Noname manuscript No.
(will be inserted by the editor)

Performance evaluation of Fractal component based systems

Nabila Salmi · **Patrice Moreaux** · **Malika Ioualalen**

the date of receipt and acceptance should be inserted later

Abstract Component based system development is now a well accepted design approach in software engineering. Numerous component models have been proposed and for most of them, specific software tools allow building Component Based System (CBS). Although these tools perform several checks on the built system, few of them provide formal verification of behavioural properties nor performance evaluation of the resulting system. In this context, we have developed a general method associating to a CBS, a formal model, based on Stochastic Well formed Nets, a class of high level Petri Nets, allowing qualitative behavioural analysis together with performance evaluation of this CBS. The definition of the model heavily depends on the (run time) component model used to describe the CBS. In this paper, we instantiate our method to Fractal CBS and its reference Java implementation Julia. The method starts from the Fractal architectural description of a system, and defines rules to systematically generate elements models of the CBS and their interactions. We then apply a structured method both for qualitative and performance analysis taking into account the given implementation of the Fractal model. The main interest of our method is to take advantage of the compositional definition of such systems to carry out an efficient analysis. The paper concentrates on performance evaluation and presents our method step by step with an illustrative example.

Keywords Modelling · performances · component · interaction · CBS · SWN · synchronous composition · asynchronous composition

N. Salmi · P. Moreaux
LISTIC, Université de Savoie, France. E-mail: {nabila.salmi, patrice.moreaux}@univ-savoie.fr

N. Salmi · M. Ioualalen
LSI, Université des Sciences et Technologie, Alger, Algérie. E-mail: {salmi,ioualalen}@lsi-usthb.dz

1 Introduction

Component based technology [28] is an attractive paradigm, widely used for the development of software and hardware systems. In this paradigm, components are developed in isolation or reused from previous works, and are then assembled to build a *Component Based System* (CBS). The main goals of such approaches are to produce high quality applications, reduce cost and time developments, and achieve more reliability, higher maintainability and easier upgrade. Since the mid'70, a lot of component models have been proposed in the literature, among them EJB, CCM and CORBA, COM+/.NET, Fractal [23, 17, 18, 22], and are used in effective applications. Building CBS is supported by sets of tools associated to each component model. These tools allow description of the CBS through Architecture Description Languages (ADL) [21] and provide the architect with several checking tools mainly based on syntactic analysis of the description and the source code of the elementary units of the component model (such as components, connectors, interfaces, configuration descriptions). They ensure for instance that interfaces required by one component are provided by another component and that the interconnection between these interfaces is provided either directly or through a compatible connector. Beyond this "static" analysis, the complexity of many CBSs requires verification of behavioural properties such as deadlock-freeness, reachability of some states and so on. This is achieved by defining a formal semantics to the component model and by (model) checking required properties against the semantic model of the CBS. We emphasize that such an analysis should be based on a *runtime* component model and not only on an architectural component model. Formal semantics of the component model is most often given by a Labelled Transition Systems (LTS) either directly or derived from a higher formal model such as process algebras or state based models (for instance Stat-

echarts, Petri Nets) generating a LTS. To cope with the classical problem of the state space explosion (huge size of the LTS of the system), model checking should be based on the formal models of the components and their composition, allowing several levels of component behaviour models (abstraction) [11] and merging of several component models into a single one (hierarchy) [20].

For what concerns performance analysis of CBSs, it is usually carried out through measures on existing systems, whereas predictive performance evaluation remains an important field of application in the perspective of software performance engineering [27]. This is the context of our work and we develop stochastic models of CBSs to provide performance indices about them during the design phase. Here also we should try to take into account the specific, component based, architecture, as for instance [25] in the context of dependability modelling using the description language AADL, and [29,16] which start from a UML design model and build a performance analysis model based on Layered Queuing Networks [14].

Among component models that gained attention these last years, the Fractal model [9] offers a hierarchical and reflective component model with dynamic configuration, component composition, management and sharing capabilities. A Fractal component is a *runtime* entity exposing provided and required services through a set of *functional interfaces*. It is also endowed with a set of control capabilities, defined through *control interfaces*. Fractal components are assembled to form a software application, either "by hand", that is directly in the Java code, but most often with the help of software development tools, like the Fractal ADL, etc. Fractal CBSs may be developed using several target programming languages (Java, C, C#, SmallTalk). For each language, a reference implementation is defined (Julia for Java, Think for C). Note that a completely specified runtime Fractal CBS cannot be modelled without taking into account specific implementation details, which implies knowing the target language of the application.

In the present paper, we focus on performance analysis of *Julia Fractal CBSs* based on formal models of the components and systematic building of the formal model of these CBSs. The main advantage of our approach is to exploit as much as possible, the compositional architecture of the system in order to reduce complexity of performance indices computation.

Components and the global system are modelled with the Stochastic Well formed (SWN) formalism [10], a special class of high level Petri nets, useful to express symmetrical behaviours and allowing performance analysis. Although we can take into account both functional and non-functional aspects of a Fractal CBS within our approach, we concentrate in this work on "stable" configurations, that is to say on Fractal architectures after initialization phase or between reconfiguration phases since we compute steady-state performance indices.

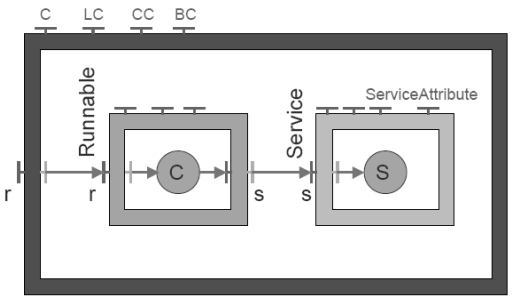


Fig. 1: The Fractal component model

We first translate systematically interfaces and interactions between Fractal components in the SWN context. Then for a given Fractal CBS, we suppose given SWN models of the primitive components. Next, from the architecture description of a Fractal system (provided by Fractal ADL), we show how to build the global SWN of the CBS. We have now a global SWN and a set of sub-SWNs modelling the components. Finally, we apply, when possible, a structured method allowing to compute performance indices. In order to apply this structured compositional method, we derive from the architecture of the CBS, a compositional view of the system at the SWN level, and we apply a modified version of our previous works [12,13] which proposed a structured analysis method for either a synchronous or asynchronous decomposition of SWNs. This method is based on a combined aggregation/tensorial representation of the underlying Markov chain of the global SWN.

The paper is organized as follows. Section 2 reminds main features of the Fractal component model and introduces an illustrative example. Section 3 presents the general method we have developed for behavioural qualitative analysis and performance evaluation of CBSs. Then, we describe in section 4 application of our method in the Fractal CBS case, providing a SWN based formal model of any such CBS. Section 5 shows how to manage a structured performance analysis of the model built previously and presents some results for the illustrative example. We conclude and describe future work in section 6.

2 The Fractal Component model

Fractal [8,9] is a general component model developed within the consortium ObjectWeb by France Telecom R&D and the INRIA. It is intended to implement, deploy, monitor and dynamically configure complex software systems, including operating systems and middleware.

2.1 Main features

A Fractal component is a *runtime* entity that interacts with its environment (i.e. other components) through well-defined *interfaces* (figure 1[1]). An interface is an access point to a component, that specifies provided services or required services exposed by other components. There are two kinds of interfaces: *server* interfaces correspond to points accepting incoming operation invocations, and *client* interfaces support outgoing operation invocations.

A Fractal component possesses two parts: a *content part* and a *controller part*. The content part consists of a finite number of other components, called *sub-components*, making the model recursive and allowing components to be nested at an arbitrary level. At the lowest level, a Fractal component is a black box, called *base* or *primitive* component, that doesn't provide introspection or intercession capabilities. A Fractal component whose content is not empty is said *composite*.

The controller part, termed *the membrane*, provides a set of *control* interfaces, supporting introspection (monitoring) and reconfiguration of internal features of the component, such as suspending and resuming activities of a subcomponent. Several control interfaces have been defined in the Fractal model specification, namely:

- The Life cycle Controller (LC) manages the component life cycle, in support for dynamic reconfiguration. Basic methods supported are starting and stopping the execution of the component.
- The Binding Controller (BC) manages connections or *bindings* to other components. It allows to bind and unbind interfaces of communicating components (see below).
- The Content Controller (CC) provides content operations such as listing, adding and removing sub-components.
- The Attribute Controller (AC) exposes getter and setter operations for attributes (configurable properties, service attribute in the figure) of a component.

The *membrane* of a component can have *external* and *internal* interfaces. External interfaces are reachable from outside the component, while internal interfaces are only reachable from its sub-components, and are not visible from the outside. External interfaces of a sub-component are *exported* by *interceptors* as an external interface of the composite parent component. Interceptors may introduce specific operations between incoming and outgoing operation invocations of an exported interface.

A component may be *shared* by several enclosing components. In this case, it is subject to the control of their

[1] Figures related to the Fractal model or Fractal CBS are reproduced from documentation on the Fractal project Web site: `http://fractal.objectweb.org`

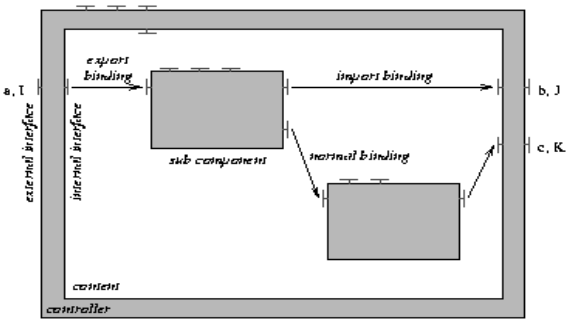


Fig. 2: Bindings between Fractal components

respective controllers. The exact semantics of the resulting configuration is determined by an encompassing component which encloses all relevant components in the configuration.

In order to define the architecture of an application, Fractal components are connected by *bindings* (see figure 2) i.e. connections between two or more components. The Fractal model specification defines *primitive* and *composite* bindings. A primitive binding is a direct connection between a client interface and a server interface. It can be a *normal binding* when the client and server interfaces are external, and the corresponding communicating components have a direct common enclosing component. It can also be an *export binding*, respectively an *import binding*, when the client interface is internal, and the server interface is external, and the component exposing the service (server) is a sub-component of the other (respectively, the client interface is external, and the server interface is internal, the component requiring the service (client) is a sub-component of the other). Whereas, a composite binding is a communication path between an arbitrary number of component interfaces. It is itself a Fractal component, built out of a combination of primitive bindings and ordinary components. Hence, binding generalizes the notion of connector in other component models.

The Fractal model specification defines a set of constraints on the interplay between functional and non functional operations, namely:

- Content and binding control operations are possible only when the component is stopped.
- A component can emit or accept invocations when started.
- A component does not emit invocations when stopped, and must accept invocations through control interfaces.

In order to define component architectures for the Fractal model, an open and extensible language has been developed: the Fractal Architecture Description Language (ADL) [6].

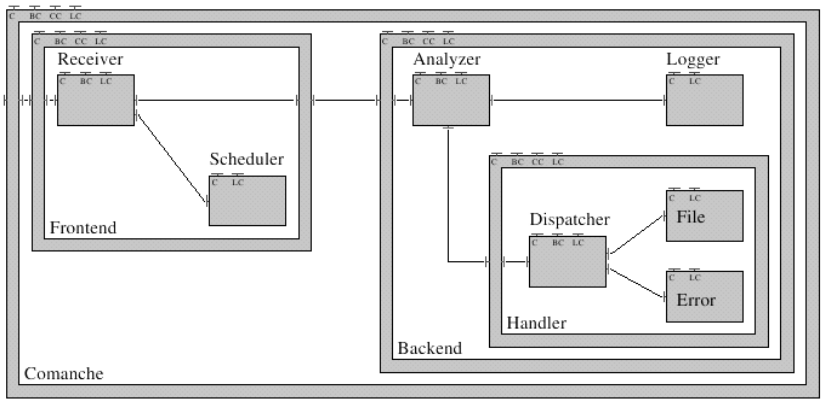


Fig. 3: A Fractal application: the Comanche Server

2.2 Fractal Architecture Description

As its name implies, an ADL definition describes a component architecture, and only that, i.e. its does not describe the resulting runtime system. It is made of an open and extensible set of ADL modules, where each module defines an abstract syntax for a given architectural "aspect", such as interfaces, bindings, attributes or containment relationships.

Fractal ADL is an XML based ADL that can be used to describe Fractal component configurations. In this paper, we only consider the ADL aspects of Fractal ADL. In fact, it is also the name of a toolchain framework allowing introduction of new tools in the generation process of a Fractal based application.

Fractal ADL is strongly typed. The first step to define a component architecture is therefore to define the types of the components. Each component type must specify what components of this type provide to, and require from other components. The definition of a primitive component is done by specifying the interfaces it provides, the interfaces it requires, and the class that implements this component.

A composite component is similarly defined by specifying its interfaces, the sub components it contains, and bindings between these sub components and with the composite component itself. Component definitions are then defined by extending existing definitions. The extension mechanism is similar to class inheritance, i.e. a sub definition can add and override elements in its super definition. This mechanism can be used to define concrete components as sub definitions of abstract component definitions.

Once the application's architecture has been defined, it can either be compiled, which gives a Java class (for the Julia implementation), in source code, or it can be directly interpreted. In both case, the Fractal ADL parser performs preliminary verifications to check the architecture and, in particular, to check that there is no missing or invalid binding.

2.3 Example: the Comanche application

Along the paper, we use an effective example to illustrate our method. This example consists of a minimal HTTP Server, *Comanche*, used in [7] to illustrate how to implement and deploy Fractal component based applications. This server accepts connections on a server socket and, for each connection, starts a new thread to handle it. Each connection is handled in two steps: the request is analyzed and logged to the standard output, and then the requested file is sent back to the client (or an error is returned if the requested file is not found).

Two main services are identified at a high level of the Comanche architecture, namely a request receiver service and a request processor service. At a lower level, the request receiver service uses a scheduler service, responsible of creating a new thread for each request. The scheduler service can be implemented in several ways: sequential, multi thread, multi thread with a thread pool, and so on. We suppose in our case that it is multithread.

In order to process the request, a request analyzer service and a logger service are used before effectively responding to a request. This response is itself constructed by a request dispatcher service, that uses a file server service, or an error manager service. The request dispatcher service dispatches requests to several request handlers sequentially, until one handler can handle the request (we can then imagine file handlers, servlet handlers, and so on).

In terms of components, the application defines a component for each service, leading thus to seven primitive components: (request) Receiver, (request) Analyzer, (request) Dispatcher, File (request handler), Error (request handler), Scheduler and Logger (see figure 3).

Some of these components are encapsulated in composite components: The Receiver and Scheduler components are included in the Frontend component, the Dispatcher, File and Error components make up the (request) Handler component, and this latter together with the Analyzer and the Logger make up the Backend component.

The Frontend and Backend composite components are themselves contained in the Comanche server application, which is the highest level component.

3 A general method for analysis of CBSs

As mentioned in the introduction, we have developed a general method for behavioural qualitative and performance analysis of CBSs. Before explaining the application of the method to Julia Fractal CBS, we present the main lines of the method. The method is based on the Stochastic Well-Formed Petri Net model (SWN), a high level (coloured) model of Petri net with probabilistic extensions for performance analysis. Our choice of the SWN formalism is first motivated by the fact that we need a state based model to be able to evaluate performance indices related to configurations of the systems (number of requests pending in some part of the system, mean usage time of some resource, etc.). Petri Nets are state based models which are well known for being able to model complex systems with concurrency and conflicts, even in the stochastic context, in contrast with Queuing networks or process algebras models for instance. Moreover, although Petri Nets are not by themselves a compositional model, interaction between Petri nets representing sub-components may be easily defined as transition or place "fusion" (merging). If complex primitive components are involved, high level Petri Nets are almost inevitably required so that the SWN model is nicely adequate. The SWN model can also take advantage of behaviourial symmetries of system's entities if there are such symmetries. Finally, SWNs are a well studied class of high level stochastic Petri nets and benefit from a large set of analysis algorithms and tools. Among formal models of Fractal CBSs, [3] also proposes an approach for specification and verification, but based on LTS derived from communicating automata networks [1,2]. However, this work is devoted to the Fractive [4] implementation of the Fractal component model and does not allow us to compute performance indices of Fractal CBSs.

3.1 The Well formed and the Stochastic Well formed Petri nets models

A Well-formed (WN) net [10] is a high level Petri net model. It is a coloured Petri net, where places and transitions are provided with a structured type of tokens.

In this model, tokens are grouped into basic classes called *colour classes*. These classes are bringed together to form a colour domain, which is associated to places and transitions. Colours of a place label its tokens, whereas colours of a transition define possible firings of the transition. Thus, an initial marking of a place is defined as a multiset (bag) of coloured tokens. A colour function is attached to each arc: its role is to define for, a given colour of the associated transition, the number of coloured tokens to add or to remove from the attached place.

A colour domain is a Cartesian product of colour classes. A total order, expressed by a successor function, can be defined on a colour class. The Cartesian product defining a colour domain can be empty (for example, in the case of a place containing neutral tokens). It can also contain repetition of a class (modelling internal synchronization of this class). A colour class, grouping colours of same nature (eg. processes, resources), can be divided into static sub-classes, where a sub-class contains colours with identical behaviours, even in terms of performance.

A colour function is built from standard operations (linear combination, composition, etc) of basic functions. The projection (denoted by X or X_i^j in figures) selects an element of a tuple; it is represented by a typed variable or by X if no confusion is possible. The synchronization/diffusion (denoted by Si or Si,k) returns the set of all colours of a class (Si) or a sub-class (Si,k). The successor function is defined for ordered classes only and returns the colour following a given colour.

A transition or an arc function can be guarded by an expression which is a linear combination of atomic predicates. An atomic predicate expresses the equality of two variables, or restricts the colour domain of a variable to a static subclass [10]. A predicate is evaluated on colours of a transition firing.

The structured definition of a WN allows us to exploit automatically system symmetries, by compacting its reachability graph, leading to a *Symbolic Reachability Graph* (SRG). An SRG is composed of *symbolic markings*, where each symbolic marking represents a set of ordinary (coloured) markings having equivalent behaviours (see [10] for more details). Several qualitative properties can be checked on the SRG (reachability of a marking, deadlock freeness, etc.)

From WNs was derived the *Stochastic Well-formed* (SWN) model, which associates to each transition an exponentially distributed delay. This delay can depend on static sub-classes of the colours considered at firing. The SRG of an SWN, augmented with stochastic firings information, results in an aggregated Markov chain of the chain derived from the coloured net. Thus, we can study performances of a system directly on this aggregated chain. Formal definitions of WN and SWN are given in appendix A.

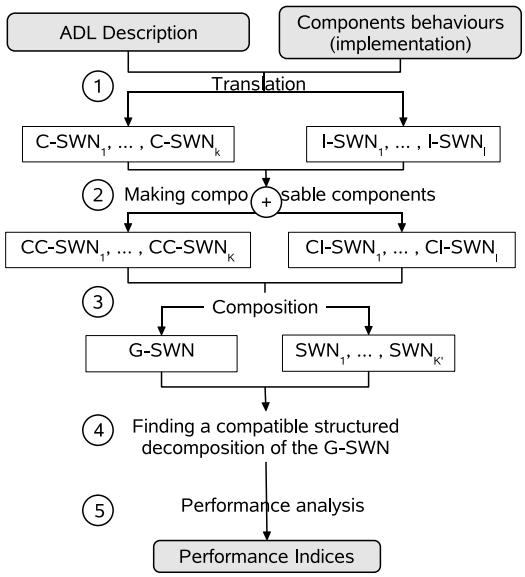


Fig. 4: Analysis method of a CBS

3.2 Overview of the method

The method starts by building a global SWN for a CBS, viewed as a composition of SWN models of components and interconnections. Then, a structured analysis method is applied for deriving performance indices.

Before giving details on the approach, we introduce some terms related to component models used in the sequel.

3.2.1 Terminology

Component SWNs An element of a CBS (component or connector) is modelled with an SWN model. This model is built from the component content behaviour or implementation and predefined sub-models modelling interfaces. The component content can be a simple component in case of a primitive component, or connected sub-components in composites. The primitive component behaviour is modelled by a SWN by an expert, where places correspond to data storage resources, state of active entities, etc. and transitions to activities consuming data and producing other data. The composite component behaviour consists of an interconnection of its sub-components behaviours with adapted interaction (or connector) components. We term *Component SWN* (C-SWN) models of primitive or composite components. Obviously, obtained C-SWNs can be gathered into libraries, and reused possibly later.

The translation of an interface of a component in the corresponding C-SWN is heavily dependent on the semantics of the runtime component model. In general, it is a subnet of the C-SWN which may consist of just a set of places, a set of transitions, or can be more complex with a combination of places and transitions. We provide translation of interfaces for Julia Fractal CBS in the next section.

Interaction SWNs Components are related by interaction relationships described in the architecture description, and so are the C-SWNs. This interaction relationships between C-SWNs are translated into *Interaction SWNs* (I-SWN). An I-SWN connects two interfaces of distinct C-SWNs. I-SWNs could be more or less complex SWNs; in the Julia Fractal case, we will show that functional interfaces could be directly interconnected, so that I-SWNs are useless.

Global SWN model of the CBS The *global SWN* (G-SWN) of a CBS is built from the set of C-SWNs and their corresponding I-SWNs. The C-SWNs are composed together with I-SWNs, through fusion of interfaces elements, obtaining thus a global model.

3.2.2 Steps of the method

We can describe our method into five main steps. The first three of them are devoted to the construction of the global SWN of a CBS and its compositional structure, and the two other steps aim to perform the analysis of the system:

1. Translation of the ADL description of the CBS, together with the description of components behaviours (i.e. source code), into the SWN framework, leading to a set of C-SWNs and a set of I-SWNs.
2. Modification of the C-SWNs and I-SWNs so that to be composable with others, in the sense of Petri net composition (fusion of places or transitions). This modification should not impact the semantics of the modelling. The set of obtained models are called *Composable Component SWNs* (CC-SWNs) and *Composable Interaction SWNs* (CI-SWNs).
3. The CC-SWNs and the CI-SWNs are then composed together through fusion of element interfaces, providing the global G-SWN model corresponding to the whole system. CC-SWNs and CI-SWNs are now seen as a unique set of subnets $\mathrm{SWN}_1, \ldots, \mathrm{SWN}_{K'}$ (see figure 4, step 3).
4. We then start from the set of $\mathrm{SWN}_{k\ k\in\{1,\ldots,K'\}}$. We search the set of SWNs $(\mathcal{N}_k)_{1\leq k\leq K''}$ representing a possible decomposition of the G-SWN, that fulfill conditions for a structured representation of the SRG and its aggregated generator (see section 5.1). These SWNs can be one of $(\mathrm{SWN}_k)_{1\leq k\leq K'}$ or a groupment of a subset of them.
5. When conditions are satisfied, the structured analysis method is applied to compute performance indices (see section 5).

4 Mapping Julia Fractal CBS to SWN models

In this section, we address the SWN modelling of Julia implementation of a Fractal CBS, following the method presented above. We assume that the CBS is defined through an ADL description and a set of Java classes corresponding to the primitive components.

4.1 General considerations

4.1.1 Modelling stable configurations and dynamics of Fractal CBS

The architecture of a Fractal CBS is defined first by an initial configuration and then may evolve through runtime reconfigurations. Although the capability of runtime evolution of the architecture is a leading property of Fractal CBSs, we claim that behavioural analysis of Fractal CBSs cannot be efficiently carried out with a single formal model of the system. In fact, any modification of the structure (binding changes, addition or deletion of sub-components, etc.) requires a specific modelling, primarily devoted to check that, starting from a configuration A, the application will eventually reach a given configuration B. In contrast, analysis of a configuration (say A or B) addresses both qualitative (reachability, deadlock freeness, etc.), and quantitative (computation of performance indices) aspects of this configuration. Moreover, in the performance evaluation context, switching from A to B probably corresponds to a "short" time period (transient phase), whereas performance indices of software systems are mainly computed over long periods (steady-state analysis).

In the present work, we do not address the verification of the reconfiguration behaviours of Fractal CBSs and we concentrate on "stable" (i.e. fixed) architectures. Hence, since control interfaces of Fractal components are used to manage the initialization and reconfiguration phases of the architecture, we do not model control interfaces.

Note however that we can *compare* performances of two configurations A and B, studying each one with our method. We have then two G-SWNs and two sets of SWN subnets.

4.1.2 Implementation dependencies

Since we wish to derive performance indices of a Fractal CBS, we emphasize that the architecture description of the CBS does not allow alone performance modelling: *it must be complemented with information from the implementation* of the component model. We can find in the literature several Fractal implementations developed in Java and C: Julia (the Java reference implementation), Fractive [4] and AOKell [26] in Java, and Think [15] the C implementation. Studying these implementations, we noted that they differ significantly. For instance, Fractive uses an asynchronous (late) operation invocation which allows the client to continue processing until it needs results returned by the service. In contrast, Julia and Think use a classical synchronous (blocking) method call. In this paper, we model the Julia implementation of Fractal.

4.1.3 Colours

Basic colour classes can model either data entities or active entities of components. Data entities consist of data flow such as requests, parameters of requests, results or even resources. Active entities are execution flows (processes and threads).

We can illustrate colour semantics by basic colours used in the Comanche example. We consider sockets, HTTP requests, files, streams, and even threads as our basic colour classes. In our model, as we abstract some component details, we use the following basic colour classes: *UC* models HTTP User requests, *IDS* models scheduled threads, *IDL* is the basic class of Log requests, *IDF* that's of File requests, and *IDE* models Error identifications.

4.2 Translation to SWN models

Building the G-SWN of a Fractal CBS starts from the low level of architecture, and goes up into the higher levels of architecture until reaching the highest level.

First, primitive components are modelled, leading to a set of C-SWNs. Obviously, abstraction may be used at this stage by selecting an appropriate level of details of primitive components. At the highest abstraction level, the content of a primitive component can be modelled by a very simple SWN (see figure 13). Modelling a component requires modelling its interfaces. Functional interfaces are modelled by transitions of the SWNs (see details below). The obtained C-SWN of a component is transformed into a CC-SWN, with the translation of functional interfaces to the adequate form as it is explained later. Next, based on bindings between components, composite components of higher levels are translated by assembling CC-SWNs of enclosed sub-components. From this translation, we obtain also a CC-SWN for each composite component. This CC-SWN may also be simplified to abstract some detailed behaviours as for primitive components (see below and figure 13).

4.2.1 Functional interfaces

Functional interfaces of a component consists of a required (client) interface which needs a service from other components, or an offered (server) interface exposing a service to others. For a composite component, these interfaces, said *external*, allow to do export and import bindings, connecting thus sub-components interfaces to functional interfaces

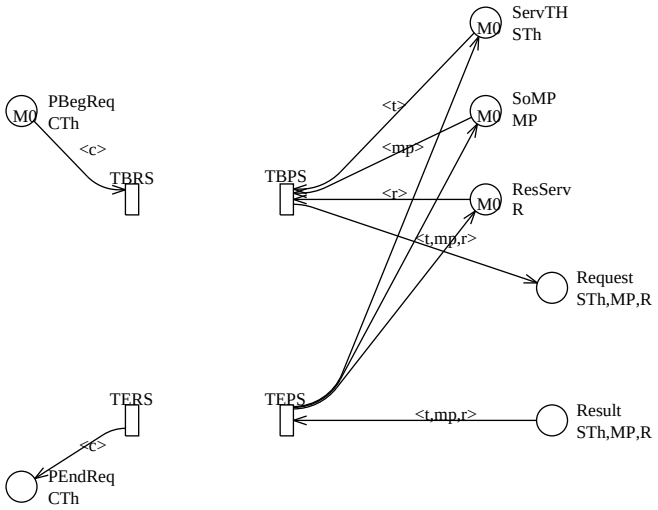


Fig. 5: SWNs models of interfaces: Client(left), server(right)

of the enclosing composite component. In each case (external or not), a functional interface is modelled with a set of coloured places and transitions as given by the following mapping rules.

Mapping rule 1 A server interface of a component, identified by a set of colours STh modeling possible server threads, offering a set MP of operations or methods with their parameters, is modeled by representing the beginning of service provided and its ending with two transitions, respectively t_{BPS} and t_{EPS} (figure 5 (right)). t_{BPS} is controlled by two places *ServTH* and *SoMP* modeling respectively server threads and methods with their parameters. Possibly, a third place *ResServ* coloured with a basic class R is used modeling specific resources needed during execution of a service. Whereas, t_{EPS} is controlled with a place *Result* coloured with tuples belonging to $STh \times MP \times R$ modeling the result of request processing.

Mapping rule 2 A client interface of a component, identified by a set of colours CTh modeling possible request threads of the client component is modeled with two transitions t_{BRS} and t_{ERS} representing the beginning of service request and its ending (figure 5, left). t_{BRS} (resp.t_{ERS}) is controlled by a place *PBegReq* (resp. *PEndReq*) coloured with CTh and modeling respectively requesting and released client threads.

The model of a server depends a priori on the invoked method and its parameters. In our mapping rule, we do not separate them: if such a level of detail is required for performance analysis, a colour class is defined with static subclasses sorting the possible pairs (method, its parameters) into disjoint subsets. For instance, when the modeller interest is to know whether the size of data sent between components impacts the overall performance of the architecture, he should model the data parameter. Note that if the pair is irrelevant for a given level of detail, we simply omit this colour class.

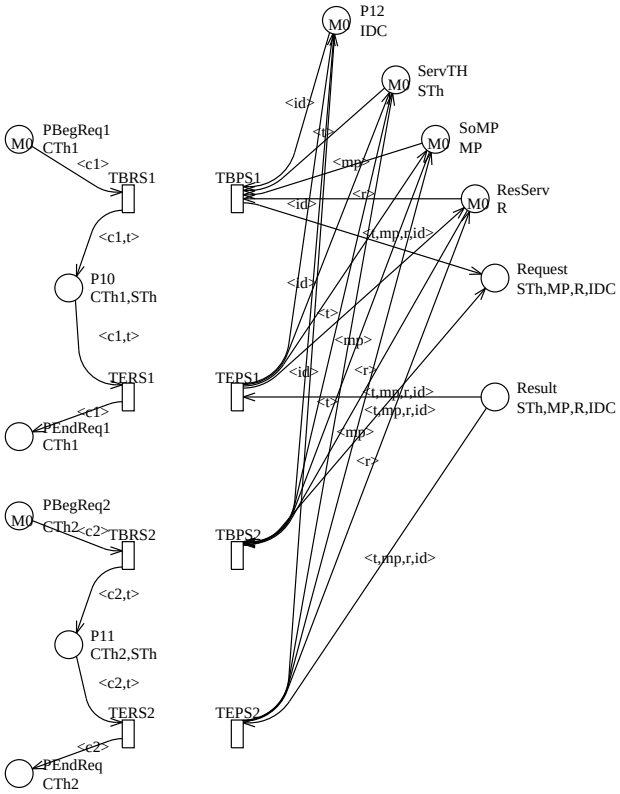


Fig. 6: CC-SWNs interfaces with multiple client interfaces for one server interface

Mapping rule 3 defines the CC-SWN, extending the client interface part of a C-SWN to allow composition of SWN and subsequent structured analysis, without modifying the semantics of the component.

Mapping rule 3: CC-SWN for clients The client interface of a C-SWN is modified, leading to a CC-SWN, by adding a place (and associated arcs) as a postcondition of the beginning transition t_{BRS} and as a precondition of the transition t_{ERS}. (see the left side of figure 5, right). The colour domain of this place is either $CTh \times STh$ or else $CTh \times STh \times IDC$ when several client components, identified with the *IDC* colour class, require the same service (see below).

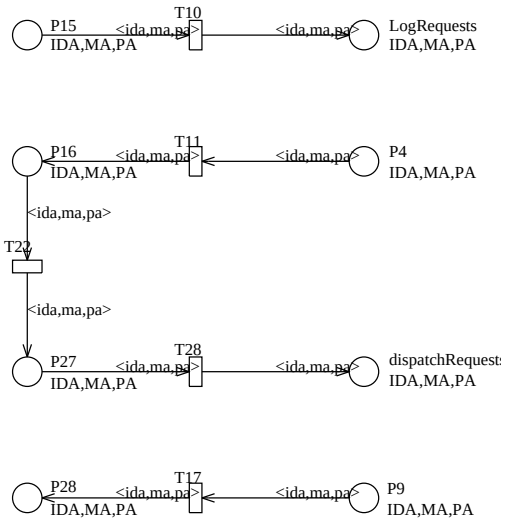


Fig. 7: SWN model of the core of the analyzer component

Dealing with multiple clients When a server interface of a component is bound to several client interfaces of other components, the C-SWN of the server must be modified in order to be composable at the same time with several models of client components, in the sense of Petri nets composition (fusion of places or transitions). This modification gives rise to a CC-SWN and is achieved by applying mapping rule 4. The resulting CC-SWN keep the same semantics as the corresponding C-SWN.

Mapping rule 4: Multiple clients for one server The server interface of a CC-SWN having multiple connected clients is modified as follows with respect to the single client case: (i) The transitions t_{BPS} and t_{EPS} of beginning and ending service are duplicated as many times as the number of clients; (ii) An *IDC* colour class is used to distinguish between several components exposing a client interface. The resulting interface is given by figure 6, right.

4.2.2 Primitive components

The C-SWN of a primitive component is built through several steps:

- Initially, model the "core" of the Fractal component behaviour by an SWN. This is done by analyzing the Java code of the component and fixing a level of details of the model.
- For each set of methods related to a server interface, model the server interface using mapping rule.
- Model internal activities to the server interface, if modelling details are required.
- For each service invocation, model the client interface using mapping rule 2.

Let us illustrate building of the CC-SWN of a primitive component with the analyzer component of the Comanche application. The implementation code of this component is given below.

```
public class RequestAnalyzer implements RequestHandler
{
  private Logger l;
  private RequestHandler rh;
  // functional concern
  public void handleRequest(Request r) throws IOException
  { r.in = new InputStreamReader(r.s.getInputStream());
    r.out = new PrintStream(r.s.getOutputStream());
    String rq = new LineNumberReader(r.in).readLine();
    l.log(rq);
    if (rq.startsWith("GET "))
    { r.url = rq.substring(5, rq.indexOf(' ', 4));
      rh.handleRequest(r);
      }
    r.out.close();
    r.s.close();
    }
  }
```

The analyzer receives requests on its server interface using two basic colours classes *IDA* and *MA*, modelling respectively identified analysis requests and related invoked methods with their parameters. It invokes two operations through two client interfaces: a log operation, and a handle operation. First, we model the core of the component, getting the SWN of figure 7. Then, we add explicit server and clients interfaces introducing request and result transitions for each interface. This gives us the C-SWN of figure 8. Next, this one is completed with places and arcs to get the CC-SWN of figure 9 (bottom).

Considering the Comanche example, figures 9, 10, 11 and 12 show the CC-SWNs of the seven components:

- The receiver exposes a server interface which uses two basic colours classes *IDC* and *M*, modelling respectively identified clients and method with their parameters. This component creates for each received request a task, and sends it to the scheduler to schedule it by creating an associated thread. Once the thread is started, it invokes an analysis request. Thus, the receiver has two client interfaces: one for invoking a schedule request and another for asking for a request analysis.
- The scheduler has only one interface which is of server type. It uses two basic colours classes *IDS* and *MS*, modelling respectively scheduled threads associated to requests and related invoked methods with their parameters.
- The analyzer: see above.
- The dispatcher processes handle operations received on its server interface, by dispatching them to either the file

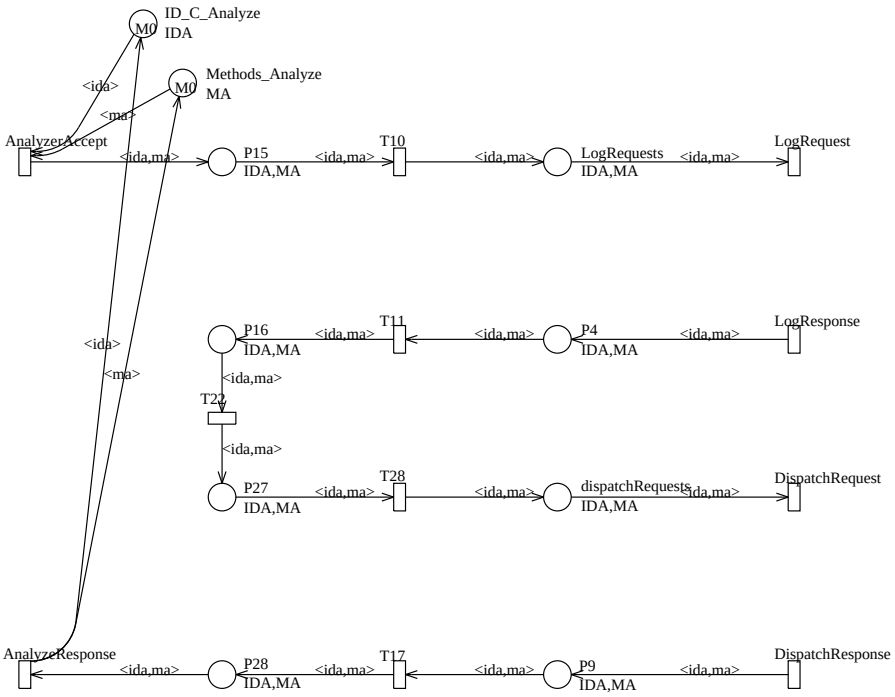


Fig. 8: C-SWN model of the analyzer component

handler or the error handler. Two basic colours classes *IDD* and *MD* are used when an operation is received, modelling respectively identified dispatching requests and related invoked methods with parameters. Dispatching is done through invocation of handle operations given inside two client interfaces.

– The logger, file handler and error handler expose one server interface each of them. The basic colour classes implied in theses interfaces are *IDL* and *ML* for respectively log requests and log methods with their parameters; *IDF* and *MF* for File requests and file methods with their parameters; and *IDE* and *ME* for Error identifications and error methods with their parameters.

We note that there is no server interface with multiple clients in this application, so that identities of the client component are not modelled in the CC-SWNs.

4.2.3 Composite components

A composite component is made up of a set of interconnected sub-components being primitive or composite themselves. We assume that CC-SWNs of its sub-components are built. Building of the SWN model of the composite requires connecting sub-components' CC-SWNs and modelling external interfaces.

In the Julia context, primitive binding of interfaces are directly translated by transition fusion of the CC-SWNs of the corresponding sub-components: associated transition (for instance Invoke service-Receive request and Receive result-Send result in figure 5) are pairwise merged; Fusion of two transitions consists in defining a unique transition and keeping associated arcs of fused transitions. Colour classes of the two transition are mapped in one to one correspondence for common parameters of the interface (name of a method for instance) and specific colour classes of each transition are kept. Hence, the colour domain of the fused transition is the Cartesian product of colour classes of the fused transition, without repetition, together with the specific colour classes of each transition. This fusion definition is different from the proposed approach of [5] where several transitions can be fused but some arcs may be duplicated on several fused transitions. We have then a CC-SWN partially modelling the composite component It is completed by modelling external interfaces as specified in mapping rule 3. As mentioned above, we can also build an abstract model of a composite component from this CC-SWN. At the highest abstraction level, we get a very elementary CC-SWN shown in figure 13.

When assembling CC-SWN models of sub-components, name conflicts (of place, transition or colour class) may occur. They are eliminated by renaming. Such a renaming re-

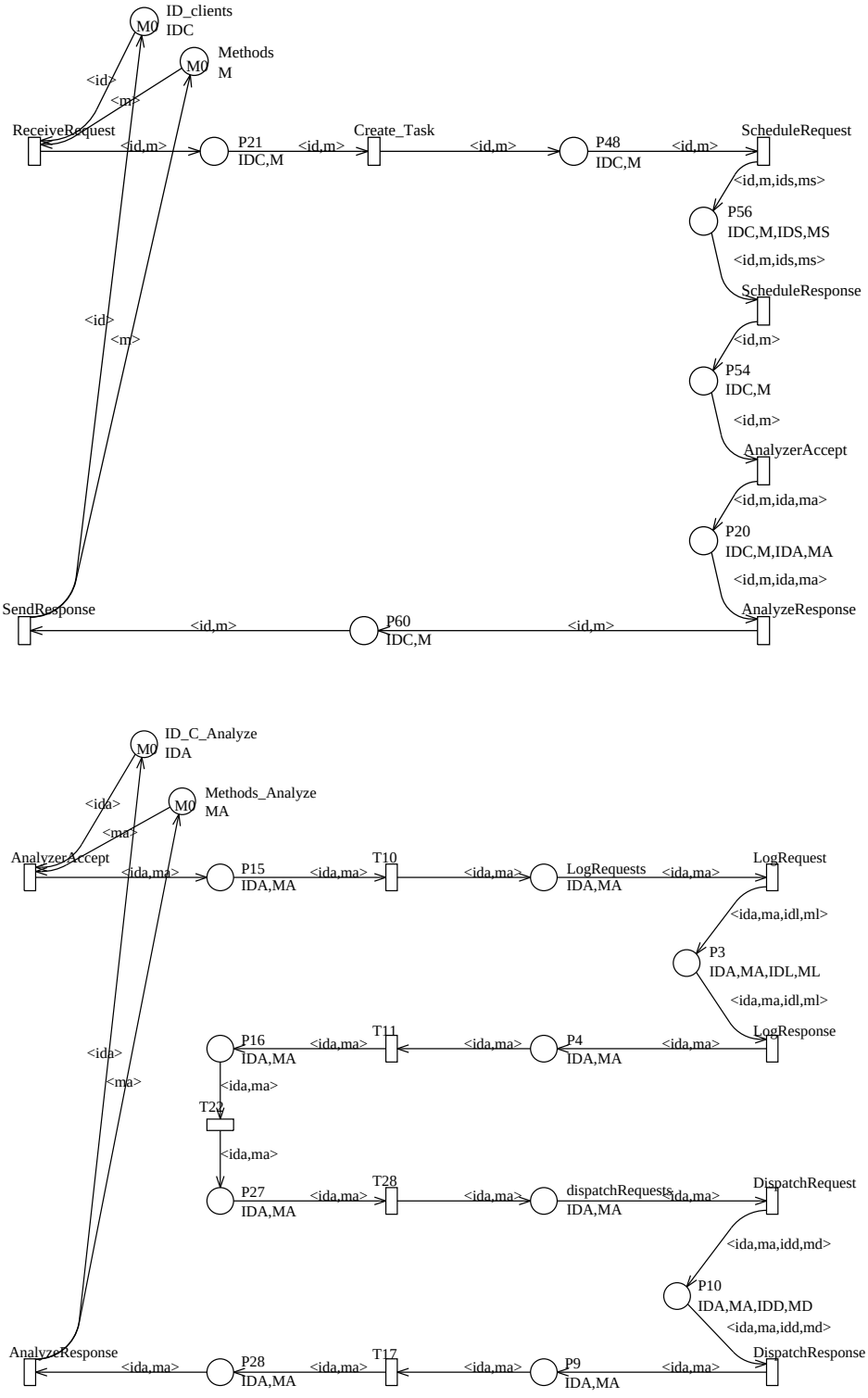


Fig. 9: CC-SWNs of the receiver (up) and analyzer (bottom) components

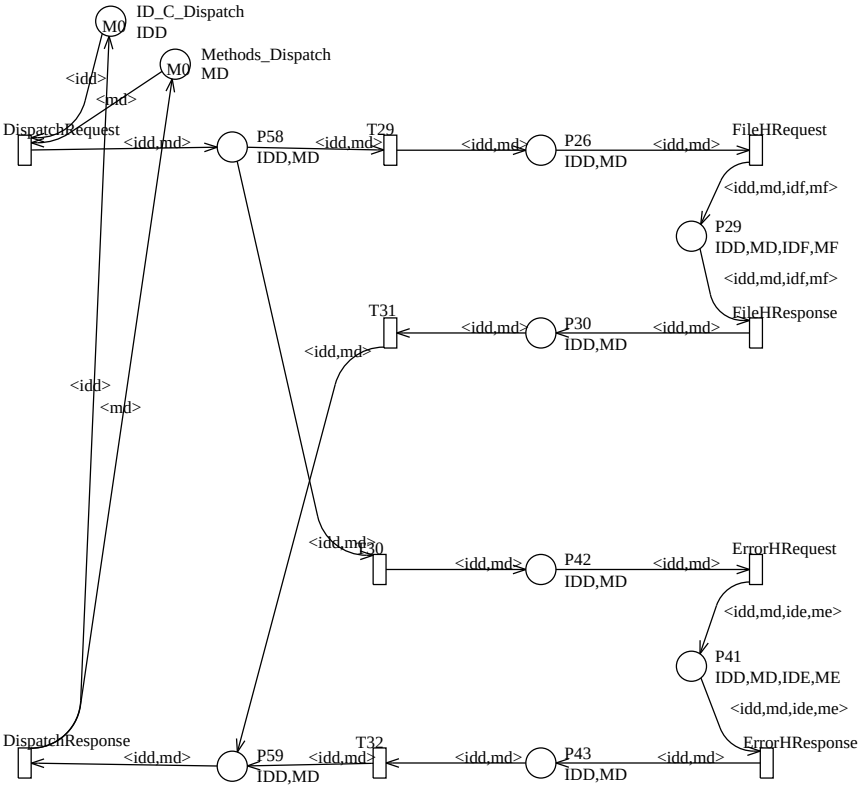


Fig. 10: CC-SWN of the dispatcher component

quires to translate analysis results (obtained properties and computed performance indices on the constructed global SWN) in the initial context of the CBS.

Note that, together with the CC-SWN of the composite, we keep track of the CC-SWNs of its sub-components; They will be used during the analysis phase.

4.2.4 Modelling the highest composite component

Starting from the first level of composite components, we successively build the CC-SWNs of the composite components up to the highest level. The resulting CC-SWN is then completed to provide the G-SWN of the application: Since we model applications with *finite* state space models, we need to "close" interfaces of the composite corresponding to the application as a whole. This is a classical method, allowing to limit the number of entities in the model. In the Petri net context, we add a Petri net to each external interface of the application with an adapted initial marking, generally an upper bound of the number of entities. An example of such a closing SWN is given in figure 14.

We summarize this modelling in the following algorithm allowing the building of the CC-SWN of a composite of N hierarchichal levels, and so the building of the G-SWN.

```
G-SWN (composite CC-SWN) building algorithm
BEGIN
Let N be the number of levels of the composite (CBS).
1. Model primitive components of level 0.
2. For (i=1; i<N; i++)
   a. For each composite C of a level i :
      (i)  Assemble CC-SWNs of components of level i-1:
           For each couple of sub-components related
           with a service invocation, merge corresponding
           transitions (TBRS,TBPS) and (TERS,TEPS).
      (ii) Each import/export binding defined in the
          Fractal ADL and related to the composite C
          corresponds to an external interface of C.
   b. Model primitive components of level i.
3. Close external interfaces of the highest level
component with a closing Petri net.
END
```

4.3 G-SWN of the Comanche application

Going back to our Comanche example, we can build the CC-SWN of the request handler composite component, then

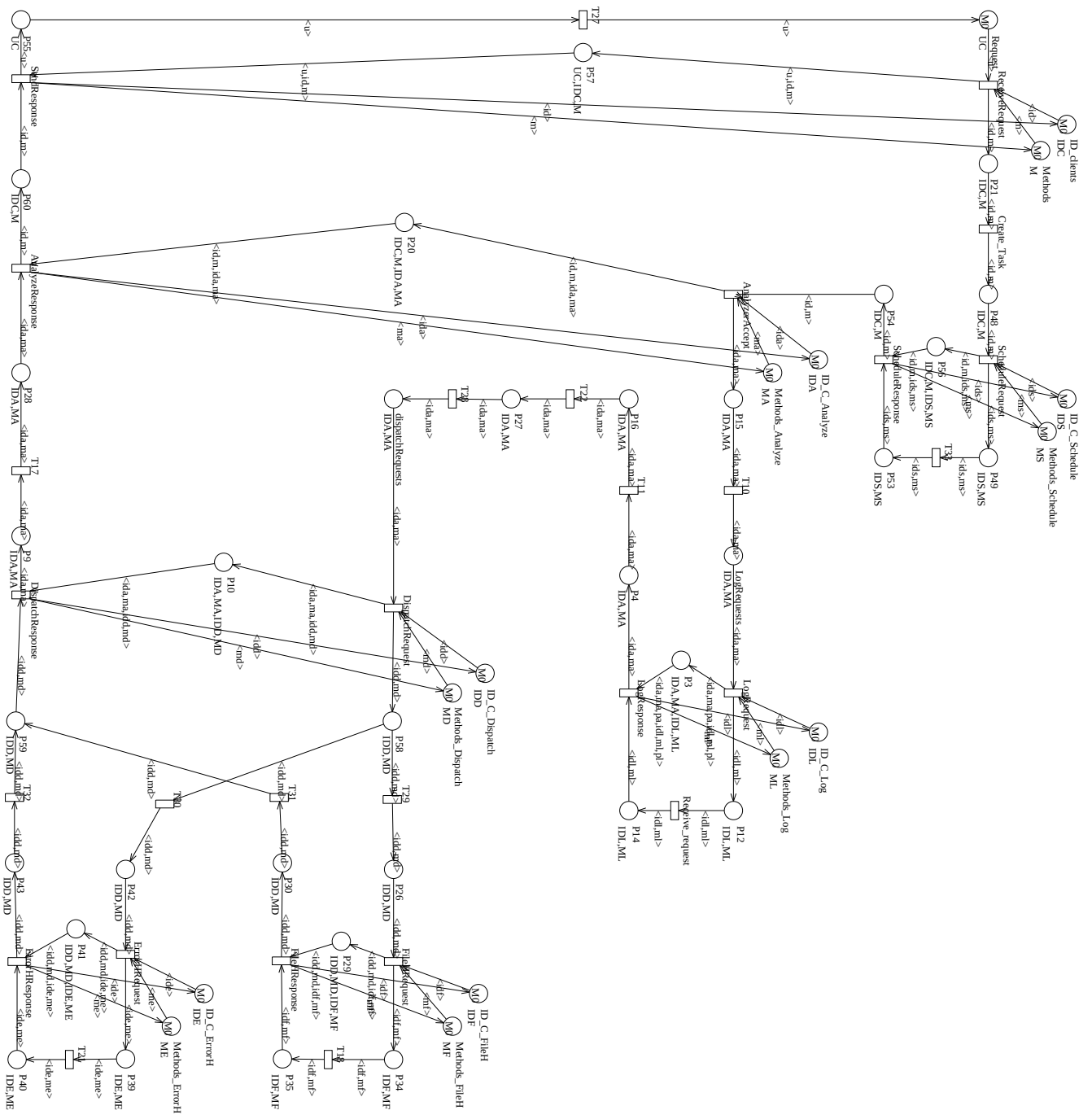


Fig. 15: The G-SWN of the Comanche application

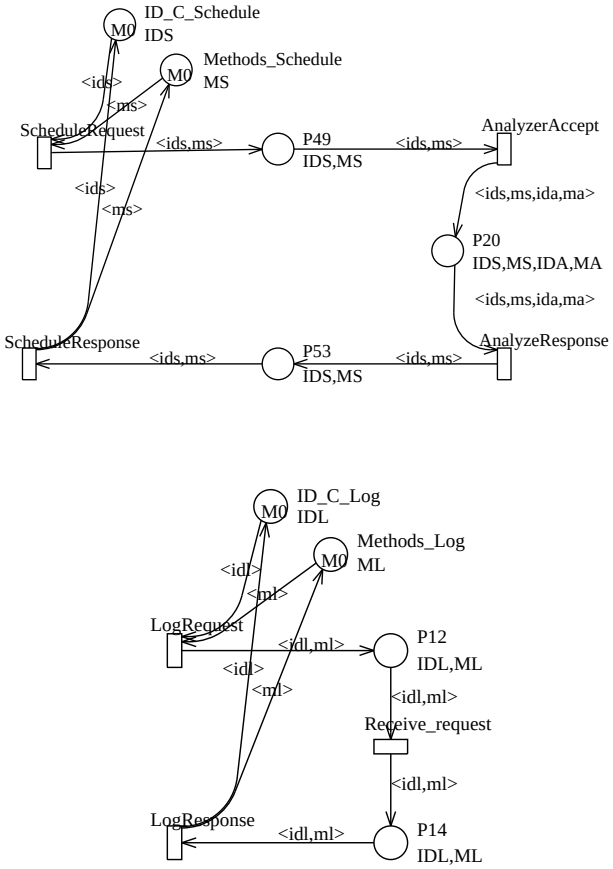


Fig. 11: CC-SWNs of the scheduler (up) and logger (bottom) components

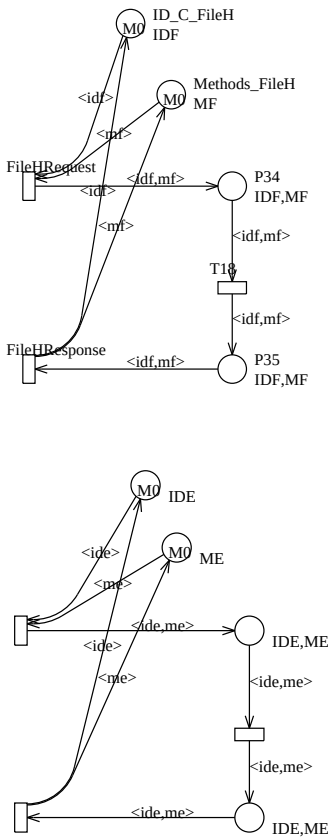


Fig. 12: CC-SWNs of the File handler (up) and Error handler (bottom) components

that of the Frontend and Backend composites, and finally we build the G-SWN given in figure 15.

5 Performance Analysis of Fractal CBSs

Analysis of a Fractal CBS can be performed through the analysis of the G-SWN obtained from the assembly of components. This approach has been followed in [5] for analysis of a different composition process of SWNs, and implemented in the Algebra tool of the GreatSPN package [24]. In our approach, we rather try to benefit from the compositionality features of the CBS, in order to provide an efficient steady-state performance analysis with regard to computation time and memory costs. For this purpose, we rely on previous work [12, 13] which defines an analysis method allowing to avoid the explicit construction of the aggregated Markov chain corresponding to the global SWN. This method uses a tensorial representation of the Symbolic Reachability Graph(SRG), hence enabling important memory and computation time savings, reducing analysis complexity. We adapt this approach to Fractal CBS.

5.1 Principle of structured analysis method

The structured analysis method was defined to apply on a global SWN N. The main idea in this method is to start from a global SWN, decompose it into several subnets, and study each subnet augmented with "parts" aggregating interactions with other subnets. These separated studies are then used to derive a tensorial representation of the generator of the underlying aggregated Markov chain of the global net, and so to compute performance indices. For this purpose, two kinds of decompositions into SWNs were defined:

- A "synchronous" decomposition (figure 16) modelling a complex synchronization of type "Rendez-vous" between two SWNs.
- An "asynchronous" decomposition (figure 17) which corresponds to an asynchronous method call or a message sending and receiving between two or more SWNs.

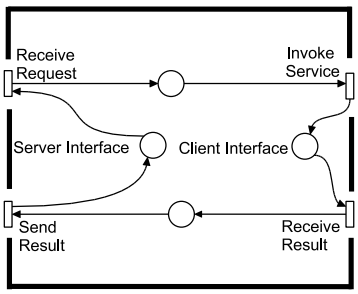


Fig. 13: A very abstract CC-SWN of a component

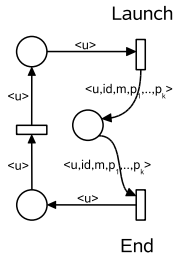


Fig. 14: Application closing subnet

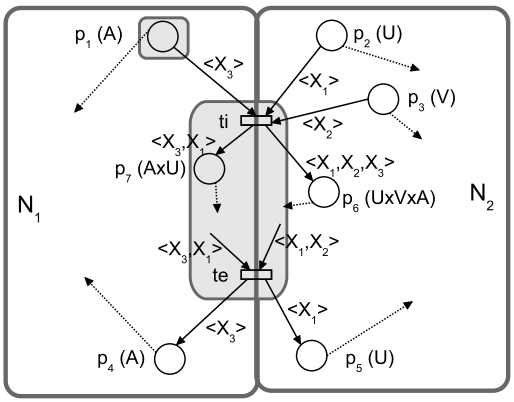


Fig. 16: Synchronous decomposition of SWNs

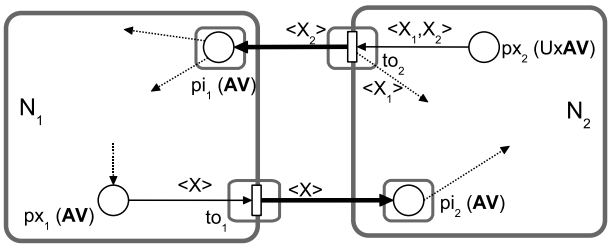


Fig. 17: Asynchronous decomposition of SWNs

Each kind of decomposition requires a set of conditions which can be checked at the SWN definition level. We refer the reader to [19] for a detailed presentation of these conditions.

We have extended and adapted this initial decomposition method to CBS. Adaptation was faced to three problems:

1. The first problem is to compose SWN models of components, as we start from the definition of components in the case of a CBS. This is in contrast to the previous method where a global SWN is decomposed into several subnets. Composition of SWNs in the Fractal framework has been explained above.
2. The second problem is to try bringing an interconnection of components into a synchronous or asynchronous composition of SWNs. This problem depends on the component model of the CBS. In the context of the Julia implementation of Fractal CBSs, we noted that service invocations of a component are in fact synchronous method calls. Thus, we model interactions between Fractal components with synchronous compositions of their corresponding SWN models as shown in the previous section.
3. The third problem consists of studying the impact of the simultanous presence of synchronous and asynchronous compositions in the same global model, as the structured method was defined for either a synchronous composition or else asynchronous composition of SWNs. In this paper, this problem does not appear, as we model Fractal components interactions with only synchronous compositions.

5.2 Structured CBS analysis algorithm

Our adapted method proceeds in several steps for analyzing a CBS. We assume that the G-SWN model of the application and the CC-SWNs of the components have been defined. These CC-SWNs make up the initial set of SWNs subnets denoted by $\mathcal{N}_k$ ($1 \leq k \leq K'$):

1. Checking applicability conditions on the G-SWN for a structured representation of the SRG and its aggregated generator.
   if they are fulfilled, goto 2
   else
   merge some of the CC-SWNs which do not satisfy applicability conditions
   to new SWNs trying to fulfill conditions.
   Each new SWN replaces the SWNs it stands for.
   Goto to 1.
2. Extension of the SWNs $\mathcal{N}_k$ to autonomous SWNs $\bar{\mathcal{N}_k}$, in order to take into consideration interaction with other subnets. These autonomous SWNs are called *extended nets* (see [19] for details).
3. Generation of the SRGs of these extended SWNs.

4. Computation of the synchronized product of these SRGs and of the tensorial representation of the generator of the underlying aggregated Markov chain.
5. Computation of the steady state distribution of the aggregated model and computation of the required performance indices.
6. Expression of the results in the initial context of the components.

We apply this method to SWNs models of Julia Fractal CBSs presented in section 4. For what concerns conditions for synchronous composition of SWNs (point 1 above), one important property must be verified: an entity of a subnet which could be synchronized (i.e. linked to an entity of another subnet during common actions) must be, at any time, either not synchronized, or else synchronized with only one entity of another subnet. This means, at the programmatic level, that such entities should not be duplicated, for instance with concurrent threads or processes, in a given component.

Structured analysis of a composition of SWNs is interesting as long as it allows for savings w.r.t. computations based on the wole G-SWN. The granularity of composed SWNs is indeed a very important parameter w.r.t. the computation time of the performance indices. Roughly and in a very imprecise way, a "large" number of "small" SWNs implies a bigger computation time than a model where "some" SWNs are merged. This is particularly due to the time taken to study each extended SWN, which exhibits more behaviours than the corresponding part of the whole system. We deduce that *component oriented design could be quite different from component oriented analysis*. Unfortunately, we think that there are no general rules on the structure of the model allowing estimation of the granularity level for best computation time. However we are working on definition of some heuristic guidelines to help the modeller in the merging process for what concerns this computation time.

5.3 Application to the Comanche example

The obtained G-SWN of the Comanche application satisfies conditions of structured analysis. So, we did not need to merge some CC-SWNs. We used our tool *compSWN* to apply our method and compute steady-state probabilities. We also used the GreatSPN environment on the G-SWN to compare results of both analysis methods. We ran the two tools on a Suse linux 9.2 workstation with 512 MO.

5.3.1 Savings with the structured method

Before giving some performance indices of the Comanche example, we first show time and memory savings due to the use of the structured analysis. To this end, we vary the cardinalities of our basic colour classes, and we study the behaviour of the solvers for several configurations (Cfi) summarized in table 1.

$\|Colour\|$	Cf1	Cf2	Cf3	Cf4	Cf5	Cf6	Cf7	Cf8
$\|UC\|$	3	5	10	2	3	5	10	20
$\|IDC\|$	1	1	1	2	2	2	2	2
$\|M\|$	2	2	2	2	2	3	3	3
$\|IDS\|$	3	5	10	2	3	5	10	20
$\|MS\|$	2	2	2	2	2	2	2	2
$\|IDA\|$	1	1	1	2	2	3	3	3
$\|MA\|$	2	2	2	2	2	2	2	2
$\|IDL\|$	2	2	2	2	2	5	5	5
$\|ML\|$	2	2	2	2	2	3	3	3
$\|IDD\|$	1	1	1	2	2	3	3	3
$\|MD\|$	2	2	2	2	2	3	3	3
$\|IDF\|$	2	2	2	2	3	5	5	5
$\|MF\|$	2	2	2	2	2	5	5	5
$\|IDE\|$	2	2	2	2	2	5	5	5
$\|ME\|$	2	2	2	2	2	4	4	4

Table 1: Various configurations for the Comanche example

We report in table 2 behaviours of the two solvers (GreatSPN and *compSWN*) for what concerns memory usage (in bytes) and computation times *for the SRG generation phase only* of the resolution, i.e. without computing steady-state probabilities nor performance indices (probabilities were found identical with the two tools, within numerical errors). We also indicate the state space sizes of the global net. Notations for tables 1 and 2 are the following: $|Colour|$ is the cardinality of the static colour subclass, NbS is the number of symbolic markings, NbO is the number of ordinary markings, TGreat is the computation time of GreatSPN, TComp is the computation time of *compSWN*, MGreat is the memory used by GreatSPN and MComp is the memory used by *compSWN*. Note that with our method and tool, we are able to compute the SRG of the G-SWN and its steady-state probabilities for all given configurations, while the computation was not possible with the GreatSPN tool for certain configurations (Cf6, Cf7, Cf8) because of the huge size of state space.

5.3.2 Performance results for the Comanche example

In this paper, we only present some of the results we have gathered and refer the reader to a forthcoming research report for a fully detailed performance analysis of the example.

Parameters of the system First, we present results for only one configuration (Cf4) of the system. Note that a colour may model a group of elementary entities, for instance a request colour can stand for 10, 100 or 1000 requests; obviously, firing rates of transitions involving this colour should be adapted to the semantics of a colour (100 requests provide a possibly 100 times (or more) slower method request

Config	NbS	NbO	TGreat(s)	TComp(s)	MGreat (B)	MComp (B)
Cf1	82	35144	4	0	402	1484
Cf2	136	239392	5	0	510	1652
Cf3	271	16139264	12	0	780	2072
Cf4	406	2392068	485	1	6305	5336
Cf5	784	24279944	4919	1	7095	6008
Cf6	1540	3656635680	-	2	-	7368
Cf7	3430	3113239552	-	3	-	10728
Cf8	7210	999926785	-	22	-	17448

Table 2: State space sizes and computation times for SRG generation of the Comanche example for various configurations

Component	Transition	Rate value
Receiver	ReceiveRequest	0.6
Scheduler	ScheduleRequest	0.9
Analyzer	AnalysisRequest	0.6
Analyzer	T10	0.75
Logger	LogRequest	0.9
Dispatcher	DispatchRequest	0.9
Dispatcher	T29	0.9
Dispatcher	T30	0.1
File Handler	FileHRequest	0.9
Error Handler	ErrorHRequest	0.1

Table 3: Transition rates of the studied configuration

rate for instance). We also take fixed rate values of a critical set of transitions, then, we vary some transition rates, and study the evolution of response time from obtained steady-state probabilities. Main transitions rate values are given in table 3 and transitions not appearing in this table have rate 1 (i.e. faster than all other transitions, rates being given in the same unit).

Response time variations We are mainly interested in computing variations of the response time with respect to several parameters: the load induced by client's requests, the rate of the analysis request, the rate of file or data retrieval and the error rate. Figure 18 shows response time variations with respect to the first two parameters. From the left diagram, we see that the CBS presents a slightly better response time as far as the client requests arrival duration increases. This a priori contradictory behaviour indicates that the system is not saturated until request arrival rate of 2. Indeed, the curve becomes flatter when the request arrival rate increases. The right diagram shows a reduced response time with the increasing of request analysis rate. This was expected since the system becomes more powerful with higher request analysis rate. However, we observe that the response time first shuts down significantly and then (0.4 and more) becomes almost stable. This phenomenon proves that the analysis is no more the bottleneck of the system for rates higher than 0.4.

Response time analysis could be completed by steady-state probabilities of markings in several components of the system.

6 Conclusion

In this paper, we have presented a method allowing to study, in an efficient way, performance indices of Fractal Component Based Systems (CBS), restricted to stable configurations (i.e. without reconfiguration) of the CBSs. We quote that implementation semantics heavily impact modelling Fractal CBSs and must be taken into account by any modelling method. In this work, we model Fractal CBS using Julia, the Java reference implementation of the Fractal component model.

The presented method is an instance of a general method for analysis of CBSs. It starts from the description of a Fractal application, given by its description expressed in the Fractal Architecture Description Language and the Java source code of the primitive components. For each component, we build a SWN model (the CC-SWN) describing its functional behaviour and its interfaces. The CC-SWNs are then interconnected following the architecture of the system, beginning from primitive components up to the highest level composite component for which we provide a global SWN (the G-SWN). We then apply, whenever possible, to the G-SWN and the CC-SWNs, a structured analysis method to compute performance indices of the system. This analysis is based on synchronous composition of SWNs we have previously developed. We show that our model of Julia Fractal CBS can indeed be analyzed with this approach which provides important savings in computation time and memory usage during the analysis. We illustrate our method with the HTTP server Comanche application. Results are computed with the help of our *compSWN* tool. Detailed performance analysis of this example will be available in a forthcoming research report.

Work in progress will first partially automate extraction of information from the description of the CBS for direct definition of the interfaces of the CC-SWNs. Moreover, gains of the structured analysis being dependent on the number

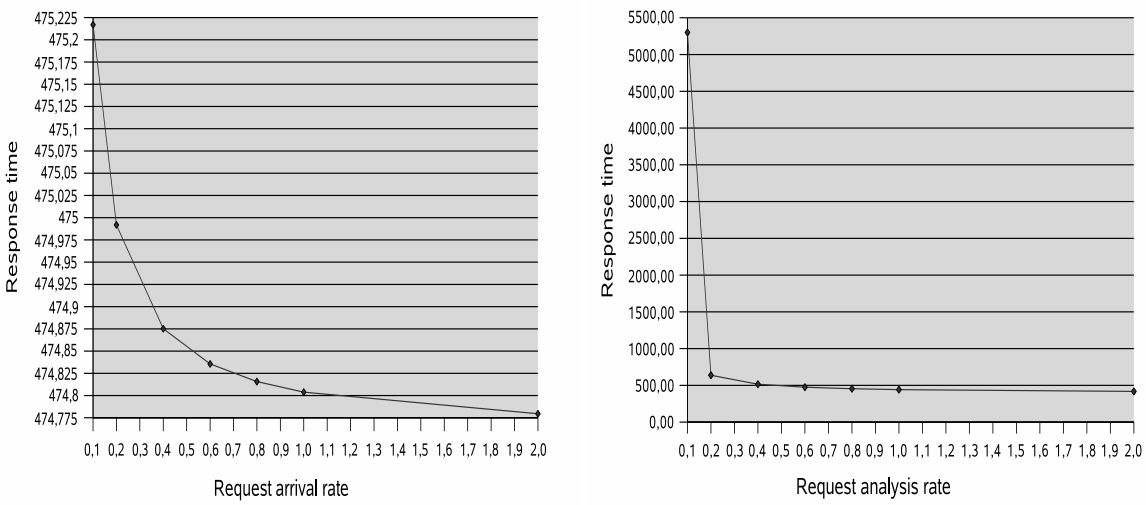


Fig. 18: Response time versus request arrival rate (left) and request analysis rate (right)

and the size of SWNs making up the global SWN, we are studying some heuristic rules to merge subnets of the model for better analysis times. Finally, we are working on modelling reconfiguration features of Fractal CBSs and verification of their behaviours.

References

1. A. Arnold. Nivats processes and their synchronization. *Theor. Comput. Sci.*, (281(1-2)):31–36, 2002.
2. A. Arnold. Finite transition systems: semantics of communicating systems. In UK Hertfordshire, editor, *Pren-tice Hall International (UK) Ltd.*, 94.
3. T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model checking distributed components : The vercors platform. In *3rd workshop on Formal Aspects of Component Systems*, Prague, Tcheque Republic, September 2006. ENTCS.
4. F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In D.C. Schmidt, R. Meersmanand Z. Tari, and al., editors, *On The Move to Meaningful Internet Systems 2003: Coopis, DOA and ODBASE*, volume 2888 of *LNCS*, pages 1226–1242. Springer Verlag, 2003.
5. S Bernardi, S. Donatelli, and A. Horváth. Implementing compositionality for stochastic Petri nets. *Int. J. STTT*, (3):417–430, 2001.
6. E. Bruneton. Fractal ADL tutorial. `file://home/nasal/doc/Fractal/fractal_adl_tutorial_index_print.html` (jan. 2007).
7. E. Bruneton. Tutorial: Developping with fractal. `http://fractal.objectweb.org/tutorial/index.html` (dec. 2006).
8. E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal component model, version 2.0-3. Technical report, Fractal team, Online documentation `http://fractal.objectweb.org/specification/` (oct. 2006), February 2004.
9. É. Bruneton, Th. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software Practice Experience*, 36(11-12):1257–1284, 2006.
10. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, November 1993.
11. L. Dias da Silva and A. Perkusich. Composition of software artifacts modelled using colored Petri nets. *Science of Computer Programming*, 56(1-2):171–189, April 2005.
12. C. Delamare, Y. Gardan, and P. Moreaux. Efficient implementation for performance evaluation of synchronous decomposition of high level stochastic Petri nets. In *On-site proceedings of the ICALP2003 Workshop on Stochastic Petri Nets and Related Formalisms*, pages 164–183, Eindhoven, Holland, June 21-22 2003. University of Dortmund, Germany.
13. C. Delamare, Y. Gardan, and P. Moreaux. Performance evaluation with asynchronously decomposable SWN: implementation and case study. In *Proc. of the 10th Int. Workshop on Petri nets and performance models (PNPM03)*, pages 20–29, Urbana-Champaign, IL, USA, September 2–5 2003. IEEE Comp. Soc. Press.
14. D.Petriu, C.Shousha, and A.Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transactions on Software Engineering*, 26(11):1049–1065, 2000.
15. J. Fassino, J. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *In Usenix Annual Technical Conference*.
16. V. Grassi, R. Mirandola, and A. Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *J. Syst. Softw.*, 80(4):528–558, 2007.
17. Object Management Group. Common object request broker architecture (CORBA) - specification, version 3.1, part 1: CORBA interoperability. `http://www.omg.org/cgi-bin/doc?pas/04-08-01.pdf` (July 2007), 2004.
18. Object Management Group. Common object request broker architecture (CORBA) - specification, version 3.1, part 2: CORBA interfaces. `http://www.omg.org/cgi-bin/doc?pas/04-08-02.pdf` (July 2007), 2004.
19. S. Haddad and P. Moreaux. Aggregation and decomposition for performance evaluation of synchronous product of high level Petri nets. Document du Lamsade 96, LAMSADE, Université Paris Dauphine, Paris, France, September 1996. .
20. Orna Kupferman and Moshe Y. Vardi. Modular model checking. In *Compositionality: The Significant Difference*, volume 1536 of *LNCS*, pages 381–401, 1998.

21. N. Medvidović and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *In IEEE Trans. On Software Engeneering*, volume 26, pages 70–93. IEEE Trans., 2000.
22. Microsoft. .Net 3.0 framework. `http://msdn.microsoft.com/netframework` (July 2007), 2007.
23. Sun Microsystems. EJB 3.0 specification, July 2007.
24. P.E. Group. GreatSPN home page: `http://www.di.unito.it/~greatspn`, 2002.
25. A.-E. Rugina, K. Kanoun, and M. Kaâniche. Aadl-based dependability modelling. Report 06209, LAAS, Toulouse, France, April 2006.
26. L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In *Proc. of Component-Based Software Engineering (CBSE'06*, volume 4063 of *LNCS*, pages 139–153, Mälardalen University, Västerås, Sweden, June 2006. Springer.
27. C.U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, Reading, Mass., 1990.
28. C. Szyperski, D. Gruntz, and S. Murer. *Component Software Beyond Object-Oriented Programming (2nd ed.)*. Addison Wesley - ACM Press, 2002.
29. X. Wu and M. Woodside. Performance modeling from software components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, 2004.

A WN and SWN formal definitions

We remind the reader with the definitions of WN and SWN. A detailed presentation of these models can be found in [10].

Definition 1 (Well-formed Petri Net (WN)) A well-formed Petri Net S is a tuple
$(P,T,C,cd,Pre,Post,Inh,Guard,Pri,M_0)$ with:

- P,T: the finite sets of places and transitions,
- $C = \{C_i / i \in I = \{1,\cdots,n\}\}$: the set of basic colour classes; C_i is possibly partitioned into into n_i static sub-classes: $C_i = \bigcup_{j=1}^{n_i} C_{i,j}$,
- cd: $P \bigcup T \rightarrow Bag(I)$. $cd(r) = C_1^{e_1} \times C_2^{e_2} \times \ldots \times C_n^{e_n}$ is the colour domain of a node r; $e_i \in \mathbb{N}$ is the number of occurrences of C_i in the colour domain of r, where Bag(I) is the set of multisets (bags) on I.
- $Pre, Post, Inh$: the input, output and inhibition standard colour functions from $C(t)$ to $Bag(C(p))$.
- $Guard(t) : C(t) \rightarrow$\{true, false\} is a standard predicate associated with the transition t. By default, $Guard(t)$ is the constant function of value True.
- $Pri : T \rightarrow \mathbb{N}$ the priority function. By default, we assume $\forall t \in T, Pri(t) = 0$;
- $M_0 : M_0(p) \in Bag(C(p))$ is the initial marking of p.

Definition 2 (Stochastic Well-formed Net (SWN)) A Stochastic Well-formed Net is a pair (S,θ) such that:

- S is a Well-Formed Net.
- θ a function defined on T such that: $\theta(t) : \tilde{cd}(t) \times \prod_{p \in P} Bag(\tilde{C}(p)) \longrightarrow R^+$.

$\theta(t)(\tilde{c},\tilde{M})$ represents:

- The weight of t for the colour c in the marking M, if $\pi(t) > 0$ (t is immediate). the firing probability of $t(c)$ in M is then: $\frac{\theta(t)(\tilde{c},\tilde{M})}{\sum_{(t',c'),M[t'(c')>} \theta(t')(\tilde{c'},\tilde{M})}$.
- The firing rate of t for the colour c in M, if $\pi(t) = 0$ (t is timed): the enabling duration before the firing of $t(c,M)$ follows an exponential probability distribution with mean $\theta(t)(\tilde{c},\tilde{M})$.

In this definition, $\tilde{c}$ is the representation of the colour c in terms of static sub-classes, and $\tilde{M}(p)$ is the representation of the symbolic marking of p in terms of tuples of static sub-classes. $\theta(t)$ depends only on static sub-classes of concerned colours.