Confirmation Engine Design Based on PSI Theory

Ondřej Dvořák
Faculty of Information Technology
Czech Technical University
Prague, Czech Republic
ondrej.dvorak@fit.cvut.cz

Robert Pergl
Faculty of Information Technology
Czech Technical University
Prague, Czech Republic
robert.pergl@fit.cvut.cz

Petr Kroha
Faculty of Information Technology
Czech Technical University
Prague, Czech Republic
kroha@informatik.tu-chemnitz.de

***Abstract*—Design & Engineering Methodology for Organisations (DEMO) is a methodology for (re)designing and (re)engineering of organisations. Having a strong theoretical background in the PSI theory (Performance in Social Interactions), DEMO deals with communication and interaction between subjects (human beings) that play a crucial role within all company processes. Advanced information systems are used to support processes and communications. In these systems, confirmations are very usual patterns. In this paper, we present a design of a confirmation engine based on the transaction axiom of the PSI theory. We discuss a theoretical background of this engine, our implementation, and how this module fits into the IT infrastructure.**

I. Introduction

During design of a commercial software system CoRiMa developed by COPS GmbH, a requirement for so-called *confirmation principle* came up. CoRiMa is a multi-user client-server application and an application platform at once. Several users communicate their demands to a server that cooperates with a risk-management banking system. CoRiMa contends with a processing of various operations carrying out necessary information (deals, foreign exchanges, balances, etc.). This information is necessary for an underlying risk-management system to do proper calculations. Since much of this information is critical for risk calculations, it cannot appear in the target risk-management system unapproved by privileged users, so-called *confirmators*. This integral principle is called the *confirmation principle*.

DEMO sees organisations as systems of actors that are in a social interaction. The actors perform so-called coordination acts. The transaction axiom of the General PSI theory declares that these acts are performed in patterns called *transactions* [6]. Within transactions, the commitments of subjects (human beings) are raised. As Dietz [6] claims: *"Carrying through a transaction is a "game" of entering into and complying with commitments."*.

In: W. Schmidt, A. Fleischmann, L. Heuser, A. Oberweis, F. Schönthaler, C. Stary, and G. Vossen (Eds.): Proceedings of the Workshop on Cross-organizational and Cross-company BPM (XOC-BPM) co-located with the 17th IEEE Conference on Business Informatics (CBI 2015), Lisbon, Portugal, July 13, 2015

The next important principle of DEMO is an operating principle stating that: *"Subjects enter into and comply with commitments regarding the products/services they bring about in coordination"* [6]. This principle clearly sees enterprises as social systems.

We believe the confirmations in CoRiMa are essentially transactions of DEMO. Users of CoRiMa are mostly human beings acting in roles of subjects. They enter into a commitment regarding certain object affirmation. The affirmation is a specific outcome of the confirmator's decision. It may be a yes/no result or something more complex, like a scale value or even a free comment.

We argue that the confirmation principle can benefit from the transaction axiom and the quoted operating principle of the General PSI theory in DEMO. In this paper, we sum up the key expectations on the confirmation principle in CoRiMa, and we map them to the DEMO methodology while outlining a possible design and a basic implementation.

The remainder of this paper is organised as follows. We begin with an overview of DEMO in section II. Then we describe what CoRiMa is, and what is the purpose of a confirmation engine in section III. We map the confirmation principle to DEMO and bring a suitable naming of all its fundamentals in section IV. On the top of the confirmation principle we introduce the confirmation engine in section V. We show how it fits the CoRiMa architecture, and we present code snippets of its main components. Finally, we present an example of a deal confirmation in section VI, we mention the related work in section VII, and we conclude the paper in section VIII.

II. DEMO Overview

DEMO stands for „Design & Engineering Methodology for Organisations", and we consider it to be the leading methodology of Enterprise Engineering discipline, as it is grounded in theories, mainly in the system ontology of Bunge, teleology, and the theory of communicative act of Habermas. At the same time, its benefits for the practical use has been proven, as documented e.g. in [8] or [1].

For a brief description of DEMO, we take a help of Op't Land and Dietz [8]:

"A complete, so-called essential model of an organisation consists of four aspect models: *Construction Model (CM)*, *Process Model (PM)*, *Action Model (AM)*, and *State Model (SM)*. The CM specifies the composition, the environment

and the structure of the organisation. It contains the identified *transaction kinds*, the associated *actor roles* as well as the information links between actor roles and transaction banks (the conceptual containers of the process history). The PM details each transaction kind according to the *complete transaction pattern*. In addition, it shows the structure of the identified business processes that are trees of transactions. The AM specifies the imperatively formulated *business rules* that serve as guidelines for the actors in dealing with their agenda. The SM specifies the *object classes*, the *fact types* and the declarative formulations of the *business rules*."

For the purpose of this paper, we need to introduce briefly the Construction Model. It shows a network of identified transaction kinds and the corresponding actor roles. For example, transaction kind `T01` (Figure 1) delivers a business service to actor role `A00`. `A00` is called the *initiator* (consumer) and `A01` the *executor* (producer). The executor of a transaction is marked by a small black diamond on the actor's role box. The solid line between `A00` and `T01` is the initiator link; the solid line between `A01` and `T01` is the executor link. Figure 1 also shows that another actor role (`A07`) needs to have access to the history of transactions `T01` (production facts as well as coordination facts (e.g., status „requested", „promised", „stated", „accepted")). This is represented by the dashed line between `A07` and `T01`. However, the diagram notation is not important for our purpose, we will just utilise the underlying concepts.

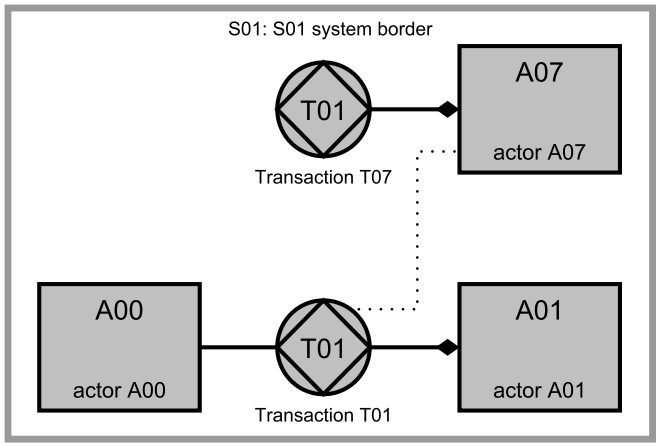

Fig. 1. Typical constructs of a DEMO Construction Model [8]

The DEMO Process Model reveals details of the transactions with the respect to complete transaction pattern. The so-called *standard transaction pattern* is depicted in Figure 2. The „happy flow" consisting of `request`, `promise`, `state` and `accept`, as is depicted in Figure 3, is also called the *basic transaction pattern*. Apart from the happy flow, `decline` may happen instead of `promise`, and `reject` may happen instead of `accept`. Then, a new attempt may be made, or `quit`, resp. `stop` may end the transaction unsuccessfully. This logic is automatically included in all DEMO transactions, which is one of the reasons why the models are so compact – it would need a lot more diagram elements to express the transaction pattern of every transaction kind using a flowchart-like notation.

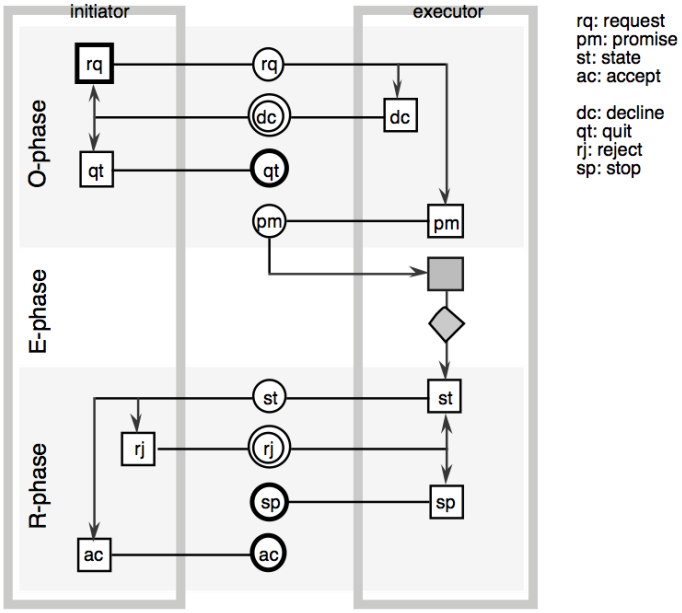

Fig. 2. DEMO Standard Transaction Pattern [2]

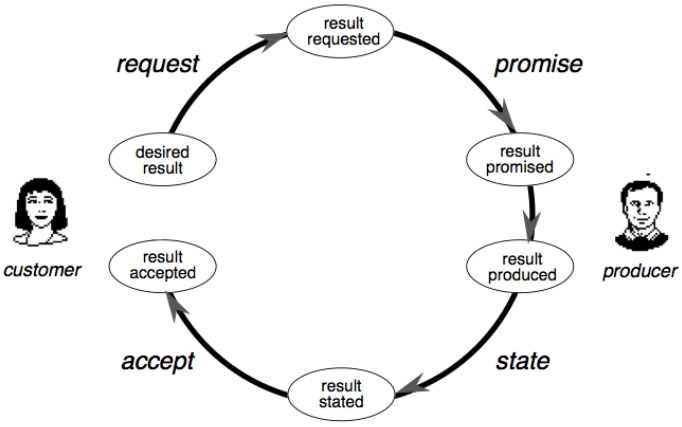

Fig. 3. Happy flow of a transaction [2]

However, real situations may become even more complicated. It is addressed by the *complete transaction pattern* in Figure 4. It incorporates the notion of *revocation* – an actor may want to „take back" their act done before[1]. If that is allowed by the other party, the transaction „rolls back" to the desired state.

The PSD (Process Structure Diagram) notation is used to depict the PM. However, we will not present it here, as we do not need it for our purpose.

[1]In the DEMO theory, nothing can disappear, so the original fact remains in the fact bank, however, the transaction flow is changed.

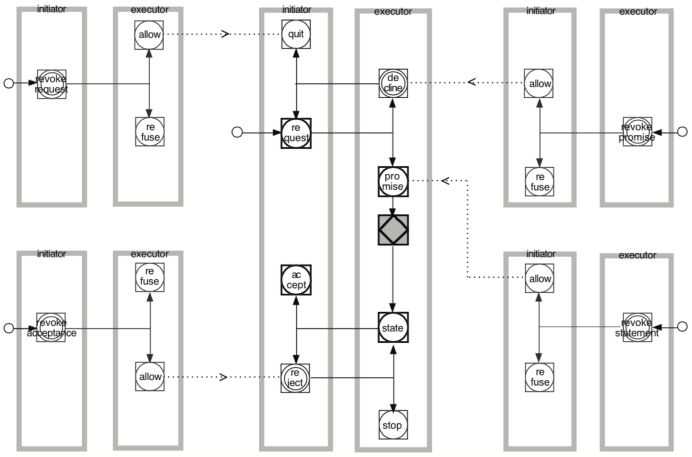

Fig. 4. DEMO Complete Transaction Pattern [3]

III. CoRiMa and the Confirmation Principle

CoRiMa is an application platform for development and execution of financial-oriented applications. It consists of a runtime and a framework. Runtime is a client-server application that can host and execute plugins composed of modules and features offered by the framework. The plugins themselves implement a client and a server core that is executable within a corresponding part of the runtime. Each plugin maintains certain instruments from a financial market. Its client-core views the instruments, its server-core transfers the instruments into an underlying risk-management system. For example, rates of currencies are displayed in the client-core, whereas the physical deposition of rates happens within the server-core.

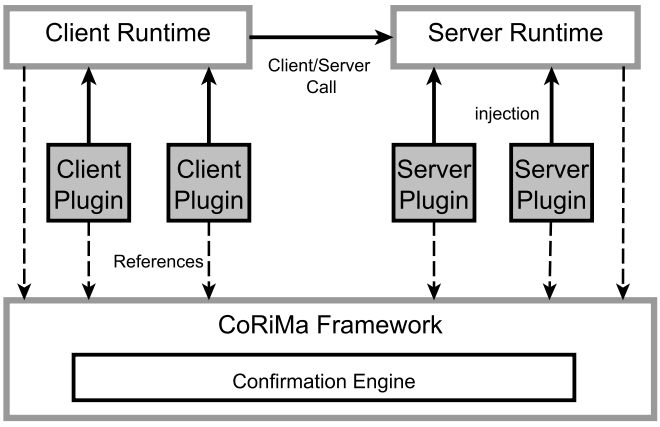

Fig. 5. CoRiMa architecture

Various instruments with which the plugins operate must undergo a specific confirmation process before landing in the risk-management system. This process must go hand in hand with the so-called *confirmation principle*. The principle guides the process through predefined mandatory and optional steps. The use-cases for confirmations can significantly differ among the instruments. Some instruments can be confirmed automatically after a given time is elapsed. Some can be changed by a confirmator, while others cannot. Some instruments can even be cancelled and consequently result in a cancellation of the confirmation. All in all, the demands for various kinds of confirmations can be custom-built.

The general idea is to design a unit in the CoRiMa framework that would meet expectations regarding the confirmation principle. It should be clearly defined and easily applicable by a developer facing a request on confirming a given instrument. We call this unit *confirmation engine*, and we elaborate it in the section V. The basic architecture of our approach is presented in Figure 5.

IV. DEMO and the Confirmation Principle

A range of similarities can be identified when comparing the confirmation principle and DEMO. Nevertheless, these can only be seen from a certain point of view. In this section, we analyse these similarities, and we evaluate a possibility of the confirmation principle benefiting from them. We follow up this section by outlining a DEMO-based design and an implementation of the confirmation engine as a software unit supporting the confirmation principle.

A. Requester, Confirmator, Confirmation Pattern, Confirmation, and Affirmation

In DEMO, a transaction is a sequence of acts involving two actor roles, an initiator and an executor. The sequence must respect the transaction pattern defining legal acts during a transaction processing. A transaction is started by the initiator who performs a request to have a desired product stated. Such an act leads to the status `requested`. The executor reacts by promising the product, and the transaction advances to the status `promised`. This act represents a guarantee that the executor is going to produce the requested product within an execution phase of the transaction[2]. As soon as the executor finishes the production, it performs the `state` act, thus bringing the transaction into the status `stated` meaning the product is ready. At this moment, the initiator can accept the product and thus the transaction ends. However, by rejecting the product, the initiator could express its disagreement with the product, and the transaction goes back to the status `stated`. Such a decision would result in a negotiation to clarify mutual expectations on the product.

In CoRiMa, the *confirmation* is a sequence of acts, too. It involves two subjects, a *requester* and a *confirmator*. As well as in DEMO, the sequence must respect a given *confirmation pattern*. The sequence is started by a requester performing a request to create a desired product. However, it is important to realize what the product actually is. The confirmation principle insists on preventing a subject to persist an object that has not been seen and subsequently confirmed by anybody else. Thus, the object itself is not a product, the sole *affirmation* pertaining to an object is. In case the confirmator has in mind to analyze the created object, it can give a promise and proceed to an

[2]Technically, it may happen that the promise is revoked later by the executor, however, we do not deal with this possibility at this place.

execution phase. The production of an affirmation must result in a state act performed by a confirmator. Again, it is highly important to realize what the state practically means in a context of the confirmation principle. The confirmator must have at least an option to express its agreement or disagreement with an object, yet there can be much more. Nevertheless, this outcome must be a property of the affirmation. As soon as the confirmation is brought to the status `stated`, the initiator can proceed with either accepting or rejecting the stated affirmation.

Let us summarize the initial mapping between DEMO and the confirmation principle. The confirmation principle is generally a PSI theory of DEMO adopted to the needs of CoRiMa. The notions of *initiator* and *executor* in DEMO are represented by the notions of *requester* and *confirmator* in the confirmation principle. The *transaction pattern* and the *confirmation pattern* both define appropriate steps of a process and relations among them. *Transaction* resp. *confirmation* is then a walkthrough in the corresponding pattern. *Product* and *affirmation* are the interests regarding of which subjects (requester and confirmator) enter into a commitment by initiating the transaction (resp. confirmation).

B. Confirmation Kind and Affirmation Kind

In DEMO, the transaction kinds imply a specific flow of allowed acts and the corresponding states of the transaction.

The same may be applied to a *confirmation kind*. The confirmation kind is based on a specific confirmation pattern. The main difference is that within the confirmation engine, the acts are performed explicitly (e.g., by calling a method `request()`). One can only define what should be an evidence that the act just happened (e.g., sending an email to its counterpart)[3]. If a certain act is senseless or not allowed for the given confirmation kind (e.g., if the requester is not allowed to disagree with a created affirmation), this must be declared. This is similar to specifying action rules in the Action Model of DEMO.

A transaction in DEMO (hopefully) results in a product successfully delivered. Each transaction kind has a specific product kind as its result. Similarly, each confirmation kind has a specific affirmation kind as its result. An affirmation is the product of a confirmation. Thus a confirmation (hopefully) results in an affirmation successfully produced. The affirmation itself is a set of properties and their corresponding values. All these properties represent the confirmator's notion regarding an object which is tasked to approve (e.g., a property called *result* can transfer an information whether the confirmator is fine with the object or not). Nevertheless, one has to introduce all required properties before a confirmation process is started. This is done by an affirmation kind which defines it.

To sum up, a transaction kind in DEMO corresponds to a confirmation kind. The product kind is represented by an affirmation kind. The product is the affirmation itself. It expresses a decision of a confirmator whether the given object is approved or not.

[3]This corresponds to facts in the PSI theory described above, however we do not go into such detail here.

DEMO	Confirmation Principle
Transaction Pattern	Confirmation Pattern
Transaction Kind	Confirmation Kind
Transaction	Confirmation
Product Kind	Affirmation Kind
Product	Affirmation
Initiator	Requester
Executor	Confirmator

TABLE I
MAPPING BETWEEN DEMO AND THE CONFIRMATION PRINCIPLE

C. Revocations

In DEMO, revocation is a situation when the initiator or the executor change their minds after performing a certain act. DEMO defines four different revocation patterns for `request`, `promise`, `state` and `accept`, as can be seen in Figure 4. If allowed by the other side, each of them can lead to a transaction step in the *standard transaction pattern*, which constitutes the complete transaction pattern. It means that not all transaction kinds allow to revoke an already performed act. Revocations may be even technically impossible (e.g., shredding of a document).

Confirmations in CoRiMa exhibit the same behaviour. For instance, shortly after a requester requests an affirmation of a given object, it can change its mind and want to take the request back. It depends on the confirmation kind if such a step is allowed, and how the confirmation engine should react. Since these situations are expectable, we want to support them, too. Thus, we have to introduce revocation patterns into the confirmation principle. To make it work, we have to define their specification for each confirmation kind individually.

D. The Confirmation Principle Summary

In this section, we clarified how DEMO and the underlying PSI theory can be mapped to the confirmation principle. We identified a mapping between a requester and an initiator, and a mapping between a confirmator and an executor. We showed that a confirmation kind maps to a transaction kind, and an affirmation maps to a product in DEMO. DEMO models successfully describe the whole enterprise process logic by transaction kinds derived from the complete transaction pattern. Thus, we argue that confirmations based on the same complete transaction pattern can handle all necessary situations of various confirmation processes. Table I presents the overview of the resulting mapping.

V. THE CONFIRMATION ENGINE

Let us recall that the confirmation engine is supposed to be a unit in CoRiMa supporting the confirmation principle. It should be universal, thus independent on objects it is used for. It must allow an easy creation of a new confirmation kind for a given type of an object. In the end, it must be able to integrate

new confirmation kinds and thereby enable the confirmation principle for any corresponding object.

In this section, we elaborate a possible design of the confirmation engine. We show how it fits the architecture of CoRiMa.

Since the implementation itself is out of scope of this paper, we only present basic code snippets in C# of its most interesting parts. We follow a similar scenario as we did while describing the mapping of DEMO to the confirmation principle in section IV. We start with an overall architecture by identifying the key components. We subsequently investigate these components deeper. We discuss a design of a requester, an executor, and a confirmation service. We show how the notions of the confirmation pattern, a confirmation, and an affirmation are embodied in these components. We continue with a confirmation and an affirmation kind, and we explain how they are integrated in the whole engine. We end up with revocations by determining their proper involvement.

A. The Overall Architecture (The Confirmation Clients and the Confirmation Service

CoRiMa is a client-server application platform. Users interact only with a CoRiMa client. In a suitable user interface, they observe and maintain financial-based data. All the data changes are communicated to a server that stores them into an underlying risk-management system. The confirmation is a process (a sequence of acts) between two users, the first one in a role of a requester and the second one in a role of a confirmator. The requester performs a data change, and a confirmator approves it before leaving it in an underlying risk-management system.

Figure 6 covers the usual confirmation process. A user in a requester role is connected to a CoRiMa client. It performs a change of a certain data object. The object is sent to a server within a request call. The server just publishes a temporary created object for the corresponding confirmator. The object is transferred to the CoRiMa client and displayed within an user interface designed for making confirmations.

A user in a role of a confirmator confirms the given object (i.e., it creates the affirmation) and sends it back to a server. Finally, the confirmed object is saved to the risk-management system by the confirmation service.

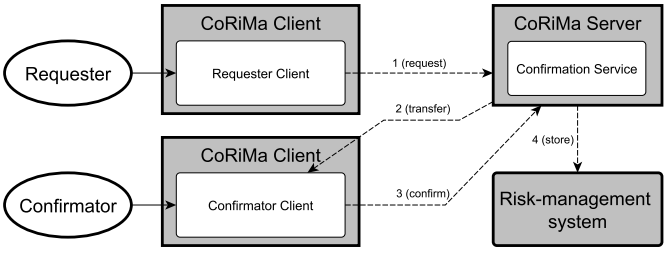

Fig. 6. General Confirmation Process

It is obvious that a unit supporting such a confirmation process must take a part on both, client, and server side. On the client side, two components must be introduced. The first component comes from the requester, the second from the confirmator point of view. Let us call them *Requester Client* and *Confirmator Client*. These components represent actors in DEMO, the initiator and the executor. They provide methods for performing all valid acts. Technically, `RequesterClient` and `ConfirmatorClient` are classes that mediate all clients' demands to the server (see the C# code snippet below). For a simplicity, access modifiers and the detailed implementation are omitted. The *ConfirmatorClient* will be implemented analogously.

```
class RequesterClient<...>
 ...
 void Request(object obj) {
    // Client-server call requesting an
    // affirmation of an object
 };
 void Quit(object obj) { ... };
 void Accept(object obj) { ... };
 void Reject(object obj) { ... };
 ...
}
```

On the server, a *confirmation service* component will take a part. It is a component responsible for reacting on acts performed on the client. It basically contains two subcomponents, *Confirmation Provider* and a *Confirmation Handler*. The Confirmation Provider publishes end points to which the client-server calls are communicated. The calls are directly forwarded to a Confirmation Handler that maintains the whole confirmation process. Technically, these subcomponents are implemented by two classes, `ConfirmationProvider` and `ConfirmationHandler`. Again, we omit the implementation details and concentrate on the general structure only. We list the code snippets below.

```
class ConfirmationProvider
 ...
 void OnRequested(
          ...,
          object obj) {
    // Forward of a request to a
       Confirmation Handler
 };
 void OnQuit(...) { ... };
 void OnAccepted(...) { ... };
 void OnRejected(...) { ...};
 void OnPromised(... { ... };
 void OnDeclined(...) { ... };
 void OnStated(...) { ... };
 void OnStopped(...) { ... };
 ...
}
```

```
class ConfirmationHandler {
 ...
```

```
void Register(...) { }

void Request(
       ...,
       object obj) {
   // Request confirmation of a given id
};
void Quit(...) { ... };
void Accept(...) { ... };
void Reject(...) { ... };
void Promise(...) { ... };
void Decline(...) { ... };
void State(...) { ... };
void Stop(...) { ... };
...
}
```

In this section, we clarified key components of the whole confirmation engine architecture. A requester and a confirmator roles, we identified in subsection IV-A, are represented by `RequesterClient` and `ConfirmatorClient`. Both mediate demands to the server side. The confirmation principle itself is realised using `ConfirmationProvider` and `ConfirmationHandler`. These constitute a Confirmation Service maintaining the entire confirmation logic. In the next section, we describe how the confirmation pattern, a confirmation, and an affirmation fits into the Confirmation Service.

B. The Confirmation Pattern in the Confirmation Service

The confirmation in CoRiMa is a sequence of legal acts (see subsection IV-B). Their legality is given by the current context of a confirmation (who is executing the act, which acts have already passed, and which object is a subject of the confirmation). Such a sequence must respect a certain confirmation pattern. We already pointed out that we suppose the standard transaction pattern in DEMO should satisfy all needs for confirming various objects. Thus, the valid subsequences of acts in a confirmation are driven by a standard transaction pattern.

In fact, the Confirmation Service, we built up in the previous section, is merely an implementation of this pattern. It contains a method for each possible act in the standard transaction pattern. It behaves in accordance to that pattern and evaluates if a given act (method call) is legal in the current situation. If so, it moves the confirmation into another state.

Each confirmation in CoRiMa is of a specific kind, a confirmation kind. The purpose of each confirmation is a creation of an affirmation. Technically, an affirmation is just a set of properties and their values. It is an instance of a class specifying these properties. Confirmation Service only have to know how to deal with such an instance. When a confirmator performs a `state` act, a new value of this instance is delivered within a client-server call. In case the Confirmation Service persists this value, a proper way of persisting it must take place. We already explained in subsection IV-B) that an affirmation kind specifies the information contained in an affirmation (being its product). It means that if a Confirmation Service knows beforehand the affirmation kind, it can persist the affirmation appropriately.

Now, we are ready to discuss which information is actually needed by a Confirmation Service to serve the acts performed on the client (e.g. `request` or `state`). It must at least know the confirmation kind, and the affirmation kind, and the object being confirmed. To identify this information smoothly, we introduce a concept of registrations. The confirmation kind is paired with the corresponding affirmation kind and registered with a confirmation handler. All client-server calls use the identifier of a registration to manifest in which confirmation kind they are interested in. The methods of `ConfirmationHandler` are enhanced as follows:

```
...
void Register(
     int registrationId,
     IConfirmationKind ck,
     IAffirmationKind ak);

void Request(
     int registrationId,
     object obj);
...
```

In this section, we outlined that the Confirmation Service implements the complete transaction pattern from DEMO. It registers pairs of confirmation kinds and affirmation kinds, and it uses these registrations for handling the client-server calls. To recognise the registered pair, `registrationId` is placed in each client-server call. In the next section, we show the purpose of a confirmation kind and an affirmation kind, and we describe how the Confirmation Service uses them.

C. The Role of a Confirmation Kind and an Affirmation Kind

The primary responsibility of a confirmation kind is to manage the process flow, namely to:

- decide if the act is allowed or not,
- specify whether the act is tacit[4] or not.

Next, in CoRiMa, it is necessary to support *custom actions* tied to different acts in different confirmation kinds. For example, sending an email after giving a `promise` is a custom action that is applied only for some confirmation kinds. Thus, the second responsibility of a confirmation kind is to specify these custom actions.

Let us consider an interface of a confirmation kind in the code-snippet below. If each method is implemented, the Confirmation Service can take it into account. For instance, if the `promise` happens, the Confirmation Service can execute `PromiseAct` defining a custom action (e.g. sending an email).

Another example is a tacit execution. Let us consider a situation when the `accept` is tacit, and the `reject` is not

[4]A tacit execution of an act means that the act is performed automatically, without being explicitly requested.

allowed. If a confirmator performs the `state`, the Confirmation Service can react by directly moving the confirmation into the state `accepted` and execute the custom method `AcceptAct`.

```
interface IConfirmationKind {
 Act RequestAct { get; set; }
 Act QuitAct { get; set; }
 Act AcceptAct { get; set; }
 Act RejectAct { get; set; }
 Act PromiseAct { get; set; }
 Act DeclineAct { get; set; }
 Act StateAct { get; set; }
 Act StopAct { get; set; }
}

class Act {
 bool IsAllowed { get; }
 bool IsTacit { get; set; }
 Action<...> Execute { get; set; }
}
```

An affirmation kind is implemented as a class having a set of custom properties characterising the affirmation. The requester and the confirmator operate with those properties while performing acts (e.g., if the requester is deciding whether the stated affirmation is acceptable or not).

To sum up, confirmation kinds define the process flow and the custom behaviour of the Confirmation Service. Affirmation kinds hold custom properties, whose values constitute the affirmation.

D. Revocations in the Confirmation Service

In CoRiMa, a revocation is a situation when a requester or a confirmator change their mind just after performing an act. If we want to support such situation, we have to enhance the confirmation senrvice.

In DEMO, four revocation patterns exist in the complete transaction pattern (section II). We believe that by extending the implementation with revocation patterns, we cover all exceptional cases within the confirmation principle. The changes will affect all the identified components. We do not pursue this topic any further here, let us just show the relevant extension of `RequesterClient`:

```
class RequesterClient<...>
 ...
 void RevokeRequest(object obj);
 void RevokeAccept(object obj);
 ...
}
```

E. Confirmation Engine Summary

We conclude our description of the confirmation engine with Figure 7 illustrating how the engine fits into the CoRiMa infrastructure.

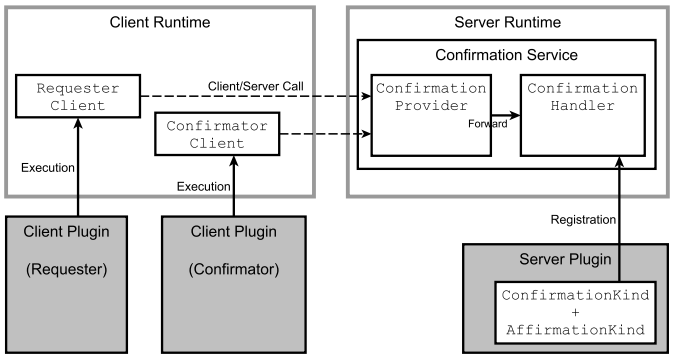

Fig. 7. Confirmation engine in the CoRiMa infrastructure

The confirmation engine consists of the following key components:

- `RequesterClient`
- `ConfirmatorClient`
- `ConfirmationProvider`
- `ConfirmationHandler`

The client plugin dedicated for requesters uses `RequesterClient` to mediate its requests to the server. The client plugin for confirmators handles its demands via `ConfirmatorClient`.

The server publishes end points using `ConfirmationProvider` that forwards all requests into `ConfirmationHandler`. This is a central unit handling the entire confirmation process. Without an extra information, this unit is useless. It must be supplemented by `ConfirmationKind` paired with `AffirmationKind`. Both are registered with `ConfirmationHandler` during the CoRiMa server startup.

VI. An Illustrative Example

Let us demonstrate how the engine can be used to address a concrete problem.

Let us consider deals and their confirmations. Each creation of a deal must be confirmed before the deal ends up in the risk-management system. It means that the created deal is not automatically persisted in the risk-management system. Instead, it stays in a temporary state until a privileged confirmator approves it. In case the confirmator is not satisfied with the deal, he can deny the creation and drop a comment explaining why.

First, we clarify the whole problem deeper and write down rules for deal confirmations. Second, we show how the rules influence the implementation. We conclude with a description of how the entire integration works.

A. Rules of the Deal Confirmation Case Study

To successfully integrate the confirmation of deals into CoRiMa, we have to scrutinize the problem and specify the rules first.

- Rule (1): The confirmator is not allowed to refuse a request to perform a confirmation. Once he is asked to confirm a deal, he has to produce a result.
- Rule (2): No revocations are allowed. Neither the requester, nor the confirmator can change their minds after performing an act.
- Rule (3): As soon as the confirmator approves or denies the deal, the confirmation is over. The requester cannot express a disagreement with the confirmator's decision.
- Rule (4): An email notification is sent to a confirmator once a request is performed.
- Rule (5): The confirmator does not have other option but to approve or deny a deal. He can just optionally leave a comment.

We described that the confirmation engine (respectively its confirmation handler) does not support any confirmation process on its own. This knowledge is represented by injected confirmation kind and an affirmation kind. Both the kinds have to be implemented and registered in the confirmation handler. Thus we have to implement the following:

- `DealConfirmationKind`, a class representing a confirmation kind. The interface `IConfirmationKind` must be implemented by this class.
- `DealAffirmationKind`, a class representing an affirmation kind.

The way the classes are implemented directly depends on the rules stated before. Let us examine each rule and describe how it is implemented:

- Rule (1): The fact that the confirmator cannot refuse a confirmation request means that the `promise` act is performed tacitly. The implementation of `DealConfirmationKind` must consider it and set a value of its `PromiseAct` property to `IsTacit=true`.
- Rule (2): No revocation is allowed, thus all the properties regarding the revocations must be set accordingly (e.g. `RevokeRequestAct.IsAllowed=false`).
- Rule (3): Because the requester cannot actually react on a decision done by a confirmator, his `accept` is tacit[5]. That means the property `AcceptAct` must reflect this.
- Rule (4): Sending of an email is a custom action. It must follow the `request`. Thus the method `Execute` of `RequestAct` must be implemented to send an email to the confirmator.
- Rule (5): The options available to a confirmator determine directly how `DealAffirmationKind` will look like. It must be a class having two properties. One of a type `enum`, having two possible values: `Approved` and `Denied`. The second property representing a comment, so most probably of a type `string`.

In this part, we described how each of the rules affects the implementation of `DealConfirmationKind` and `DealAffirmationKind`. In the next part, we register them in a confirmation handler. We show how the requester and the confirmator undertake the expected deal confirmation process.

[5]This means that the result phase will practically lack the accept act, however, formally, it is present, so the transaction axiom is still valid.

B. Deal Confirmation Process

Let us demonstrate the entire deal confirmation process now. It is depicted in Figure 8.

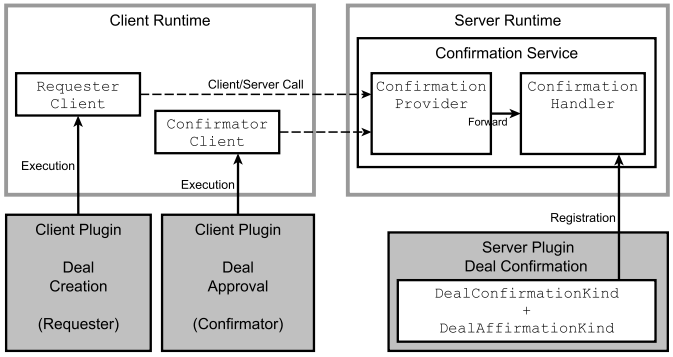

Fig. 8. Deal Confirmation Process

During a CoRiMa server startup, the server-side Deal Confirmation plugin is loaded and `DealConfirmationKind` together with `DealAffirmationKind` are registered in the confirmation handler under a specific identifier `Id`, let us assume the number `248`. This identifier must be known to both the client and the server-side.

A pair of client-side CoRiMa plugins must be present to provide user interfaces for the deal confirmations. One plugin aimed for the creation of the deal, the other is dedicated for its approval. Once a requester creates a deal, the deal must be confirmed. It is done by pressing a Request button. The command related to this button is implemented as follows:

```
OnRequestExecuted(Deal deal) {
  var rct = new RequesterClient(248);
  rct.Request(deal);
 ...
}
```

This method results in a client/server call to the confirmation provider. The provider forwards the call to the confirmation handler asking for a confirmation identified by `248`. The confirmation handler searches for the confirmation kind registered under the number `248`. Subsequently, the handler changes the status of the confirmation to `requested`, and it executes the method `Execute` of the corresponding `RequestAct`. Since `RequestAct` of the `DealAffirmationKind` has to send an email to the confirmator, it is done consequently. Because the `PromiseAct` is tacit, the handler instantly adjusts the status of the confirmation to `promised`.

Now, the whole process waits for the confirmator's act. The client-side plugin Deal Approval responsible for making approvals is designed for this purpose. Its implementation is more or less analogical to the previous one, thus we do not elaborate it any further.

VII. Related Work

Formetis is a Dutch commercial company that has developed and successfully applied a DEMO Engine (also called a DEMO Processor) for its customers, as documented in [5] and [7]. The Engine enables the construction and execution of DEMO models. It is also implemented in the .Net platform using the C# language. The engine itself is independent on the technological environment. It has been used to implement desktop workflow applications and an educational application http://demoworld.nl.

The first DEMO Engine application in production is a case management system for a company that provides energy and utility services. The customers – citizens – are active co-producers of the service by providing information, coordination of some tasks, approval of decisions, etc. The contract covers issues such as type of services provided, costs, costs calculation procedures, conditions for payments, letters, mails, instructions for the subcontractors, etc. There is an enforced compliance to legal procedures, policies, conditions, approvals etc.

Formetis's solution is currently the leading industrial solution of a software system based on DEMO and its underlying theories. However, its scope is much broader than our confirmation engine. It is a complete general workflow engine, while our confirmation engine is a rather lightweight, compact module for CoRiMa fully focused on its specific task. Overall, technically, it would be perfectly possible to use Formetis' DEMO engine as a confirmation engine, however, from the software engineering point, often lightweight focused solutions may be preferred, as is the case of CoRiMa.

Agreement Technologies [9] may seem similar to our approach, however there is a fundamental difference. Our approach addresses confirmations of ontological transactions, which cannot be automated [4], while agreement technologies are focused on automated agents, i.e. the infological level.

VIII. Conclusion and Future Work

In this paper, we mapped the PSI theory to the confirmation principle. We designed and outlined an implementation of a confirmation engine, as a unit supporting the confirmation principle in the CoRiMa framework. The described implementation has been fully implemented and deployed to a testing environment of a banking institution. As we do not have feedback from the operation yet, we cannot perform a thorough evaluation. However, the validation by the customer passed successfully. It proves that the concepts were designed adequately, and the mapping fulfilled its goal.

Due to space limitations, we covered just the essential aspects of the implementation. The goal of this paper was to show that utilising the PSI theory for a design of the confirmation engine in CoRiMa brought considerable benefits. Mainly, designing the confirmation pattern by mapping the corresponding concepts from the PSI theory results in a guarantee that all possible confirmation situations are covered.

Since CoRiMa is a running project, we now want to focus on improvements of the engine. We want to emphasize the implementation of simple and self-explaining components to speed-up an integration of requests on confirming certain objects.

Acknowledgment

We would like to express a special thanks to mr. Robert Lukas who came up with a concept for a confirmation principle in CoRiMa. Indeed, this concept evoked an idea that such a kind of request perfectly fits DEMO methodology. This opens up a space for further mapping of the theoretically grounded DEMO to a commercial software requirements.

This paper was written with the support of the SGS15 CTU grant no. 118/OHK3/1T/18.

References

[1] Céline Décosse, Wolfgang A. Molnar, and Henderik A. Proper. What Does DEMO Do? A Qualitative Analysis about DEMO in Practice: Founders, Modellers and Beneficiaries. In Wil van der Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, Clemens Szyperski, David Aveiro, José Tribolet, and Duarte Gouveia, editors, *Advances in Enterprise Engineering VIII*, volume 174, pages 16–30. Springer International Publishing, Cham, 2014.

[2] Jan L. G. Dietz. *Enterprise ontology: theory and methodology*. Springer, Berlin; New York, 2006.

[3] Jan L.G. Dietz. *THE ESSENCE OF ORGANIZATION - AN INTRODUCTION TO ENTERPRISE ENGINEERING*. Sapio bv, 2012.

[4] Jan L.G. Dietz. *Red garden gnomes don't exist*. Sapio Enterprise Engineering, 2013.

[5] Sérgio Guerreiro, Steven J. H. van Kervel, André Vasconcelos, and José Tribolet. Executing Enterprise Dynamic Systems Control with the Demo Processor: The Business Transactions Transition Space Validation. In Wil van der Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, Clemens Szyperski, Hakikur Rahman, Anabela Mesquita, Isabel Ramos, and Barbara Pernici, editors, *Knowledge and Technologies in Innovative Information Systems*, volume 129, pages 97–112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[6] Dietz Jan and Hoogervorst Jan. Theories in Enterprise Engineering Memorandum - PSI.

[7] Steven JH Van Kervel, John Hintzen, Tycho van Meeuwen, Joost Vermolen, and Bob Zijlstra. A professional case management system in production, modeled and implemented using DEMO. In Molnar, Wolfgang A., Henderik A. Proper, Jelena Zdravkovic, Peri Loucopoulos, Oscar Pastor, and Sybren de Kinderen, editors, *Complementary proceedings of the 8th Workshop on Transformation & Engineering of Enterprises (TEE 2014), and the 1st International Workshop on Capability-oriented Business Informatics (CoBI 2014)*, volume 1182, Geneva, Switzerland, July 2014. Technical University of Aachen.

[8] Martin Op 't Land and Jan L. G. Dietz. Benefits of Enterprise Ontology in Governing Complex Enterprise Transformations. In Wil van der Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, Clemens Szyperski, Antonia Albani, David Aveiro, and Joseph Barjis, editors, *Advances in Enterprise Engineering VI*, volume 110, pages 77–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[9] Sascha Ossowski. *Agreement Technologies*. Springer, 2013.