Style-Based Model Transformation for Early Extrafunctional Analysis of Distributed Systems

Julien Mallet, Siegfried Rouvrais

Style-Based Model Transformation for Early Extrafunctional Analysis of Distributed Systems

Julien Mallet and Siegfried Rouvrais

Institut TELECOM; TELECOM Bretagne
Technopole Brest-Iroise, CS 83818, 29238 Brest Cedex 3, France
{julien.mallet,siegfried.rouvrais}@telecom-bretagne.eu

Abstract. In distributed environments, client-server, publish-subscribe, and peer-to-peer architecture styles are largely employed. However, style selection often remains implicit, relying on the designer's know-how regarding requirements. In this paper, we propose a framework to explicitly specify distributed architectural styles, as independent models of the application functionalities. To justify feasibility and further benefits of our approach, we formally define three classical distributed architectural styles in a process calculus. Our proposal then opens up the way to a systematic composition of functional models with architectural style models as an endogenous transformation. Comparative analysis of extrafunctional properties could then be proposed at the early design stages to guide the architect in stylistic choices.

1 Introduction

Architectural styles build up conventional structures for designing large systems at a software architecture level. Different architectural styles enforce different quality attributes for a system [1]. Within distributed systems, an application often relies on an architectural style which defines connections between application components (*e.g.* simple message interaction models, client-server, publish-subscribe, peer-to-peer). Most often, style selection remains implicit and tacit [2], relying on the architect's know-how regarding requirements. Choosing an inappropriate architectural style can lead to major impacts on the properties of a system or application [3]. Moreover, extrafunctional properties such as security, performance, reliability or scalability are not easily grasped at an abstract description level. Such concerns thus tend to be forwarded to the end of the design process lifecycle, though they are rough to manage once a style has been selected and a system designed. They are however the critical selection criteria to better manage the development process, regarding system's internal and external properties.

Specifying a distributed system's software architecture classically requires to model architectural components and connectors, and some of their extrafunctional properties. However connectors, as communication mediums, are parts

of distributed styles having their own comprehensive, intrinsic, and emergent properties. To manage extrafunctional properties at early design stages, we propose to specify distributed architectural styles independently of the functional model. By separating concerns in a framework, we then propose a model transformation corresponding to a composition of an abstract functional model with styles predefined in a repository. Functional and especially extrafunctional analysis could then be investigated to compare models and guide the architect faced with several distributed design alternatives. To justify the approach, we restrict to three common distributed styles descriptions in this paper, using structure diagrams and process calculus: client-server [4], publish-subscribe [5] and peer-to-peer [6]. A distributed version control system with its functional model is proposed as a case study.

The remainder of this paper is organised as follows. Section 2 introduces our framework and proposes some common distributed architectural styles and their specifications using a process calculus. Section 3 presents an independent functional model and proposes a specification example on the version control system case study. The systematic composition of a functional model with an architecture style model is described in section 4 through the application example. Section 5 addresses extrafunctional properties integration in the framework, while section 6 presents related work. Finally, section 7 concludes this paper with a summary and an outline of further research.

2 A Framework with Distributed Architectural Styles

The motivation of the proposed framework presented in figure 1 is to guide the architect in choosing the right distributed architecture style in conformance with extrafunctional requirements. It follows a model driven engineering approach [7] and addresses the quality of the target system's software architecture for early design decisions. First, the designer specifies system functionality using

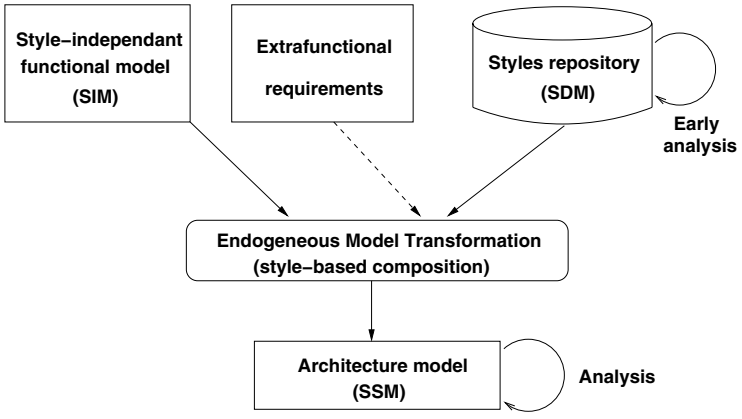


Fig. 1. The overall framework

a style-independent model (SIM). This model is purely functional: the services are provided in an ideal structureless-environment. However, application components in a distributed system require some interaction mechanisms for coordination and communication. As early proposed in the software architecture community [8], architectural styles provide conventional structures for building large systems [9,10]. Such structures are the primary models for distributed interactions. Connectors, as interaction mechanisms, are the principal structural elements for a boxology [3] of distributed architectural styles. Within distributed systems, the client-server architectural style [4], based on the request-reply protocol, is still predominant. With Web generalisation, related styles like service-oriented or *Representational State Transfer* (REST [11]) architectures are now taking a major position. So far, mechanisms for coordination and communication based on push-like models (*e.g.* message sending) or pull-like models (*e.g.* request-reply) are structural elements of architectural styles. Thus, we propose to gather distributed architecture styles in a repository of style definition models (SDM). Those models, having their own extrafunctional characteristics, can be early analysed in light of extrafunctional requirements.

Extrafunctional properties (*e.g.* security, performance, scalability) are key elements to guide the design decisions. Our transformation model consists in composing the required functional services with a given distributed style available in the repository. A distributed architectural style candidate can first be selected in front of quality attributes. The resulting style specific model (SSM) represents a possible system's architecture model which can be compared with other SSMs using functional and extrafunctional analysis. The framework makes it possible to choose, at upper stages, possible styles thanks to the extrafunctional properties. After analysis, if a proposed style specific model does not guarantee the extrafunctional requirements, the architect may modify or weaken some of the requirements until finding an appropriate style, or introduce specific mechanisms (*e.g.* patterns) in the specified system's software architecture to satisfy requirements.

2.1 Three Classical Distributed Architecture Style Models

Client-server, publish-subscribe, and peer-to-peer styles are largely used for distributed applications. Such styles encourage reusability, system comprehension, and analysis by using well-known interaction mechanisms. These mechanisms predominantly rely on push or pull models. Style variants exist, but they share common characteristics at an abstract level. In the classical client-server style, a client component requests a service through a remote invocation to a server component. Often synchronous, this interaction mechanism follows a pull model based on a request-reply protocol. In the publish-subscribe style, components are either announcers or listeners of events. By registering through an event manager, listeners are asynchronously informed of events most often through a push model. In the decentralised peer-to-peer style, where a peer represents a component, overlays, as logical networks, are dynamically constructed or maintained. For instance, by using a pure pull model between peer neighbours or by

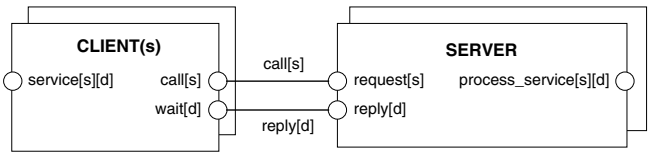
combining the pull with a push model restricted in depth, this style is much adopted in mobile or ubiquitous environments.

The distributed architecture styles have emergent extrafunctional properties. For example, the publish-subscribe and peer-to-peer styles are mostly known to be scalable and reliable. Moreover, the publish-subscribe style generally guarantees the anonymity of the announcers. However, these intrinsic properties mainly arise from empirical studies and not from systematic evaluations on style models.

2.2 Modelling Architectural Styles

Magee and Kramer [12] provide elements of style specification using the Finite State Processes (FSP) process calculus. For our purpose, we expand their style examples to create a first repository of distributed styles, independent of any application functionalities. Other formal approaches could have been addressed, but FSP process calculus, with its associated LTSA tool (*i.e.* model-checker), is suitable for a comprehensible demonstrator for a model transformation. For the sake of clarity, the three classical distributed styles addressed in this paper are shown hereafter as structure diagrams of processes (*i.e.* components as boxes in figures) and ports or events (*i.e.* bullets in figures). Note that the structure diagrams are only graphical representations of the FSP expressions as defined in [12] (*e.g.* the notion of provided/required ports does not exist). Elements of syntactical FSP expressions for client-server and publish-subscribe styles can be found in [12]. We detail the FSP expression only for the client-server style.

Figure 2 presents the generic client-server style, through the structure diagram, where several clients call services from the server and obtain the associated result. The clients are introduced by the processes CLIENT (stacked boxes represent identical processes). A given client will always request the same service




```
set Data = {d1,d2,d3,d4}
set ServiceId = {s1,s2}
set Clients ={c1,c2,c3}
CLIENT(S='si)
  = (call[S] -> wait[d:Data] -> service[S][d] -> CLIENT)+{call[ServiceId]}.
SERVER
  = (request[s:ServiceId] -> process_service[s][d:Data] -> reply[d] -> SERVER).
||CS_EX
  = (c1:CLIENT('s1) || c2:CLIENT('s2) || c3:CLIENT('s1) || Clients:SERVER)
    /{forall [c:Clients]{[c].call/[c].request,[c].reply/[c].wait}}.
```

Fig. 2. Structure diagram and FSP expression of a client-server style

(introduced by its parameter s). Furthermore, each CLIENT process is statically linked with one distinct SERVER process that responds to its request (there are as many SERVER as CLIENT processes). For model transformation, in order to further compose the style-independent and style definition models, the clients provide a service[s][d] event corresponding to an external call to the service s that returns a result d, and the server offers a process_service event corresponding to an external service computation.

In addition, figure 2 presents the corresponding FSP expression and an example of an instanced client-server style (process ||CS_EX). Three sets are introduced: Data as the possible results, ServiceId as the service names offered by the server and Clients as the identifiers of the client processes. A CLIENT process calls the service (event call[S]), then awaits synchronously the result (event wait), forwards it through the service event to external components and iterates. Similarly to [12], the CLIENT process uses the alphabet extension operator (noted +) in order to ensure a suitable synchronisation between the clients and the server and takes the service name as parameter (its default value is si). The SERVER process awaits a service request (event request), then requests the result computation (event process_service) and returns the result (event reply). Finally, the ||CS_EX client-server style example is the parallel composition of the clients (in our case, three clients: c1, c2 and c3) with as many server processes prefixed by the identifier of the corresponding client (expression Clients:SERVER). Corresponding to architectural attachments, the mapping between client and server events associates, respectively, the call and wait events of the client with the request and reply events of the corresponding server.

Note that the definition of the client-server style is generic : we just have to define the Data, Client and ServiceId specific sets in order to instanciate the style to a client-server application.

Relying on [12], figure 3 presents the publish-subscribe style where one announcer (process ANNOUNCER) publishes events to zero or more listeners (process LISTENER). The EVENTMANAGER process carries out the event broadcasting. A listener can register his/her interest in a particular pattern p with the event manager through the register[p] event. Each time the announcer produces the pattern, only the registered listener is notified. Finally, a listener can deregister himself through the deregister event. In order to specify this style in FSP, one event manager process is introduced per listener. When an event announcement is produced, the event managers forward it to their associated registered listeners.

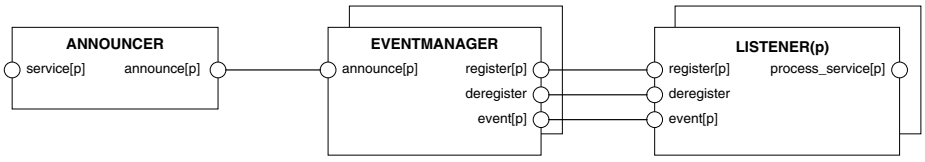


Fig. 3. Structure diagram of a publish-subscribe style

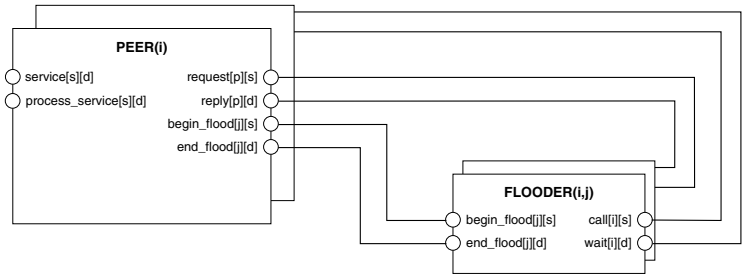


Fig. 4. Structure diagram of a pure peer-to-peer style

Finally, figure 4 presents the peer-to-peer style in its flooding version where peers collaborate through a pull model (*e.g.* like in the Gnutella scheme). Since many variants of peer-to-peer styles exist, we limit our example to the simple flooding version. Each peer is represented by a PEER process taking its number as parameter. It offers two services as interface: service[s][d] for requesting a service s returning the result d and process_service[s][d] for requesting external components. A FLOODER process per peer is associated with each link between peers. In the figure 4, the peers of number i and j are neighbouring. The FLOODER sends the requests to the peer neighbours (event call) then awaits the response from the latter (event wait). Therefore, the topology of the considered peer-to-peer network is modelled by the FLOODER and PEER links. At the time of a service[s][d] event, the PEER process initiates the flood request to its neighbours with the begin_flood event. The FLOODER process carries out the flood, then obtains the result d thanks to the wait[i][d] event. Finally, a PEER process receives the result through the end_flood event and transmits it to the requester.

The styles presented above and contained in the repository are distributed architecture styles rather than communication ones in the sense that they describe interaction between components. Due to the style transformation (presented section 4), the intended interaction mechanisms will be introduced into the functional model using the selected style model.

3 Functional Model

In the framework, the designer first specifies his system in a style-independent model, with a pure functional point of view set in an ideal non-constrained environment: extrafunctional properties are not taken into account, and architectural stylistic elements are abstracted.

Thereafter, we exemplify this principle on a version control system as a classical distributed case study. A version control system (*e.g.* CVS) allows several users/developers to modify a set of shared files concurrently. Each developer has a copy of the files (most often in a repository) which he/she can modify locally, using a write command. The local modifications are spread to other developers

with a commit command, using the versioning system. Update command brings local copies up-to-date with the last shared version.

3.1 Pure Functional Model of a Versioning System

The style-independent functional model of a distributed application is defined by introducing, for each user, the three following components (defined by one or more processes as the application requires):

- **User:** operations from the user's point of view (*i.e.* update, write, and commit in the example). It introduces the available services thanks to shared events;
- **RemoteState:** operations for data information obtained from other users due to distribution (*i.e.* future components for interaction mechanisms);
- **Safe:** business rules specified through authorised sequences of events in the system, to guarantee application requirements in functional terms.

Our abstract level approach is general and can be applied to other distributed applications as long as they could be decomposed into the three previous components. For our case study, each preceding component is specified by one FSP process (*i.e.* User, RemoteState, Safe) as presented in the structure diagram of figure 5. Each User process of the system takes a unique user number I as parameter. It holds and updates the state of its local copy of the repository (*i.e.* either it is identical to the reference repository or it contains some update). Any user can modify the file locally (event write), update his/her local copy with the repository (event update) or update the repository with his/her local copy (event commit). Moreover, the User process offers a service to read its local state thanks to the localState event. In the structure diagram, the User(I) and User(J) are similar except for their number. We distinguish them in order to present the specific shared events between User and RemoteState processes.

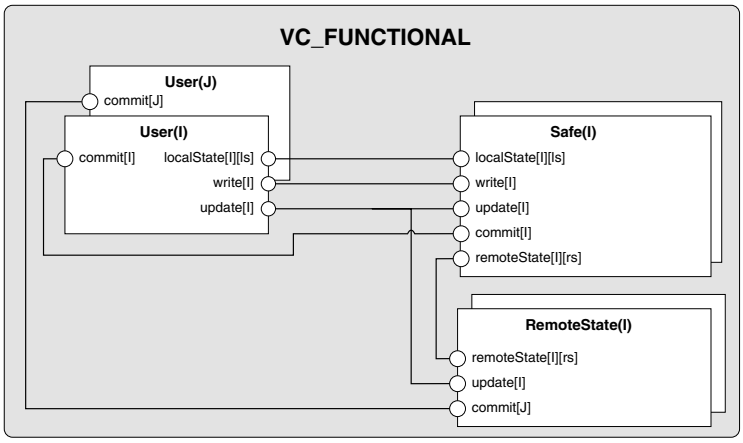


Fig. 5. Functional structure diagram of a source control management system

The RemoteState(I) process abstracts the overall remote state for the i^{th} User process. The remote state is either remotely non modified (other users have not modified the repository since the last local update) or remotely modified (one or more other user has modified the repository). So, when another User(J) process, distinct from User(I), performs a commit, the remote state associated with I becomes remotely modified. When User(I) produces an update event, the remote state becomes remotely non-modified. Furthermore, RemoteState provides a service allowing its state to be read (event remoteState).

Finally, Safe(I) processes ensure the right update policy (*i.e.* each User(I) process has to obtain a local copy that is consistent with the repository before updating the repository with its changes). These processes define the functional properties that the whole system has to ensure by specifying the authorised sequences of events.

There are several implementations of versioning systems, introducing architectural styles more or less implicitly (*e.g.* CVS and Subversion rely on the client-server style, SVK uses the peer-to-peer style). As we can see in our case study, the functional model does not imply any style.

3.2 Facilitating Functional Model Generation

We propose to formally specify style-independent models at an abstract level using a process calculus. A functional model is a composition of processes having remote states definitions. However, specifying processes could be difficult for the designer unfamiliar with formal methods. Tools supporting functional model elaboration have been designed in the process calculus community. For instance, based on scenarios specified as sequence diagrams or message sequence charts, FSP expressions could be generated to assist the architect [13].

4 Functional and Style-Based Model Transformation

Once a functional model of the system has been specified by the designer, it can be related to a certain architectural style model taken from the repository. Thanks to process calculus specifications, the transformation is achieved by process composition and event renaming. This model transformation is endogenous, *i.e.* the source and target models are still FSP processes. The generic transformation process is based on the following steps:

1. Choice of an architectural style in the repository (note that the designer could be egged on a choice due to early analysis on style models);
2. Selection of functional model event(s) in order to introduce the style;
3. Definition of event relabelling between functional model and style.

These above steps are devolved to the designer. Then, based on them, a systematic transformation can be applied in order to obtain the style specific model. The target style specific model is simply a process parallel composition, where events have been relabelled. We detail two generic transformations on our case

study, respectively to produce client-server and publish-subscribe style specific models. The transformation will link the external events of a style (*e.g.* service and process_service for the client-server style) with shared events of the functional model (*e.g.* commit, update and remoteState for the case study). These later events can be seen as join points for the style introduction.

4.1 Two Style Specific Models of the Case Study

A Client-Server Versioning System. The client-server style is introduced through the remoteState, update and commit events shared by the User, Safe and RemoteState processes from the functional model detailed in section 3. The events are transformed into a request-reply interaction between clients and a server. The resulting structure diagram is given in figure 6. Three kinds of components are identified: CS_GEN, VC_CLIENTS, and VC_SERVER. CS_GEN is the client-server process described in section 2. VC_CLIENTS represents the n clients of the distributed versioning system, each one maintaining the state of the local copy of a user and ensuring the access policy to the repository. Finally, VC_SERVER is the system server that holds and maintains the remote state of each user.

Figure 7 presents the FSP expression (for simplicity, only the components previously described are given). The processes from the functional model are unchanged (*i.e.* User, Safe and RemoteSate). In this case, the system is composed of three users identified by u0, u1 and u2. ServiceId is the set of events used as join points in order to introduce the style. The ||CS_GEN component contains a CLIENT process per event and user. Thus, for the remoteState event, there are three CLIENT processes, prefixed by u1.remoteState, u2.remoteState and u3.remoteState respectively. The ||VC_CLIENTS component is the parallel composition of the User and Safe processes. Events are relabelled in order to

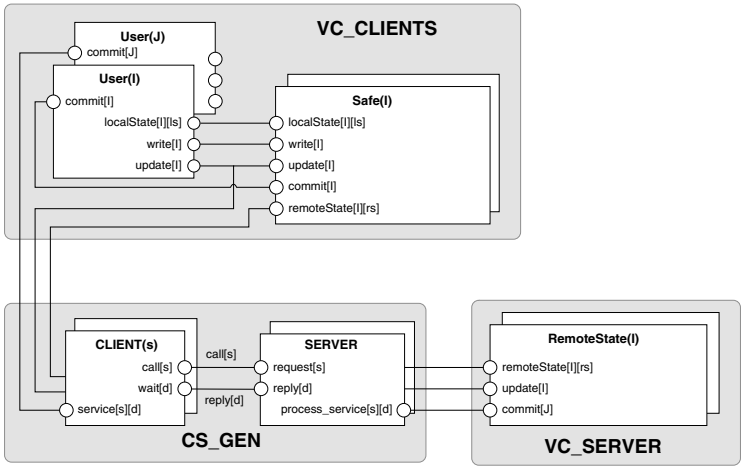


Fig. 6. Structure diagram of the client-server specific model

```
set Users = {u0,u1,u2}
const NUser = #Users
range U =0..NUser−1
set ServiceId = {remoteState,update,commit}
||CS_GEN
   = (forall[c:Users](forall[s:ServiceId]([c].[s]:CLIENT(s)))
      || Users[s:ServiceId]:SERVER)
     /{forall [c:Users]{forall [s:ServiceId]{
                    [c].[s].call/[c].[s].request,[c].[s].reply/[c].[s].wait}}}.
||VC_CLIENTS
  = forall[i:U](User(i)||Safe(i))
    /{forall[j:U]{[@(Users,j)].remoteState.service['remoteState]/remoteState[j],
                  [@(Users,j)].update.service['update]/update[j],
                  [@(Users,j)].commit.service['commit]/commit[j]}}.
||VC_SERVER
  = (forall[i:U](RemoteState(i)))
    /{forall[j:U]{[@(Users,j)].remoteState.process_service['remoteState]/remoteState[j],
                  [@(Users,j)].update.process_service['update]/update[j],
                  [@(Users,j)].commit.process_service['commit]/commit[j]}}.
||VC_CS =(VC_CLIENTS || CS_GEN || VC_SERVER).
```

Fig. 7. Case study client-server FSP expression

match the style events. For instance, the remoteState[0] event of User(0) is relabelled u0.remoteState.service['remoteState] (in FSP, the expression @(Users, j) denotes the j^{th} element of Users). The ||VC_SERVER component composes the RemoteState processes and relabels events equally. Finally, the ||VC_CS final system is the parallel composition of the three previous components.

A Publish-Subscribe Versioning System. For the publish-subscribe style, we choose the commit event shared by the User and RemoteState processes in order to introduce the style. This event is transformed into a push mode interaction: each User notifies all the others that he/she has modified the repository. The resulting structure diagram is given in figure 8. Three kinds of component are introduced: VC_ANNOUNCERS, VC_LISTENERS and PS_GEN for the style. The first one contains the User processes that act as announcers of commit events. The VC_LISTENERS component contains the RemoteState processes acting as listeners of the same events. At commit time, the User(J) process announces the commit[J] event to the event manager, which broadcasts it to all the RemoteState processes. The users communicate through the event manager to notify repository updates.

We have shown throughout this example that a functional model could be systematically composed with different styles provided in a common repository. Nowadays, the transformations are handmade; we have not yet an automatic tool that produces the target style specific model but we plan to describe formally the style transformation as a FSP expression transformation. Indeed, the transformation consists in composing the FSP expression of the functional model with

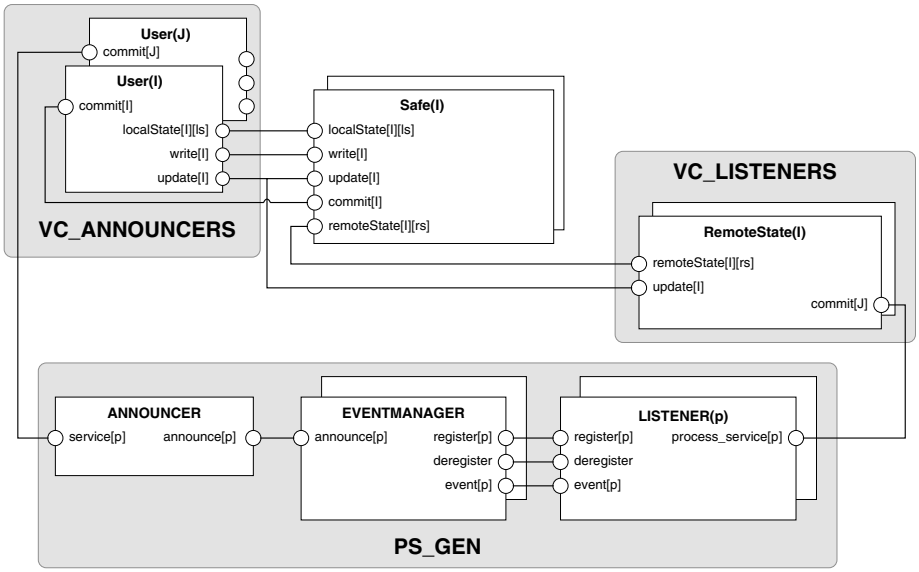


Fig. 8. Structure diagram of the publish-subscribe specific model

the chosen style one and relabelling the events in order to match the functional model ones with the external events of the style.

Further, style variants are also to be considered. In fact, RemoteState could be distributed either locally on a User or within a particular component. A range of style interaction models can then be made available according to the required distribution. After composition, a model checker is the primary tool used to verify conformity with functional requirements. A generated model can be verified through a LTSA model checker (*e.g.* liveness, progress). For the versioning system example, we can check that the update policy is preserved after the style introduction (*i.e.* there is no deadlock due to the SAFE processes). But extrafunctional properties should be the key concerns for selection.

5 Preparing Architecture Quality Analysis

For the moment, our framework focuses mainly on the functional services and properties of the system's software architecture. However, extrafunctional properties are also to be taken into account in the style definition models and in style-specific models so as to compare architectural choices. Some styles are intrinsically known to meet extrafunctional requirements more easily. In particular, peer-to-peer distributed systems are well recognised for scalability and reliability characteristics; publish-subscribe systems support extensibility, anonymity of actors and dynamicity of incoming/outgoing participants; client-server systems are often considered better for flexibility but worst for availability due to single points of failure.

Using our approach, some security properties such as message authenticity, confidentiality and integrity may be early verified, *e.g.* based on earlier formal work of Schneider [14]. Pursuant to this proposition, the system is specified in the CSP process calculus (quite similar to FSP) and includes an additional enemy process in order to model potential security attacks (*e.g.* message leakage, message alteration). The security properties are then introduced as properties on event traces. For example, for message confidentiality, it can be stated that each message received by the enemy process must have been sent to it before. This ensures that the message can only be accessed by the component which was intended to receive it. Security analysis on a style model could then warn anonymity weaknesses in face of strong security requirements. The designer could then have a look at other styles or investigate the introduction of security mechanisms after model transformation. In order to preserve architecture quality after refinement by introducing such mechanisms, cross-cutting concerns between extrafunctional properties need to be addressed. Extrafunctional properties influence each other [15] and can lead to conflicts in face of requirements (*e.g.* a security mechanism impacts performance issues).

On the other hand, extensions to process algebras allow to introduce a timed interpretation, *i.e.* to specify time performance properties as computing time or message transfer time. These existing works seem a good starting point to introduce extrafunctional properties into the framework. Their integration and the extension to other properties (*e.g.* scalability, dependability) are under investigation.

6 Related Work

An application can rely on several styles. As an early example, Garlan and Shaw [8] have defined a collection of architectural styles showing how different architectural solutions for a same problem offer different benefits. For example, they outline four distinct architectural designs for the *Key Word In Context* (KWIC) system (*i.e.* shared data, abstract data types, implicit invocation, and pipes and filters). However, their early proposal does not open up the way to a systematic transformation of functional models with different styles. In a distributed system, selected interaction mechanisms impact locally on extrafunctional properties of a point-to-point interaction. But the choice of the right architectural style also broadly depends on the emergent properties addressed by the overall structure. To guide the selection, a formal specification of common distributed styles is a prerequisite for early analysis. Moreover, styles could be combined [16] to meet specific requirements, encouraging analysis assistance. Our proposal tends to go one step further in this direction.

In the last ten years, a number of architecture description languages have been proposed to represent software architectures (*e.g.* [17,18,19]). More recently, development and deployment of large distributed systems also conduce to rely on component models with dedicated languages (*e.g.* CCM, Fractal, GCM). Some of those architecture description languages open up the way to analysis by

incorporating formal specifications (*e.g.* CSP or pi-calculus, Z, OCL, types, graph grammars or chemical abstract machine). However, their usage does not directly dissociate styles from system models and therefore limits the definition of a fixed repository of styles independent of the architect's know-how.

To the best of our knowledge, three mature frameworks provide some architecture stylistic guidance. Morisawa and Torii [20] have restricted their exploration to the client-server style alternatives and propose to evaluate them under some of the ISO 9126 quality issues (*e.g.* data security, reply to user for time performance). Metrics with maximum range are fixed on properties. The target style may be selected thanks to size and distance functions regarding requirement criteria. It is worth noting that several extrafunctional characteristics are not taken into account at the level of early design choices. By separating concerns, the ISO 42010:2007 recommended practice [21] now provides some elements within a conceptual framework for describing and analysing complex architectures in terms of architectural viewpoints. However, there is still a recognised gap between requirements and architectural description phases.

The NFR framework [22] considers extrafunctional goals to guide the designer and cover a far-reaching area of extrafunctional requirements. A nonfunctional requirement is defined in a tree description as a combination of lower level requirements (*e.g.* security is a combination of confidentiality, integrity and availability) and pattern-mechanisms to meet them (*e.g.* confidentiality can be ensured by using authentication and access matrix). Informal positive and negative contributions between mechanisms and requirements are elaborated by an expert to guide the selection. Architectural patterns can also be attached to a tree, based on property contributions known by experience. In this case, an architecture guidance is a mechanisms and patterns proposal in conformance with requirements.

After identifying actors and goals for a system, the i^* framework [23] permits to represent dependencies between components (*e.g.* tasks, resources) and then elaborates alternative architectures. By selecting predefined architectural patterns corresponding to quality attributes, refined solutions of an instance system are then evaluated through metrics (actor-based and dependency-based). This framework, used for system reengineering (*i.e.* SAR*i*M and PR*i*M methods) under quality issues, fruitfully distinguishes functional and extrafunctional aspects. However, its pattern-based approach is flexible but less rigorous than a formal transformation, and stylistic choices remain handmade at a pattern-level rather than at a high structural one.

7 Conclusion and Perspective

Designing software architecture of good quality, satisfying requirements, is recognised as a complex task. To facilitate construction in the lifecycle, well-known architectural abstractions could be employed at the early design stages. However, a style choice could considerably impact extrafunctional properties in the rest of the design cycle with sometimes large influences at the implementation stages. Requirement specifications do not always impose or promote style models.

Depending on the functional and extrafunctional properties, an architect faced with alternatives could rely on his/her know-how. Thus, alternatives scope and criteria for style selection often remain implicit [2] and tend to be put forward in the design process.

It is critical to better manage extrafunctional properties in the architecture design as an engineering discipline. Architecture analysis at the early stages is crucial to satisfy requirements in the final application or system and limits tacit choices for the architect. By specifying the behaviour of an application only in terms of functional concerns, independently of a style, our conjecture prepares for the separation of distribution concerns. Indeed, some extrafunctional properties [15] are intrinsic to specific styles [16] (*e.g.* reliability in client-server or scalability in peer-to-peer).

Therefore, we have proposed in this paper a model-driven framework to shed light on the appropriateness of separating functional system concerns from distributed architectural style. Relying on a process calculus, our formal design process enables different architectural solutions to be systematically generated by a transformation model. We tackled style independent and style specific models at an abstract specification level, and have shown through a classical distributed application that functional models could be expressed by a designer independently of interaction mechanisms. Based on an expandable repository of styles, a functional model of an application could be systematically composed with alternative styles for further comparative analysis before development. The applicability of our approach has been justified, through three distributed styles, with a classical distributed version control system case study. For the architect faced with design alternatives, our framework thus provides early formal support and allows to address the software architecture quality at higher design stages.

Following the NFR framework [22] proposal for quality attributes, future work will take into consideration other extrafunctional properties for styles and analyse their impact on the composition process. Is a property simply intrinsic or not to a style? How to match the extrafunctional requirements with the results of analysis in our approach? Noting that resulting models are analysable with respect to some extrafunctional properties, they could be extended with patterns of mechanisms to meet requirements. Finally, a process calculus is not ideally suited to all of the extrafunctional concerns. Further, it might restrict the expandableness of the style repository (*e.g.* in order to address more dynamic notions). Other formalisms could also be considered in our framework to meet requirement specifications and to increase the expressiveness of style description.

References

1. Bhattacharya, S., Perry, D.E.: Predicting architectural styles from component specifications. In: Proceedings of the 5th Working IEEE/IFIP Conf. on Software Architecture, pp. 231–232. IEEE Computer Society Press, Los Alamitos (2005)
2. Kruchten, P., Lago, P., van Vliet, H., Wolf, T.: Building up and exploiting architectural knowledge. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214. pp. 43–58. Springer, Heidelberg (2006)

3. Shaw, M.: Comparing architectural design styles. IEEE Software 12(6), 27–41 (1995)
4. Orfali, R., Harkey, D., Edwards, J.: The essential client/server survival guide. John Wiley and Sons, Chichester (1996)
5. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
6. Androutsellis-Theotokis, S., Spinellis, D.: A survey of peer-to-peer content distribution technologies. ACM Computing Surveys 36(4), 335–371 (2004)
7. OMG: Object Management Group (Lillerand, J., Mukerji, J. (eds.)) Model Driven Architecture Guide, version 1.0.1 (June 2003), `http://www.omg.org/docs/omg/03-06-01.pdf`
8. Garlan, D., Shaw, M.: An introduction to software architecture. Advances in Software Engineering and Knowledge Engineering 2, 1–39 (1993)
9. Bushmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A system of patterns. John Wiley and Sons, Chichester (1996)
10. Shaw, M., Clements, P.: Toward boxology: preliminary classification of architectural styles. In: Proceedings of the second international software architecture workshop (ISAW-2) on SIGSOFT 1996 workshops, pp. 50–54. IEEE Computer Society Press, Los Alamitos (1996)
11. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. 2(2), 115–150 (2002)
12. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley and Sons, Chichester (2006)
13. Uchitel, S., Chatley, R., Kramer, J., Magee, J.: LTSA-MSC: Tool support for behaviour model elaboration using implied scenarios. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619. pp. 597–601. Springer, Heidelberg (2003)
14. Schneider, S.: Security properties and CSP. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp. 174–187. IEEE Computer Society Press, Los Alamitos (1996)
15. Chung, L., Nixon, B.A., Yu, E.: Using non-functional requirements to systematically select among alternatives in architectural design. In: First International Workshop on Architectures for Software Systems (IWASS), pp. 31–43 (1995)
16. Mehta, N., Medvidovic, N.: Composing architectural styles from architectural primitives. In: Proceedings of the 9th European Software Engineering Conference (ESEC), pp. 347–350. ACM press, New York (2003)
17. Magee, J., Kramer, J.: Modelling distributed software architectures. In: First International Workshop on Architectures for Software Systems (IWASS) (1995)
18. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
19. Zhang, S., Goddard, S.: xsadl: An architecture description language to specify component-based systems. In: Proceedings of the IEEE Int. Conference on Information Technology: Coding and Computing, pp. 443–448. IEEE Computer Society, Los Alamitos (2005)
20. Morisawa, Y., Torii, K.: An architectural style of product lines for distributed processing systems, and practical selection method. In: ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 11–20. ACM, New York (2001)

21. ISO: International Organization for Standardization: Systems and Software Engineering – Recommended practice for architectural description of software-intensive systems. ISO/IEC DIS 42010, 90.92 review stage (December 2007)
22. Chung, L., Gross, D., Yu, E.: Architectural design to meet stakeholder requirements. In: Donohue, P. (ed.) Software Architecture, First Working IFIP Conference on Software Architecture (WICSA1), Vienna, Austria, pp. 545–564. Kluwer Academic Publishers, Dordrecht (1999)
23. Grau, G., Franch, X.: A goal-oriented approach for the generation and evaluation of alternative architectures. In: Oquendo, F. (ed.) ECSA 2007. LNCS, vol. 4758. pp. 139–155. Springer, Heidelberg (2007)