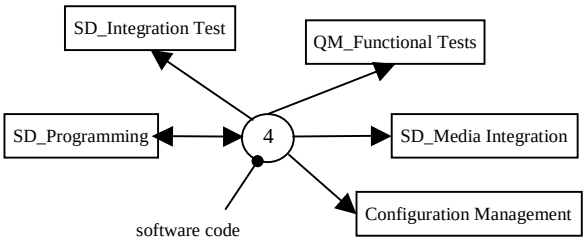


Figure 4: Document view of document type "software code"

## 4 Analysis of Process Autonomies

We now analyze the example of a software process landscape introduced in the previous section. Table 2 presents an initial overview of the extend of the upper process landscape levels. For each activity modelled at the top level of the process landscape the first table column indicates the number of affiliated subactivities which are refined further. These subactivities represent the successors of each top level activity. The second table column

shows the number of all subactivities affiliated to one top level activity which are modelled as leaves within the activity tree (partially illustrated in figure 1). The maximum number of refinement levels and the number of interface documents are listed for each top level activity in table columns three and four.

| top level activity | number of refined subactivities (successor) | number of all subactivities (leaves) | max number of refinement levels | number of interface documents |
|---|---|---|---|---|
| **Project Mgmt** | 2 | 12 | 3 | 8 |
| **Quality Mgmt** | 3 | 8 | 3 | 7 |
| **Software Developm.** | 4 | 14 | 4 | 9 |
| **Media Mgmt** | 3 | 10 | 3 | 5 |
| **Configuration Mgmt** | | 7 | 2 | 7 |

Table 2: Complexity of the entire example process landscape

As already observable in figure 1, the top level of the example software process landscape consists of five activities. They represent the key processes of the software development for multimedia applications. Already at the second level of abstraction they have been refined with different degrees of detail. For example, "Project Management" is divided into two further refined subactivities, whereas "Configuration Management" is refined by a set of subactivities which are not refined any further.

The total number of refinement levels and leaves of each top level activity gives no information about the content's complexity of the modelled process landscape. This depends on the process modeller who decides individually about the level of modelling detail in each modelling phase. The same holds for the number of interface documents. The more detailed activities are modelled, then the more interfaces result without indicating any conclusions about the concrete number and variety of information objects to be exchanged. Therefore, we do not discuss the relation of activities to subactivity numbers. The total number of all activities within the examined software process landscape is much higher. It increases further if we extend the process landscape by adding information about temporal and causal dependencies of the modelled activities. With PLL, we only model the upper levels of a process landscape with already complex activities where control flow and different states of documents are not considered.

Due to the length of this paper we restrict ourselves to the discussion of autonomy types only at some selected granularity levels (see table 1 in section 2.4):

- data autonomy of a single activity (section 4.1)
- data autonomy of activities at the same abstraction level (section 4.2)
- operation autonomy of activities which take place at the same location (section 4.3)
- communication autonomy of activities which take place at the same location (section 4.4)
- communication autonomy of activities communicating between two locations (section 4.5)

We analyze their different degrees and discuss their impact on the management of the software process landscape. Therefore, we merge subsets of attribute values relevant for granularity levels and/or locations to be considered. We focus on situations identified within the example landscape where autonomy analysis turned out to be useful. Expected analysis benefits are

- identification of inconsistencies within the modelled process landscape
- verification of expected landscape characteristics
- identification of improvement possibilities concerning the autonomy of activities

### 4.1 Data Autonomy of a Single Activity

Figure 1 in section 3.2 shows activity "SD_Storyboard Production" as refining part of activity "SD_Design". If we want to consider the data autonomy of "SD_Storyboard Production", we have to focus on attributes with respect to all access relations starting with this activity. Figure 5 shows attribute values assigned by functions *d-aut* and *per*. "SD_Storyboard Production" is related via write access to five documents of different types. For the property of data autonomy only those relations are important where *d-aut* (SD_Storyboard Production, document type $d_i$)= 1. This is the case for relations $z_1$ = (SD_Storyboard Production, storyboard document) and $z_2$ = (SD_Storyboard Production, refined didactical design). In accordance with the regulations in section 3 we have to check the consistency condition for *d-aut* ($z_1$) and *d-aut* ($z_2$). It requires that

1. *d-aut* ((SD_ Storyboard Production, storyboard document)) = 1 ⇒
   ∀ w ∈ V, w ≠ SD_ Storyboard Production: *d-aut* ((w, storyboard document)) = 0
2. *d-aut* ((SD_ Storyboard Production, refined didactical design)) = 1 ⇒
   ∀ w ∈ V, w ≠ SD_ Storyboard Production: *d-aut* ((w, refined didactical design)) = 0

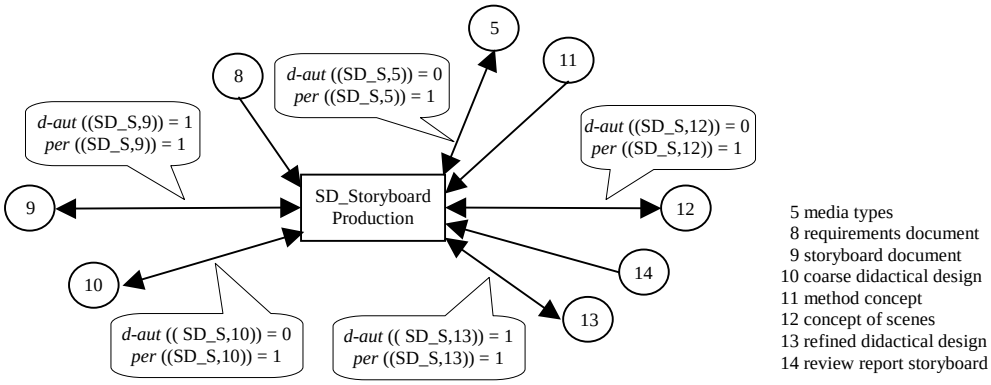


Figure 5: Data autonomy of activity "SD_Storyboard Production"

For checking the first condition, we have a closer look at the document view of document "storyboard document". Figure 6 represents this document view together with the affiliated attribute values. In addition to activity "SD_Storyboard Production" there are three other activities related to document "storyboard document". For relation (SD_Media Integration, storyboard document), which is the only additional write access to this document, *d-aut* is zero. This means that the first consistency condition is fulfilled.

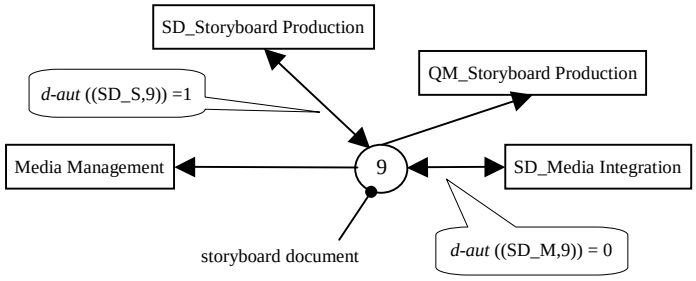


Figure 6: Document view of document "storyboard document" with affiliated attribute values

Figure 7 shows the document view of document "refined didactical design". In this case, we identify relation (SD_Media Integration, storyboard document) as only additional write access to "refined didactical design" where *d-aut* is zero again. If it had been values "undefined" it would have to be changed to zero to obtain consistency.

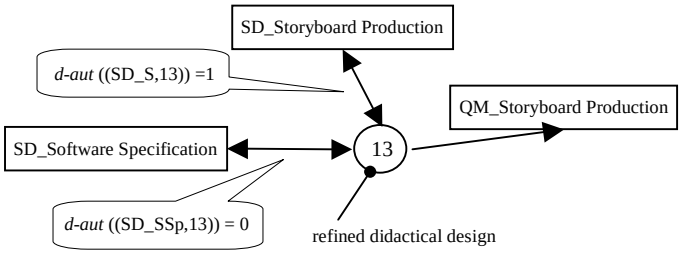


Figure 7: Document view of document "refined didactical design"

Coming back to the data autonomy of activity "SD_Storyboard Production", *d-aut* (SD_Storyboard Production, software code) = *d-aut* (SD_Storyboard Production, refined didactical design) =1 is necessary but not sufficient for this property. The concerned relations also have to be persistent. Figure 5 shows, that this additional condition is also fulfilled.

Summarizing the discussion of data autonomy for activity "SD_Storyboard Production", we have identified two of five relations where "SD_Storyboard Production" is data autonomous. With 40% data autonomous relations in total we call "SD_Storyboard Production" weakly data autonomous (see section 2). Considering the task of activity "SD_Storyboard Production" within the software process landscape, this degree seems to be realistic. The activity has to support the software development by providing suitable software code needed by many other activities. It is an important service activity where the requirements for the data to be produced are defined by others and the produced data is used by others.

Weak data autonomy indicates a suitable degree for service activities. The described situation illustrates a useful evaluation of a suitable degree of data autonomy. It supports the process management by checking the distribution of data autonomy within a software process landscape. Only if the autonomy degree is not an expected one, the associated activities need to be analyzed further.

### 4.2 Data Autonomy of Activities at the Same Abstraction Level

In the previous subsection we have analyzed the data autonomy of activity "SD_Storyboard Production". We now want to discuss this property for its complete abstraction level. This means analysis of data autonomy for the refined activity "SD_Design" (see also figure 1). There are two other activities at this abstraction level, namely "SD_Software Architectural Design" and "SD_Software Specification". We first have to calculate their degree of data autonomy in order to find out the degree of data autonomy for activity "SD_Design".

Figure 8 shows that activity "SD_Software Architectural Design" has only one of two relevant relations where *d-aut* = 1 and *per* = 1, namely the relation to "architectural design document". Its document view in figure 9 shows that the consistency condition is also fulfilled, because "SD_Software Architectural Design" is the only related activity with *d-aut* = 1. Therefore, this activity is semi-autonomous.

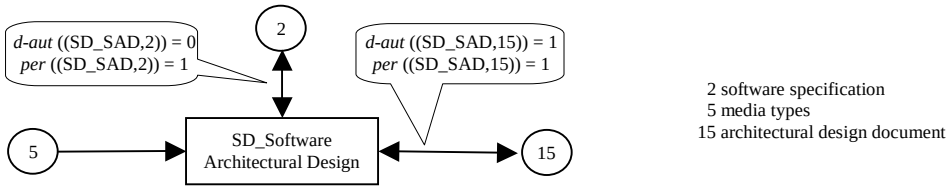


Figure 8: Data Autonomy of activity "SD_ Software Architectural Design"

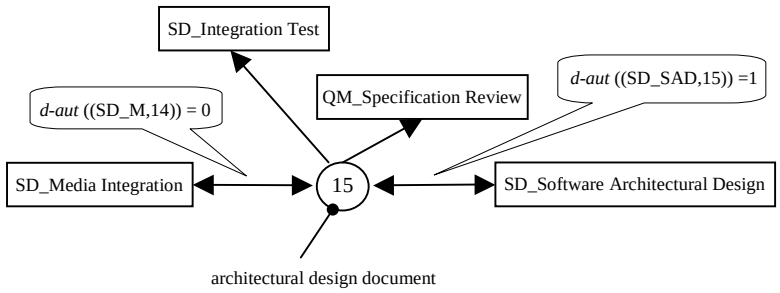


Figure 9: Document view of document "architectural design document" with affiliated attribute values

Figure 10 shows that activity "SD_Software Specification" is non-autonomous. The value of *d-aut* for relation (SD_Software Specification, software specification) is one, but the persistency is zero. Therefore, we can now calculate the degree of data autonomy for activity "SD_Design", representing the same abstraction level for activities "SD_Storyboard Production", "SD_Software Architectural Design" and "SD_Software Specification". We calculate the autonomy degree of higher levels by considering the ratio of the relation set with *d-aut* (z) = 1 and *per* (z) = 1 to those without data autonomy: with three out of nine data autonomous relations (= 33%) activity "SD_Design" is called weakly autonomous. Obviously, modelling further activities as elements of "SD_Design" requires recalculation of the autonomy degree.

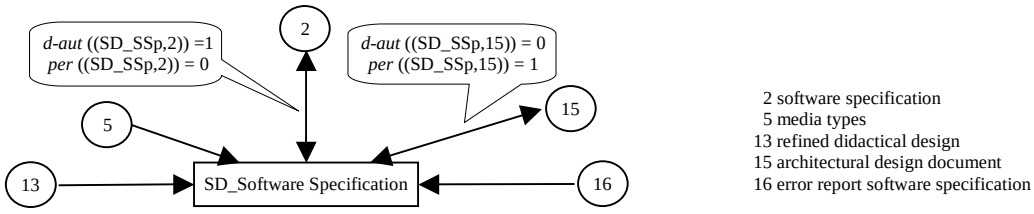


Figure 10: Data autonomy of activity "SD_Software Specification"

For process management it is useful to consider especially those activities at a high level of abstraction, which are either non- or completely autonomous with respect to data autonomy. Unusual high or low degrees indicate states of software process landscape parts where data autonomy might not be determined correct. For example for the refined activity "SD_Definition Phase" (see also figure 1), we measured first no data autonomy. The reason was an undefined autonomy value where the responsible roles did not come to an agreement yet. In most cases some subactivities are at least weakly autonomous like it has been in the example for subactivities "SD_Software Architectural Design" and SD_Software Specification".

### 4.3 Operation Autonomy of Activities which take Place at the Same Location

In the following we analyze the property of operation autonomy for activities which take place at the customer's side. In our example these are activities "SD_Requirements Acquisition", "SD_Requirements Analysis" and "SD_Storyboard Production". In figure 11, they are arranged with all document types they are related with. For the sake of clarity, we assigned only the results of function *op-aut* to the relation arrows except for those representing both, read and write access.

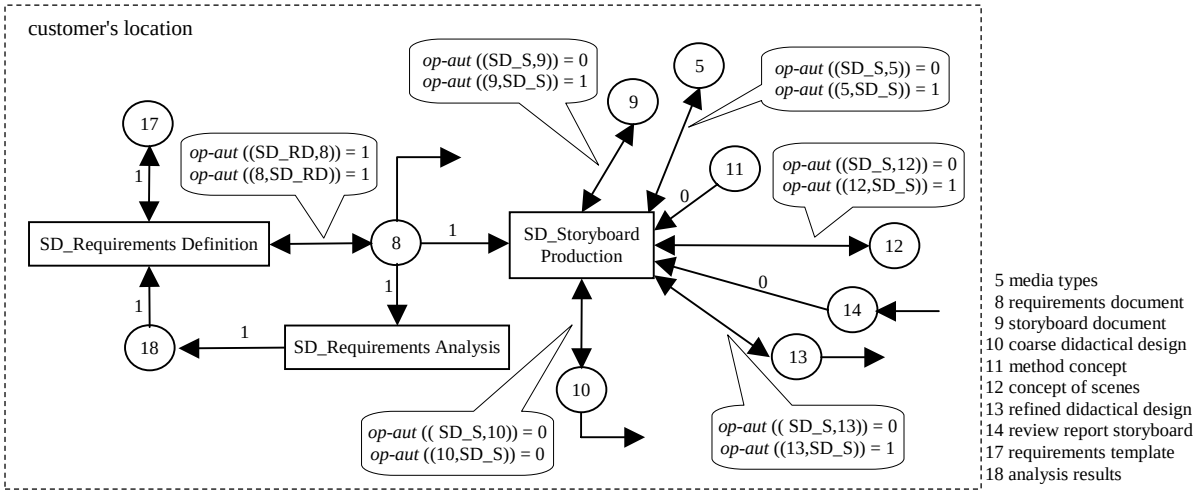


Figure 11: Activities which take place at the customer's side

Activity "SD_Requirements Definition" is completely operation autonomous, because for every relation function *op-aut* = 1. The activity produces the requirements documentation without guidelines from other activities. The templates it uses are defined by the activity itself. Analysis results which cause changes to the requirements document have no impact on the way how activity "SD_Requirements Definition" implements these changes.

"SD_Requirements Analysis" uses the requirements document in order to prepare the design phase of the software development. Some of the analysis results may require changes to the requirements document. Because *op-aut* = 1 for the relations to documents "requirements document" and "analysis results", activity "SD_Requirements Analysis" is also completely operation autonomous.

"SD_Storyboard Production" is related to most of the documents in a non-autonomous way with respect to operation autonomy. For example, the documents describing the method concept and the concept of scenes (no. 11, 12) may have impact on the way the storyboard is produced. The report of the storyboard review (no. 14) also changes the production process, if there are e.g. some issues missing which require the use of additional techniques. Therefore, activity "SD_Storyboard Production" cannot produce the storyboard document independent on other activities. Only a previous storyboard version and changes in the fine didactical concept or

in the concept of scenes should have impact on the production process of the next version. This is where *op-aut* (z) = 1 with "SD_Storyboard Production" as affiliated activity. With four of 13 relations, where *op-aut* (z) = 1, we call activity "SD_Storyboard Production" weakly operation autonomous.

Now we can calculate the autonomy degree for the total granularity level: *op-aut* (z) = 1 for nine of 12 relations at this location. In other words, the set of activities of this software development project taking place at the customer's location is strongly autonomous (see section 2). A high degree of operation autonomy supports the software process management because it does not have to spend much time organizing and managing those processes. It is sufficient to check their output data in order to start the following processes with correct input data.

### 4.4 Communication Autonomy of Activities which take Place at the Same Location

Figure 12 shows activities which take place at one location. Because all activities concern developing tasks, we call it the developers' location. If we want to analyze communication autonomy at the granularity level of one location we have to focus on information exchange within this location. In our example in figure 12, the elements of the software process landscape participating at this communication are shaded grey: activities "SD_Programming", "SD_Media Integration" and "SD_Integration Test" exchange documents "software code", "multimedia application" and "error report integration" with each other. "SD_Prototyping" also takes place at the developers' location, but at this level of abstraction and location it does not exchange data with other activities at this location.

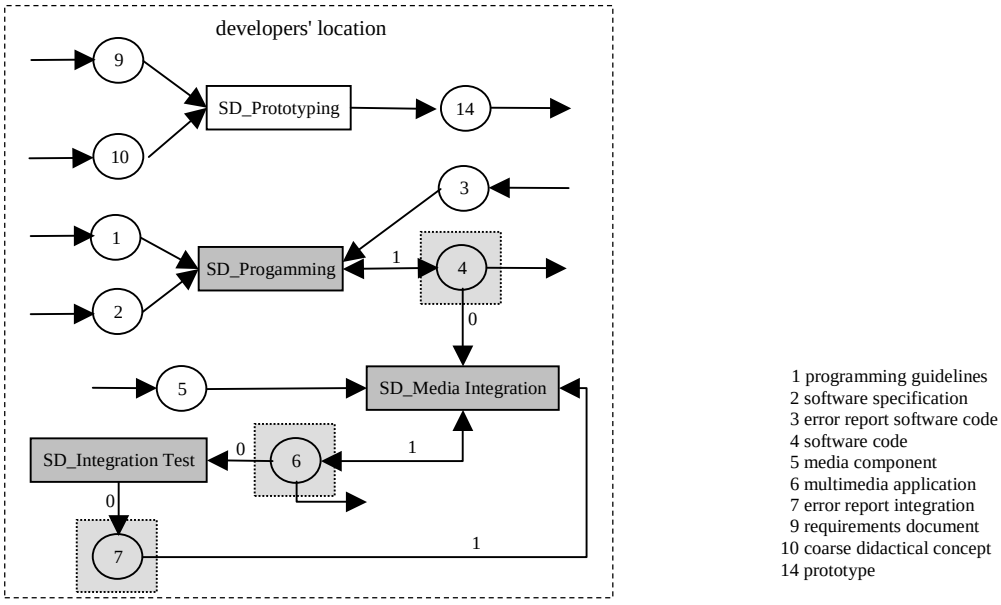


Figure 12: Activities which take place at the developers' location

Dotted frames around grey shaded document types denote three different communication channels. In order to keep figure 12 simple and comprehensible we list their associated communication attributes in table 3. We assign again only the results of function *com-aut* to the relevant relation arrows in figure 12. In the case of a bidirectional arrow the value is assigned to write access. A first look at figure 12 shows that *com-aut* is defined properly: each communication channel contains one relation with *com-aut* = 1 and the other with *com-aut* = 0 (see definition in section 3).

Table 3 shows three communication channels we have to analyze before calculating the autonomy degrees of activities "SD_Programming", "SD_Media Integration" and "SD_Integration Test". The channels are each indicated by two lines surrounded by double frames. The software code e.g. is exchanged via the communication channel between activity "SD_Programming" and "SD_Media Integration". This channel is not congruent because *priv* (software code, SD_Media Integration) is "undefined". The value has to be changed to 1, because "SD_Programming" is defined as communication autonomous with respect to document "software code" (see figure 12) and therefore determines the value of this attribute.

"SD_Media Integration" sends document "multimedia application" to "SD_Integration Test" via the second communication channel (see also dotted frame around document type no. 6 in figure 12). This communication

channel is defined in an inconsistent way: "SD_Media Integration" sends the document not changeable whereas "SD_Integration Test" expects to receive it changeable. Because *com-aut* (SD_Media Integration, multimedia application) = 1, *change* (multimedia application, SD_Integration Test) has to be adapted.

| | persistent | synchronous | private | encoded | changeable |
|---|---|---|---|---|---|
| **SD_Programming, software code** | 1 | 0 | 1 | 0 | 1 |
| **software code, SD_Media Integration** | 1 | 0 | undefined | 0 | 1 |
| | | | | | |
| **SD_Media Integration, multimedia application** | 1 | 0 | 0 | 1 | 0 |
| **multimedia application, SD_Integration Test** | 1 | 0 | 0 | 1 | 1 |
| | | | | | |
| **SD_Integration Test, error report integration** | 0 | 1 | 1 | undefined | 0 |
| **error report integration, SD_Media Integration** | 1 | 1 | 1 | 1 | 0 |

Table 3: Communication channels and related attributes at the developer's location

The last two lines in table 3 describe the communication channel via which "SD_Integration Test" sends an error report of the integration test to " SD_Media Integration". This report is only kept persistent by the receiving activity. As already mentioned with the formal introduction of function *com-aut*, we do not need to adapt this attribute value because it has no impact on the operativeness of the communication channel. However, we have to adapt the value of *coded* (SD_Integration Test, SD_Media Integration) and change it to 1. This is determined by the communication autonomy of activity " SD_Media Integration" with respect to document "error report integration" (see figure 12).

After the adaption of some communication channel attributes it is now possible to calculate the autonomy degrees for activities "SD_Programming", "SD_Media Integration" and "SD_Integration Test". At this granularity level we call "SD_Programming" (completely) autonomous. "SD_Media Integration" is strongly autonomous because for two of three affiliated relations the value of function *com-aut* is 1. "SD_Integration Test" is non-autonomous.

In order to judge these three autonomy degrees process management can follow a general rule: the more relations to different information objects an activity has, the higher the degree of communication autonomy should be. Otherwise this activity would have an enormous effort to manage the communication infrastructure needed to fulfil the probably heterogeneous requirements for information exchange. In our example "SD_Programming" is completely autonomous and "SD_Media Integration" is strongly autonomous. The two activities have the most relations of all to different information objects at the developers' location. Therefore, the different degrees of communication autonomy indicate a satisfactory value. The benefit of analyzing this special situation within the examined software process landscape was at least the identification and removement of inconsistencies for two communication channels and the verification of suitable autonomy degrees.

### 4.5 Communication Autonomy of Activities Communicating between two Locations

In this section we analyze communication autonomy between activities at the developers' location and at the department for quality management. The latter department is settled at a separate location. Figure 13 shows three documents which are exchanged between the two locations, namely "programming guidelines", "error report software code" and the "software code" itself. Not all activities are participating at the external information exchange. Some are only communicating within its location (see section 4.4). Activities participating at the communication with another location, and the affiliated interface documents and communication channels are shaded grey in figure 13. The results of function *com-aut* are assigned to each relation arrow. In the case of a bidirectional arrow the value is assigned to write access as in figure 12.

Table 4 shows the attributes of the affected communication channels in detail. "QM_Development Guidelines" sends the programming guidelines to "SD_Programming" at the developers' location. The sender keeps the document persistent, the receiver does not. But this fact has no impact on the communication infrastructure. The value of function *priv* (programming guidelines, SD_Programming) has to be changed from 1 to 0 because activity " QM_Development Guidelines" determines the value for this communication channel and has already defined the value to be 0.

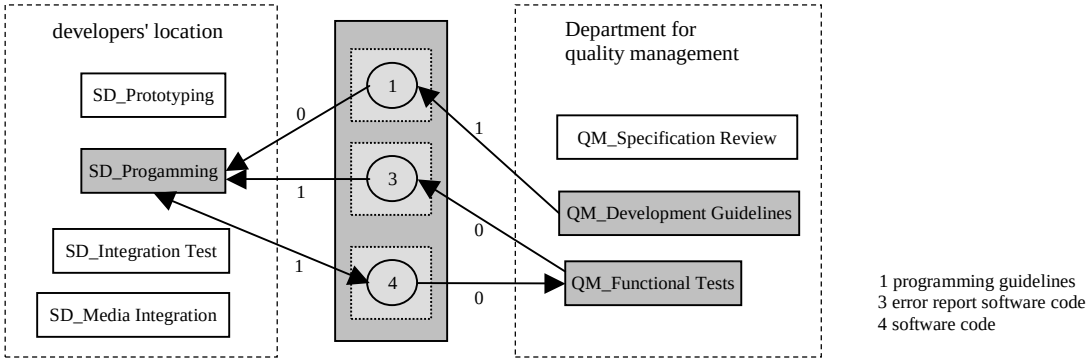


Figure 13: Communication between developer's location and department for quality management

| | persistent | synchronous | private | encoded | changeable |
|---|---|---|---|---|---|
| **QM_Development Guidelines, programming guidelines** | 1 | 0 | 0 | 0 | 0 |
| **programming guidelines, SD_Programming** | 0 | 0 | 1 | 0 | 0 |
| **SD_Programming, software code** | 1 | 0 | 1 | 0 | 1 |
| **software code, QM_Functional Tests** | 1 | 0 | 1 | 0 | 0 |
| **QM_Functional Tests, error report software code** | 0 | 1 | 1 | undefined | 0 |
| **error report software code, SD_Programming** | 0 | 1 | 1 | 0 | 0 |

Table 4: Communication channels between developer's location and department for quality management

The second communication channel in table 4 describes the attributes for sending software code from "SD_Programming" to "QM_Functional Tests". In this case we have to adapt the value of function *change* (software code, QM_Functional Tests) to 1. "SD_Programming" is defined as communication autonomous with respect to this information exchange (see figure 13) and therefore sends the software code changeable.

For the third communication channel between the two locations, we just have to specify the value of function *coded* (QM_Functional Tests, error report software code) which is "undefined". "SD_Programming" is also communication autonomous with respect to this information exchange which means that the value of the affected function has to be changed to zero.

Considering now the communication autonomy degrees for the two locations, we call activity "SD_Programming" strongly autonomous and "QM_Development Guidelines" weakly autonomous. "QM_Functional Tests" is non-autonomous. In total, developers' location is more communication autonomous than the department of quality management. It is the task of the software company to evaluate if this ratio of communication autonomy is wanted or not. A software process modeller only supports this evaluation by presenting the properties in a suitable and transparent way. In general, one should take care that locations receiving a lot of data are at least half autonomous with respect to communication autonomy. This minimizes the effort for maintaining the underlying communication infrastructure.

In our example process landscape we increased the degree of communication autonomy for the Department for Quality Management concerning activities communicating with a fourth location, where the content and didactical management tasks take place. This location and the affiliated activities have not been discussed in the previous section in order to keep the discussed process landscape simple and clear. We mention it here to show a further situation where autonomy analysis turned out to be useful for the identification of improvement potential: the increase of communication autonomy for the Department of Quality Management decreased the effort for communication infrastructure maintenance.

### 4.6 Evaluation Approach

Until now, we have discussed single results of our autonomy analyses. Table 5 summarizes these results according to the classification frame introduced in section 2.4. Although only some results concerning different parts of the software process landscape are presented, the table already indicates process specific features which could be measured and checked by autonomy analysis.

| type of autonomy / granularity of autonomy | data autonomy | operation autonomy | communication autonomy |
|---|---|---|---|
| SD_Storyboard Production | 40% | | |
| activities at customer's side | | 75% | |
| activities at developers' location | | | 100%, 75%, 0% |
| activities at the same abstraction level of SD_Design | 33% | | |
| activities communicating between developers' location and Department for Quality Management | | | 66% / 33% |

Table 5: Some analysis results of the example software process landscape.

Considering the measured degrees of data autonomy, we determine e.g. weak autonomy for the set of activities belonging to "SD_Design". This conforms to the idea of design activities which have to deal with data identified already in the requirements definition phase. During the design phase this data will not be modified but arranged to form a software specification. Only activity "SD_Software Architectural Design" creates a new set of data, namely the architectural design. The activity is also responsible for changes in the architectural design and is therefore rightly identified as semi-autonomous.

The set of activities taking place at the customers' side is strongly autonomous (75%) with respect to operation autonomy. "SD_Requirements Definition" and "SD_Requirements Analysis" are elements of this set which are completely autonomous. This fact approves the accepted opinion that though these activities have to deal with concrete data, the way they work is highly creative and does not follow rules defined by others and the sequence of their definition resp. analysis steps cannot be predicted in advance.

Finally, we have analyzed the communication autonomy of activities at one location (internal communication) and between two different locations (external communication). The resultant ratio of 66% to 33% communication autonomy for the external communication between the developers' location and the Department of Quality Management demonstrates a suitable ratio. For activities at the developers' location the amount of communication and communication partners is much higher than for the Department of Quality Management because many consultations and iterations are necessary. Therefore, the developers' location should be more communication autonomous than the locations the developers communicate with in order to keep the underlying communication infrastructure simple and maintainable. A great variety of different communication channels would increase the effort and costs for maintenance. More generally, the rule formulated in section 4.4 for internal communication (the more relations to different information objects an activity - here a set of activities - has, the higher the degree of communication autonomy should be), is also valid for external communication issues.

Summarizing the evaluation approach, our analysis aims, namely the identification of inconsistencies (indicated in subsections 4.2, 4.3, 4.4), the verification of process landscape characteristics (indicated in subsections 4.1, 4.3, 4.4) and the identification of improvement potential (indicated in subsection 4.5) have been reached. We have identified some general rules for the evaluation of autonomy supporting process management when checking a software process landscape. Orocess management benefits from strongly autonomous activities because they allow on focussing on the management of less or non-autonomous activities.

## 5 Conclusions

We have presented a classification for different autonomy properties of a software process landscape at different granularity levels and to various degrees. PLL as an underlying process modelling language serves as a formal basis to model and to analyze the upper levels of a software process landscape. The discussion in the previous section shows that knowledge about the autonomy of process landscape parts supports process management. Unusually high or low degrees indicate autonomy characteristics which have to be considered in more detail. Therefore, they serve as indicator of where to start the analysis of the huge number of attribute values.

With the classification and discussion of autonomy with respect to software process landscapes, we finally have discussed all characteristics serving as precondition for a database management system to be named federated. Therefore, we also apply the term "federated" to the area of software process modelling research: we call a software process landscape federated, if the integrated process models are distributed, cooperating but autonomous to various degree.

Our future research will focus on tool support for modelling and analysis of software process landscapes with special interest in communication and autonomy properties. We have already started with the specification of a process landscape editor and an analysis component supporting PLL. It allows us to model a software process landscape at different abstraction levels, and to analyze different properties in order to identify some improvements. Furthermore, we also will review our experiences with autonomy properties by applying the analysis steps to additional and more complex software process landscapes.

## References


[ALO96] I. Alloui, S. Latrous, F. Oquendo, *A Multi-Agent Approach for Modelling, Enacting and Evolving Distributed cooperative Software Processes*, In: C. Montangero (ed.), Proceedings of the 5th European Workshop on Software Process Technology EWSPT'96, Nancy, France, October 1996, appeared as Lecture Notes in Computer Science No. 1149, pages 225-235

[Bir99] A. Birk, *Autonomous Systems as distributed embedded devices*, 1999, http://arti.vub.ac.be/~cyrano/AUTOSYS/index.html

[Chr92] G. Chroust, *Modelle der Software-Entwicklung*, Oldenbourg Verlag, 1992, in German

[CDF98] G. Cugola, E. Di Nitto, A. Fuggetta, *Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems*, In: Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan, April 1998, pages 261-270

[CMW94] S. Chawathe, H. Molina, J. Widom, *Flexible Constraint Management for Autonomous Distributed Databases*, Data Engineering Bulletin, Vol. 17, No. 2, June 1994

[CW90] I.J. Cox, G.T. Wilfong, *Autonomous Robot Vehicles*, Springer, 1990

[DF98] V. Decugis, J. Ferber, *Action selection in an autonomous agent with a hierarchical distributed reactive planning architecture*, In: K.P. Sycara, T. Finin, M. Woolridge (eds.), Proceedings of the Second International Conference on Autonomous Agents (Agents'98), ACM Press, 1998

[Gre95] M. Greenwood, *Coordination Theory and Software Process Technology*, In: Wilhelm Schäfer (ed.), Software Process Technology, Proceedings of the 4th European Workshop on Software Process Technology, EWSPT'95, Noordwijkerhout, The Netherlands, April 1995, appeared as Lecture Notes in Computer Science No. 913, pages 209-213

[GG95] G. Graw, V. Gruhn, *Distributed Modelling and Distributed Enaction of Business Processes*, In: Software Engineering – ESEC95, 5th European Software Engineering Conference, Sitges, Spain, September 1995, appeared as Lecture Notes in Computer Science No. 989, pages 8-27

[GK88] H. Garcia-Molina, B. Kogan, *Node autonomy in distributed systems*, In: Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, December 1988, pages 158-166

[GW99] V. Gruhn, U. Wellen, *Software Process Landscaping: An Approach to Structure Complex Software Processes*, International Process Technology Workshop IPTW, Villard de Lans, France, September 1999

[GW00a] V. Gruhn, U. Wellen, *Process Landscaping – Eine Methode zur Geschäftsprozessmodellierung*, In: Wirtschaftsinformatik, Vol. 4, Vieweg Verlag, pages 297-309, August 2000, in German

[GW00b] V. Gruhn, U. Wellen, *Simulating a Process Landscape*, ProSim2000 International Workshop on Software Process and Simulation Modelling, Imperial College London, July 2000

[Hän97] M. Hänßle, *Entwicklung eines Vorgehensmodells für Multimedia-Projekte*, master thesis at the University of Dortmund, Department of Computer Science, September 1997, in German

[Ham94] J. Hammer, *Resolving Semantic Heterogenity in a Federation of Autonomous, Heterogeneous Database Systems*, Technical Report USC-CS-94-569 (Ph.D. thesis), University of Southern California, Los Angeles, May 1994

[Obb95] J.H. Obbink, *Process differentiation and integration: the key to just-in-time in product development*, In: Wilhelm Schäfer (ed.), Software Process Technology, Proceedings of the 4th European Workshop on Software Process Technology, EWSPT'95, Noordwijkerhout, The Netherlands, April 1995, appeared as Lecture Notes in Computer Science No. 913, pages 79-92

[Sof99] SofTec NRW research group, *Softwaretechnische Anforderungen an multimediale Lehr- und Lernsysteme*, September 1999, http://www.uvm.nrw.de/News/AktuellesFS, in German

[Pet81] J.L. Peterson, *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, 1981
[Rei86] W. Reisig, *Petrinetze – Eine Einführung*, 2. edition, Springer, 1986, in German
[SL90] S.P. Sheth, J.A. Larson, *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases*, ACM Computing Surveys, Vol. 22, No. 3, September 1990
[VP86] J. Veijalainen, R. Popescu-Zeletin, *On multi-database transactions in cooperative, autonomous environment*, Technical Report, Hahn-Meitner Institut, Berlin, Germany, 1986