Model-Driven Refinement of Software Architectures with π-ARL

FLAVIO OQUENDO
University of South Brittany
VALORIA – BP 573 – 56017 Vannes Cedex
FRANCE

Abstract: In model-driven development, architecture descriptions and their refinements are explicitly represented and manipulated as models. π-ADL and π-ARL are formal (executable) architecture description and refinement languages providing architecture-centric modelling constructs. When applied, refinement actions expressed in π-ARL refine architecture description models described in π-ADL outputting new refined models described in π-ADL. Enabling model-driven refinement of software architectures is a new challenge for the model-driven development of complex software systems. This paper gives an overview of π-ARL and illustrates the expressiveness and usefulness of model-driven refinement with π-ARL through a case study.

Key-Words: Stepwise Refinement, Model-Driven Development, Software Architectures

1 Introduction

Software architecture has emerged as an important subdiscipline of software engineering. Enabling stepwise model-driven refinement of software architectures is a new challenge for the model-driven development of complex software systems.

All forms of engineering rely on models to build systems. Models are at the heart of the ArchWare[1] approach. Indeed, ArchWare supports full model-driven development, i.e. the system models have sufficient detail to enable the generation of a full software application from the models themselves. Indeed, "the model is the code", i.e. the focus is on modelling and code is mechanically generated from models. In ArchWare, models are architecture-centric (run-time) models. They are executable and support analysis and refinement.

In ArchWare model-driven development, architecture description and its stepwise refinement are explicitly represented and manipulated as models. In stepwise refinement, software architectures are designed such that:

- the architecture description starts at a high level of abstraction,
- subsequent refinement steps reveal further details,
- each refinement step decreases underspecification, in the sense of "yet unfinished" parts, of the previous architecture description.

An important feature of refinement languages in model-driven development is the integration of description and transformation constructs into the same framework, so that a smooth transition can be made from an abstract, platform-independent model, down to a concrete, platform-specific model of the system. Thus, an abstract – platform-independent – architecture description can be refined to a concrete – platform- specific – architecture description following a Model-Driven Architecture approach [4]. Figure 1 depicts the ArchWare model-driven approach for describing and refining architectural models [16].

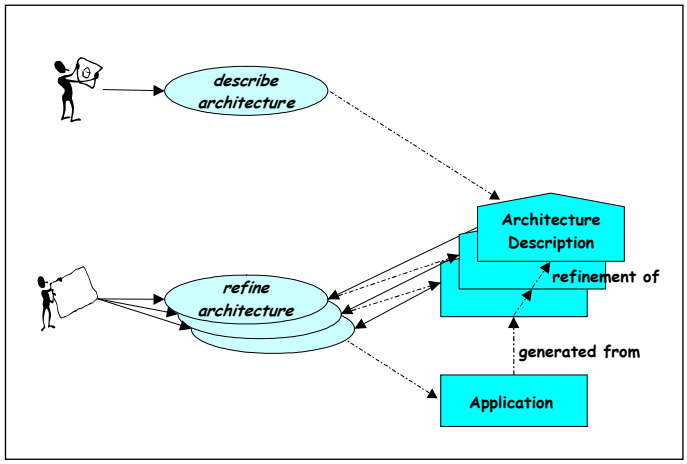

Figure 1. Model-Driven Development in ArchWare

In order to support model-driven architecture refinement, a formal language should be able to cope with underspecification and very high-level descriptions. During successive stages, it should be possible to complete and refine the initial model, until the intended system is precisely described. At each stage the intermediate model should be reified; in fact, the outcome of each refinement action should provide a guiding principle for the decisions in the next refinement action.

Formally, some refinement step carries a proof obligation, since it should be verified that the refined model does not introduce any behaviour that is excluded by the higher-level model. It is a proof obligation of each refinement action to formally verify that these assumptions hold.

In order to support model-driven refinement of software architectures, a novel architecture refinement language has been designed in ArchWare: π-ARL [14]. π-ARL is an executable model-driven architecture refinement language providing architecture-centric refinement primitives and constructs for their compositions. When applied, refinement actions expressed in π-ARL refine architecture description models described in π-ADL [13][15] outputting new refined models described in π-ADL.

This paper gives an overview of π-ARL (including a brief presentation of π-ADL), and then illustrates the expressiveness

[1] The ArchWare European Project is partially funded by the Commission of the European Union under contract No. IST-2001-32360 in the IST-V Framework Program (2002-2005).

and usefulness of model-driven refinement with π-ARL through a case study. The case study addresses the description and refinement of the software architecture of a login system. It covers a simple, yet frequent, model-driven architecture refinement that would be problematic for most other refinement techniques. Several refinement steps are performed, each dealing with a simple refinement, in order to achieve a more concrete architecture. The remainder of the paper is organised as follows. Section 2 briefly describes the ArchWare architectural languages π-ADL and π-ARL. Section 3 presents the case study. Section 4 compares π-ARL with related work and section 5 concludes this paper.

2 Architectural Languages

The ArchWare architectural languages comprise an architecture description language, the π-ADL, and an architecture refinement language, the π-ARL. A detailed description of π-ADL, illustrated with examples, is provided in [13] and a detailed description of π-ARL, also illustrated with examples, is provided in [14].

2.1 π-ADL Architecture Description Language

π-ADL [13] supports description of software architectures from a runtime perspective. In π-ADL, an architecture is described in terms of components, connectors, and their composition. Figure 2 depicts its main constituents[2].

Components are described in terms of external ports and an internal behaviour. Their architectural role is to specify computational elements of a software system. The focus is on computation to deliver system functionalities.

Ports are described in terms of connections between a component and its environment. Their architectural role is to put together connections providing an interface between the component and its environment. Protocols may be enforced by ports and among ports.

Connections are basic interaction points. Their architectural role is to provide communication channels between two architectural elements. Connections may be unified to enable communication.

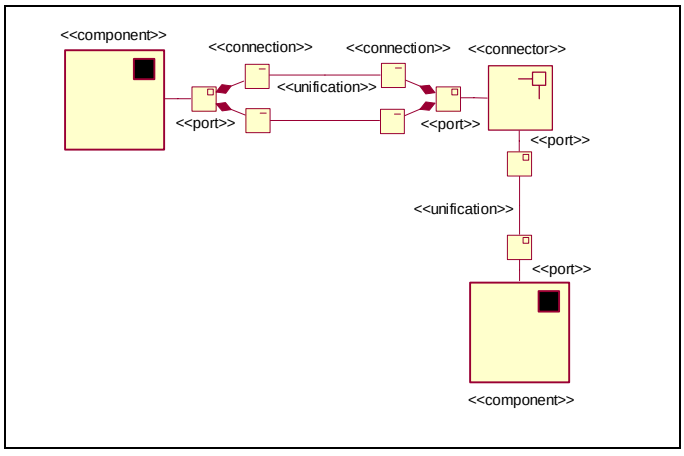

Figure 2. Architectural concepts in π-ADL

A component can send or receive values via connections. They can be declared as output connections (values can only be sent), input connections (values can only be received), or input-output connections (values can be sent or received).

Connectors are special-purpose components. They are described as components in terms of external ports and an internal behaviour. However, their architectural role is to connect together components. They specify interactions among components.

Therefore, components provide the locus of computation, while connectors manage interaction among components. In order to have actual communication between two components, there must be a connector between them.

A connection provided by a port of a component is attached to a connection provided by a port of a connector by unification or value passing. Thereby, attached connections can transport values (that can be data, connections, or even architectural elements).

From a black-box perspective, only ports (with their connections) of components and connectors and values passing through connections are observable. From a white-box perspective, internal behaviours are also observable.

Components and connectors can be composed to construct composite elements, which may themselves be components or connectors. Composite elements can be decomposed and recomposed in different ways or with different components in order to construct different compositions.

Composite components and connectors comprise external ports (i.e. observable from the outside) and a composition of internal architectural elements. These external ports receive values coming from either side, incoming or outgoing, and simply relay it to the other side keeping the mode of the connection. Ports can also be declared to be restricted. In that case, constituents of composite elements can use connections of restricted ports to interact with one another but not with external elements.

Architectures are composite elements representing systems. An architecture can itself be a composite component in another architecture, i.e. a sub-architecture.

2.2 π-ARL Architecture Refinement Language

Software applications are usually developed in several refinement steps. In π-ARL, the underlying approach for architectural refinement is underspecification. The decrease of this underspecification establishes a refinement relation for architectural elements.

The refinement relation in π-ARL, from an external or internal point of view, comprises four forms of refinement: behaviour, port, structure, and data refinements. The most fundamental notion of refinement in π-ARL is behaviour refinement. The other forms of refinement imply behaviour refinement modulo port, structure and data mappings.

In general, architectural refinement is a combination of the four forms of refinement. For instance, an architect can define an abstract architecture, then "data" refine that architecture in order to introduce base and constructed data types, then "port" refine the architecture to have ports with finer grain connections carrying data of different types, then "structure" refine its composite behaviour by adding new finer grain connectors, and so on.

π-ARL provides constructs for defining refinements of the four forms cited so far, according to external or internal points of view. Composite refinements can be defined in terms of refinement primitives and composite refinements themselves. Refinement primitives comprise:

- adding, removing, replacing or transforming data type declarations of an architecture,

[2] The *UML Profile for π-ADL* is used for presenting diagrammatic models.

- adding, removing, replacing or transforming ports of an architecture,
- adding, removing, replacing or transforming output and input connections of ports of an architecture,
- transforming the behaviour of an architecture or the behaviour of a component or connector in an architecture,
- adding, removing, replacing or transforming components or connectors in an architecture,
- exploding or imploding components or connectors in an architecture,
- unifying or separating connections of ports in an architecture.

These primitives, applied step by step, allow the incremental transformation of an architecture description. These transformations are enforced to be refinements if preconditions of refinement primitives are satisfied and proof obligations discarded. A refinement engine based on rewriting logics runs the refinement descriptions expressed in π-ARL generating further refined architectures. Code is generated from refined (concrete) architectures.

3 Case Study

In order to illustrate how π-ARL can be used to formally support the model-driven refinement of a software architecture, we present in this section a case study on the stepwise refinement of the abstract architecture of a login system. First we will present the abstract architecture description of the login system with π-ADL. Then we will refine the abstract architecture with π-ARL in order to obtain a more concrete architecture.

3.1 Describing the Architecture with π-ADL

To start, let us present the abstract architecture description of the login system. We will present a black-box description of the architecture focusing on interface (i.e. ports and their connections) of components and connectors. Then we will present, as an example, the internal behaviour of a connector. Finally the encompassing structure (i.e. binding among components and connectors using connection unifications) is described.

The login system supports the creation of new logins by receiving a new user identification (userId) and a password to be stored under this userId. Concurrently, it supports checking of existing logins by answering requests for the password of a certain existing userId by sending the password stored under this userId.

Using π-ADL, the login system, seen as a whole, can be formally described as follows.

```
architecture LoginManager is abstraction() {
   type UserId is Any. type Password is Any.
   type Login is tuple[UserId, Password].
   port update is {  connection in is in(Login) }.
   port request is { connection userId is in(UserId).
                     connection password is out(Password)
   }  assuming {
      protocol is { (          via userId receive any. true*.
                               via password send any )* }
   }.
   …
}
```

In this interface description of the login system, it is represented as a composite component, named *LoginManager*, having ports[3] *update* and *request*. These ports represent the interaction of the *LoginManager* system with its environment. Type *UserId* is the set of all possible userIds and type *Password* is the set of all possible passwords. *Login* is the tuple type *tuple[UserId, Password]*, i.e. the set of all possible logins associating userId and password. Two ports are declared: *update* that comprises the connection *in* for receiving new login entries and *request* that comprises the connections *userId* and *password* for answering requests. The protocol enforced by this port is that requests for the password of a certain userId, which are received via the connection *userId*, are answered by sending (after processing) a password via the connection *password*. For each userId received there must be a password sent before accepting the next userId.

The login system is composed of a login user interface (UI) and a login database manager. The login UI acts as a client of the database manager that acts as a server managing the login data. A new login entry received from the environment first undergoes some processing in the login UI and is then forwarded to the remote database manager that stores its data. Figure 3 outlines the abstract architecture of the system in terms of its components and connectors.

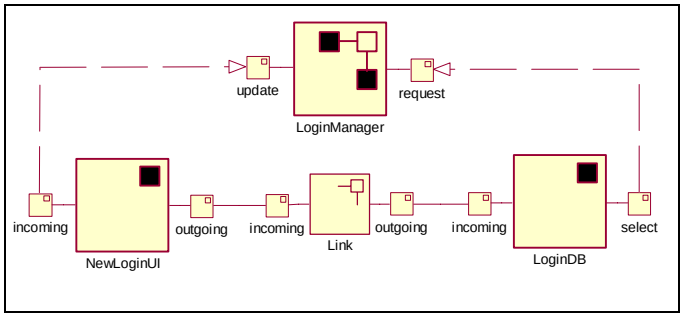

Figure 3: Outline of the *abstract architecture*

The architecture consists of a login UI component *NewLoginUI*, a database manager component *LoginDB*, and a connector *Link* to connect them together. These components and connector can be formally described in π-ADL as follows.

```
component NewLoginUI is abstraction() {
   type UserId is Any. type Password is Any.
   type Login is tuple[UserId, Password].
   port incoming is { connection in is in(Login) }.
   port outgoing is { connection toLink is out(Login) }.
   …
}  assuming {
   protocol is { (  via incoming::in receive any. true*.
                    via outgoing::toLink send any )* }
}
```

In component *NewLoginUI*, two ports are declared: *incoming* that comprises the connection *in* for receiving new login entries and *outgoing* that comprises the connection *toLink* for forwarding these logins. The protocol enforced by the two ports is that a value received via the connection *in* is (after processing) forward by sending it via the connection *toLink*. For each new login entry received there must be a login sent before accepting the next new login.

[3] By syntactic convention, ports that are not explicitly declared as restricted are external free ports.

```
component LoginDB is abstraction() {
   type UserId is Any. type Password is Any.
   type Login is tuple[UserId, Password].
   port select is { connection userId is in(UserId).
                    connection password is out(Password)
   } assuming {
      protocol is { (        via userId receive any. true*.
                             via password send any )* }
   }.
   port incoming is { connection fromLink is in(Login) }.
   …
}
```

In component *LoginDB*, two ports are declared: *select* that comprises the connection *userId* for receiving userId values and the connection *password* for sending the password value stored under this userId, and *incoming* for receiving new login entries to be stored.

```
connector Link is abstraction() {
   type UserId is Any. type Password is Any.
   type Login is tuple[UserId, Password].
   port incoming is { connection toLink is in(Login) }.
   port outgoing is { connection fromLink is out(Login) }.
   …
} assuming {
   protocol is { ( via incoming::toLink receive login : Login.
                   via outgoing::fromLink send login )* }
}
```

In connector *Link*, two ports are declared: *incoming* that comprises the connection *toLink* for receiving login entries and *outgoing* that comprises the connection *fromLink* for forwarding these entries. The protocol enforced by the two ports is that login entries received via the connection *toLink* are immediately forward by sending it via the connection *fromLink*.

This black-box description of the *LoginManager* architecture can be further detailed to achieve a white-box description of the architecture that encompasses interface, behavioural and structural aspects. For instance, the behaviour of the connector *Link* can be formally described in π-ADL as follows.

```
connector Link is abstraction() {
   …
   behaviour is {
      via incoming::toLink receive login : Login.
      via outgoing::fromLink send login.
      behaviour()
   }
} assuming { … }
```

In connector *Link*, the behaviour specifies that login entries received via the connection *toLink* are immediately forward by sending it via the connection *fromLink*. The behaviour is recursively defined. Once a login entry is handled, it continues with the same (recursive) behaviour for the next entry.

Using the components *NewLoginUI* and *LoginDB* and the connector *Link*, the abstract architecture *LoginManager* can be composed in π-ADL as shown below, thereby providing the structure of the architecture in terms of attached components and connector.

```
architecture LoginManager is abstraction() {
   …
   behaviour is   compose {  ui is NewLoginUI()
                  and        lk is Link()
                  and        db is LoginDB()
} where {  ui::incoming relays update
   and     ui::outgoing unifies lk::incoming
   and     lk::outgoing unifies db::incoming
   and     request relays db::select
   }
}
```

In the architecture, the component instances *ui* and *db* are connected using the connector *lk*. In order to actually connect them, connections must be unified[4]. Connection *toLink* of port *outgoing* of component *UI* is unified with connection *toLink* of port *incoming* of connector *lk*. Connection *fromLink* of port *outgoing* of connector *lk* is unified with connection *fromLink* of port *incoming* of component *db*.

Besides connecting component instances together, the architecture must express the binding between external ports and ports of components. This binding is expressed by connection relay. Connection *in* of external port *update* is relayed to connection *in* of port *incoming* of component *UI*. Connection *userId* of external port *request* is relayed to connection *userId* of port *select* of component *db*. Connection *password* of port *select* of component *db* is relayed to connection *password* of external port *request*.

3.2 Refining the Architecture with π-ARL

The software architect can refine the previously described abstract architecture to obtain a more concrete architecture where, for instance, security is improved. This could be achieved by encrypting the passwords that are transmitted: for each new login entry, the *NewLoginUI* will encrypt the password to transmit and the *LoginDB* will decrypt it to store in the database.

The architect is not interested in the algorithmic aspects of the password encryption. S/he just consider that the encrypted password is itself an element of *Password*, and that there is a function *encrypt : Password → Password* that handles the encryption for a single password in the login UI. Another function *decrypt : Password → Password* decrypts the passwords. S/he can assume that *for all password*: *decrypt(encrypt(password)) = password*.

In order to refine the architecture, the following actions could be carried out. The *NewLoginUI* could be extended with an encrypting component. For each new login entry the password related to a userId is encrypted and forwarded. The *LoginDB* could be extended with a decrypting component to decrypt passwords received.

One possible architectural refinement to achieve this architecture is to introduce two components, *Encryptor* and *Decryptor*, that encrypts and decrypts passwords, respectively.

In the sequel, we present how this model-driven refinement can be carried out with π-ARL by a software architect.

[4] If connections have the same names in different ports, identifying ports is enough to express unifications (if connection names are different, then they must be explicitly unified).

1st step: adding components

As first step, the architect could introduce the two new empty components, *Encryptor* and *Decryptor*. They should not be connected to any other component in the architecture.

In π-ARL, this refinement could be expressed as follows.

```
architecture LoginManagerRef1 refines LoginManager using {
   components includes { Encryptor is abstraction() { deferred } }.
   components includes { Decryptor is abstraction() { deferred } }
}
```

Note that the behaviours of the components that have been added are completely undefined. It is worth noting that these refinement actions do not change the behaviour of the architecture.

2nd step: adding output and input connections

The architect could now add a typed output connection *toCoLink* to *Encryptor*, a typed output connection *toDLLink* to *Decryptor*, and an output connection *toENLink* to *NewLoginUI*. S/he could also add an input connection *fromENLink* to *Encryptor*, an input connection *fromCoLink* to *Decryptor*, and input connection *fromDLLink* to *LoginDB*.

In π-ARL, this refinement could be expressed as follows.

```
architecture LoginManagerRef2 refines LoginManagerRef1 using {
   Encryptor::types includes { UserId is Any. Password is Any.
      Login is tuple[UserId, Password] }.
   Decryptor::types includes { UserId is Any. Password is Any.
      Login is tuple[UserId, Password] }.
   Encryptor::connections includes
      { toCoLink is out(Login). fromENLink is in(Login) }.
   Decryptor::connections includes
      { toDLLink is out(Login). fromCoLink is in(Login) }.
   NewLoginUI::connections includes {
      outgoing::toENLink is out(Login) }.
   LoginDB::incoming::connections includes {
      fromDLLink is in(Login) }
}
```

The behaviour of the system itself is unchanged, since the new connections are not yet used in the architecture.

3rd step: adding and connecting connectors

The architect could now add connectors *CoLink* to connect the *Encryptor* and the *Decryptor*, *ENLink* to connect the *Encryptor* and the *NewLoginUI*, and *DLLink* to connect the *Decryptor* and the *LoginDB*.

Of course, the addition of connectors could, as they are first-class citizens as components, follow the same basic steps: addition of connectors without connections, addition of output connections, and addition of input connections.

The behaviour of the system itself is unchanged, since the connections *toENLink* and *fromDLLink* added to the *NewLoginUI* and the *LoginDB* respectively are not yet used by their behaviours in the architecture.

In π-ARL, this refinement could be expressed as follows.

```
architecture LoginManagerRef3 refines LoginManagerRef2 using {
   connectors includes {
      CoLink is abstraction() {
         connection toCoLink is in(Any).
         connection fromCoLink is out(Any).
         behaviour is { deferred }
      }.
      ENLink is abstraction() {
         connection toENLink is in(Any).
         connection fromENLink is out(Any).
         behaviour is { deferred }
      }.
      DLLink is abstraction() {
         connection toDLLink is in(Any).
         connection fromDLLink is out(Any).
         behaviour is { deferred }
      }.
   connections unifies { CoLink::toCoLink with Encryptor::toCoLink.
      CoLink::fromCoLink with Decryptor::fromENLink.
      ENLink::toENLink with NewLoginUI::outgoing::toENLink.
      ENLink::fromENLink with Encryptor::fromENLink.
      DLLink::toDLLink with Decryptor::toDLLink.
      DLLink::fromDLLink with LoginDB::incoming::fromDLLink }
}
```

4th step: refining the behaviour of added components

The architect could now refine the behaviour of added components in π-ARL. The *Encryptor* and *Decryptor* components could be refined as follows.

```
architecture LoginManagerRef4 refines LoginManagerRef3 using {
   Encryptor::behaviour becomes abstraction() {
      encrypt is function(p : Password) : Password { deferred }.
      replicate
         via fromENLink receive log : Login.
         via toCoLink send tuple(log::userId, encrypt(log::password))
   }.
   Decryptor::behaviour becomes abstraction() {
      decrypt is function(p : Password) : Password { deferred }.
      replicate
         via fromCoLink receive log : Login.
         via toDLLink send tuple(log::userId, decrypt(log::password))
   }
}
```

The *Encryptor* applies the encryption function to the password received from its input connection to send the encrypted password via its output connection. The *Decryptor* applies the decryption function to the encrypted password received from its input connection to send the decrypted password via its output connection.

As the behaviour of these components were unspecified, that is their specifications were deferred until now, this refinement is obviously correct. The structure of the system remains unchanged.

5th step: refining the behaviour of added connectors

The architect could now refine the behaviour of added connectors to carry the login entries by *ENLink*, the encrypted login entries by *CoLink* and the decrypted login entries by

DLLink. This is accomplished by refining the behaviour of these connectors as follows.

```
architecture LoginManagerRef5 refines LoginManagerRef4 using {
   ENLink::behaviour becomes abstraction() {
      replicate    via toENLink receive log : Login.
                   via fromENLink send log
   }.
   CoLink::behaviour becomes abstraction() {
      replicate    via toCoLink receive log : Login.
                   via fromCoLink send log
   }.
   DLLink::behaviour becomes abstraction() {
      replicate    via toDLLink receive log : Login.
                   via fromDLLink send log
   }
}
```

6th step: refining the behaviour of existing components

The architect could now refine the behaviour of existing components of the abstract architecture in order to take into account the new introduced components and connectors. This is accomplished by refining the behaviour of the *NewLoginUI* and *LoginDB* as follows.

```
architecture LoginManagerRef6 refines LoginManageRef5 using {
   component NewLoginUIRef1 refines NewLoginUI using {
      connections replaces outgoing::toLink by outgoing::toENLink
   }.
   component LoginDBRef1 refines LoginDB using {
      connections replaces incoming::fromLink
      by incoming::fromDLLink
   }
}  assuming {
   property passwordIntegrity is
            { forall p : Password | decrypt(encrypt(p)) = p }
}
```

The architect assumes that encrypting and decrypting the password via *CoLink* yields the same password as that transmitted through *Link*. Formally:

$$\forall\ p : \text{Password} \bullet \text{decrypt}(\text{encrypt}(p)) = p$$

That is, in order to guarantee that the value-passing behaviour of the new architecture refines the value-passing behaviour of the abstract architecture, the architect assumes that this property will hold in the system. In fact, it becomes a proof obligation. Thereby, if the assumption holds, the refinement is guaranteed.

7th step: removing disconnected connectors

After this refinement, the connector *Link* is no more used and can therefore be discarded by the architect.

```
architecture LoginManagerRef7 refines LoginManagerRef6 using {
   connectors excludes Link
}
```

The following figure depicts the resulting architecture.

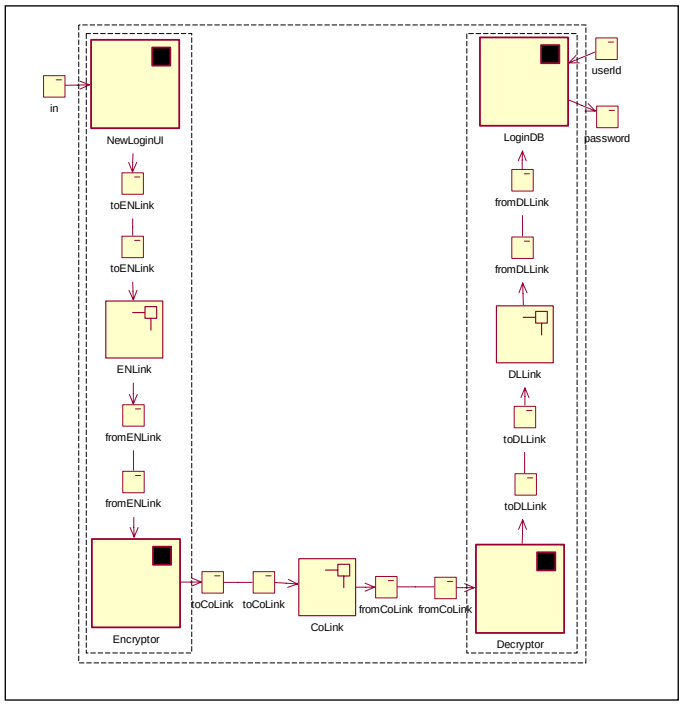

Figure 4. More concrete architecture

8th step: imploding sub-architectures as components

In the last refinement step, in order to get a concrete architecture that is better structured, the architect could implode the *NewLoginUI* and the *Encryptor* as one component and the *LoginDB* and the *Decryptor* as another. Figure 4 depicted the architecture before this refinement step. Imploding sub-architectures as components yielding a composite *CoNewLoginUI* and a composite *CoLoginDB* can be expressed in π-ARL as follows.

```
architecture CoLoginManager refines LoginManagerRef7 using {
   components implodes { NewLoginUIRef1
      and ENLink and Encryptor } as CoNewLoginUI.
   components implodes { LoginDBRef1
      and DLLink and Decryptor } as CoLoginDB
}
```

The following figure shows the architecture after that refinement step, comprising composite components.

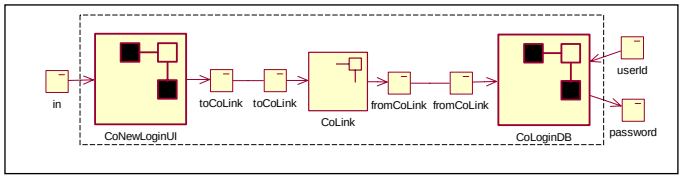

Figure 5. Imploding sub-architectures as components

After the application of all these refinement steps, the transformations yield a more concrete architecture, encrypting and decrypting passwords that are passed between *NewLoginUI* and *LoginDB*. The architecture that is obtained by refinement is behaviourally equivalent to the initial abstract architecture, but is more detailed with respect to security-related features.

Composing refinement actions for reuse

A composite refinement action can then be defined in π-ARL by combining these different refinement steps in order to capitalize this refinement expertise. This composite refinement action can then be applied on different abstract architectures yielding the same kind of transformations.

4 Related Work

Enabling stepwise architecture refinement is a new challenge for the architecture-centric model-driven development of complex software systems. With the exception of a variant of FOCUS [19], i.e. FOCUS/DFA [17], RAPIDE [9] and SADL [12], there is no proposal for a rigorous calculus based on architectural terms as there are rigorous calculus for refinement of programs [2][3]. In the case of SADL the refinement is only structural. In the case of RAPIDE it is only behavioural (supported by simulations). In both cases, clear architectural primitives for refining architectures are not provided and the refinement supported is only partial. π-ARL like the B [1] and Z [7] formal methods, provides operations to transform specifications. However, unlike FOCUS, B and Z, π-ARL has been specially designed to deal with architectural elements of any architectural style.

5 Concluding Remarks

π-ARL provides a novel language that on the one side has been specifically designed for architectural refinement taking into account refinement of behaviour, port, structure, and data from an architectural perspective and on the other side is based on preservation of properties. The core of π-ARL is a set of architecture transformation primitives that support refinement of architecture descriptions. Transformations are refinements when they preserve properties of the more abstract architecture. Core properties are built-in. Architectural style-specific or architecture-specific properties are user defined. The underlying foundation for architected behaviours [5][6] is based on the higher-order typed π-calculus [11] [18].

Refinement actions are defined in π-ARL (using a textual notation – presented in this paper – or a visual notation – under development). These actions can be combined to carry out complex refinements. Assumptions made during refinement are proof obligations to guarantee the correctness of the refinements. They can be mechanically checked using the π-ARL toolset [10] that includes a refinement engine interpreting π-ARL and orchestrating a model checker, a prover and specific evaluators.

π-ARL provides the required key features for supporting formal architecture-centric model-driven development. Thus, an abstract – platform independent – architecture can be refined to a concrete – platform specific – architecture following a Model-Driven Architecture approach [4].

By addressing software development as a set of architecture-centric model refinements, the refinements between models become first class elements of the software engineering process. This is significant because a great deal of work takes places in defining these refinements, often requiring specialized knowledge on source and target abstraction levels, for instance knowledge on the source application logics and on the targeted implementation platforms. Efficiency and quality of software systems can be improved by capturing these refinements explicitly and reusing them consistently across developments. Thereby, user-defined refinement steps can be consistently defined, applied, validated, and mechanically automated with π-ARL.

Both π-ADL and π-ARL have been applied in several industrial case studies and pilot projects at Thésame (France) and Engineering Ingegneria Informatica (Italy). The pilot project at Thésame aimed to architecting and refining agile integrated business process systems. The pilot project at Engineering Ingegneria Informatica aimed to architecting and refining federated knowledge management systems.

Future work is mainly related with the formal development of an architecture-centric formal method. This formal method, called the π-Method, like formal methods such as B, FOCUS, VDM [8], and Z, aims to provide full support for formal development. Unlike these formal methods that do not provide any architectural support, the π-Method has been built from scratch to support architecture-centric model-driven development.

References

[1] Abrial J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996.

[2] Back, R-J.: Refinement Calculus, Part II: Parallel and Reactive Programs. Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Springer, 1990.

[3] Back, R-J., von Wright, J.: Refinement Calculus, Part I: Sequential Nondeterministic Programs. Proceedings of REX Workshop: Refinement of Distributed Systems. Springer, 1989.

[4] Brown A.W.: Model Driven Architecture: MDA and Today's Systems. The Rational Edge, February 2004.

[5] Chaudet C., Greenwood M., Oquendo F., Warboys B.: Architecture-Driven Software Engineering: Specifying, Generating, and Evolving Component-Based Software Systems. IEE Software Engineering Journal, Vol. 147, No. 6, UK, 2000.

[6] Chaudet C., Oquendo F.: A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems. Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00). IEEE Computer Society, Grenoble, September 2000.

[7] Davies J., Woodcock J.: Using Z: Specification, Refinement and Proof. Prentice Hall Series in Computer Science, 1996.

[8] Fitzgerald J., Larsen P.: Modelling Systems: Practical Tools and Techniques for Software Development. Cambridge University Press, 1998.

[9] Luckham D.C., Kenney J.J., Augustin L.M., Vera J., Bryan D., Mann W.: Specification and Analysis of System Architecture Using RAPIDE. IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995.

[10] Megzari K., Oquendo F.: The ArchWare Architecture Refinement Toolset. Deliverable D6.3, ArchWare European RTD Project, IST-2001-32360, April 2004.

[11] Milner R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, 1999.

[12] Moriconi M., Qian X., Riemenschneider R.A.: Correct Architecture Refinement. IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995.

[13] Oquendo F.: π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. ACM Software Engineering Notes, Vol. 29, No. 3, May 2004.

[14] Oquendo F.: π-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures. ACM Software Engineering Notes, Vol. 29, No. 5, September 2004.

[15] Oquendo F.: Formally Describing Dynamic Software Architectures with π-ADL. WSEAS Transactions on Systems, Vol. 3, No. 8, October 2004.

[16] Oquendo F. et al.: ArchWare: Architecting Evolvable Software. Proceedings of the 1st European Workshop on Software Architecture, LNCS 3047, Springer, St Andrews,UK, May 2004.

[17] Philipps J., Rumpe B.: Refinement of Pipe and Filter Architectures. Proceedings FM'99, Springer, LNCS 1708, 1999.

[18] Sangiorgi D., Walker D.: The π-Calculus: A Theory of Mobile Processes. Cambridge University Press, 2001.

[19] Stolen K., Broy M.: Specification and Development of Interactive Systems. Springer Verlag, 2001.