A Petri-Net-Based Process Engine

Holger Burbach
Volker Gruhn
LION GmbH
Universitätsstraße 140
D-44799 Bochum
Federal Republic of Germany
[burbach,gruhn]@lion.de

Abstract

Software process management and business process management have been areas of research for some years now. In both areas, the notions of process guidance, process control and process enaction have been intensively discussed. A component used to govern a process according to its model is usually called a process interpreter or a process engine. While the question of suitable enactable modeling languages is thoroughly discussed, the question what the key functionalities of a process engine are is hardly considered. In this article, we discuss the key functionality of a process engine and we delineate the algorithm implemented by the process engine in the FUNSOFT net approach. Moreover, we discuss the communication links between a process engine and other components of a process management environment.

Keywords
process management, process enaction, process engine, FUNSOFT nets

1 Introduction

Software process management is an area of increasing interest [6, 14, 16]. The research in this area has yield basic progress in software process modeling and analysis. Several prototypes of systems supporting software process modeling and analysis (and to some extent also process enaction) have been developed [1, 2, 15, 12, 18]. Despite some major process modeling and analysis projects, process enaction experience for large-scale software processes is hardly available.

The management of business processes has gained some attention by the use of buzzwords like *business process (re-)engineering, process innovation, lean management, and total quality management* [9, 19]. Enaction of business processes is often called *workflow management* [11].

In the following, we use the term *process* as abbreviation of *software process or business process*. To manage a process means:

- to model a process. A **process model** is understood as a description of properties of processes. It defines activities to be carried out in processes, their order, types of objects to be manipulated during the process and organizational entities involved in the process. In the software process community, the question of what a software process modeling language should look like is discussed intensively [1, 15], but no consensus of what the ideal process modeling language is has been reached. Instead of that, a

common understanding of entities to be captured in process models has been achieved [13].

- to analyze process models. Process model analysis helps to identify deadlocks within and between processes, expected process duration, and idle times of resources needed in processes.

- to enact processes, i.e. to govern a real process on the basis of prescriptions and definitions given in the model. A component governing a real process is called **process engine** in the following. Process enaction is also called process guidance (in case of rather loose coupling between process participants and the process engine), process monitoring (in case of just recording what happens without influencing the course of the process), or process control (in case of rather strict control of process participants by the process engine) in the literature [4].

While process modeling and process (model) analysis have been extensively discussed in the literature, process enaction mechanisms and functionalities are hardly subject to discussions. This is the more surprising, the more process enaction is claimed to be the ultimate goal of process management. In this article, we identify basic functionalities of process engines and we propose an implementation of these functionalities. For doing so, we relate to the FUNSOFT net approach to process management.

In section 2, we discuss the general architecture of enaction components in order to set the scene. Section 3 explains those features of the FUNSOFT net approach which determine its process engine implementation. This is done by means of a software process example. Then, section 4 discusses the process engine algorithm implemented in the FUNSOFT net approach and the communication links of FUNSOFT net process engines. In section 5, we discuss the enaction of processes following the process model described in section 3. Finally, section 6 concludes this article with reporting our experience in process enaction.

2 Architecture of a process management environment

The ultimate goal of process management is to provide guidance and assistance to people participating in real-world processes and, thus, to ensure that actual processes do not deviate from their models. This ultimate goal requires that prescriptions defined in process models are checked at process runtime in order to drive processes into the *right* direction. First of all, that means to identify which activities can be executed in the current process state. Activities which can be executed are either offered to all persons who have the permission and the qualification to participate in them (if these activities require human interaction) or they are executed automatically as soon as all their inputs are available.

Independent from the process modeling language used, process enaction has to provide some basic functionalities. In order to identify these typical functionalities we start from the reference architecture proposed by the workflow management coalition [17]. This architecture is shown in figure 1. It identifies the *Workflow Enactment Engine* as central enaction component, which communicates with:

- modeling tools (1) (in order to allow the modification of process models),

- an administration and monitoring tool (5) (in order to allow to start, interrupt, resume, finish processes),

- worklist tools (2) and other workflow engines (4) (in order to allow interoperability between workflow tools of different vendors), and

- invoked applications (3) (in order to allow the integration of existing software into new workflow models).

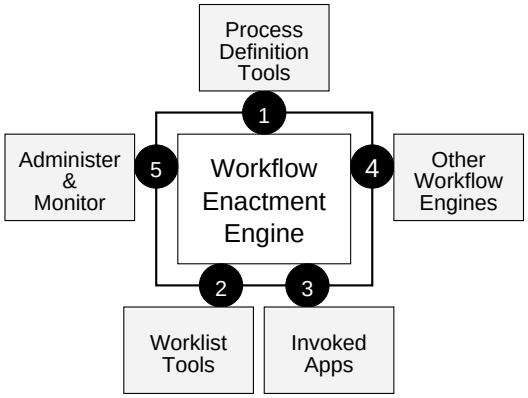

Figure 1: Reference architecture of workflow management systems

Based on the experience gathered in developing:

- the process-centered software engineering environment ALF [4],
- the software process management environment MELMAC [3], and
- the business process management environment LEU [5],

we propose a more detailed architecture for process enaction components. This architecture is still independent from the process modeling language used, but it reveils some more details than the general architecture shown in figure 1.

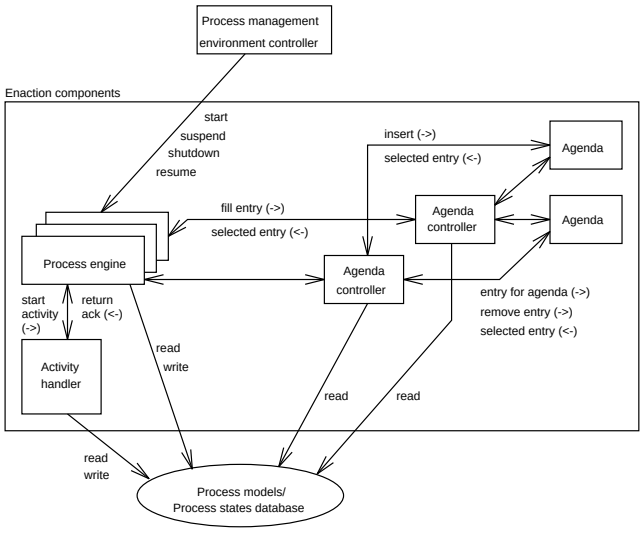

Figure 2: Architecture of enaction components

Figure 2 represents a general architecture of enaction components. Boxes represent components (i.e. module hierarchies). Arrows between boxes indicate that there is a *call*-relationship

between these components. The annotations of arrows give examples of demanded services. The component *Process engine*, for example, calls the service *start activity* from component *Activity handler*. Between some components we find double-headed arrows (e.g. between *Process engine* and *Agenda controller*). This indicates a bidirectional communication between these components. In that case, the arrow annotations indicate which service is demanded by which component (e.g. the *Process engine* asks the *Agenda Controller* to fill entries into agendas (service *fill entry*), and the *Agenda controller* returns entries which have been selected (service *selected entry*)). All these components could be implemented as individual operating system processes, which communicate with each other by message transfer and by means of a commonly used database (*Process models / Process states database*).

The main functionality of the enaction components and their interfaces to other components are discussed in the following:

- There is one **Process engine** for each process running. It can be useful to understand a large project as a set of communicating software processes in order to keep them manageable [7]. Therefore, more than one process engine may be necessary. Process engines can be started, suspended, shut down, and resumed by the *Process management environment controller*. The process engine for a process P identifies all activities of P for which all inputs are available and which, therefore, could be executed. In case of automatic activities, it starts them immediately by sending messages to the *Activity handler*. In case of manual activities, corresponding entries are sent to the process engine's *Agenda controller*. To identify executable activities, a process engine accesses the database that stores process models and process states. When an activity is finished, the state of the process concerned is modified.

- The only functionality of the **Activity handler** is to receive execution requests from all process engines, to start the requested activities, and to return whether or not the execution has been successful.

- As soon as a process engine is started, an **Agenda controller** is created. An agenda controller administrates all manual activities of its process. It is connected to the agendas of all persons who may participate in the process. The set of persons who may participate can be identified by checking permissions which are also stored in the underlying database. If a manual activity is executable, the process engine sends a message to the agenda controller. The agenda controller identifies the process participants who have the permission to participate in this activity, and sends a corresponding message to the agendas of these potential participants. If a manual activity is selected by a process participant, this information is returned to the agenda controller and from there it is passed to the process engine. The process engine starts the activity at the workstation of the participant (with the help of the *Activity handler*).

- There is one **Agenda** for each process participant. As soon as a participant logs into the process management environment, an agenda is started and automatically connected to the agenda controllers of all processes the participant has permissions to take part in.

The most sophisticated part of the enaction algorithm is implemented by the process engine. The structure of this algorithm is language-independent, but, of course, individual functions are closely related to the process modeling language used. In this article, we discuss the process engine algorithm as implemented in the FUNSOFT net approach. For that purpose, the FUNSOFT net example of section 3 highlights all FUNSOFT features which are relevant to enaction. This discussion is not only a prerequisite for understanding the process engine implementation, but it also helps to illustrate the price paid (in terms of algorithmic complexity) for a very labor-saving process modeling language.

3 The FUNSOFT net approach to process management

In the FUNSOFT net approach, which, for example, is implemented in the LEU environment [5] and in the MELMAC environment [2], process models are created by integrating data models, activity models, and organization models.

Data models describe types of objects to be manipulated and the relationships between object types. They are represented by extended entity/relationship models.

Activity models describe the activities to be executed within processes and their order. They are represented by FUNSOFT nets [8]. FUNSOFT nets are high-level Petri nets adapted to the requirements of process modeling. T-elements of FUNSOFT nets represent activities to be executed in a process. They are called agencies. An agency is activatable (i.e. it can be fired) as soon as all its input parameters are available. To fire an agency means to execute the activity represented by the agency. S-elements of FUNSOFT nets represent object stores. They are called channels. An object being stored in a channel is also called a token marking the channel (in Petri net terminology). Edges from channels to agencies describe, that an object stored in the channel is used as input parameter for the agency, edges from agencies to channels describe where output parameters of agencies are stored.

Table 1 describes how general Petri net terminology, FUNSOFT net terminology and process modeling terminology match. This survey may be helpful for readers being familiar with one of those worlds.

Process modeling terminology	Petri net terminology	FUNSOFT net terminology
activity	T-element	agency
object store	S-element	channel
object/document	token	token
object in a certain state	token marking an S-element	token marking a channel
data flow	edge	edge
activity execution	T-element firing	agency firing
executable activity	activatable T-element	activatable agency

Table 1: Mapping of process modeling, Petri net, FUNSOFT net terminology

Organization models are represented by organization charts. Data models, activity models, and organization models are integrated. Persons and sets of permissions are associated with organizational entities. Thereby, it is defined which persons are allowed to participate in which activities, and which objects of which types can be manipulated by whom. Object types are assigned to channels. A channel to which a certain type is assigned can only be marked with objects of that type.

In the rest of this article, we discuss those features of FUNSOFT nets which determine the process engine algorithm. Later, in section 4, we revert to these features by explaining how they are handled by the enaction algorithm. For a complete discussion of FUNSOFT nets we refer to [8].

In the following example, we consider a resource management process in which all resources are administrated. For this example, we focus on computer hardware and computer software. From time to time, resources are examined in order to decide whether resource replacements or purchases are necessary. If it is decided to replace a resource, either a new resource is designed, e.g. by writing a new program to replace an existing one, or a new resource is bought. Figures 4 to 7 describe this process. The structure of the example discussed is illustrated in figure 3. It shows that the process models fro *resource management* and *resource*

purchase are related by interface channels. The process model *resource test* is bound to two agencies of process model *resource management*. Finally, process model *perform resource test* is a refinement of an agency of process model *resource test*.

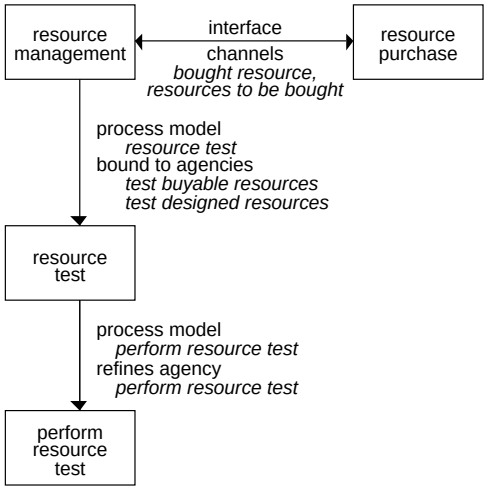

Figure 3: Structure of the software process example

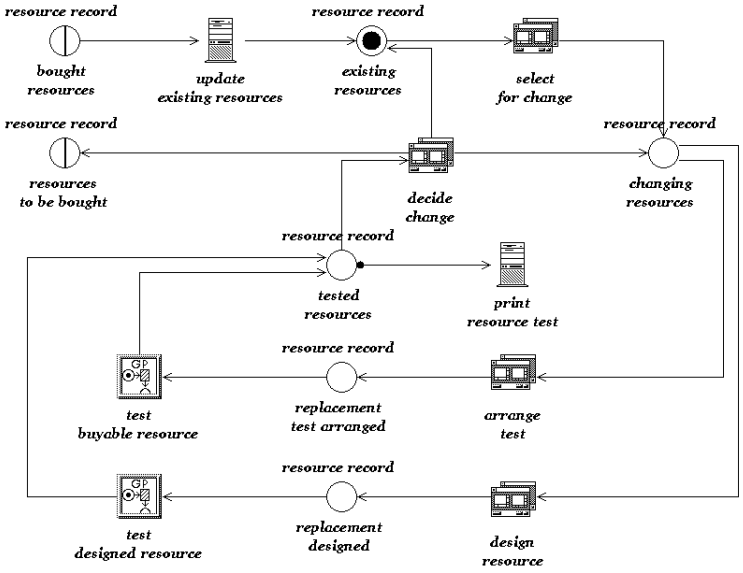

Figure 4: Resource Management

In figure 4 to figure 7 we recognize different icons representing agencies:

- agencies representing manual activities (e.g. *select for change* in figure 4),
- agencies representing automatic activities (e.g. *print resource test* in figure 4),
- agencies with processes bound to them representing the call of processes from within other processes (e.g. *test buyable resource* in figure 4; described below),
- refined agencies which are used to hide details on a lower level (corresponding to the notion of T-element refinement as defined in [10]) (not occurring in the example),

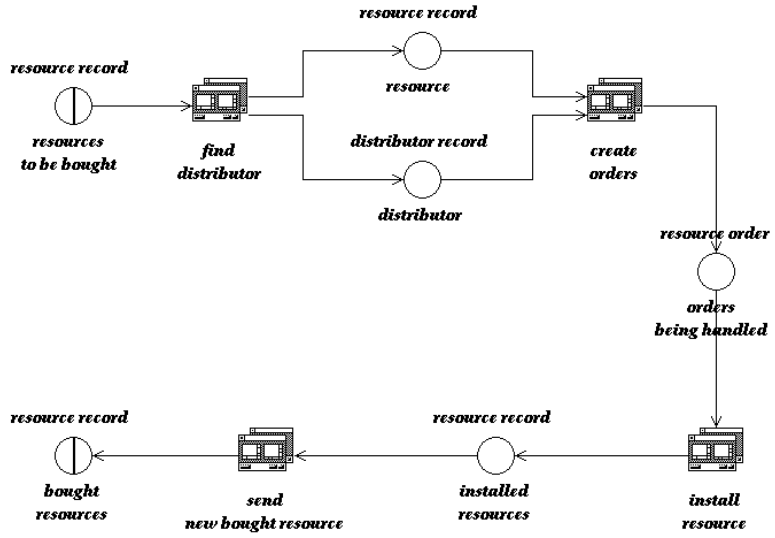

Figure 5: Resource Purchase

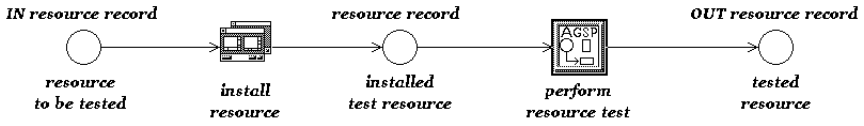

Figure 6: Resource Test

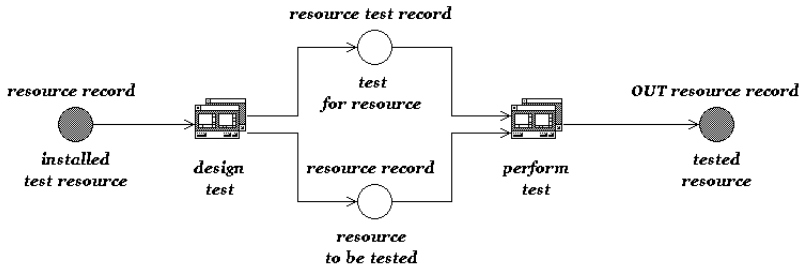

Figure 7: Perform Resource Test

- refined agencies marked as an agenda bypass. That means, all agencies in such a refinement represent closely related activities which have to be carried out by only one person (e.g. *perform resource test* in figure 6; described below).

A channel containing at least one object is represented by a circle containing another filled circle. Circles with a vertical line inside represent interface channels (see below).

The process creates and manipulates objects of several types. Since the main subject of this article is the enaction algorithm, the data model is not discussed here. The name of an object type assigned to a channel can be found above the channel. The meaning of the object types used in the example is as follows:

- An object of type *Resource Record* contains all information about a concrete resource, e.g. name, location, manufacturer. It also contains references to optional replacement resources together with the descriptions of tests of these resources.
- An object of type *Resource Distributor Record* contains all information about a distributor.
- An object of type *Test Record* contains all information about a resource test.
- An object of type *Resource Order* contains all information about an order being sent to a distributor.

All modeling concepts mentioned in the following example are explained in a glossary at the end of this section. Terms appearing in the glossary are underlined when they are used for the first time. At the start of the process, the channel *existing resources* in figure 4 contains records of all available resources.

A time-dependent predicate[1] is assigned to the agency *select for change*. The predicate causes the agency to fire once a week. Thus, the agency *select for change* appears in the agenda at a specified moment, and can be selected together with a resource record taken from the channel *existing resources*.

We assume that the value of the laddering attribute[2] for the agency *select for change* has been set to 5 when editing the process model. This means that five process participants can in parallel select resources to be changed.

When the agency *select for change* is executed, it moves the resource record selected into the channel *changing resources*. As soon as this channel contains at least one resource record, the agencies *arrange test* and *design resource* appear in the agenda of the members of the department of resource management. They either arrange a test for a resource that is bought, or they start the design of a resource that is developed internally. For each member of the department the laddering attribute of the two agencies in the process model has to be increased in order to enable them to work in parallel. After testing the bought or the internally developed resource, a resource record is written into the channel *tested resources*.

In our example, the process model *resource test* of figure 6 is bound to the agencies[3] *test designed resource* and *test buyable resource* of figure 4. This binding is based on the channel assignments as described in table 2 (i.e. the IN channel of process model *resource test* is mapped to channel *replacement test* arranged in the context of agency *test buyable resource* of figure 4).

In the process *resource test*, first the resource to be tested has to be installed. A member of the test group will be in charge of the test installation. To notify him of the installation

[1] Compare to *Predicates* in the glossary.
[2] Compare to *Laddering attribute* in the glossary.
[3] Compare to *Processes bound to agencies* in the glossary.

required, the agency *install resource* appears in his agenda. After the installation, a single tester can design and carry out the test. These two tasks are described in figure 7, which shows a refinement of the agency *perform resource test* of figure 6.

The refined agency *perform resource test* in figure 6 is marked as an agenda bypass[4]. That means that all activities of the resource test have to be carried out by only one process participant. Thus, after carrying out the *design test* activity (which is the first activity of the refinement of agency *perform resource test* described in figure 7) all other activities of this refinement are immediately started as soon as they are activatable. The process participant does not notice that different agencies are fired, because the agenda is bypassed.

When the test of a resource is finished and the record arrives in the channel *tested resources* of figure 4, the agency *print resource test* is automatically[5] executed by the process engine. The agency prints the description and the results of the latest test found in the record.

In order not to disturb the execution of the agency *decide change* the agency *print test* copies[6] the resource record.

A predicate assigned to the agency *print test* ensures that the agency is executed only once for each record arriving in the channel. In parallel to the execution of *print resource test*, the agency *decide change* is put into the agenda of the head of the department of resource management. Depending on the decision taken, the resource record is written into one of three possible postset channels:

- if a replacement resource is to be bought, the record is written into the channel *resources to be bought*,
- if the resource is not to be replaced anymore, the record is written back into the channel *existing resources*,
- if an alternative replacement resource has to be considered, the record is written back into the channel *changing resources*.

The interface channel[7] *resources to be bought* is used to exchange resource records between the resource management process (figure 4) and the purchase process (figure 5). In the purchase process, resources are bought after it has been decided to do this in the resource management process.

For each resource record arriving in the channel *resources to be bought* (figure 5), a distributor is looked for in the agency *find distributor*. The pair of resource and distributor is written into the postset channels which are read by the agency *create orders* (figure 5). Within this activity, the order is created and sent to the distributor. The object representing the order is written into the channel *orders being handled*.

For each ordered resource delivered, the agency *install resources* reads the respective object from channel *orders being handled*, creates a resource record, and writes it into the channel *installed resources* after the resource has been installed. The agency *send new bought resource* reads all the records and writes the resource records into the channel *bought resources*. This channel is accessed by the agency *update existing resources* of figure 4. Thus, the new resource record is put into the resource pool.

In the following glossary, the modeling concepts mentioned in the example above are explained in more detail. What impact these concepts have on the process engine algorithm is discussed in section 4.

[4] Compare to *Agenda bypasses* in the glossary.
[5] Compare to *Automation attribute* in the glossary.
[6] Compare to *Copy access to channels* in the glossary.
[7] Compare to *Interface channels* in the glossary.

Predicates:

A predicate is a boolean function specifying conditions which have to be fulfilled before an agency can be fired. With the help of predicates, conditions depending on the values of objects can be specified. Furthermore, predicates can be used to specify time-dependent conditions.

Laddering attribute:

Each agency has an attribute called *laddering attribute*. The value of this attribute is a positive integer with 1 as default. This value specifies how often an agency can be fired simultaneously. It eases modeling, because the degree of parallelism can easily be defined and modified.

Processes bound to agencies:

Binding a process model to an agency supports the reuse of process models and is a mechanism to structure complex process models. In a process model, which is supposed to be bound to an agency, input and output channels have to be identified explicitly. This is necessary in order to specify how the process model is integrated into the overall process model. These channels have *IN* respectively *OUT* as prefix of their names. In binding a process model to an agency, these channels are mapped to channels in the pre- and postset of the agency the model is bound to. Agencies with process models bound to them are represented by special icons (e.g. agency *test buyable resource* in figure 4). In this example, the mapping of IN and OUT channels of the process model *resource test* to the pre- and postset channels of the agencies *test buyable resources* and *test designed resource* is described in table 2.

When an agency to which a process model is bound is fired, a new son process is created. The objects to be read by the firing agency are shifted to the son process according to the channel assignments in the two process models. Then, the son process is started like any other process. When there are no activatable or active agencies left in the son process, the objects in the *OUT* channels are shifted back to the father process according to the channel assignment. These objects are treated as if they had just been created by the bound agency when terminating its firing.

Agenda bypasses:

Refined agencies have a boolean attribute called agenda bypass. Its default value is *FALSE*. If the value is set to *TRUE*, the agency is represented by a special icon (e.g. the agency *perform resource test* in figure 4). The agenda bypass attribute determines how manual agencies of the refinement are treated. The first agency of such a refinement selected by a process participant initiates that a new son process is created. The treatment of this son process resembles the treatment of processes bound to agencies. However, agenda bypasses use the same process engine for father and son process. The treatment of such a bypass son process differs from processes bound to agencies in the way activatable agencies are handled.

- First of all, only agencies belonging to the bypassed refinement may fire. Other agencies are suspended as long as the bypassed refinement is carried out.
- The second difference is the way objects are handled after the agency has fired. For channels into which objects have been written, it is checked whether they belong to the refinement or whether they are in the postset of the refined agency. If they are in the postset, the objects are immediately shifted back to the father process with all consequences for activation checks.
- The third difference is the way manual agencies are handled. If an agency of the bypassed refinement has already been selected by a process participant, then all activatable manual agencies are handled as if this process participant had selected them already. The bypass handling is terminated automatically when there is no activatable or active agency left in the bypassed refinement (i.e. the refining FUNSOFT net is dead).

Agenda bypasses ensure that certain parts of a software processes are carried out by only one process participant. The process participant is not known in advance, but the person who starts such a process part is also responsible for the rest of this process part.

Automation attribute:
Each agency has a boolean attribute called *automation attribute*. The default value is *FALSE*. When editing a process model, this attribute may be set to *TRUE*. The attribute specifies whether the firing of the agency may be started without informing any process participant about it.

Copy access to channels:
Each edge linking an agency with one of its preset channels[8] has a boolean attribute called *copy flag*. Its default value is *FALSE*. If the value of this attribute for an actual edge is *TRUE*, a small filled circle is displayed at the start of the edge (compare edge between channel *tested resources* and agency *print resource test* of figure 4). If the copy flag is set to *TRUE*, the agency accessing a channel using this edge reads a copy of the object. The object will not be removed from the channel. Conflicts between agencies with copy access are solved in a way allowing the maximum number of firings, i.e. if possible firings of agencies with copy access will be executed first. The copy flag enables several agencies to access the same object in an easy way.

Interface channels:
Circles with a vertical line inside represent interface channels. Interface channels in different process models represent only one channel. That means, access to an interface channel has side-effects to all occurrences of that interface channel.

If an object is written into an interface channel, all running processes in the models of which the interface channel occurs receive a copy of the object. If there is no running process with read access to the interface channel, the next process with read access receives the object. Interface channels allow easy data exchange between processes. A process using interface channels can be easily replaced by another process without having to change other processes, too.

4 The Process Engine Algorithm

In this section, we describe the enaction algorithm implemented in the FUNSOFT net process engine and its embedding in the other enaction components.

The task of the process engine is to interpret a process model, identify activatable activities and manage their execution. To realize this, the engine has to create agenda entries and to start agency firings while working together with the other enaction components (see figure 2).

In order to describe the process engine algorithm, we first have a look at the structure of the engine which is displayed in figure 8. An arrow from a box A to a box B means that B is used by A. Each box represents a module. The *coordination* module supervises the activities of the whole engine. The *IPC (Inter-Process Communication) receiver* module analyses the messages sent from the agenda controller or the activity handler and triggers the respective reactions of the engine. The *execution* module implements the algorithm described below. It interprets processes with the help of different basic services (summed up as a virtual component called *basic services*). They comprise submodules, e.g. for implementing attributed directed graphs as an abstract data type or for database access. The *IPC sender* module sends the results of the interpretation as messages to the respective operating system

[8] A channel s is called a preset channel of an agency t, if there is an edge from s to tt. All preset channels of an agency t are called preset of t.

	IN resource record	OUT resource record
agency *test buyable resource*	replacement test arranged	tested resources
agency *test designed resource*	replacement designed	tested resources

Table 2: Mapping of IN and OUT channels to pre- and postset of agencies to which process model *resource test* is bound

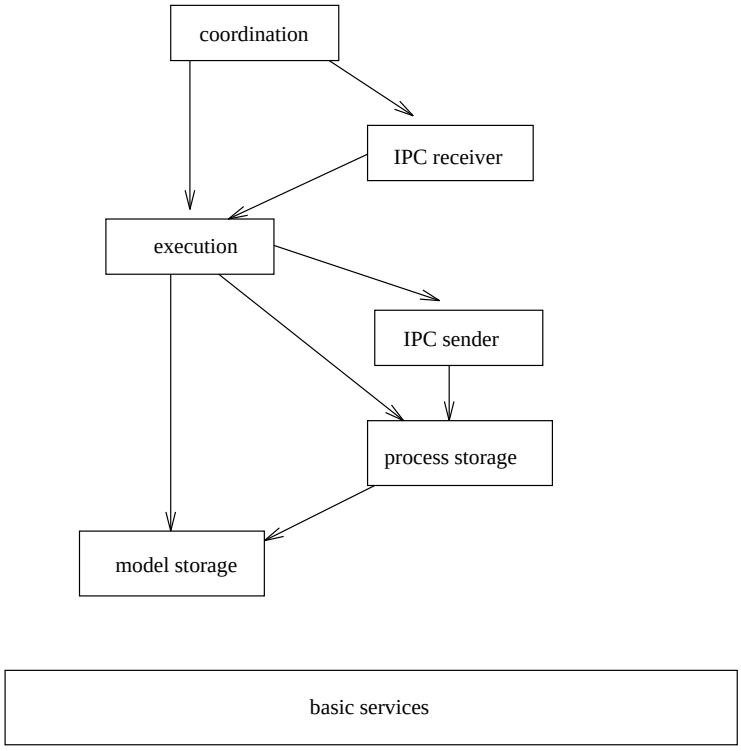

Figure 8: The Process Engine Architecture

processes implementing the other enaction components. The *model storage* module administrates the process models, the *process storage* module administrates the information about processes which are currently enacted. While process model information is not modified during process enaction (except for situations in which processes are interrupted, the process model is modified, and the process is resumed [3]), process state information is updated after each activity execution. All modules mentioned so far use the *basic services* module (for reasons of simplicity the arrows from all other modules to the *basic services* module are omitted in figure 8).

One central entity of the enaction algorithm is the agency incarnation. An agency incarnation is created whenever the preset of an agency is marked and, thus, the agency is candidate to fire. Roughly speaking, the only functionality of the enaction algorithm is to identify agency incarnations, to check if they are executable (with respect to predicates and conflicts) and to distribute them to the *right* process participants. Due to the central role of the notion *agency incarnation* we describe its structure below.

```
typedef struct
        {
            unsigned int number;
                  /* number of the incarnation */

            AGENCY_INCARNATION_STATUS ActivationStatus;
                  /* status of agency incarnation ( conflict / no conflict
                     / in agenda / firing started / firing ended ) */

            unsigned int RecipientIndex;
            char *LayoutString;
            char *DisplayString;
                  /* management data for communication with interpreters
                     performing the firing of the incarnation */

            unsigned int TotalPredCount;
                  /* number of predicate evaluations to be performed */
            unsigned int DonePredCount;
                  /* number of predicate evaluations already
                     performed */
            unsigned int TruePredCount;
                  /* number of predicate evaluations with result TRUE */

            BOOLEAN OutOfDate;
                  /* flag for management synchronization with other
                     processes; TRUE: incarnation is not valid any more
                     and should be deleted as soon as possible */

            BOOLEAN CopyConflictExcluded;
                  /* flag for storing access status when checking copy
                      conflicts;
                      TRUE: non-copy conflict already detected;
                      FALSE: ... not detected yet */

            MODEL_AGENCY BypassAgency;
                  /* optional reference to the decomposed agency in the
                     process model where the agenda bypass is marked */

            PROCESS_INDEX BoundProcess;
                  /* optional reference to the process which is bound to
```

```
                    the agency the incarnation belongs to */
    } AGENCY_INCARNATION_RECORD;
```

All agency incarnations of a process are stored in the *process storage* data structure of the engine (compare figure 9). All tasks of the engine can be described in terms of

- creating, deleting, modifying the agency incarnations, and
- copying references to the agency incarnations to various locations.

Besides the process storage data structure, there are two major places agency incarnations are referred to:

- the agenda (where agency incarnations are offered to process participants), and
- the agency incarnation list which contains references to exactly those agency incarnations which are currently checked for executability by the process engine.

In figure 9, the data flow of agency incarnations is shown. The boxes with dotted lines represent data structures which are internal to the process engine. The other boxes represent enaction components. The ovals assigned to arrows between data structures and enaction components represent routines. The routine *process agency incarnation list* (arrow from *agency incarnation list* to *agenda*), for example, means that the processing of an agency incarnation list results in manipulating the agency incarnations offered in the agenda, and that this manipulation is based on information available in the *agency incarnation list*. Another example is the routine *activation check* (arrow from *process storage* to *agency incarnation list*). When it is executed, it reads management data from the process storage data structure in order to perform activation checks. If an agency is activatable, an agency incarnation is created and appended to the agency incarnation list.

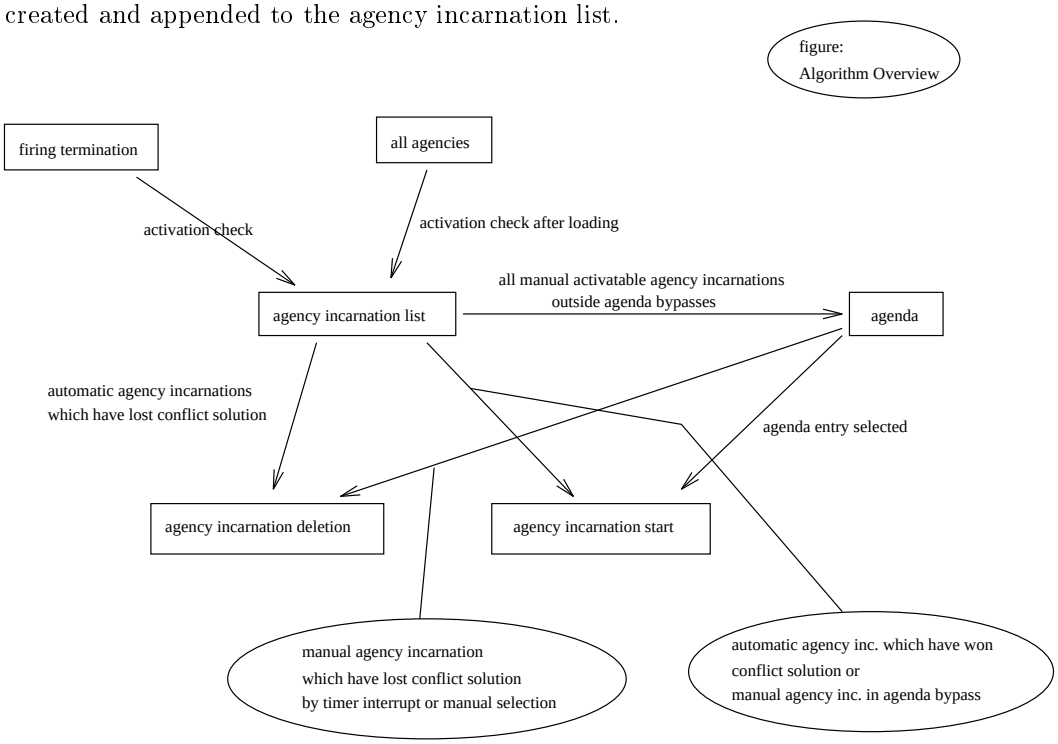

Figure 9: Agency Incarnation Data Flow

The process engine (central box of figure 9) communicates with other enaction components by message transfer. In fact, it can receive message of the following types and reacts to them as follows:

- `Start new process <process_model_id>`:
  A process following process model `<process_model_id>` is created and started. It is interpreted by a new process engine.
- `Stop process <process_id>`:
  The execution of process `<process_id>` is stopped. All data of the process being stored in the database remains there while the process engine terminates.
- `Start existing process <process_id>`:
  The execution of the already existing process `<process_id>` is resumed. The process was previously stopped with the message `stop process <process_id>`.
- `Shutdown process <process_id>`:
  First, the command `stop process <process_id>` is executed. However, before the process engine terminates, it deletes the process from the database.
- `Recognize agenda selection <agency_incarnation_id> in process <process_id>`:
  All actions necessary to execute activity `<agency_incarnation_id>` are initiated. This may mean to start a piece of software, a standard tool, or to send a message to someone.
- `Recognize timer event <agency_incarnation_id> in process <process_id>`:
  When a conflict between a manual and an automatic agency occurs, the process engine generates a timer event, which cause the alarm timer to ring after some time. This ringing of the timer causes a message to be sent to the engine. Then, the process engine solves the conflict by starting the automatic agency using the `<agency_incarnation_id>` of the process `<process_id>` as specified in the message.
- `Recognize agency termination <agency_incarnation_id>`:
  The termination of an agency is handled by the process engine. `<agency_incarnation_id>` specifies the agency incarnation used to manage the firing. The new state is calculated, and the process state database is updated.
- `Recognize data in interface channel <channel_id> in <process_id>`:
  A companion process engine has written an object into an interface channel `<channel_id>` of process `<process_id>`. The message-receiving process engine identifies if agencies became activatable by the object written into channel `<channel_id>`.
- `Check predicate in <process_id>`:
  From time to time this message is sent to each process engine in order to examine all time-dependent predicates in process `<process_id>`.

For each of the above message types, the algorithm contains a routine to handle it. These routines are based on shared subroutines. While the message types can easily be mapped to functionalities visible to process participants, the subroutines used are close to the modeling concepts of FUNSOFT nets. Thus, the relationship between message-handling routines and concept-related subroutines bridges the gap between core functionalities of process enaction and the handling of language-dependent modeling concepts.

Figure 10 depicts this relation. On the left hand side, message handling routines are shown. In the middle of figure 10, we recognize the core functions of the process engine. The arrows from the left to the middle show which message handling routines use which core functions. Certain core functions (*select start*, *distribute user*) are not directly used by the message handling routines, but they are internally called by other core functions. On the right of the figure, the modeling concepts are listed. An arrow from a message handling routine or from

a core function to a modeling concept means that the routine / function is impacted by the modeling concept. The core function *activation check*, for example, deals with predicates, laddering attributes, and agenda bypasses, but is not concerned with automation attributes, processes bound to agencies, copy access to channels, and interface channels.

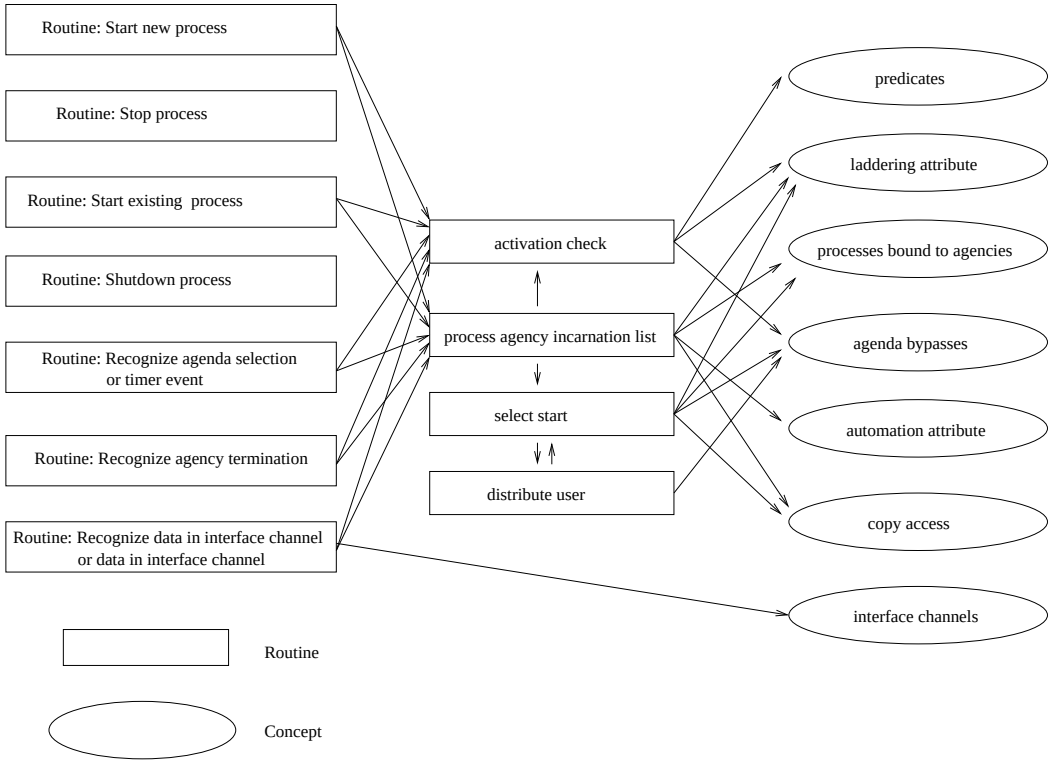

Figure 10: Algorithm routines and their relationship to modeling concepts

Within the process engine algorithm, key functionalities related to the modeling concepts are used. These are identified below:

- **Predicates:**
  Predicates have to be consulted when the activatability of agencies is checked. This key functionality is called **check predicate activatability**.
- **Laddering attribute:**
  It is necessary to handle records for each firing of an agency in order to implement the management of the laddering attribute. This record is called agency incarnation (see above). The key functionality is called **handle firing management record**.
- **Processes bound to agencies:**
  The process engine has to be able to handle more than one process in order to deal with processes bound to agencies. The key functionality is called **handle process hierarchy**.
- **Agenda bypasses:**
  In agenda bypasses, firings have to be started without displaying anything in the agenda. This key functionality is called **automatic agency execution**. In addition to this, the functionality **handle process hierarchy** is used to handle the automatic firings of the bypass as a unit within a separate process.

- **Automation attribute:**
  If the automation attribute of an agency is set in such a way that it indicates that the agency is to be executed automatically, this agency is not displayed in the agenda. Therefore, the automation attribute requires the same key functionality as agenda bypasses: **automatic agency execution**.

- **Copy access to channels:**
  For a given set of agencies, the engine has to be able to detect a firing order of maximum length with respect to the existing copy edges. This functionality is called **order firing sequence**.

- **Interface channel:**
  The process engine has to be able to communicate with other engines in order to implement data exchange via interface channels. The key functionality is called **recognize interface channel object**.

The key functionalities listed above are mentioned in the explanation of the routines for processing the messages received by the process engine. Whenever the key functionalities are needed in these routines, they are mentioned in the routine description. They are put in braces, printed in bold type and in this form appended to commands which recognize them. In the algorithm first the routines for the processing of messages are described. Then the subroutines used by this routines follow. The routines *Start new process*, *Stop process* and *Shutdown process* are left out because their actions are quite obvious.

Routine: Start existing process

This routine is called after a process has just been loaded.
Input: process

1. Execute *activation check* (see below) for all agencies of the process. Create a list of all agency incarnations of activatable agencies. (**handle firing management record**)
2. Execute *process agency incarnation list* (see below) for the agency incarnation list just created. (**handle firing management record**)
3. Execute *start existing process* for all son processes of the input parameter process. (**handle process hierarchy**)
4. If there is no agency incarnation existing in the input parameter process at the moment, then
   (a) If there are no son processes of the input parameter process then delete it. (**handle process hierarchy**)
   (b) Perform the dead net check recursively for the father process of the input parameter process. (**handle process hierarchy**)
5. else trigger the repetitive check of time dependent predicates by *Check predicate* messages.

Routine: Recognize agenda selection or timer event

This routine is called after an agenda entry is selected or the timer alarm has rung.
Input: process, agency incarnation

1. Consider all agency incarnations being in conflict to the given agency incarnation. They are either displayed in the agenda or assigned to timer alarms. Delete all those agency incarnations and their agenda entries respectively delete the pending timer alarms. (**handle firing management record**)
2. If there is a predicate assigned to the agency the incarnation belongs to, then evaluate it once again for the objects to be used for the firing. If the predicate issues FALSE, then delete the agency incarnation and leave the routine. (**handle firing management record; check predicate activatability**)

3. Execute *select start* (see below) for the input parameters.
4. Execute *activation check* (see below) for all agencies of which agency incarnations have just been deleted, for the agency of the incarnation which has just been started, and for all agencies in the postset of the channels being in the preset of the agency which has been started. Create a list of all agency incarnations created during the activation check. (**handle firing management record**)
5. Execute *process agency incarnation list* for the agency incarnation list just created.

Routine: Recognize agency termination or data in interface channel

This routine is used to process the actions of the engine after objects have been written into the postset of an agency. Therefore, it can be used for the two messages mentioned above.
Input: process, agency incarnation

1. If the process is used to manage an agenda bypass, then
   (a) Check all channels into which objects have just been written whether they belong to the part of the model the bypass has been marked for.
   (b) Shift all objects from those channels which have failed the check to the father process. (**handle process hierarchy**)
   (c) Execute *Recognize agency termination* for the father process as if the agency incarnation belonged there. (**handle firing management record, handle process hierarchy**)
2. Execute *activation check* for all agencies the preset of which has changed (i.e. an object has been written) or which have ended their firing. Generate a list of all agency incarnations created during the activation check. (**handle firing management record**)
3. Execute *process agency incarnation list* for the list. (**handle firing management record**)
4. If there is no agency incarnation existing at the moment, the net is dead. (**handle firing management record**)
5. If the net is dead and the process is bound to an agency, then shift the objects to the father process and execute *Recognize agency termination* for the respective agency incarnation used to manage the firing of the bound process. (**handle firing management record, handle process hierarchy**)
6. If the net is dead and there are no son processes of the input parameter process, delete the process and perform the dead-net-check recursively for the father process of the input parameter process. (**handle process hierarchy**)

Routine: Check predicate

This routine is used to ensure that time-dependent predicates are checked from time to time.
Input: process

1. Execute *activation check* (see below) for all agencies of the process, which have time dependent predicates assigned to them. Generate a list of all agency incarnations of activatable agencies.
2. Execute *process agency incarnation list* for the agency incarnation list just created. (**handle firing management record**)

Subroutines used by the message processing routines:

- Routine: *Activation check*
  This routine is used to check agency activatability.
  Input: process, agency
  Output: flag (activatable /not activatable), agency incarnation

1. If there are already as much agency incarnations as the laddering attribute permits, then the agency is not activatable. (**handle firing management record**)
2. If there are channels in the preset of the agency which do not contain any objects, then the agency is not activatable.
3. If there is a predicate assigned to the agency, then check it for all preset object combinations. If there is no object combination which the predicate issues TRUE for, then the agency is not activatable. (**check predicate activatability**)
4. Since the control flow has reached this point, the agency is activatable. Create a new incarnation of the agency. (**handle firing management record**)

- Routine: *Process agency incarnation list*
  This routine is used to process incarnations of activatable agencies. It generates all reactions of the process engine possible after a successful activation check.
  Input: process, agency incarnation list

  While the agency incarnation list is not empty (**handle firing management record**):

  1. Test whether there is an agency incarnation (say A) for which holds:
     incarnation A is in conflict with at least one other agency incarnation and if it is in conflict with another agency incarnation (say B), then incarnation A accesses the respective channel using copy access[9]. (**order firing sequence**)
  2. If there is an agency incarnation which has passed the upper test, then
     (a) Select this incarnation for further processing. (**order firing sequence**)
  3. else
     (a) Consider the first agency incarnation (say C) of the agency incarnation list.
     (b) If it is an incarnation of a manual agency and the process or one of its father processes is not used to manage an agenda bypass with a user already assigned to it, then select it for further processing.
     (c) If the process is used to manage an agenda bypass and a user is already assigned to it, then consider all agency incarnations, otherwise consider only those of manual agencies:
     Collect from the agency incarnation list all incarnations which are in conflict to the considered one (named C above). Enter the considered one (named C) into the collection, too. (**automatic agency execution**)
     (d) Select one incarnation from the collection in a randomly equally distributed way. Select it for further processing below. (**automatic agency execution**)
     (e) Delete all incarnations from the agency incarnation list except the one just selected for further processing.
  4. If the selected agency incarnation is an incarnation of a manual agency, then
     (a) If the process or one of its father processes is used to manage an agenda bypass with a user already assigned to it, then
        i. Consider the agency incarnation selected above for further processing. It belongs to a certain agency. If there is a process model bound to the agency, then
           A. Create a new son process bound to the agency incarnation. (**handle process hierarchy, handle firing management record**)
           B. Shift the objects to be used by the agency incarnation to the input channels of the bound process after deleting all initial objects already existing in those channels. (**handle process hierarchy, handle firing management record**)

[9] In order to perform as much firings as possible these agency incarnations have to be started before all others.

C. Execute *start existing process* for the new son process. (**handle process hierarchy**)

ii. else start the firing of the agency incarnation. (**automatic agency execution, handle firing management record**)

(b) else display the agency incarnation in the agenda. (**handle firing management record**)

5. else if the agency incarnation is in conflict to an incarnation of a manual agency and if it has never been assigned to a timer alarm since has been created, then assign it to a timer alarm,
6. else execute *select start* for the process and the agency incarnation selected for further processing.
7. Execute *activation check* (see below) for the agency of the incarnation just processed.
8. Execute *activation check* for all agencies incarnations if which have just been deleted.
9. Append all agency incarnations created by activation check during this run of the loop to the agency incarnation list.

- Routine: *Select start*
  This routine selects the proper way for the start of an agency incarnation as soon as is has been decided to perform the start.
  Input: process, agency incarnation

1. If the agency the incarnation belongs to is in a part of the process model an agenda bypass is marked for[10], then
   (a) Create a new son process for managing the agenda bypass. (**handle process hierarchy**)
   (b) Shift the objects to be used by the agency incarnation to the respective channels of the new son process after deleting all initial objects already existing in those channels. (**handle process hierarchy**)
   (c) Delete the agency incarnation.
   (d) Execute *start existing process* for the new son process. (**handle process hierarchy**)
2. else if there is a process model bound to the agency, then (**handle firing management record**)
   (a) Create a new son process bound to the agency incarnation. (**handle process hierarchy**)
   (b) Shift the objects to be used by the agency incarnation to the input channels of the bound process after deleting all initial objects already existing in those channels. Perform the shifting of the objects with respect to copy accesses. (**handle process hierarchy, copy access of channels**)
   (c) Execute *start existing process* for the new son process. (**handle process hierarchy**)
3. else start the firing of the agency incarnation. (**handle firing management record**)
4. If it is an incarnation of a manual agency, and if the process or one of its father processes is used to manage an agenda bypass, and the agency incarnation has just been selected in the agenda, then (**handle firing management record**)

[10] This condition is different to: the process or one of its father processes is used to manage an agenda bypass!

(a) If the input parameter process is not the one used to manage the agenda bypass, then find the father process used for the management.

(b) Execute *distribute user* for the input parameter process respectively the process just found, the input parameter agency incarnation, and the user who has performed the selection in the agenda.

- Routine: *Distribute user*
  This routine stores information about a user after he has selected an agenda entry belonging to an agenda bypass.
  Input: process, agency incarnation, user

  1. Assign the user to the process.
  2. Delete all existing agenda entries of the process and execute *select start* for them. (**automatic agency execution**)
  3. Execute *distribute user* for all son processes of the input parameter process. (**handle process hierarchy**)

5 An enaction scenario

After having described details about the process engine and the environment it works in we now give an example of its behavior. The example shows how the engine interprets the model described in section 3.

Figure 11 depicts the process participants, the agenda contents, and the communicating operating system processes. We identify the process engine with *leupe*, the agenda with *leuae*, the process in charge of displaying dialogues (understood as one kind of implementing manual agencies) with *leude*, the process executing automatic agency firings and evaluating predicates with *leuce*, and the agenda controller with *leuae*. The communication links are represented by lines linking the boxes which specify the processes.

Suppose that channel *existing resources* in figure 4 contains a resource record of a database. Suppose that the enaction of the process had been stopped (situation 1 in figure 11) and is going to be resumed now by the administrator (situation 2 in figure 11).

A process engine is started. The message *start existing process* is sent to it with the process as parameter. The engine reads the process and the respective model. Then, the algorithm is started (situation 3 in figure 11). Only the agency *select for change* which is activatable. The examination of the laddering attribute of the agency yields that an incarnation may be created. So the processing of the agency is continued. The predicate assigned to the agency is evaluated (situation 4 in figure 11). It yields *false*. So there is no activatable agency. Since there is a time dependent predicate, the engine does not automatically execute the command *shutdown process*, but continues evaluating the predicate from time to time. On Monday morning, the predicate yields *true*. An agency incarnation for agency *select for change* is created and inserted into the agency incarnation list. Thus, the list contains one element now. Since it is the only element no conflicts have to be solved. The incarnation is displayed in the agendas and deleted from the list, but kept in memory for further processing as soon as the agenda entry is selected. The agenda entries are displayed depending on the permissions of the different process participants. If additional process participants log in, they may not get the entry in their agenda (situation 5 in figure 11).

Suppose that the agenda entry is selected with the record of the database as parameter. The entry is deleted from all agendas. A firing is started and the corresponding dialogue is displayed. A new activation check is performed for agency *select for change* in the same way as has taken place the first time. Since the laddering attribute had been set to 5, another agency incarnation may be created. Predicate evaluation still yields *true*. An agenda entry

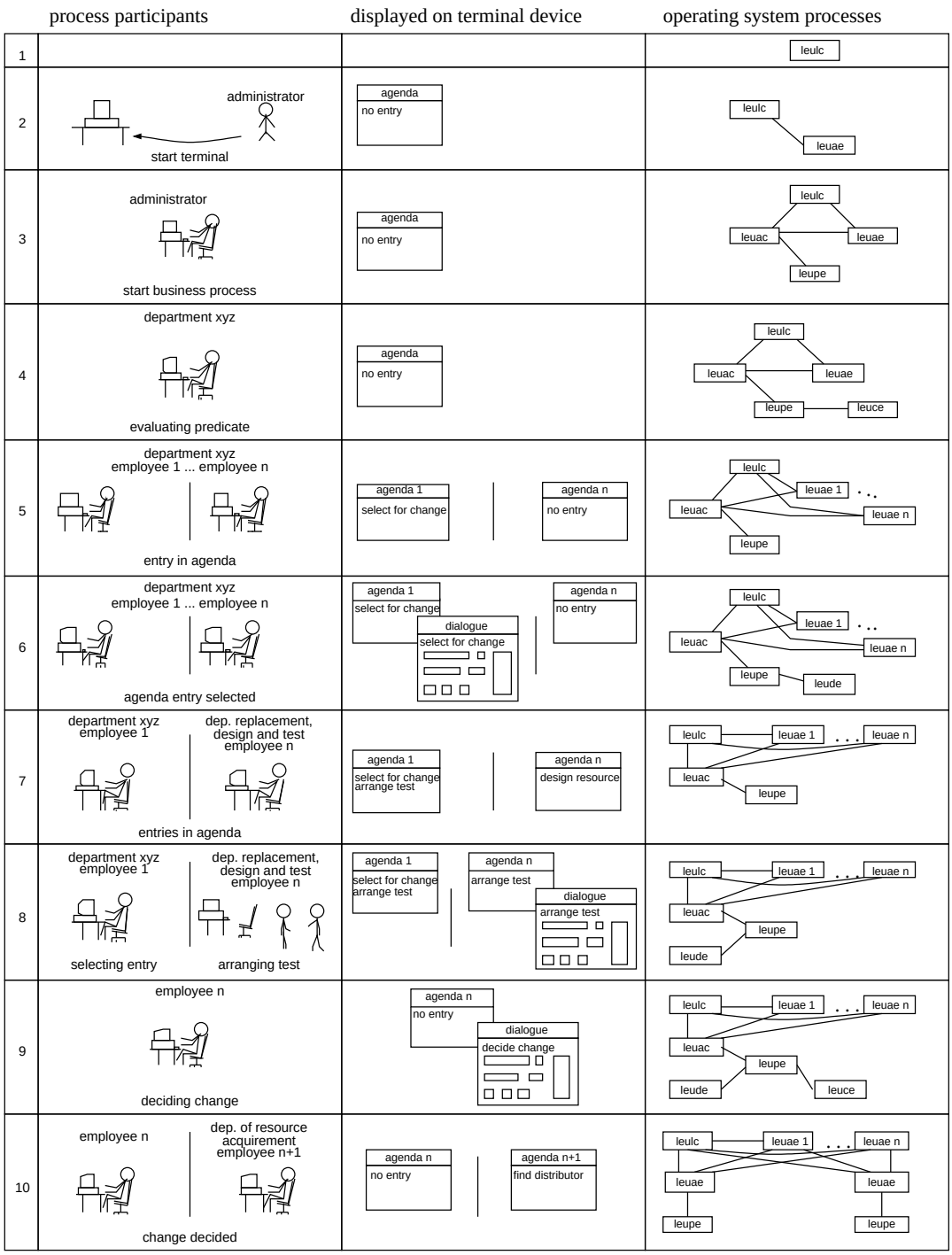

Figure 11: Scenario Situations

is created again (situation 6 in figure 11). This may happen again and again until the fifth agency firing is started. During the following activation check the examination of the laddering attribute yields that no further agency incarnations may be created. No additional processing is performed after detecting this.

When the dialogue of agency *select for change* dealing with the database record is terminated, this record is written into channel *changing resources*. An activation check is performed for the agencies *select for change*, *arrange test* and *design resource*. After the termination of the dialogue just mentioned, the related agency incarnation is deleted. So the activation check of agency *select for change* yields that an agency incarnation may be created again. The predicate evaluation still yields *true* (similar to situation 4 in figure 11). There are no conflicts to solve for agency *select for change*. Thus, it the agency displayed in the agenda again (situation 5 in figure 11).

The examination of the laddering attribute of agency *arrange test* yields that an agency incarnation may be created. There are no predicates to be evaluated. Thus, an incarnation is created. Since agency *arrange test* is a manual agency, it is displayed in the agenda. The same holds for agency *design resource*. This way we get three agenda entries after the termination of agency *select for change*. They are displayed according to the permissions of the different process participants (situation 7 in figure 11).

Suppose that a member of the department of resource management selects the agenda entry of agency *arrange test* together with the database resource record as parameter. Since agency *arrange test* is in conflict to agency *design resource*, its entry is removed from all agendas in addition to the selected entry. The firing of agency *arrange test* is started. The incarnation of agency *design resource* is deleted. For both agencies an activation check is performed. We suppose that channel *changing resources* had been filled with additional objects by parallel firings of agency *select for change*. So the presets of both agencies *design resource* and *arrange test* are filled. Since their laddering attributes still permit the creation of agency incarnations and there are no predicates new agency incarnations and agenda entries are created (situation 8 in 11).

The next interesting part of the algorithm is used when firing agency *test buyable resource*. This is done after a test of the replacement database has been arranged and when it is subject to be performed. As soon as the agenda entry of this agency is selected, the firing is executed by creating a new son process. The objects are shifted to channel *resource to be tested* in figure 6. Then, an activation check is performed for the created process as already described. The operating system processes communicating with the process engine do not recognize the creation of son processes. Thus, the resulting situation is similar to, for example situation 5 in figure 11.

After the new database has been installed and the resource record has been written into channel *installed test resource*, an entry for agency *design test* in figure 7 appears in the agenda in the way already described. When the entry is selected, the algorithm recognizes that agency *perform resource test* is refined and marked as agenda bypass. Therefore, a new son process is created to handle the bypass. Again the creation of the son process is not recognizable from outside the process engine. The user who has selected the agenda entry is assigned to the new process in order to enable the process engine to automatically display all dialogues of the new process on his terminal without displaying any entry in the agenda. The database resource record in channel *installed test resource* is shifted to its companion channel in the new son process. The son process is executed in the way already described.

As soon as the test of the new database is finished, the resource record is written into channel *tested resource*. When writing the object, the algorithm recognizes that channel *tested resource* does not belong to the agenda bypass. The object is shifted back to the father process which is bound to agency *test buyable resource* in figure 4. Since the net of the agenda bypass process is dead the bypass process is deleted. Then the dead net check

is carried out for the father process of the process just deleted. This is the bound process which is dead, too. Therefore the resource record is shifted from channel *tested resources* in figure 6 to channel *tested resources* in figure 4. Then the bound process is deleted, too.

When the object reaches channel *tested resources* after the firing of agency *test buyable resource*, activation checks are carried out for this agency as well as for the agencies *decide change* and *print resource test*. The laddering attribute of agency *test buyable resource* permits the creation of a new agency incarnation. However, the preset of the agency is empty. This prevents further processing of the agency. The laddering attribute of agency *print resource test* permits the creation of a new agency incarnation, too. Its preset is filled. The predicate ensuring that agency *print resource test* is executed only once for each record yields *true*. Thus, a new agency incarnation is created and inserted into the incarnation list which had been empty before.

In the same way, a new incarnation for agency *decide change* is appended to the incarnation list now consisting of two elements.

At this point, the algorithm realizes that agency *print resource test* fulfills the special requirement for copy conflicts, which permits the agency to be executed at once. When starting the firing, the incarnation is removed from the incarnation list. The other incarnation remaining in the list is handled as already described for manual agencies resulting in situation 9 in figure 11.

When agency *decide change* terminates its firing the resource record is written into channel *resources to be bought*. When writing the object, it is realized that this channel is an interface channel. Therefore the object is passed to channel *resources to be bought* in figure 5. There, the object is handled as if it was created during the termination of an agency firing. We suppose that the respective process had been started resulting in situation 10 in figure 11.

6 Experiences and Conclusions

In using FUNSOFT nets for process modeling for about eight years, we came along different requests for more comfortable process modeling. Some of these requests resulted in modifications of the modeling tools, a very few had some influence on the modeling concepts.

In FUNSOFT nets we tried to keep the number of modeling constructs as small as possible as long as people were able to model things with the available constructs. Nontheless we had to integrate some extensions which were raised in a project, in which about 40 modelers described all business processes of several housing construction and administration companies. These extensions (agenda bypasses, processes bound to agencies) were not needed in the earlier software process modeling experiments, but they eased modeling of certain situations substantially.

These extensions meant to also extend the enaction algorithm. The initially simple FUNSOFT net enaction algorithm had to be extended to cope with additional modeling concepts. The decision to integrate different process engines by message passing on the hand and by using a common database also meant to extend the enaction algorithm (in this case by communication facilities between process engines).

The extensions discussed were integrated, but resulted in an enaction algorithm, which - even though its internal structure remains clear - became more and more difficult and, thus, less extensible. Based on this experience, it was rather difficult to decide between modeling comfort (usually demanding new modeling constructs) and ease of enaction (demanding that as few constructs as possible have to be taken into consideration).

Summing this up, we believe that the number of modeling concepts should be kept as small

as possible. Comfortable modeling should not justify any extension. But, of course, the modeling language must allow to express business process situations as they are (without burdening them under too much syntactical details demanded by the modeling language). Only then, it is reasonable to let people, who know *their* processes, take part in process modeling.

References

[1] S. Bandinelli, A. Fugetta, and S. Grigolli. *Process Modelling In-the-Large with SLANG.* In *Proceedings of the* 2^{nd} *International Conference on the Software Process - Continuous Software Process Improvement*, pages 75–83, Berlin, Germany, February 1993.

[2] W. Deiters and V. Gruhn. *Managing Software Processes in MELMAC.* In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 193–205, Irvine, California, USA, December 1990.

[3] W. Deiters, V. Gruhn, and H. Weber. *Software Process Evolution in MELMAC.* In Daniel E. Cooke, editor, *The Impact of CASE on the Software Development Life Cycle*. World Scientific, Series on Software Engineering and Knowledge Engineering, 1994.

[4] J.-C. Derniame and V. Gruhn. *Development of Process-Centered IPSEs in the ALF Project. Journal of Systems Integration*, 4(2):127–150, 1994.

[5] G. Dinkhoff, V. Gruhn, A. Saalmann, and M. Zielonka. *Business Process Modeling in the Workflow Management Environment LEU.* In P. Loucopoulos, editor, *Proceedings of the* 13^{th} *International Conference on the Entity-Relationship Approach*, pages 46–63, Manchester, UK, December 1994. Springer. Appeared as Lecture Notes in Computer Science no. 881.

[6] C. Ghezzi, editor. *Proceedings of the* 9^{th} *International Software Process Workshop*, Airlie, VA, US, October 1994.

[7] G. Graw and V. Gruhn. *Process Management in-the-Many.* In W. Schäfer, editor, *Software Process Technology - Proceedings of the* 4^{th} *European Software Process Modeling Workshop*, pages 163–178, Noordwijkerhout, Netherlands, April 1995. Springer. Appeared as Lecture Notes in Computer Science 913.

[8] V. Gruhn. *Validation and Verification of Software Process Models.* In A. Endres and H. Weber, editors, *Proceedings of the European Symposium on Software Development Environments and CASE Technology, Königswinter, FRG*, pages 271–286, Berlin, FRG, June 1991. Springer. Appeared as Lecture Notes in Computer Science 509.

[9] K. Hales and M. Lavery. *Workflow Management Software: the Business Opportunity.* Ovum Ltd., London, UK, 1991.

[10] P. Huber, K. Jensen, and R.M. Shapiro. *Hierarchies in Coloured Petri Nets.* In *Proc. of the* 10^{th} *Int. Conf. on Application and Theory of Petri Nets*, pages 192–209, Bonn, FRG, 1989.

[11] S. Jablonski. *Functional and Behavioral Aspects of Process Modeling in Workflow Management Systems.* In *Connectivity '94 - Workflow Management - Challenges, Paradigms and Products*, pages 113–133, Linz, Austria, October 1994. R. Oldenbourg, Vienna, Munich.

[12] G.E. Kaiser, P.H. Feiler, and S.S. Popovich. *Intelligent Assistance for Software Development and Maintenance. IEEE Software*, 5(3):40–49, May 1988.

[13] J. Lonchamp. *A Structured Conceptual and Terminological Framework for Software Process Engineering.* In *Proceedings of the* 2^{nd} *International Conference on the Software Process - Continuous Software Process Improvement*, Berlin, Germany, February 1993.

[14] L. O'Conner, editor. *Proceedings of the* 2^{nd} *International Conference on the Software Process - Continuous Software Process Improvement*, Berlin, Germany, February 1993.

[15] B. Peuschel and W. Schäfer. *Concepts and Implementation of a Rule-based Process Engine.* In *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[16] W. Schäfer, editor. *Software Process Technology - Proceedings of the* 4^{th} *European Workshop on Software Process Modelling*, Noordwijkerhout, The Netherlands, April 1995. Springer. Appeared as Lecture Notes in Computer Science 913.

[17] K.D. Swenson. *Interoperability Through Workflow Management Coalition Standards.* In *Proceedings of the Workflow 1994*, pages 185–197, San Jose, US, August 1994.

[18] R.N. Taylor, F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. *Foundations in the ARCADIA Environment Architecture.* In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–13, Boston, 1988. Appeared as Software Engineering Notes, 13(5), November 1988.

[19] B. Warboys. *Reflections on the Relationship Between BPR and Software Process Modelling.* In P. Loucopoulos, editor, *Proceedings of the* 13^{th} *International Conference on the Entity-Relationship Approach*, pages 1–9, Manchester, UK, December 1994. Springer. Appeared as Lecture Notes in Computer Science no. 881.