Dependability and Performance Assessment of Dynamic CONNECTed Systems

Antonia Bertolino, Antonello Calabro, Felicita Di Giandomenico, Nicola Nostro

Dependability and Performance Assessment of Dynamic CONNECTed Systems

Antonia Bertolino, Antonello Calabró,
Felicita Di Giandomenico, and Nicola Nostro

Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche
via Moruzzi 1, I-56124, Italy
{*name.surname*}@isti.cnr.it

Abstract. In this chapter we present approaches for analysis and monitoring of dependability and performance of CONNECTed systems, and their combined usage. These approaches need to account for dynamicity and evolvability of CONNECTed systems. In particular, the chapter covers the quantitative assessment of dependability and performance properties through a stochastic model-based approach: first an overview of dependability-related measurements and stochastic model-based approaches provides the necessary background. Then, our proposal in CONNECT of an automated and modular dependability analysis framework for dynamically CONNECTed systems is described. This framework can be used off-line for system design (specifically, in CONNECT, for CONNECTor synthesis), and on-line, to continuously assess system behaviour and detect possible issues arising at run-time. For the latter purpose, a generic, flexible and modular monitoring infrastructure has been developed. Monitoring is at the core of the CONNECT vision, in order to ensure run-time observation of specified quantitative properties and possibly trigger adequate reactions. We focus here on the interaction chain between monitoring and analysis, to allow for on-line continuous validation of specified dependability and performance properties. Illustrative examples of applications of analysis and monitoring are provided with reference to the CONNECT Terrorist Alert scenario.

1 Introduction

Modern software applications are more and more conceived as dynamically adaptable and evolvable sets of components that must be able to modify their behaviour at run-time to tackle the continuous changes in the unpredictable open-world settings [BGD06]. On the other hand, these systems are increasingly pervasive and their improper behaviour will produce effects on our everyday lives and business, which can range from annoying ones up to sometime even critical consequences. Therefore, we need to ensure that these dynamic systems provide the required non-functional properties, such as reliability, availability, performance, security and trust, and so on, and continue to do so even after evolution and adaptation.

In such partially unknown and evolving contexts, dependability analysis [Lap95] calls for on-line support to enhance the accuracy of preliminary estimates performed at design time. Indeed, as we entrust more and more responsibilities to distributed software systems, the need arises to augment them with powerful oversight and management functions in order to allow continuous and flexible monitoring of their behaviour.

In this chapter we tackle the challenge of dependability and performance analysis in dynamic CONNECTed systems. We present our preliminary results obtained in the context of the European Project CONNECT [CON13], which considers dynamic environments populated by heterogeneous Networked Systems within disconnected isles, willing to communicate with each other despite differing and evolving technologies.

In CONNECT, communication between heterogeneous systems is seamlessly supported by CONNECT Enablers, which make on-the-fly interoperation possible by synthesising and deploying mediating software bridges, called CONNECTors. Specifically, the main Enablers in the CONNECT architecture include (please refer to [GG11] for a complete description): *Discovery*, which discovers mutually interested Networked Systems (NSs), and retrieves information on the NS interfaces; *Learning*, which possibly completes the specifications of the NSs through a learning procedure; *Synthesis* and *Deployment*, which perform, respectively, the dynamic synthesis of the mediating CONNECTors and their deployment; *Dependability&Performance*, which uses a model-based analysis to support *Synthesis* in the definition of a dependable CONNECTor; *Security&Trust*, which assess and enforce security, privacy and trust aspects, and finally *Monitoring*, which continuously monitors the deployed CONNECTor to update the CONNECTor specification used by the other Enablers with run-time data.

To accomplish dependability and performance analysis in such a complex and evolving context, both off-line and on-line approaches are pursued, to cover a wider range of needs. As commonly intended in the literature, off-line analysis refers to activities devoted to analyse the system at hand before deployment, or even after its deployment but in isolation with respect to the system in operation. On the contrary, on-line analysis refers to activities performed while the system is in operation, so accounting for the detailed system and environment aspects during that specific system execution. We adopt the off-line and on-line terms with such meanings.

Analysis at the early stage of a development process is of paramount importance to achieve the required functional and non-functional properties. In fact, early evaluation of the concepts and architectural choices prevents wasting time and resources by promptly identifying possible design deficiencies or helping in performing design decisions by comparing different alternative architectural solutions and selecting the most suitable one.

Nevertheless, the incomplete a priori knowledge about the operating system and environment unavoidably undermines the accuracy of the considered elements and, hence, of the analysis results. In this perspective, monitoring becomes a key technological enabler for dependability assurance, as it provides the enabling infrastructure to prolong software lifecycle after deployment, by supporting run-time verification and on-line adaptation.

Therefore, in Connect, both an automated approach to off-line dependability analysis adopting model-based analysis, to support the design of dependable connectors, and event-based monitoring, to support dependability and performance run-time analysis, are under definition and development. In this chapter, the two approaches are first individually presented, pointing out their respective architectures and their role in the Connect framework. Then, we focus on their synergic use, to allow refining model-based dependability and performance analysis in distributed dynamic systems through monitoring.

The continuous interplay and refinement between model-based analysis and run-time monitoring is today emerging as an irremissible direction of software development. A software module (the whole software system or part of it) is repeatedly analysed through model-based analysis and refined in its design until it proves to satisfy specified non-functional quantitative requirements. Once such a proper design is obtained, it is deployed in a suitable computing environment and put in operation. At run-time, the deployed software system must be monitored to be sure that its execution respects the required properties. Data collected through monitoring constitute invaluable information to be exploited for: i) validating the models generated through model-based software engineering sub-process, and ii) continuously refining the analysis by overcoming the possible inaccuracy in the values of the model parameters due to incomplete knowledge or to evolution of the elements involved in the analysis.

The rest of the chapter is structured as follows. Section 2 presents some background material about dependability and performance properties and related analysis approaches. The approaches undertaken in Connect to perform dependability and performance assessment of Connected systems are dealt with in Section 3, namely model-based analysis and event-based monitoring. In addition to presenting them individually, emphasis is put on their synergic use and the current steps towards their integration are illustrated. A case-study accounting for representative aspects of the Connect context is then elaborated in Section 4, which allows to provide preliminary illustrative examples of the analysis performed both off-line and on-line. Related work is briefly overviewed in Section 5, while conclusions and future perspectives are drawn in Section 6.

2 Background

In this section we provide some background material about dependability, performance and monitoring.

2.1 Dependability, performance and related assessment metrics

Dependability has been defined in the 90*'s* as the ability of a system to provide its intended services in a justifiable way [Lap95,ALRL04]. Such ability of the system is generally measured against the following attributes (see Figure 1):

availability, *reliability*, *safety*, *integrity*, *maintainability*. *Availability* is defined as the readiness for correct service and is generally computed as the ratio between the up-time of the system to the duration of the considered time period. *Reliability* is defined as the continuity of correct service and is typically expressed by using the notions of mean time between failures (MTBF) and mean time to recover (MTTR) for repairable systems, and with mean time to failure (MTTF) for non-repairable systems. *Safety* is the absence of catastrophic consequences. This attribute is a special case of reliability: a safe state, in this case, can be either a state in which a proper service is provided, or a state where an improper service is provided due to non-catastrophic failures. *Integrity* is defined as the absence of improper system state alterations. *Maintainability* is the ability to undergo modifications and repairs.

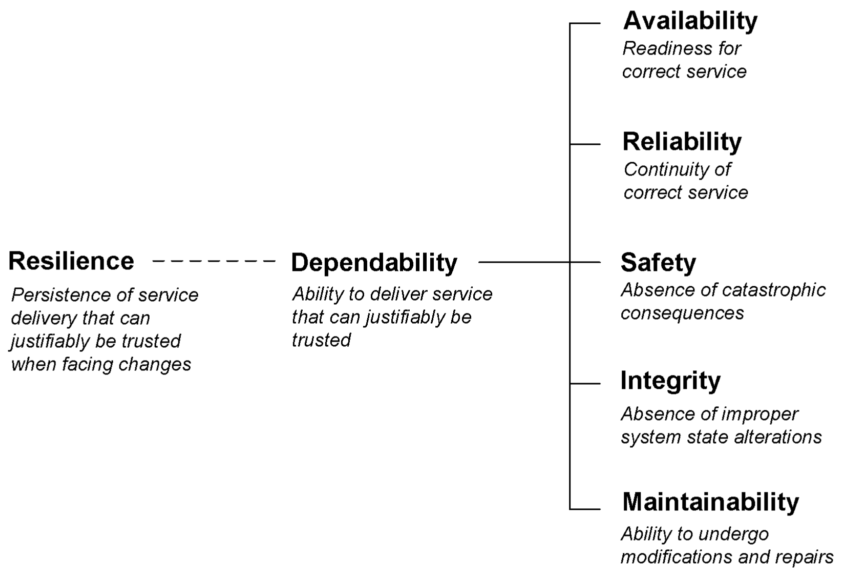

Resilience
Persistence of service delivery that can justifiably be trusted when facing changes
Dependability
Ability to deliver service that can justifiably be trusted
Availability
Readiness for correct service
Reliability
Continuity of correct service
Safety
Absence of catastrophic consequences
Integrity
Absence of improper system state alterations
Maintainability
Ability to undergo modifications and repairs

Fig. 1. Classical dependability attributes and resilience

Dynamic and evolvable systems generally need to cope with unanticipated conditions that might cause system failures. In these cases, the concept of dependability can be naturally extended to *Resilience*, i.e., the persistence of service delivery that can justifiably be trusted when facing changes [Lap08]. Possible changes can be classified according to their *nature* (e.g., functional, environmental, technological), *prospect* (e.g., foreseen, foreseeable, unforeseen), and *timing* (e.g., short, medium or long term).

Performance is the ability of a system to accomplish its intended services within given non-functional constraints (e.g., time, memory) [iee90]. Typically, performance of a system can be characterised with the following attributes (see

Figure 2): *timeliness*, *precision*, *accuracy*, *capacity* and *throughput*. *Timeliness* is the ability of the system to provide a service according to given time requirements, e.g., at a given time and within a certain time frame. *Precision* is the ability of the system to provide the same results when repeating measurements under unchanged conditions. *Accuracy* is the ability of the system to provide exact results, i.e., results that match the actual value of the quantity being measured. *Capacity* is the ability of the system to hold a certain amount of data or handle a certain amount of operations. *Throughput* is the ability to handle a certain amount of operations or data in a given time period.

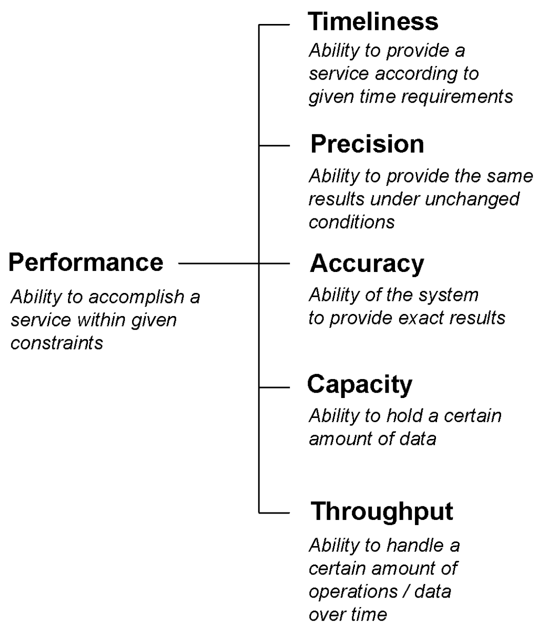

Performance
Ability to accomplish a service within given constraints
Timeliness
Ability to provide a service according to given time requirements
Precision
Ability to provide the same results under unchanged conditions
Accuracy
Ability of the system to provide exact results
Capacity
Ability to hold a certain amount of data
Throughput
Ability to handle a certain amount of operations / data over time

Fig. 2. Performance attributes

Quantification of dependability and performance attributes is of paramount importance in the process of determining whether a system meets its specification and to compare possible alternative design solutions leading to the most effective system realization. This is accomplished through the definition of appropriate metrics for the dependability and performance attributes. In general, a number of metrics can be defined for a given attribute; as an example, the following metrics allow to quantify *Availability*, that is the alternation between deliveries of proper and improper service:

- A(t) is 1 if service is proper at time t, 0 otherwise;
- E[A(t)] (Expected value of A(t)) is the probability that service is proper at time t;

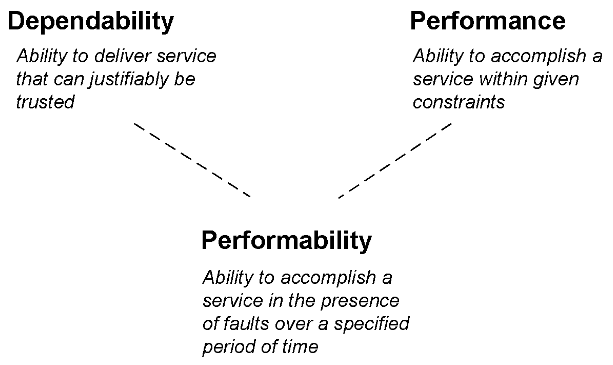

Dependability
Ability to deliver service that can justifiably be trusted
Performance
Ability to accomplish a service within given constraints
Performability
Ability to accomplish a service in the presence of faults over a specified period of time

Fig. 3. Performability and its relation with Dependability and Performance

- A(0,t) is the fraction of time the system delivers proper service during [0,t];
- E[A(0,t)] is the expected fraction of time the system delivers proper service during [0,t].

Similarly, *Performance* metrics typically include:

- the number of jobs processed per time unit, as a measure of throughput;
- the time to process a specific job, as a measure of the response time;
- the maximum number of jobs that may be processed per time unit, as a measure of the capacity.

Most practical *Performance* measures are very application specific, and measure times to perform particular functions or, more generally, the probability distribution function of the time to perform a function.

A measure of special interest introduced to evaluate degradable systems, i.e., systems that are still able to provide a proper service when facing faults, but with degraded level of performance, is *Performability.* This indicator combines the concepts of performance and dependability and represents the ability of a system to accomplish its intended services in the presence of faults over a specified period of time [Mey92]. Performability allows to evaluate different application requirements and to assess dependability-related attributes in terms of risk versus benefit.

2.2 Stochastic model-based approaches for early prediction of dependability and performance metrics

Fault forecasting and evaluation approaches are very suited to detect errors and deficiencies at design time, that could otherwise be very costly or even catastrophic when discovered at later stages.

Modelling is composed of two phases:

- The construction of a model of the system from the elementary stochastic processes that model the behaviour of the components of the system and

their interactions. These elementary stochastic processes relate to failures, to repair, service restoration and possibly to system duty cycle or phases of activity;
– Processing the model to obtain the expressions and the values of the dependability measures of the system.

Research in dependability analysis has developed a variety of models, each one focusing on particular levels of abstraction and/or system characteristics. As reported in [NST04], important classes of model representation include: *i)* Combinatorial Methods (such as Reliability Block Diagrams); *ii)* Model Checking; and *iii)* State-Based Stochastic Methods. In the CONNECT project, approaches at both points ii) and iii) are employed; in this chapter the emphasis is on State-Based Stochastic Methods, which support the explicit modelling of complex relationships (e.g., concerning failure and repair processes), and their transition structure encodes important sequencing information; for discussing Stochastic Model-Checking and for a compared evaluation of their usefulness in the CONNECT environment instead we refer to [CON10]. Concerning Combinatorial Methods, they have not been considered in CONNECT, since they do not easily capture certain features relevant for the project's context such as stochastic dependence and imperfect fault coverage,

State-Based Stochastic Methods use state-space mathematical models expressed with probabilistic assumptions about time durations and transition behaviours. State-based stochastic models can be classified in Markovian and non-Markovian according to the underlying stochastic process [CGL94,Hav01,Tri02]. A wide range of dependability modelling problems fall in the domain of Markovian models, for example when only exponentially distributed times occur. Markov chains (DTMC and CTMC) [How71,MFT00,Hav01,Tri02], Stochastic Petri nets (SPN) [Mol82,CBC$^+$93,Bal01] and Generalized Stochastic Petri nets (GSPN) [ABC84] are among the major Markovian models. However, there is also a great number of real circumstances in which the Markov property is not valid, for example when deterministic times occur; non-Markovian models are used for this type of problems. In past years, several classes of non-Markovian approaches have been defined [BT98], such as Semi-Markov Stochastic Petri Net (SMSPN's) [CGL94], Markov Regenerative Stochastic Petri Nets (MRSPN's) [CKT94] and Deterministic and Stochastic Petri Nets (DSPN's) [AC87]. Some major methods for analytically solving the non-Markovian models are discussed in [BPTT98,MFT00,Ger01]. A short survey on State-Based Stochastic Methods and automated supporting tools for the assisted construction and solution of dependability models can be found in [BCG05].

The proposal of an automated dependability analysis framework for dynamically CONNECTed systems will be discussed in 3.1. Two implementation of the analysis engine are being pursued: one based on the Stochastic Activity Networks (SAN) formalism and related Möbius tool, and the other on the PRISM tool. In this chapter, we focus on the Möbius implementation only; details on the PRISM-based implementation can be found in [CON11b]. The Stochastic Activity Networks (SAN) formalism is one of the most powerful (in term of modelling capabilities) stochastic extensions to Petri nets and is supported by the Möbius tool. SAN formalism and the Möbius tool are very commonly used in dependability analysis and therefore they have been initially chosen for dependability analysis in CONNECT. In the following, we provide some background on SAN and Möbius to get the reader more familiar with (part of the) dependability models we are going to define in this chapter.

Stochastic Activity Networks (SAN) SAN are stochastic extensions of Petri Nets introduced in [MM84] and formally defined in [SM02]. They have a graphical representation and consist of four primitive objects: *places*, *activities*, *input gates* and *output gates*. Places in SANs have the same interpretation as in Petri Nets, i.e., they hold tokens. The number of tokens in a place is referred to as the marking of that place, and the marking of the SAN is the set of all place markings.

There are two types of activities: instantaneous and timed. Timed activities represent actions that have a duration that impacts the performance of the modelled system, e.g., message transmission time, recovery time, time to fail. The duration of each timed activity is expressed via a time distribution function. Both instantaneous and timed activities may have *case probabilities*. Each case probability stands for a possible outcome of the activity, and can be used to model probabilistic aspects of the system, e.g., probability for a component to fail.

Gates connect activities and places. Input gates (indicated as red/grey triangles) are connected to one or more places and one single activity. They have a predicate, a boolean function of the markings of the connected places, and an output function. When the predicate is true, the gate holds. Output gates (indicated as black triangles) are connected to one or more places, and to the output side of an activity. If the activity has more than one case, output gates are connected to a single case. Output gates have only an output function. Gate functions (both for input and output gates) provide flexibility in defining how the markings of connected places change when the delay represented by an activity expires.

Properties of interest. Properties of interest are specified with *reward functions*. Each reward function is a C++ function that specifies how to measure a property on the basis of the marking of the SAN. There are two kinds of reward functions: *rate reward* and *impulse reward*. Rate rewards can be evaluated at any time instant. Impulse rewards are associated with specific activities and they can be evaluated only when the associated activity completes. Measurements can be conducted at specific time instants, over periods of time, or when the system reaches the steady state.

Möbius Möbius [CCD$^+$01] provides an infrastructure to support multiple interacting modelling formalisms and solvers. The main features of the tool include:

- *Multiple modelling languages*, based on either graphical or textual representations. Supported model types include stochastic extensions to Petri nets (e.g. SAN), Markov chains and extensions, and stochastic process algebras.
- *Hierarchical modelling paradigm.* Models are built from the ground up. First the behaviour of individual components is specified, and then a model of the complete system is created by combining these components.
- *Customized measures of system properties*, with ability to construct detailed expressions that measure the exact information desired about the system (e.g., reliability, availability, performance, and security). Measurements can be conducted at specific time points, over periods of time, or when the system reaches steady state.
- *Study the behaviour of the system under a variety of operating conditions.* The functionality of the system can be defined as model input parameters, and then the behaviour of the system can be automatically studied across wide ranges of input parameter values.
- *Distributed discrete-event simulation.* The tool evaluates the custom measures using efficient simulation algorithms to repeatedly execute the system.
- *Numerical solution techniques.* Exact solutions can be calculated for many classes of models, and advances in state-space computation and generation techniques make it possible to solve models with tens of millions of states.

Möbius allows to combine (atomic) models to form the *Composed model.* To this purpose, it supports the two operators *Rep* and *Join* to compose subnetworks. Join is used to compose two or more SANs. Rep is a special case of Join, and is used to construct a model consisting of a number of replicas of a SAN. Models in a composed system interact via *Place Sharing.* Place Sharing is a composition formalism based on the notion of sharing places via an equivalence relation. It supports the transient and steady-state analysis of Markovian models, the steady-state analysis of non-Markovian DSPN-like models [Sha93], and transient and steady-state simulation. More information can be found in the web site: http://www.crhc.uiuc.edu/PERFORM.

2.3 Run-time Analysis via Monitoring

The ultimate goal of CONNECT, i.e., achieving automated and eternal interoperability among heterogeneous and evolvable Networked System, strongly relies on the adoption of on-line approaches, and therefore on a pervasive monitoring infrastructure.

More in general, as systems become more and more dynamic, distributed and evolvable, the capability of effectively gathering run-time information about their execution and/or their surrounding environment becomes an indispensable tool for many functionalities. Schroeder [Sch95], for example, identified the following seven monitor functionalities: Control, Correctness checking, Debugging&Testing, Dependability, Performance evaluation, Performance enhancement, and Security. In modern applications certainly further functionalities can be identified, such as Learning, Accounting, Trust management, and so on.

Although in this chapter we focus on the use of monitoring for dependability and performance evaluation, the monitoring infrastructure that we built in Connect (described in the next section) has been conceived to be general and flexible, and not restricted to this purpose. Here below, as a background we provide a basic introductory overview of monitoring concepts.

For the purpose of monitoring, the actions performed by the object under observation are abstracted into *events*. In particular, simple or primitive events are directly produced by the observed object, whereas *complex* events can be defined from simple events by using operators of a suitable *event algebra* [Zim99]. Event specification requires a careful design and configuration activity that is central to the overall setup of a monitoring system. In [SK88], events that may happen in a distributed system are divided in *local, non-local, global.* Local events are produced on a single node, which means that observing them does not require addressing the problems that are related to distribution and inter-node synchronization. Non-local events, on the other hand, are (composite) events whose observation requires considering and correlating events originated from more than a single node. Global events are a special case of non-local events, and require considering *all* the nodes of a system. Recognizing complex events in distributed loosely-coupled environments is not trivial [Fid96], as it requires establishing in which order two or more constituent events (originated from different nodes) happened. As noted by Lamport in his well-known 1978 paper, *in a distributed system it is sometimes impossible to say that one of two events happened first* [Lam78]. Fortunately, the observability problem does not necessarily arise in all distributed systems. For example, if one aims at identifying the service that takes the maximum average response time among a set of services, the problem is not really distributed, as the observation is in fact local and there is no need for aggregated interpretation.

Even though more than 20 years old, Joyce definition of monitoring as *the process of dynamic collection, interpretation, and presentation of information concerning objects or software processes under scrutiny* [JLSU87] remains still relevant and suggests some reflections.

Firstly, it qualifies monitoring as a *dynamic* activity, to underline that it is inherently concerned with the *execution phase* of a system, as opposed to (static) activities that are carried out in the development/coding phase. With this same meaning we also speak of *run-time* monitoring. Secondly, the definition identifies several different activities, namely collection, interpretation, and presentation, as part of monitoring. Each of these activities is meant to address a specific problem and may use dedicated techniques. This is why research on monitoring appears fragmented: the many works on monitoring are hard to compare as most of them focus only on some of the aspects of monitoring. A first attempt to overview the most important problems and issues about monitoring in distributed systems is [MSS94].

Monitor generic architecture Figure 4 depicts the main architectural elements of a generic monitoring system. Elaborating on [MSS94], we identified the following five core functions.

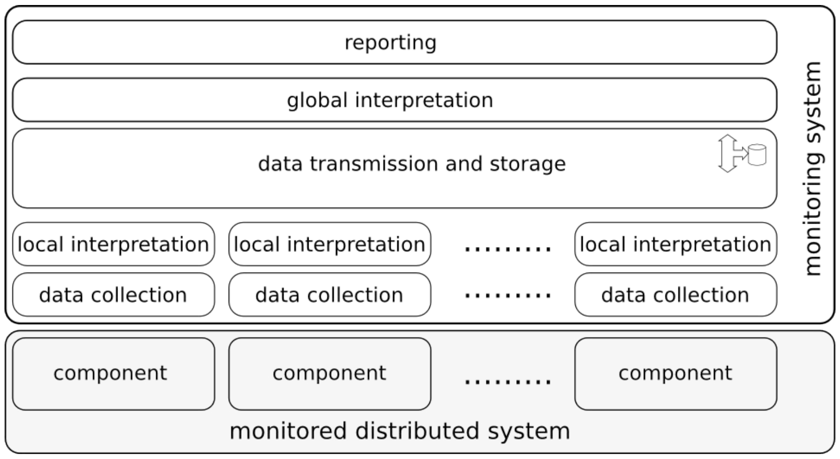


Fig. 4. Architectural elements of a monitoring system

Raw data collection. The lowest layer of a monitoring framework is realized by a set of sensors or probes: these fire a primitive event whenever the observed entity performs some specified computation steps (actions). This is done locally on the entity under observation. Therefore, data collection is the function through which a monitor can produce the largest impact on the system under observation. We expand more on it later.

Local interpretation. The process of local interpretation is concerned with making sense (locally) of the information extracted by probes. This is achieved by applying a filter that extracts interesting sequences of events out of the raw data collected by probes. In practical implementations, data collection and filtering can overlap to some extent: a sort of rough preliminary filtering is done if the events emitted in the data collection step are not just the result of reaching a given point in the execution but also of some other logic or processing embedded in the probe.

Data transmission and storage. In distributed systems, the relevant events that are revealed locally need to be collected at one or more sites where they can be aggregated with analogous data coming from other nodes. Transmission may occur immediately, to reduce detection latency, or may be delayed, using buffering, e.g., to cope with network congestion (at the expense of memory occupation or CPU cycles if compression is used).

Global interpretation (also known as "correlation" or "aggregation"). This function makes sense of pieces of information that come from several nodes and puts them together in order to identify interesting conditions/events that can be observed only at an aggregated level. Architectures with more than one level of aggregation are possible, and are usually adopted when enhanced scalability is necessary [MCC04]. When moving from local to global observation, observability issues must be taken into account. Suitable timestamping and synchronization facilities must be used as appropriate.

Reporting. The information provided as the output of monitoring can be used for a variety of purposes and should be presented in a way that is meaningful to the "consumer" of the monitoring system. The consumer can be a piece of software itself or a human. In both cases the results of the final interpretation phase must undergo an elaboration in order to express output data in a suitable format. In the former case, this format should be machine-readable; in the latter, it must be shown either as a textual report or it may use interactive GUIs, graphics, animations and so on.

The first core function, data collection, determines the monitoring system *intrusiveness* [Sch95], i.e., the level of interference imposed upon the observed application. Intrusive monitors may alter the behaviour that they want to observe. This phenomenon is referred to as the probe effect (or sometime the "Heisenberg effect") [Gai86]. It concerns especially monitoring of performance-related characteristics, but may also impact functional properties, since the process might alter the timing of events and therefore cause wrong behaviour that otherwise would not happen [JLSU87]. Analogously, faults (that would have happened otherwise) can be masked as an effect of the interplay of the subject system and the monitoring.

The collection of data can be done according to two styles [HBPU06]:

- *Instrumentation:* some code is inserted in the application to be monitored in order to emit an event when the control flow reaches a certain point in the execution. This can be realized in different forms at different levels of abstraction. For example, source code instrumentation techniques include statements in the original program to be monitored. Their outcome can be guarded by a condition, whereby it is possible to refine the event definition and to emit an event only when the guard condition is satisfied.
- *Interception:* when adopting this style, data collection is achieved through a proxy-like probe that is put on the wire and snoops interactions, as in [BGG04]. Although this approach may not be as flexible as the previous, it has the advantage of being non-intrusive, therefore it is well suited in other contexts where the control over the system is distributed/partioned across several organizations.

Orthogonally to the instrument/intercept criterion, the techniques for collecting monitoring data can also be distinguished according to whether the collection is based on sampling or complete executions are observed. Most sampling

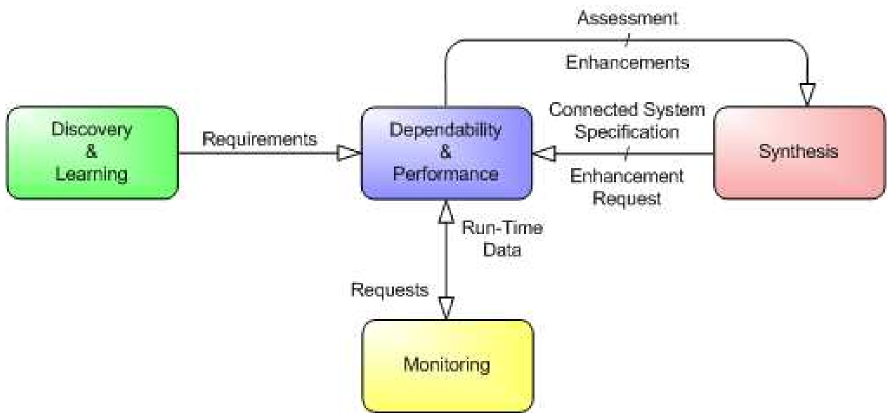

Assessment / Enhancements
Discovery & Learning
Requirements
Dependability & Performance
Connected System Specification / Enhancement Request
Synthesis
Run-Time Data
Requests
Monitoring

Fig. 5. Input-Output Relations between DePer and the Other Enablers

approaches sample on a time basis; however certain (composite) events may go undetected if some of the constituent events are discarded by the sampling. A possible way to address this problem is by sampling in space, rather than in time, i.e., by alternating the processes (or components) that are chosen as the target of monitoring, in such a way that, locally to the target, the observation is complete and no event is discarded.

3 Dependability Assessment Approach in CONNECT

Assessment techniques are sought in Connect to ensure that Networked Systems as well as the generated bridging Connectors satisfy specified levels of accomplishment for dependability and performance requirements, according to pertinent metrics. In the following, we present two approaches under development in Connect for this purpose: the Dependability&Performance Enabler (DePer) and the Monitoring Enabler (Glimpse). The two approaches are first described individually and then their combined use to enhance dependability and performance assessment of the system under analysis is discussed.

3.1 DePer

DePer provides support to the definition of a Connector that allows NSs to interact with a desired level of dependability and performance properties.

Before presenting the architecture of this Enabler, we briefly discuss its relations with other Enablers of the Connect architecture. In [GG11], a complete overview of Connect Enablers and their role is provided. Adopting a DePer-centric view, here we restrict to those having input-output relations with DePer, as shown in Figure 5 (to make the chapter self-complete, the role of relevant Enablers has also been synthetically recalled in Section 1).

According to the CONNECT vision, a Networked System broadcasts a *connect request* whenever a new connection to a service is needed. The connect request contains a description of the requested service together with a specification of the required dependability and performance level for the service. When Discovery detects a connect request, it looks for available Networked Systems that can provide the requested service. If such systems are found and operate a communication protocol different from that of the Networked System that made the connect request, Discovery triggers the process of creating a suitable CONNECTor that enables interoperation. The Synthesis Enabler, on the basis of the specification of the communication protocols, produces a mediating CONNECTor. Before CONNECTor deployment, Synthesis activates DEPER to evaluate if the CONNECTed system that will be obtained satisfies the non-functional requirements expressed by the Networked Systems. If the non-functional requirements are satisfied, the CONNECTor is deployed; otherwise, Synthesis is supported by DEPER in the definition of possible enhancements that can be applied. To take into account dynamic system changes once the CONNECTor is deployed, the Monitoring Enabler, if requested by other Enablers, continuously observes the run-time behaviour of the CONNECTed system and provides them related information, in accordance with received requests.

Therefore, as shown in Figure 5, the joint activity of Discovery and Learning provides the dependability requirements; Synthesis provides the specification of the CONNECTed system, and possibly requests a dependability enhancement; Monitoring provides run-time data on the execution of the deployed CONNECTor. The dependability and performance assessment and the enhancements produced by DEPER are used by Synthesis.

The architecture of the DEPER Enabler is shown in Figure 6 and also preliminarily described in [MMDGar]. Currently, this Enabler accommodates dependability and performance analysis performed through both the stochastic state-based and the stochastic model-checking approaches. Actually, the architecture is general and other analysis methods could be easily included by specifying and implementing an appropriate Analysis Engine module. The Selector and Aggregator modules, at the entrance and exit of the architecture, allow the selection of the analysis method and the aggregation of the analyses results (in case more than one method is applied), respectively. More details on each module are provided in the following.

At the time of writing, a prototype which partially implements the DEPER architecture is under development (http://dcl.isti.cnr.it/DEA/). It is based on the SAN formalism and the Möbius tool, already introduced in Section 2. For those modules already considered in the implementation, some details are also provided in the following description.

3.2 Selector

The Selector module activates, depending on the characteristics of the specification of the CONNECTed system and of the requirements, the most suitable analysis engine among those available to the Enabler. In fact, the employed engines implement different approaches to analyse dependability and performance properties of a CONNECTed system. Each approach has its own advantages regarding modelling capability, specification of properties, and scalability; hence, besides using the different engines to cross validate the results and to improve the confidence in the correctness of the models, they actually complement each other. In the study conducted during the first year [CON10], the characteristics of the stochastic model checking and state-based stochastic methods evaluation approaches have been already pointed out.

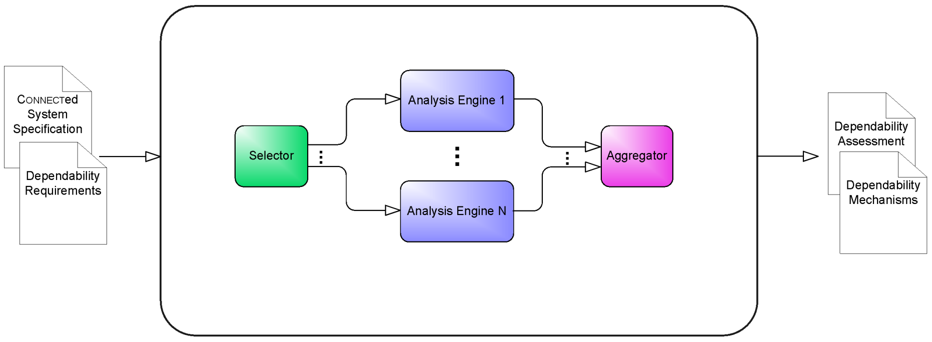


Fig. 6. Architecture of the Dependability&Performance Analysis Enabler

3.3 Aggregator

The Aggregator module is in charge of selecting the analysis results to be provided in output to the Synthesis Enabler, in case more than one Analysis Engines have been activated for a CONNECTed system specification. Therefore, when only one kind of analysis is performed, based on the choice made by the Selector module, Aggregator just conveys the analysis results to the output interface of DEPER. Instead, when more analysis methods are activated, their results are collected by Aggregator and elaborated according to some criteria to derive the output results.
The first step to be performed is a comparison among the values provided by the different methods to check whether they are in agreement (within a certain tolerance degree, to cope with natural dissimilarities inherent to the use of different methods). In the case of a matching comparison, the reliance in each of the employed approaches is increased (cross-validation) and all the analysis results can be equally considered valid, so anyone of them can be output as final analysis values. Alternatively, some form of mediation could be made on the obtained multiple results (e.g., the average), to balance the effects of single method's

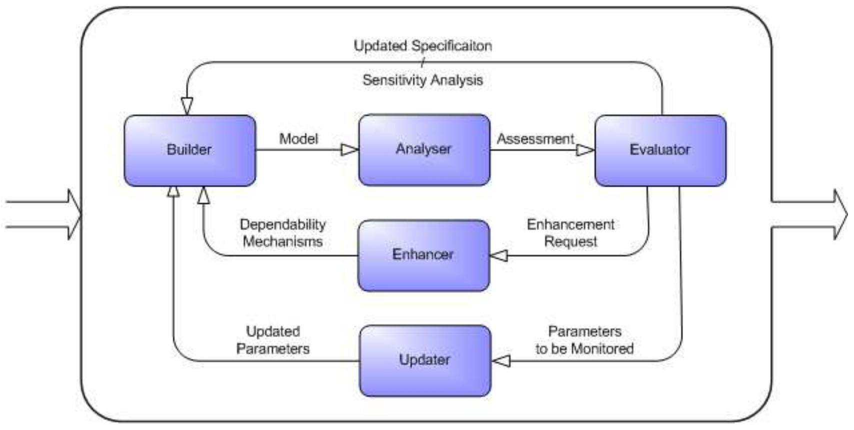


Fig. 7. Architecture of the Dependability&Performance Analysis Engine

approximations. A mismatch, instead, would be the symptom of erroneous/-too inaccurate analysis by at least one of the applied methods. Let us recall that DePer in Connect is based on an automated procedure, starting from given specifications of the Connected system and of the dependability and performance requirements, partially implemented at the current stage. Therefore, once fully automated, we expect that the case of mismatch would be removed by construction; however more investigations are necessary on this issue. The implementation of this module has been deferred, at the moment.

3.4 Dependability&Performance Analysis Engine

The Dependability&Performance Analysis Engine is logically split into five main functional modules (see Figure 7): Builder, Analyser, Evaluator, Enhancer and Updater.

Builder The Builder module takes in input the specification of the Connected system from Synthesis, and the dependability and performance requirements from Discovery/Learning. The module produces in output a dependability and performance model of the Connected system suitable to assess the given dependability and performance requirements.

Specification of the Connect*ed system.* With reference to recent works on synthesis of mediating Connectors [SI10] and automata discovery/learning [RSB05], the specification of the Connected system is given with Labelled Transition Systems (LTSs) annotated with non-functional information necessary to build the dependability and performance model of the Connected system. An LTS is an abstract machine that represents the sequence of actions performed by the system. Formally, an LTS is a tuple $(\mathcal{S}, \mathcal{S}_0, \mathcal{L}, \mathcal{T})$, where $\mathcal{S}$ is a set of states,

$\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of initial states, $\mathcal{L}$ is a set of labels, and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is a transition relation. Annotations include, for each labelled transition, the following fields: *time to complete*, *firing probability*, and *failure probability*. The values for these parameters could be exact values or ranges of values (ranges are especially appropriate when the exact estimate is not possible, given uncertainties of the environment).

Dependability and performance requirements. In our architecture, the dependability and performance properties required by the Networked Systems are translated by Discovery/Learning into *metrics* and *guarantees*. Metrics are arithmetic expressions that describe how to obtain a quantitative assessment of the properties of interest of the CONNECTed system. They are expressed in terms of transitions and states of the LTS specification of the Networked Systems. Guarantees are boolean expressions that are required to be satisfied on the metrics.

Dependability and performance model. The dependability and performance model of the CONNECTed system is specified with a formalism that allows to describe complex systems that have probabilistic behaviour, e.g., stochastic processes.

Implementation. The prototype implementation of the Builder module takes in input the LTS of the connected system described with Finite State Processes (FSP) [MK06]. The dependability model of the system is specified with Stochastic Activity Networks (SANs), already introduced in section 2. The SAN model is obtained from the LTS model by using the theory of regions [ER90]. A region identifies a set of states in the LTS such that all transitions with the same label either enter, exit, or never cross the boundary of the region. Each region in the LTS corresponds to a place in the derived SAN model, and each labelled transition in the LTS corresponds to an activity in the SAN model. A similar approach has already been used in other works to translate LTSs into Petri Nets (see, for instance, [CKLY98], [BS02] and [CCK09]). In order to have a well-defined probabilistic model, non-deterministic choices among k transitions outgoing from an LTS state are mapped in the SAN model into instantaneous activities with k case probabilities. The metric is an arithmetic expression that may contain a predefined set of functions (see Table 1 for some examples). The guarantee is given by a boolean expression on the metric and a set of constraints on the connected system model (e.g., constraints on the time frame of evaluation of the metric). Statistical operators (e.g., *mean* and *variance*), comparison and logical operators can be used in the expression.

Analyser The Analyser module takes in input the dependability and performance model from the Builder module and the dependability and performance requirements from Discovery/Learning. The module builds a reward model, i.e., a model that enables a quantitative assessment of the metrics of interest, and makes use of a solver engine to obtain a quantitative assessment of the dependability and performance metrics.

Function	Description
$timeFrame(s) : \mathcal{S} \rightarrow \mathbb{R}^+$	returns the interval of time when the system is in state s
$minTimeStamp(tr) : \mathcal{T} \rightarrow \mathbb{R}^+$	returns the first instant of time when transition tr fires
$avgTimeStamp(tr) : \mathcal{T} \rightarrow \mathbb{R}^+$	returns the average instant of time when transition tr fires
$maxTimeStamp(tr) : \mathcal{T} \rightarrow \mathbb{R}^+$	returns the last instant of time when transition tr fires
$\#(tr, t1, t2) : \mathcal{T} \times \mathbb{R}^+ \times \mathbb{R}^+ \rightarrow \mathbb{N}$	returns the number of times transition tr fires during the time frame $[t1, t2]$
$\#(l, t1, t2) : \mathcal{L} \times \mathbb{R}^+ \times \mathbb{R}^+ \rightarrow \mathbb{N}$	returns the number of times transitions with label l fire during the time frame $[t1, t2]$

Table 1. Examples of predefined functions that can be used in the metric expression

Reward model. The reward model is the dependability and performance model extended with reward functions. Reward functions allow to specify properties of interest: they return a value depending on the system state, and can be evaluated either at an instant of time or accumulated over a time frame.

Solver. The solver engine evaluates the reward functions defined in the reward model. The evaluation can be performed either through analytical approaches or through simulation, depending on the metrics under evaluation and on the mathematical representation of the involved phenomena.

Implementation. The prototype implementation of the Analyser is based on Möbius, already introduced in section 2. In Möbius, each reward function is a C++ function that returns a value depending on the marking of the SAN. There are two kinds of reward functions: *rate rewards* and *impulse rewards*. Rate rewards are used to implement time-based reward functions. Impulse rewards are used to implement action-based reward functions, i.e., they are associated with specific activities and can be evaluated only when the associated activity completes. The reward functions are automatically derived from the metrics expression as follows: the metric is mapped into its syntax tree to decompose the metric into a combination of basic functions; the basic functions are translated into C++ functions by using a predefined repository of function templates (currently under construction). For instance, with reference to the functions shown in Table 1, a rate reward template is used to translate $timeFrame(s)$, while an impulse reward template is used to translate $\#(tr, t1, t2)$. Then, the quantitative assessment of the metric is obtained from the assessment of the reward functions by merging the results according to the arithmetic operations specified in the syntax tree of the metric expression.

Evaluator The Evaluator module reports to Synthesis if the CONNECTed system satisfies the dependability and performance requirements provided by Discovery/Learning. In the case of requirements mismatch, Evaluator sends a warning message to Synthesis, and may receive back a request to evaluate if enhancements can be applied to improve the dependability (and/or performance, depending on the received request) level of the CONNECTed system.

In view of the synergic cooperation with the monitoring infrastructure, this module also informs the Updater module, which is in relationship with the Monitor Enabler, about the model parameters for on-line observation.

Requirements mismatch. If the requirements are not satisfied, Evaluator may receive a request to explore one of the following three directions for improvements:

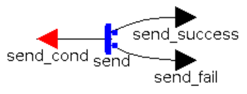
1. Update the specification of the CONNECTor to take into account an alternative CONNECTor deployment (e.g., a deployment that uses a communication channel with lower failure rate). Upon receiving this request, the Evaluator triggers a new analysis that considers the updated specification of the CONNECTor.
2. Enhance the specification of the CONNECTor by including dependability mechanisms, which are counter-measures to contrast failure modes affecting performance and/or dependability metrics (e.g., a message retransmission technique). Upon receiving this request, it is first necessary to understand which are the failure probabilities mostly impacting on the metrics evaluation, so as to include primarily dependability mechanisms capable of limiting the effects of such highly impacting failures. To this end, Evaluator builds a sensitivity analysis campaign to instruct the Builder module on the creation of dependability and performance model variants, each of which considers a specific subset of failure probabilities, among those foreseen. Whenever a variant is generated, the Analyser module performs the assessment of the metrics on the generated model. Evaluator collects the analysis results and, after all variants have been analysed, produces a ranking of the failure probabilities. This ranking is used by Evaluator to iteratively activate the Enhancer module until one of the following conditions is met: the guarantees are satisfied, or Enhancer signals that all possible dependability mechanisms have been explored.
3. Apply a combination of the previously mentioned enhancements.

Enhancer The Enhancer module is activated by Evaluator when the guarantees are not satisfied and Synthesis makes a request to enhance the CONNECTor with dependability mechanisms. Enhancer is instructed by the Evaluator module on the requirements mismatch and the failure probability that needs to be improved. Then, it performs the following actions: (i) selects the dependability mechanisms that can be employed, among those available, to improve the failure probability indicated by the Evaluator module; (ii) instructs the Builder module on the application of the selected dependability mechanisms in the CONNECTed system model (one model variant will be generated for each dependability mechanism selected, generally only one) and triggers a new analysis for each of the generated models.

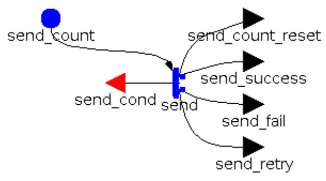
Dependability mechanisms. Typically, dependability mechanisms are based on the application of redundancy, e.g., duplication of system channels, or retry of message transmissions over system channels. The dependability mechanism, in this context, will be embedded in the synthesised CONNECTor, because Networked Systems are not under the control of the framework. Nevertheless, the dependability mechanisms embedded in the CONNECTor can be employed to improve, to some extent, the dependability and performance level of the Networked Systems. For example, the reliability level of a transmission performed by a Networked System can be improved through timeouts or message retransmissions applied at the CONNECTor level.

Implementation. We developed ad hoc dependability models for a set of relevant dependability mechanisms, and a set of rules to automate the application of the mechanisms in the SAN model of the connected system. The ad hoc models can be parametric; for instance, a retransmission mechanism is parametric with respect to the maximum number of allowed retransmissions. As an example of mechanism and application rule, in Figure 8 we graphically show a retransmission mechanism and how to modify the original model in order to apply message retransmissions to a send operation. Specifically, the original model contains a timed activity `send` that models the send operation, an input gate `send_cond` that specifies the enabling condition of the activity, and two output gates, `send_success` and `send_fail`, that specify the output functions in the case of correct and faulty behaviour. The enhanced model is obtained from the original model by adding the following elements: a place `send_count`, with initial marking the maximum allowed number of retransmissions; an output gate `send_count_reset`, which resets the marking of `send_count` to its initial value when the `send` succeeds; an output gate `send_retry`, which reactivates `send` as long as `send_count` contains tokens, and resets the marking of `send_count` after performing all retransmission attempts.

In a more structured vision, the dependability mechanism(s) suitable to improve on a given failure probability are determined through an ontology of dependability mechanisms, such as that reported in [ReS08]. The definition of such an ontology is planned as future work.



(a) Basic model



(b) Enhanced model

Fig. 8. Example of dependability mechanism and application rule

Updater The Updater module interacts with the Monitoring Enabler to refine the accuracy of model parameters through on-line observations. Inaccuracy of the non-functional values used in the off-line analysis at CONNECTor design time is mainly due to two possible causes: i) limited knowledge of the NSs characteristics acquired by Learning/Discovery Enablers; ii) evolution along time of the NSs, as naturally accounted for in the CONNECT context.

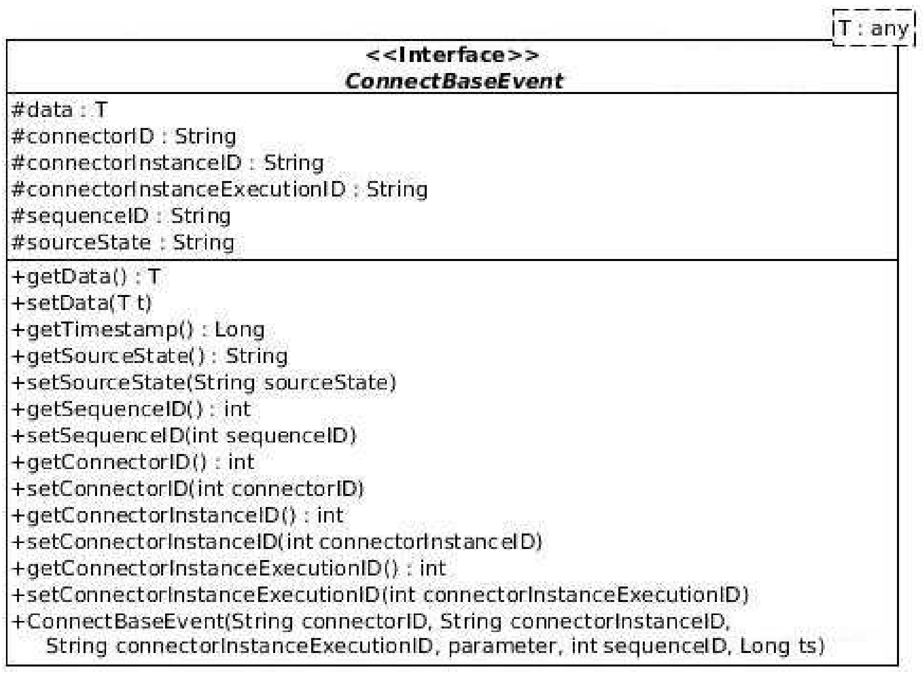
Updater receives inputs from both internally to DEPER (from the Evaluator) and externally (from the Monitor Enabler). From the former, for each CONNECTor ready to be deployed it receives the model parameters to convey to the Monitor for run-time observations. From the latter, it receives a continuous flow of data for the parameters under monitoring relative to the different executions of the CONNECTor. Accumulated data are processed through statistical inference techniques. If, for a given parameter, the statistical inference indicates a discrepancy between the on-line observed behaviour and the off-line estimated value used in the model resulting into a significant deviation of the performed analysis, a new analysis is triggered by instructing the Builder module to update the CONNECTed system model.

Details on the implementation between Updater and Monitoring are illustrated in Subsection 3.6.

Statistical Inference Techniques. As reported in [Tri02], methods of statistical inference applied to a collection of elements under investigation (called population), allow to estimate the characteristics of the entire population. In our case, the collection of values relative to each parameter under monitoring constitute a population to which such techniques are applied.

3.5 GLIMPSE

In CONNECT we developed a monitoring infrastructure, called GLIMPSE, aimed at covering the main function[MSS94] for the monitoring discussed in section 2.3. We tried to implement these five core function in a modular and flexible way, aiming to support behavioural learning, performance, reliability assessment, security and trust also in non-CONNECT context.
GLIMPSE is an acronym for "Generic fLexIble Monitoring based on a Publish-Subscribe infrastructurE". The architecture of GLIMPSE is shown in Figure 10; details of each component are in the following.
Probes. There are entrusted to *Collection* and *Local interpretation* functions. Probes are usually realized by injecting code into the existing software or by using a proxy. The probes that we used in GLIMPSE are already injected into the CONNECTor during its synthesis phase. When primitive events occur into the software, probes send them to the Monitoring Bus component (described below).
An event, in the current context, represent a transition between two states of an LTS. In our implementation we collect all the information that may be useful at GLIMPSE to infer complex event occurrences for the analysis in the `ConnectBaseEvent` interface.



```
T : any
<<Interface>>
ConnectBaseEvent
#data : T
#connectorID : String
#connectorInstanceID : String
#connectorInstanceExecutionID : String
#sequenceID : String
#sourceState : String
+getData() : T
+setData(T t)
+getTimestamp() : Long
+getSourceState() : String
+setSourceState(String sourceState)
+getSequenceID() : int
+setSequenceID(int sequenceID)
+getConnectorID() : int
+setConnectorID(int connectorID)
+getConnectorInstanceID() : int
+setConnectorInstanceID(int connectorInstanceID)
+getConnectorInstanceExecutionID() : int
+setConnectorInstanceExecutionID(int connectorInstanceExecutionID)
+ConnectBaseEvent(String connectorID, String connectorInstanceID,
   String connectorInstanceExecutionID, parameter, int sequenceID, Long ts)
```

Fig. 9. ConnectBaseEvent Interface

The event description is shown in figure 9.
Examinating more in details some of the parameters composing the `ConnectBaseEvent` interface we can find:

- `connectorID` : defines the identity of the CONNECTor
- `connectorInstanceID` : used to define the execution of the CONNECTor

Monitoring bus. The monitoring bus is the communication backbone where all information (events, requests, responses) is sent on by: Probes, Enablers, Complex Event Processor and by all the services using GLIMPSE. To obtain a better decoupling and to keep asynchronous communication, in our implementation we decided to adopt public-subscribe paradigm, which is implemented with messages queue through ServiceMix4 and Java Message Service. There are many commercial products that implement Enterprise Service Bus, we prefer ServiceMix for its strong compatibility and simplicity interacting with the rule engine used for this prototype.

However, in designing the GLIMPSE architecture, we adopted a model-driven which allow GLIMPSE to use different rule languages. The handling of the Monitoring Bus is devoted to the Manager component, analysed below.
The usage of a messaging system in Enterprise Service Bus (ESB) allows GLIMPSE to use a variety of protocols such as HTTP/SOAP and REST. Moreover, with the

usage of JBI [jbi] components, Glimpse interact even with legacy systems, binary transports, document-oriented transports, and Remote Procedure Call [rpc] systems. Adopting a messaging system reduces execution bottlenecks that might occur using Remote Procedure Call or Database-centric architecture.
Complex Event Processor. The Complex Event Processor (CEP) is the rule engine that allows to infer complex events from primitive sent on the Monitoring Bus by Probes.
The CEP is instructed at runtime by the Manager component that, after analysing the consumer (Enabler) request expressed sending a JMS message on wich payload is written in XML using the `ComplexEventRuleActionList` schema (see Listing 1.1), load the new inference rules on the engine. There are several rule engines that can be used for this task (like Drools Fusion [dro], RuleML [rul]), in the existing prototype wu used Drools Fusion.
The schema of `complexEventActionList` is shown in Listing 1.1.
The `complexEventActionList` specification supports the use of heterogeneous rule languages, as it is natively unbound to any. Examinating the Listing 1.1, at line 30, we found the RuleBody field that is used by the Enabler to set the Drools rule for the requested evaluation. Into the field RuleType (line 32), the Enabler will set the type of rule language requested for the evaluation, this will allow to use more rule languages even at run-time.
Consumer. In Glimpse, a `Consumer` may be a learning engine, a dependability analyser or a simple customer that requests some information to be monitored. The basic requirement to interact with Glimpse is to be able to send/receive JMS messages and to raise correct query to the inference engine (CEP). The `Consumer` sends a request to the Manager using the Monitoring Bus and waits for the evaluation results on a dedicated response channel provided and notified by the Manager.
Manager. It manages all the communications among its components. The Manager component is the orchestrator of all the Glimpse architecture. Specifically, the Manager fetches requests coming from Consumers (Enablers), analyses them and instructs the Probes. Then, it instructs the CEP Evaluator, creates and notifies to the Consumer a dedicated channel on which it will provide results produced by the CEP Evaluator. (For more information about interaction, sec:3.6). The most valuable support provided by the `Manager` is the handling of all the knowledge base loaded into the CEP.
Glimpse *Implementation.* A prototype of Glimpse is available for public download at http://labse.isti.cnr.it/tools/glimpse.

3.6 Integrated run-time analysis

Synergic use of DePer and Glimpse is pursued to allow automated refinement of dependability and performance analysis through inspection of run-time data, as preliminarly described in [BCDG+ar]. Precisely, feedbacks from run-time executions of the CONNECTed system as collected from Glimpse are used by DePer to enhance the accuracy of model parameters adopted in the analysis performed at design time.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule" xmlns:tns="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule" elementFormDefault="qualified">


    <element name="ComplexEventRuleActionList" type="tns:ComplexEventRuleActionType"></element>

    <complexType name="ComplexEventRuleActionType">
      <sequence>
        <element name="Insert" type="tns:ComplexEventRuleType"
          maxOccurs="unbounded" minOccurs="0">
        </element>
        <element name="Delete" type="tns:ComplexEventRuleType"
          maxOccurs="unbounded" minOccurs="0">
        </element>
        <element name="Start" type="tns:ComplexEventRuleType"
          maxOccurs="unbounded" minOccurs="0">
        </element>
        <element name="Stop" type="tns:ComplexEventRuleType"
          maxOccurs="unbounded" minOccurs="0">
        </element>
        <element name="Restart" type="tns:ComplexEventRuleType"
          maxOccurs="unbounded" minOccurs="0">
        </element>
      </sequence>
    </complexType>

    <complexType name="ComplexEventRuleType">
      <sequence>
        <element name="RuleName" type="string" maxOccurs="1" minOccurs="1"></element>
        <element name="RuleBody" type="string" maxOccurs="1" minOccurs="0"></element>
      </sequence>
      <attribute name="RuleType" type="string"></attribute>
    </complexType>
</schema>
```

Listing 1.1. ComplexEventRuleActionList Schema

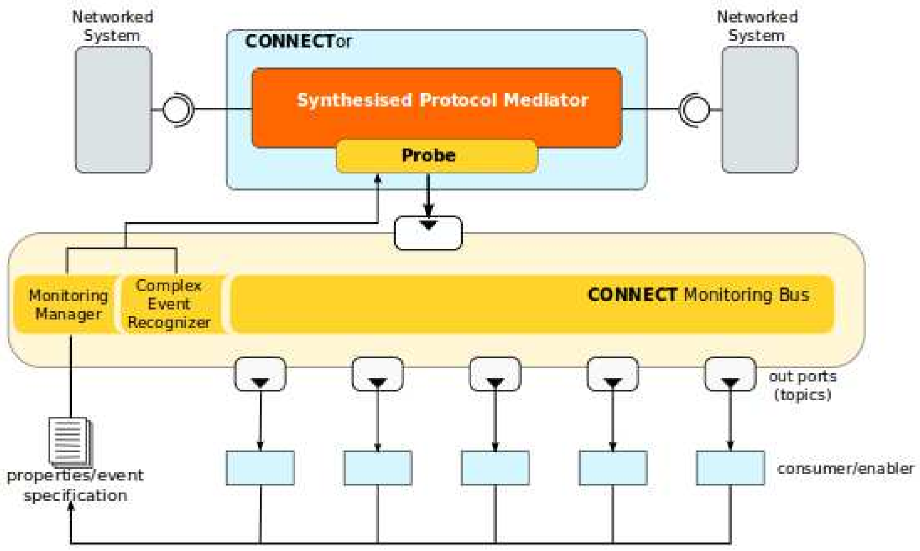


Fig. 10. The architecture of GLIMPSE

The interactions between DEPER and GLIMPSE Enablers start after the DEPER Enabler determines that the synthesised CONNECTor satisfies the required dependability and performance level. Specifically, after the analysis phase, if compliance with requirements is verified with the consequent deployment of the CONNECTor, DEPER informs the GLIMPSE on which are the parameters (among those used in the dependability analysis) relative to CONNECTor and NSs, that must be kept under observation at run-time. GLIMPSE, upon receiving the request, properly instructs the probes embedded in the CONNECTor.

Run-time data relative to parameters under observation are sent by the GLIMPSE to DEPER. DEPER, through its Updater module, continuously performs statistical analyses on the collection of data received to verify whether the accuracy of the model parameters used in the analysis is good enough for the analysis results to be still valid, or the CONNECTor no longer satisfies the requirements and needs adjustments. In this latter case, a new analysis adopting the updates parameters values is triggered.

Figure 11 shows the interaction between DEPER and GLIMPSE. DEPER is shown inside the dotted box at the top of the Figure, while GLIMPSE is shown again in a dotted box at the bottom left side of the Figure. For clarity, only relevant modules involved in the cycle with Monitoring, are shown. Inside DEPER, the activities from receiving the CONNECTed System specifications till conclusion of the evaluation phase are represented. Since we want to show the interaction between DEPER and GLIMPSE, we depicted the case where the dependability and performance requirements are met, so the Evaluator module reports to Synthesis that the CONNECTor can be deployed and triggers Updater on sending

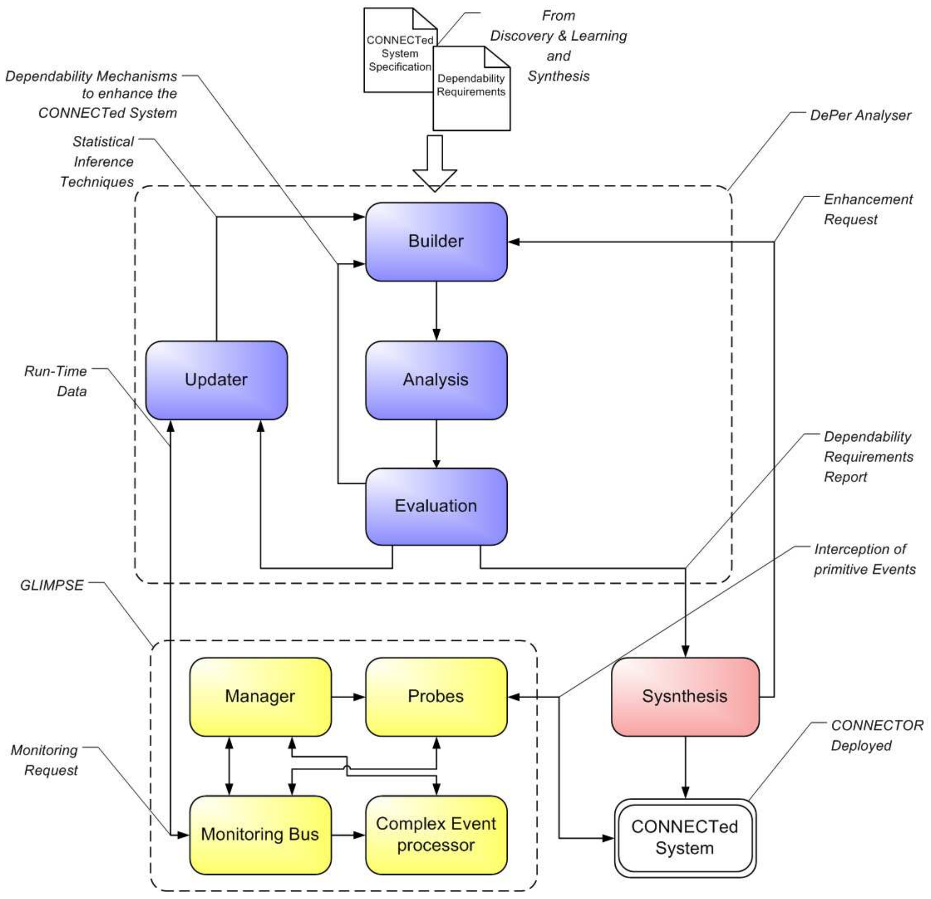

CONNECTed System Specification
Dependability Requirements
From Discovery & Learning and Synthesis
Dependability Mechanisms to enhance the CONNECTed System
Statistical Inference Techniques
DePer Analyser
Enhancement Request
Builder
Updater
Analysis
Evaluation
Run-Time Data
Dependability Requirements Report
Interception of primitive Events
GLIMPSE
Manager
Probes
Sysnthesis
Monitoring Request
Monitoring Bus
Complex Event processor
CONNECTOR Deployed
CONNECTed System

Fig. 11. Interactions between DePer and Glimpse

monitoring requests to Glimpse. Such requests are received by the Manager component of Glimpse through a service channel on the Monitoring Bus, which instructs the Probes and the Complex Event Processor and creates a dedicated communication channel on the Monitoring Bus, used to provide results to DePer.
Then, Probes start intercepting events of interest, when they occur. If DePer requests consist of complex events, their composing primitive events are elaborated by the rule engine Complex Event Processor, and resulting values are computed.
Responses to monitoring requests so determined are sent to DePer through a dedicated channel on the Monitoring Bus.
Once the Updater module of DePer receives the run-time data from Glimpse, it applies to them statistical inference techniques to determine the actual values of the corresponding model parameters. In case the newly determined values are

outside the range assumed in the analysis at design time, the analysis model is updated with the new values and solved again. If the new analysis evidences that the deployed Connector needs adjustments, a new synthesis-analysis cycle is started in cooperation with the Synthesis Enabler and a notification is sent to Glimpse about stopping monitoring the no more satisfactory Connector.

Implementation. In the implementation performed so far, DePer and Glimpse interact by using a Publish/Subscribe protocol. The interaction pattern is shown as a sequence diagram in Figure 12 where we intentionally left out system start-up operations (for a more detailed sequence diagram, see [CON11a]). Whenever Monitoring Enabler receives a request message on the service channel (see message 2 on Figure 12), a new channel dedicated to the requesting Enabler is set up to communicate the monitored values.

Glimpse sends response messages to DePer Enabler as soon as the aspect of interest is available (see message 8 on Figure 12).

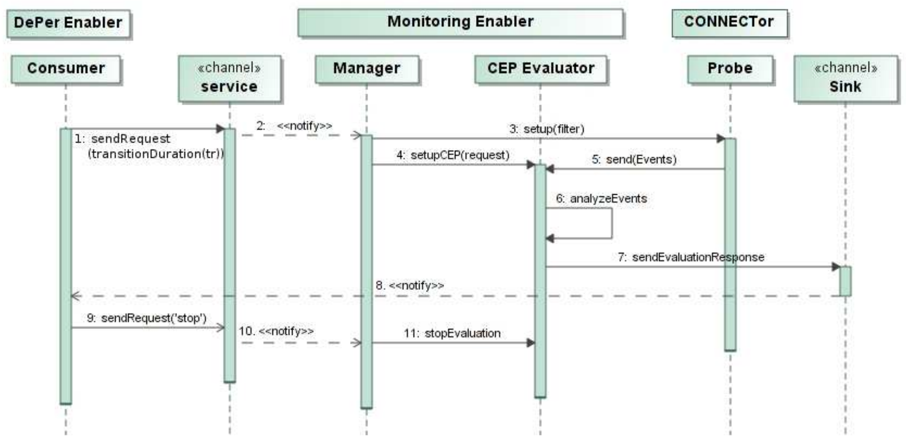


Fig. 12. Sequence Diagram of the Basic Interaction Pattern between DePer and Glimpse

The two Enablers exchange JMS messages whose payload is expressed in XML language. The payload of the XML, contains a `ComplexEventRuleActionList` xml object, which determines a lists of possible actions to execute on the Monitoring Enabler knowledge base. The schema of ComplexEventRuleActionList is shown in Listing 1.1.

4 Example

In this section we first introduce an example scenario and then we show how we apply the presented approaches to it.

4.1 The Terrorist Alert Scenario

We consider the CONNECT Terrorist Alert scenario [CON11c], depicting the critical situation that during a show in the stadium, the control center spots one suspect terrorist moving around. The alarm is immediately sent to the Police.

Policemen are equipped with ad hoc handheld devices which are connected to the Police control center to receive command and documents. Precisely, the policemen can share documents with the Police control center and with other policemen through a *SecuredFileSharing* application, for example a picture of a suspect terrorist.

Unfortunately, the suspect is put on alert from the police movements and tries to escape, evading the Stadium.

In such an emergency situation, there may be various cases in which CONNECT can be of help. As described in [CON11c], the police could for example be directly put in connection with various surveillance systems in the zone to receive videos or pictures in their devices. We focus on the case that a policeman that sees the suspect running away can dynamically seek assistance to capture him from civilians serving as private security guards in the zone of interest. To get help in following the moves of the escaping terrorist and capturing him, the policeman sends to the civilian guards an alert message in which one picture of the suspect is distributed.

On their side, to perform their service, the guards are equipped with smart radio transmitters which run an *EmergencyCall* application. This transmission follows a two steps protocol. We assume in fact that the guards that control a zone are CONNECTed in groups, and that for each group there is a Commander on duty. The protocol followed in the *EmergencyCall* application is that a request message is first sent from the guards control center to the Commander. As soon as the Commander replies with an acknowledgement of receipt, a message with details of the emergency is forwarded to all security guards. On correct receipt of the alert, each guard's device automatically sends an ack to the control center.

The two applications, *SecuredFileSharing* and *EmergencyCall*, in this scenario represent the two Networked Systems, which are not a priori compatible; hence a CONNECTor bridging between the policeman device and the guard device must be deployed.

In the following we show the LTSs modelling the two applications above mentioned.

SecuredFileSharing

- The peer that initiates the communication (hereafter denominated the *coordinator*) sends a broadcast message (`selectArea`) to selected peers (the Police control center or policemen) operating in a specified area of interest. In the SecuredFileSharing application, the coordinator can be either the Police control center or a policeman.
- The selected peers reply with an `areaSelected` message.
- The coordinator sends an `uploadData` message to transmit confidential data to the selected peers.

– Each selected peer automatically notifies the coordinator with an `uploadSuccess` message when the data have been successfully received.

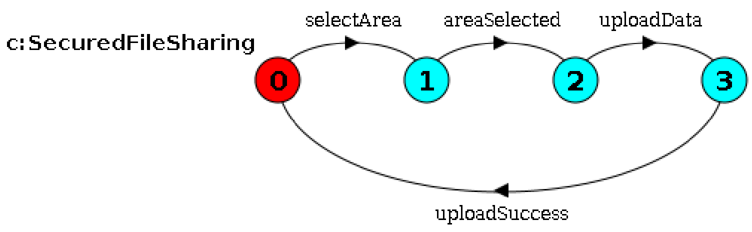


Fig. 13. LTS of the *SecuredFileSharing* Application

EmergencyCall

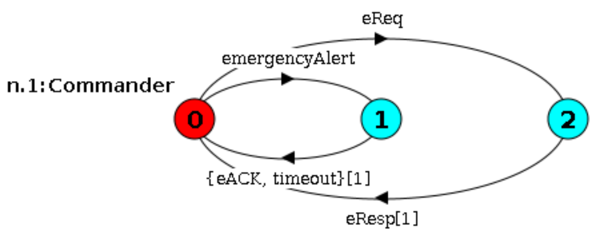
– The guards control center sends an `eReq` message to the commanders of the patrolling groups operating in a given area of interest.
– The commanders reply with an `eResp` message.
– The guards control center sends an `emergencyAlert` message to all guards of the patrolling groups; the message reports the alert details.
– Each guard's device automatically notifies the guards control center with an `eACK` message when the data has been successfully received and a timeout is triggered after a time interval if not all guards sends back the `eAck` message. The timeout represents the maximum time that the CONNECTor can wait for the `eAck` message from the guards.

To allow a Policeman and the guards in the zone where the suspect has escaped to communicate we need to synthesize on-the-fly a CONNECTor. Precisely, we need to mediate between the LTSs shown in Figures 13 and 14, respectively. We briefly summarise the needed mappings below.

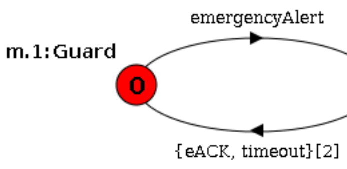
CONNECTor

– The `selectArea` message of the policeman is translated into an `eReq` message directed to the commander of the patrolling group operating in the area of interest.
– The `eResp` message of the commander is translated into an `areaSelected` message for the policeman.
– The `uploadData` message of the policeman is translated into a multicast `emergencyAlert` message.
– The `eACK` messages automatically sent by the guards' devices that correctly receive the `emergencyAlert` message are collected and then translated into a single `uploadSuccess` message for the policeman.

The LTS of the CONNECTor in the case of a patrolling group consisting of one commander and two other guards is shown in Figure 15.



(a) Commander



(b) Other guards

Fig. 14. LTSs of the *EmergencyCall* Application

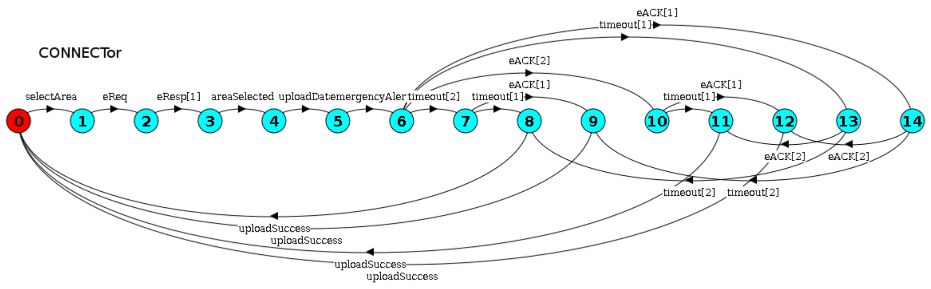


Fig. 15. LTS of the CONNECTor

4.2 Off-line analysis

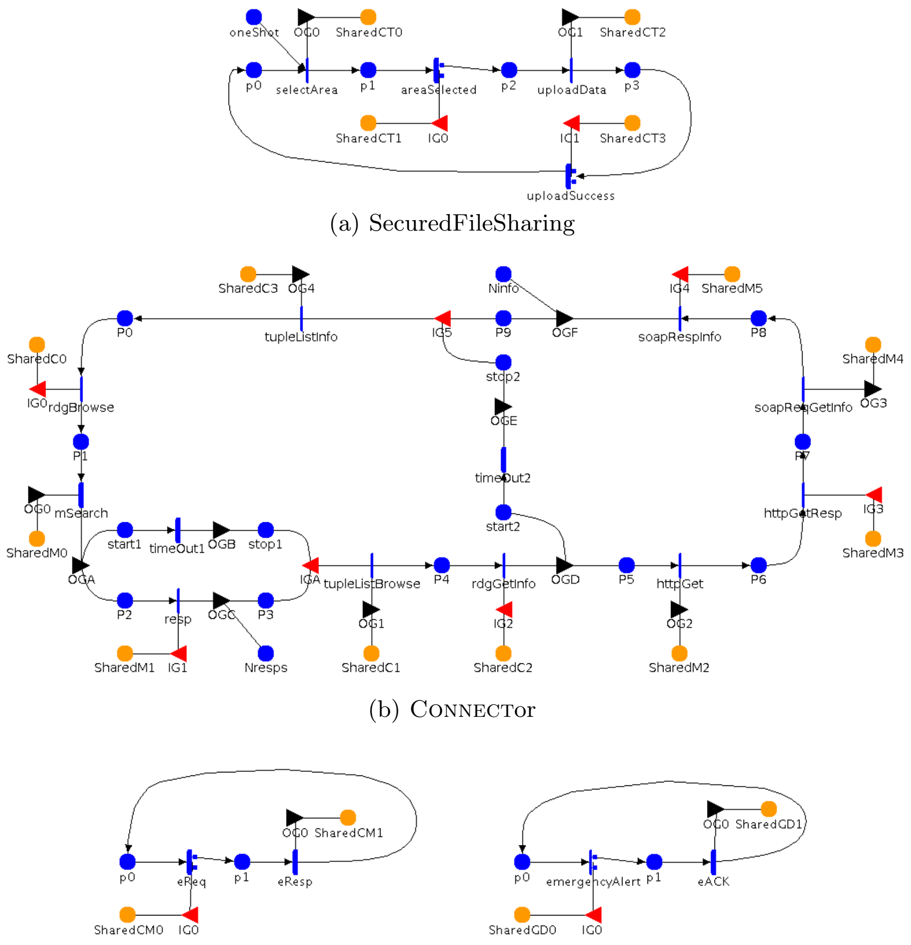
In this section, first we show the SAN models of the case study under analysis, then the results of the analysis obtained through Möbius.

SAN Models The SAN models of guard, commander, CONNECTor, and SecuredFileSharing are shown in Figure 16. The model of the CONNECTed system is obtained by composing, via place sharing, the SAN models of SecuredFileSharing, commander, CONNECTor and guards (the SAN model of the guards is obtained by replicating a guard with the Rep operator). There is a shared place for each pair of activities that represent send/receive actions: send activities add tokens in the shared place, while receive activities remove tokens from the shared place and use the marking of the shared place as enabling condition. Note that, in general, a send activity may control $n > 1$ receive activities (e.g., in the case of a message with multicast/broadcast addresses); in this case, the send activity will add n tokens to the shared place to allow the simultaneous enabling of the receive activity of n receivers.

Timing aspects for send/receive actions are taken into account in the SAN models as follows: when n receive activities complete simultaneously after a send action completes, the receive activities are instantaneous and the send activity is timed; when n receive activities complete independently after a send action completes, the receive activities are timed and the send activity is instantaneous. Timeouts are modelled with timed activities that force the enabling of other activities.

In the following we describe in detail the behaviour of the model of CONNECTed system. In the description, we will use the prefixes `C`, `G`, `CON`, and `S` to disambiguate the names of local places, activities and gates of commander, guards, CONNECTor, and SecuredFileSharing.

Initially, all places in the models have zero tokens, except `p0`, which contains one token in all models. The SecuredFileSharing starts the communication, because `S.selectArea` is the only enabled activity. When `S.selectArea` completes, one token is placed in `S.p1` and one token in `SharedCT0`. At this point, `S.selectArea` is enabled. When `S.selectArea` completes, one token is placed in `S.p1` and the number of tokens in `SharedCT0` is increased. The activity `CON.selectArea` is now enabled, when it completes one token is moved from `SharedCT0` to `CON.p1`, and `CON.eReq` becomes enabled. When `CON.eReq` completes, the marking changes as follows: *commNum* tokens are placed in `SharedCM0`, because *commNum* commanders must be involved in the communication; *commNum* tokens are placed in `CON.p2`, because the CONNECTor must wait for one `eResp` from each commander. When the CONNECTor receives a response from each commanders, (i) for each response received one token is placed in `CON.p3`; (ii) when each commanders has sent a response `CON.areaSelected` is enabled, one token is placed in `CON.p4` and the number of tokens in `SharedCT1` is increased. At this point `S.areaSelected` is enabled, when it completes one token is moved from `SharedCT1` to `S.p2`, and `S.uploadData` becomes enabled. A token is placed in `S.p3` and the number of token in `SharedCT2` is increased. Activity `CON.uploadData` is now enabled, when it completes one token is moved from `SharedCT2` to `CON.p5`, which enables the activity `emergencyAlert`. When `emergencyAlert` completes $commNum + guardNum$ tokens are placed both in `SharedGD0` and `CON.p6`, and the number of tokens in `CON.start1` is increased. At this point activities `CON.timeOut1` and `G.emergencyAlert` are both enabled. The first one represents the CONNECTor's timeout on the maximum waiting time; while the second one enables the activity `G.eACK` which increases the number of tokens in `SharedGD1`. At this point activity `CON.eACK` is enabled and the number of tokens in `CON.p7` and `CON.Nresps` is increased, until the timed activity `CON.timeOut1` completes. The activity `uploadSuccess` becomes enabled when $commNum+guardNum$ tokens are placed in `CON.p7`, this means that the CONNECTor has received all responses, or when the number of tokens in `CON.stop1` is



(a) SecuredFileSharing

(b) CONNECTOr

(c) EmergencyCall, commander

(d) EmergencyCall, guard

Fig. 16. SAN Models

greater than zero, this means that the time associated to the activity `timeOut1` has elapsed, `CON.timeOut1` completes. The number of tokens in `CON.Nresps` represents the number of guards that have recived the `emergencyAlert` and have sent back the `eACK` before the timeout.

State-based Stochastic Analysis The analysis performed through Möbius consists in: i) two measures of latency, at varying the number of guards and for different traffic patterns; ii) a measure of coverage in case of failure.

Latency. This property is measured from the moment when the control center starts to send the initial request `selectArea` to the time it receives `uploadSuccess`. The latency is specified by accumulating over time the following rate reward function:

```
double latency() {
    if ( SecuredFileSharing->p1->Mark() > 0
         || SecuredFileSharing->p2->Mark() > 0
         || SecuredFileSharing->p3->Mark() > 0 )
       { return 1; }
}
```

Latency2. It is also useful to know the trend of the amount of time spent on waiting for `eAck`, given different values of **T**, the duration of the timeout shown in the model of the Connector,.
This property is specified by accumulating over time the following rate reward function:

```
double latency2() {
    if ( connector->start1->Mark() > 0 && connector->p6->Mark() > 0 )
       { return 1; }
}
```

Coverage. This property is associated to the real value m/n, where n represents the total number of guards and commanders, and m represents how many of them send back their respons to the connector within **T** time units, after they receive the request `emergencyAlert`.
Coverage is specified by accumulating over time the following impulse reward on `CON.uploadSuccess` (`guardNum` and `commNum` are two parameters of the composed model, and hold the number of guards and commanders respectively):

```
double coverage() {
 return ( (double) connector->Nresps->Mark() ) / ( guardNum + commNum );
}
```

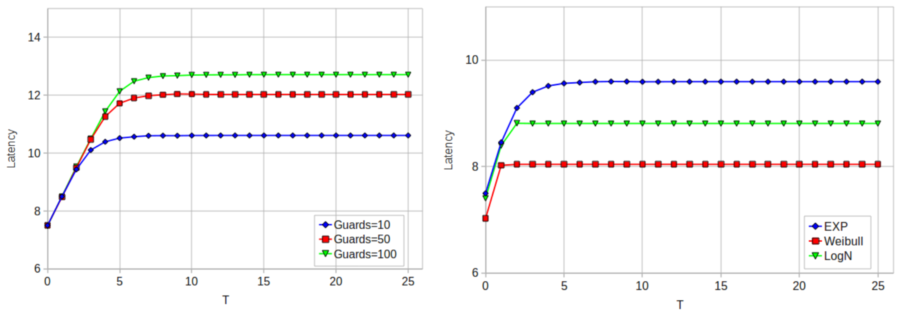
Connected systems may include an arbitrary large number of Networked Systems. Therefore, we investigated the scalability of the SAN model of the Connected system by analysing large networks. The developed SAN model of the Connected system is parametric with respect to the number of guards and commanders.

We successfully assessed coverage and latency for scenarios with hundreds of guards and two commanders. Figure 17(a) shows the analysis results for latency in scenarios with at most 100 guards. We can notice that, for low values of the timeout **T**, it is not possibile to appreciate differences in latency at increasing the number of guards. In fact, due to the short duration of **T** the guards do not have enough time to send a response. When **T** becomes greater than 8 time units, it is possible to observe how the number of guards affects the value of latency: as expected, increasing the number of guards leads to an increase of

latency.
The number of batches needed to reach a confidence level of 95% and a confidence interval of 10% for the considered models was always below 10K, because the models are relatively simple.

Latency for different traffic patterns CONNECTed systems are expected to be a mix of heterogeneous user applications, each of which may have different characteristics and requirements. Currently, there is no single traffic distribution that can efficiently capture the traffic characteristics of all types of networks under every possible situation. A large number of empirical studies have shown that network traffic is self-similar and that it generally exhibits multiple timescale behaviour [LTWW94]. These aspects can be modelled with subexponential distributions, such as Weibull and Lognormal.

We investigated the effect of different subexponential distributions on latency by changing the probability distribution function of the timed activities. For a fair comparison, we have chosen distribution parameters that allow the same mean value in all cases. The analysis results are shown in Figure 17(b). We can notice that different traffic patterns lead to different latency profiles. Similarly to the previous analysis, the latency assumes a constant value when the timeout **T** reaches a certain value (5 time units in this case), after which it is possible to appreciate how different traffic patterns affect the latency.



(a) Latency for different number of guards (b) Latency for different traffic patterns

Fig. 17. Latency for Different System Size and Different Traffic Patterns

Coverage in the case of failures Communication in the real-world can be subject to failures. Therefore, failure modes need to be accounted for when setting up the system model. Failure modes can pertain the value domain (e.g., wrong output), and/or the time domain (e.g., omission). In this section, we assess coverage in the case of omission failure of the messages sent and received in the EmergencyCall application. Figure 18 shows the coverage profiles for different probability *P(ECallFailure)* of failures of `EmergencyCall` communications. The analysis is performed with two commanders and two guards. The figure shows that variations in the failure probability significantly affect the coverage metric. The lower values shown by all the curves on the left side of the figure (that is, at initial values of **T**) are due to the fact that, given the short duration of **T**, the guards do not have enough time to send a response.

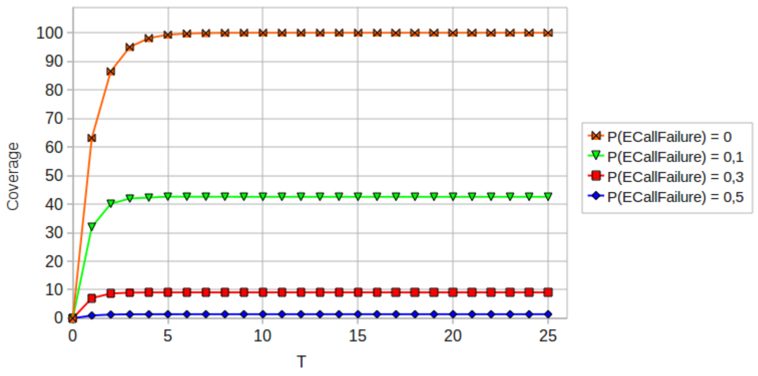


Fig. 18. Coverage for Different *P(ECallFailure)* of Failures of `EmergencyCall` Communications

4.3 On-line analysis

In the following, we focus on the Enablers interactions only, leaving out of scope the actions taken by DePer Enabler once it obtains the values observed at run-time from the Glimpse Enabler. We show a basic interaction between DePer and Glimpse Enablers, with reference to the Terrorist Alert Scenario [CON11c]. As summarised in Section 4.1, the scenario considers the interactions between police and patrolling civil guards. It is assumed that policemen and guards are both equipped with mobile devices, but they use different communication protocols. Hence, we intend to use the Connect infrastructure to enable the direct interoperation between a policeman and guards in the zone.

What we want to monitor at run-time is the latency between two states of the LTS. Specifically, we want to monitor latency of two transitions from the LTS shown in Figure 13.

The parameters under monitoring are the duration of the transitions executed by the NS requesting the communication, on which timeouts have been setup in the ConnectOr specification to limit the waiting periods. Therefore, having feedbacks on real executions is useful to improve the timeout calibration.

From Figure 13, the events to be monitored are the consumer transitions `selectArea` and `areaSelected`. The request messages sent by DePer Enabler to Glimpse are shown in Listing 1.2.

The Glimpse infrastructure, more specifically, the Manager component, receives the DePer requests and sets up the ComplexEventProcessor with the provided rule.

The events flowing in from Probes are structured on a `ConnectBaseEvent` object (see Figure 9), that provides all the necessary informations for an accurate pattern recognition.

According to the scenario, the peer that initiates the communication sends a broadcast message `selectArea` to selected peers (the Police control center or policemen) operating in a specified area of interest.

The event generated from the Probe instrumented into the peer software component is shown in Figure 19 and flows in into the Glimpse infrastructure stream of events.

When the selected peer replies (Police control center or policemen), another event is fired and sent on the Monitoring Bus, the `areaSelected` event.

The rule `computation time`, (lines 15-20) in Listing:1.2, used the timestamp contained into the two different events, matching: connectorID, sequenceID, ConnectorInstanceID, ConnectorInstanceExecutionID of each event. This rule, is able to calculate the latency (line 27) and provide it to the DePer.

Indeed, the rule `pending request` in the Listing1.2, (lines 40-42), computes the number of incoming requests into the Connector and provide it to DePer in order to evaluate coverage metric.

With those results, DePer is able now to evaluate the behaviour of the Connector and if this is not compliant to the expected values, it may contact the Syntesis Enabler requiring a new synthesis process.

Using a CEP able to infer more complex rules and patterns along with an event-driven architecture approach for dependability and performance analysis, may be beneficial in order to provide a cross-checking validation between runtime value and analysis expected value.

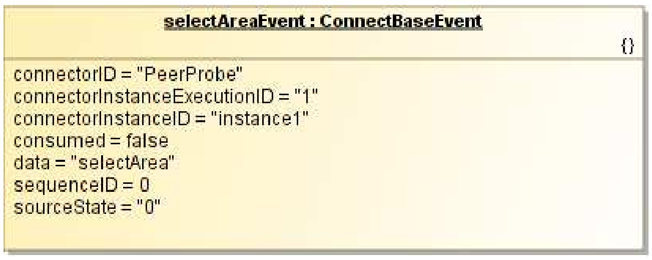

selectAreaEvent : ConnectBaseEvent
{}
connectorID = "PeerProbe"
connectorInstanceExecutionID = "1"
connectorInstanceID = "instance1"
consumed = false
data = "selectArea"
sequenceID = 0
sourceState = "0"

Fig. 19. The selectArea Event Sent from Peer Probe

```
<?xml version="1.0" encoding="UTF-8"?>
<ComplexEventRuleActionList xmlns="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule ./ComplexEventRule.xsd">
    <Insert RuleType="drools">
        <RuleName>transitionDurationRule</RuleName>
        <RuleBody>
    [...]
    rule "computation time"
    no-loop
    salience 999
    dialect "java"
    when
      $aEvent : SimpleEvent(this.data == "selectArea", this.getConsumed == false);
      $bEvent : SimpleEvent(this.data == "areaSelected",
                  this.getConsumed == false ,
                  this.getConnectorID == $aEvent.getConnectorID ,
                  this.getConnectorInstanceID == $aEvent.getConnectorInstanceID ,
                  this.getConnectorInstanceExecutionID == $aEvent.getConnectorInstanceExecutionID ,
                  this after $aEvent);
    then
      $aEvent.setConsumed(true);
      $bEvent.setConsumed(true);
      SatisfiedRequest sr = new SatisfiedRequest();
      sr.setIncoming($aEvent);
      sr.setOutcoming($bEvent);
      sr.setDuration(DroolsUtils.latency($aEvent.getTimestamp(),$bEvent.getTimestamp()));
      insert(sr);
      retract($aEvent);
      retract($bEvent);
      ResponseDispatcher.NotifyMe(drools.getRule().getName(),"DePer Module", sr.getDuration());
    end

    rule "pending request"
    no-loop
    salience 999
    dialect "java"
    when
      $total : Number()
      from accumulate($nEvent : SimpleEvent(data == "selectArea")
            from entry-point "DEFAULT",
            count($nEvent))
    then
      ResponseDispatcher.NotifyMe(drools.getRule().getName(),"DePer Module", "PENDING: " + $total);
    end
        </RuleBody>
    </Insert>
</ComplexEventRuleActionList>
```

Listing 1.2. Sample Request from Dependability&Performance Enabler

5 Related work

This work spans over automated model-based dependability analysis and event-based monitoring.

Research on definition and development of transformation-based verification and validation environments are being pursued since several years. Providing automatic/automated transformations methods from system specification languages to modelling languages amenable to perform dependability analysis has been recognized as an important support for improving the quality of systems. In addition, it favours the application of verification and validation techniques at industry level, where these methods are not widely used primarily due to the high level of abstractness of the mathematical modelling and analysis techniques. To provide some examples, the Viatra tool [CHM$^+$02] automatically checks consistency, completeness, and dependability requirements of systems designed using the Unified Modeling Language. The Genet tool [CCK09], based on the theory of regions [ER90], allows the derivation of a general Petri net from a state-based representation of a system. Our work addresses the transformation from the LTS formalism, as system specification language, to SAN, as dependability modelling language. Since there are some steps in common with the Genet tool and related theory, we partially reused available results from this previous study.

Similarly to Glimpse, also [PSB04] presents an extended event-based middleware with complex event processing capabilities on distributed systems. Similar to Glimpse this work adopts a publish/subscribe infrastructure but it is mainly focused on the definition of a complex-event specification language. The aim of Glimpse is to give a more general and flexible monitoring infrastructure for achieving a better interpretability with many possible heterogeneous systems.

Another monitoring architecture for distributed systems management is presented in [HAwM99]. Differently from Glimpse, this architecture employs a hierarchical and layered event filtering approach. The goal of the authors is to improve monitoring scalability and performance for large-scale distributed systems, minimizing the monitoring intrusiveness.

Many works focus on the definition of expressive complex event specification languages [MSS97,CM94,CM10]. Among these languages, GEM [MSS97] is a generalized and interpreted event monitoring language. It is rule-based (similar to other event-condition-action approaches) and also provides a tree-bases detection algorithm taking into account communication delay. Also the Snoop language [CM94] follows an event-condition-action approach supporting temporal and composite events specification but it is especially developed for active databases. A more recent formally defined specification language is TESLA [CM10]. It has a simple syntax and a semantics based on a first order temporal logic. The authors of [CM10] also provide an efficient event detection algorithm by translating TESLA rules into automata. Some existing open-source event processing engines are Drools Fusion [dro] and Esper [esp]. They can fully be embedded in existing Java architectures and provide efficient rule processing mechanisms. In our prototype we used Drools because ServiceMix offers it as business rule engine.

Preliminary studies that attempt combining off-line with on-line analysis have already appeared in the literature. A major area on which such approaches have been based is that of autonomic computing. Among such studies, in [MT06], an approach is proposed for autonomic systems, which combines analytic availability models and monitoring. The analytic model provides the behavioural abstraction of components/subsystems and of their interconnections and dependencies, while statistical inference is applied on the data from real time monitoring of those components and subsystems, to assess parameter values of the system availability model. Through on-line monitoring and estimation of system availability, adaptive on-line control of system availability can then be obtained. In [RP10], an approach is proposed to carry out run-time reliability estimation, based on a preliminary modelling phase followed by a refinement phase, where real operational data are used to overcome potential errors due to model simplifications. The model is based on Discrete Time Markov Chain, and a prototype version of the monitoring system has been implemented, that is initially trained with the reference model and the preliminary reliability estimation, and then uses operational data to compute the on-line reliability level.

Our approach aims at proposing powerful evaluation and monitoring supports able to cover, individually, a wide spectrum of needs inside the Connect framework (quantitative assessment of a variety of dependability and performance metrics on one side and generic monitoring infrastructure useful to a variety of Connect Enablers on the other side), and at exploiting their synergic usage to lead to higher accuracy of dependability and performance analysis results.

6 Conclusions and outlook

We have presented the directions currently pursued in the Connect project for the assessment of dependability and performance related properties of dynamic evolving systems. In particular, we focused on usage of stochastic model-based approaches, both at design time, for the early evaluation of the relevant non-functional requirements, and at run-time, for the continuous checking of system behaviour based on the actual data collected by the publish-subscribe monitoring infrastructure.

In line with the tutorial flavour of the chapter, we first provided basic introductory concepts and bibliography to model-based analysis of dependability attributes, relying on the SAN formalism and the Möbius tool. We also overviewed event-based monitoring and current research directions. We then described the solutions developed in the Connect project, which include the DePer modular infrastructure and the flexible Glimpse monitor, and discussed their interconnection to bring dependability and performance analysis to on-line stage.

The presented solutions mostly exploit advanced state-of-art results. The value brought forward by the Connect project stays mainly in their combined engineering and in the integration with the other Connect Enablers. The infrastructure resulting from the interfacing of DePer and Glimpse has been conceived with the highest flexibility and modularity in mind, so to allow for future further expansions, for example by including differing analysis engines, as we already show in [DGKM+10] for stochastic model checking.

At the time of writing, the implementation of the presented framework is still on-going and therefore our future work in the short term will of course involve the experimentation and refinement of the proposed approaches. More importantly, we intend to make the framework *model-driven*, so to make it more general and reusable. We are defining a property meta-model, a first release of which is available at http://labsewiki.isti.cnr.it/labse/tools/cpmm/public/main. The meta-model specifies non-functional properties, both qualitative and quantitative, to be evaluated. The idea then is that specific property models conforming to such meta-model can be used to automatically drive both DePer analysis, by providing in input the requested dependability and performance metrics, and probe instrumentation of the Connect monitoring Enabler.

Acknowledgements

The work reported in this chapter has been partially supported by the EU-funded Connect project (FP7231167) and stems from the collaborative effort of many colleagues from the Connect project, whom we would like to thank collectively, in the difficulty to list each single contribution. Among them, however, a special acknowledgement goes to the contributions from Marco Martinucci, Paolo Masci and Antonino Sabetta, formerly part of the CNR-ISTI Connect team, who have conceived with us the architectures of the presented DePer and Glimpse infrastructures, contributed to their development, and shared with us lot of discussions, insights, writing, and perspectives. We and the project owe much to them for the presented achievements.

References

[ABC84] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[AC87] M. Ajmone Marsan and G. Chiola. On Petri nets with deterministic and exponentially distributed firing times. In G. Rozenberg, editor, *Advances in Petri Nets 1987*, volume 266 of *LNCS*, pages 132–145. Springer-Verlag, 1987.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[Bal01] G. Balbo. Introduction to stochastic petri nets. In J.-P. Katoen, H. Brinksma, and H. Hermanns, editors, *Lectures on Formal Methods and Performance Analysis : First EEF/Euro Summer School on Trends in Computer Science Berg en Dal, The Netherlands, July 3-7, 2000, Revised Lectures*, volume 2090 of *Lecture Notes in Computer Science*, pages 84–155. Springer-Verlag, 2001.

[BCDG$^+$ar] A. Bertolino, A. Calabrò, F. Di Giandomenico, M. Martinucci, and P. Masci. Automated refinement of dependability analysis through monitoring in dynamically connected systems. In *Proc. IEEE International Symposium on Autonomous Decentralized Systems*, Tokyo, Japan, March 2011, to appear.

[BCG05] A. Bondavalli, S. Chiaradonna, and F. Di Giandomenico. Model-based evaluation as a support to the design of dependable systems. In Hassan B. Diab and Albert Y. Zomaya, editors, *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, pages 57–86. Wiley, 2005.

[BGD06] L. Baresi, C. Ghezzi, and E. Di Nitto. Toward open-world software: issues and challenges. *Computer*, 39(10), 2006.

[BGG04] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202, New York, NY, USA, 2004. ACM.

[BPTT98] A. Bobbio, A. Puliafito, M. Telek, and K. S. Trivedi. Recent developments in non-Markovian stochastic Petri nets. *Journal of Circuits, Systems and Computers*, 8(1):119–158, 1998.

[BS02] U.A. Buy and G. Singal. Toward efficient algorithms for generating compact petri nets from labeled transition systems. In *COMPSAC '02*, pages 717–722, Washington, DC, USA, 2002. IEEE Computer Society.

[BT98] A. Bobbio and M. Telek. Non-exponential stochastic Petri nets: an overview of methods and techniques. *Computer Systems Science and Engineering*, 13(6):339–351, 1998.

[CBC$^+$93] G. Ciardo, A. Blakemore, P.F. Chimento, J.K. Muppala, and K.S. Trivedi. Automated generation and analysis of markov reward models using stochastic reward nets. In C. Meyer and R.J. Plemmons, editors, *Linear Algebra, Markov Chains, and Queueing Models, IMA Volumes in Mathematics and its Applications, volume 48*, pages 145–191. Springer-Verlag, 1993.

[CCD$^+$01] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Mobius modeling tool. In *9th Int. Workshop on Petri Nets and Performance Models*, pages 241–250, Aachen, Germany, September 2001. IEEE Computer Society Press.

[CCK09] J. Carmona, J. Cortadella, and M. Kishinevsky. Genet: A tool for the synthesis and mining of petri nets. In *ACSD '09*, pages 181–185, Washington, DC, USA, 2009. IEEE Computer Society.

[CGL94] G. Ciardo, R. German, and C. Lindemann. A characterization of the stochastic process underlying a stochastic petri net. *IEEE Transactions on Software Engineering*, 20(7):506–515, 1994.

[CHM$^+$02] Gyorgy Csertan, Gabor Huszerl, Istvan Majzik, Zsigmond Pap, Andras Pataricza, Daniel Varro, and Dniel Varr. Viatra - visual automated transformations for formal verification and validation of uml models. In *17th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 267–270, 2002.

[CKLY98] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8):859–882, 1998.

[CKT94] H. Choi, V. G. Kulkarni, and K. S. Trivedi. Performance modeling using Markov regenerative stochastic Petri nets. *Performance Evaluation*, 20(1–3):339–356, 1994.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994.

[CM10] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *Proceedings of DEBS*, pages 50–61, 2010.

[CON10] CONNECT Consortium. Deliverable 5.1 – Conceptual Models for Assessment & Assurance of Dependability, Security and Privacy in the Eternal CONNECTed World, 2010.

[CON11a] CONNECT Consortium. Deliverable 4.2 – Further development of learning techniques, 2011.

[CON11b] CONNECT Consortium. Deliverable 5.2 – Design of Approaches for Dependability and Initial Prototypes, 2011.

[CON11c] CONNECT Consortium. Deliverable 6.1 – Experiment scenarios, prototypes and report Iteration 1, 2011.

[CON13] EU FP7 Project CONNECT (FP7–231167), 2009–2013.

[DGKM+10] F. Di Giandomenico, M. Kwiatkowska, M. Martinucci, P. Masci, and H Qu. Dependability analysis and verification for connected systems. In Tiziana Margaria and Bernhard Steffen, editors, *Proc. ISOLA 2010 - Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *LNCS*, pages 263 – 277. Springer, 2010.

[dro] Drools fusion: Complex event processor. http://www.jboss.org/drools/drools-fusion.html.

[ER90] A. Ehrenfeucht and G. Rozenberg. Partial (set) 2-structures. Part I: basic notions and the representation problem. *Acta Inf.*, 27(4):315–342, 1990.

[esp] Esper: Event stream and complex event processing for java. http://www.espertech.com/products/esper.php.

[Fid96] Colin Fidge. Fundamentals of Distributed System Observation. *IEEE Softw.*, 13(6):77–83, 1996.

[Gai86] Jason Gait. A Probe Effect in Concurrent Programs. *Softw., Pract. Exper.*, 16(3):225–233, 1986.

[Ger01] R. German. Non-Markovian analysis. In E. Brinksma, H. Hermanns, and J. P. Katoen, editors, *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *LNCS*, pages 156–182. Springer-Verlag, 2001.

[GG11] Nikolaos Georgantas and Paul Grace. The CONNECT architecture. In *11th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Connectors for Eternal Networked Software Systems (SFM-11:CONNECT). LNCS series*, 2011.

[Hav01] B. R. Haverkort. Markovian models for performance and dependability evaluation. In J.-P. Katoen, H. Brinksma, and H. Hermanns, editors, *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *LNCS*, pages 38–83. Springer-Verlag, 2001.

[HAwM99] Ehab Al-Shaer Hussein, Hussein Abdel-wahab, and Kurt Maly. HiFi: A New Monitoring Architecture for Distributed Systems Management. In *Proceedings of ICDCS*, pages 171–178, 1999.

[HBPU06] Hesham Hallal, Sergiy Boroday, Alexandre Petrenko, and Andreas Ulrich. A formal approach to property testing in causally consistent distributed traces. *Formal Asp. Comput.*, 18(1):63–83, 2006.

[How71] R. A. Howard. *Dynamic Probabilistic Systems: Markov Models*, volume 1 of *Decision and Control*. John Wiley and Sons, New York, 1971.

[iee90] *IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[jbi] Jbi: Java business integration. http://jcp.org/aboutJava/communityprocess/final/jsr208.

[JLSU87] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lap95] J.C. Laprie. Dependable computing and fault tolerance: concepts and terminology. In *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*, pages 2+, 1995.

[Lap08] J.C. Laprie. From dependability to resilience. In *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, 2008.

[LTWW94] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, 1994.

[MCC04] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[Mey92] John F. Meyer. Performability: A retrospective and some pointers to the future. *Perform. Eval.*, 14(3-4):139–156, 1992.

[MFT00] J. K. Muppala, R. M. Fricks, and K. S. Trivedi. Techniques for system dependability evaluation. In W. K. Grassmann, editor, *Computational Probability, Vol. 24 of Operations Research and Management Science*, pages 445–480. Kluwer Academic Publishers, The Netherlands, 2000.

[MK06] J. Magee and J. Kramer. *Concurrency: state models & Java programs.* John Wiley & Sons, New York, NY, USA, 2006.

[MM84] A. Movaghar and J. F. Meyer. Performability modelling with stochastic activity networks. In *1984 Real-Time Systems Symposium*, pages 215–224, Austin, TX, December 1984. IEEE Computer Society Press.

[MMDGar] P. Masci, M. Martinucci, and F. Di Giandomenico. Towards automated dependability analysis of dynamically connected systems. In *Proc. IEEE International Symposium on Autonomous Decentralized Systems*. IEEE, Tokyo, Japan, March 2011, to appear.

[Mol82] M. K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, 31(9):913–917, 1982.

[MSS94] Masoud Mansouri-Samani and Morris Sloman. Monitoring distributed systems. pages 303–347, 1994.

[MSS97] Masoud Mansouri-Samani and Morris Sloman. GEM: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[MT06] Kesari Mishra and Kishor S. Trivedi. Model based approach for autonomic availability management. In *ISAS 2006*, volume 4328 of *LNCS*, pages 1–16. Lecture Notes in Computer Science, 2006.

[NST04] David M. Nicol, William H. Sanders, and Kishor S. Trivedi. Model-based evaluation: from dependability to security. *IEEE Transactions on Dependable and Secure Computing*, 1:48–65, January-March 2004.

[PSB04] P.R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44 – 55, jan. 2004.

[ReS08] ReSIST Consortium. EU project ReSIST: Resilience for Survivability in IST. Deliverable D33: Resilience-explicit computing. Technical report, 2008. http://www.resist-noe.org/.

[RP10] K. S. Trivedi R. Pietrantuono, S. Russo. Online monitoring of software system reliability. In *Proc. EDCC '10 - 2010 European Dependable Computing Conference*, pages 209–218. IEEE Computer Society, 2010.

[rpc] RPC: Model for programming in a distributed computing environment. http://msdn.microsoft.com/en-us/library/ms691207(VS.85).aspx.

[RSB05] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: a library for automata learning and experimentation. In *FMICS '05*, pages 62–71, New York, NY, USA, 2005. ACM.

[rul] Ruleml: The rule markup initiative. http://ruleml.org.

[Sch95] Beth A. Schroeder. On-Line Monitoring: A Tutorial. *Computer*, 28(6):72–78, 1995.

[Sha93] B. P. Shah. Analytic solution of stochastic activity networks with exponential and deterministic activities. Master's thesis, University of Arizona, USA, 1993.

[SI10] Romina Spalazzese and Paola Inverardi. Mediating connector patterns for components interoperability. In *Proceedings of ECSA2010, LNCS 6285*, pages 335–343, 2010.

[SK88] Madalene Spezialetti and John P. Kearns. A General Approach to Recognizing Event Occurences in Distributed Computations. In *ICDCS*, pages 300–307, 1988.

[SM02] William H. Sanders and John F. Meyer. Stochastic Activity Networks: formal definitions and concepts. pages 315–343, 2002.

[Tri02] K. S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications.* John Wiley & Sons, New York, second edition, 2002.

[Zim99] D. Zimmer. On the semantics of complex events in active database management systems. In *Proceedings of the 15th International Conference on Data Engineering*, pages 392–, Washington, DC, USA, 1999. IEEE Computer Society.