A Distributed Monitoring System for Enhancing Security and Dependability at Architectural Level

Paola Inverardi and Leonardo Mostarda

Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
{inverard,mostarda}@di.univaq.it

Abstract. In this work we present the DESERT tool that allows the automatic generation of distributed monitoring systems for enhancing security and dependability of a component-based application at architectural level. The DESERT language permits to specify both the components interfaces and interaction properties in term of correct components communications. DESERT uses these specifications to generate one filter for each component. Each filter locally detects when its component communications violate the property and can undertake a set of reaction policies. DESERT allows the definition of different reaction policies to enhance system security and dependability. DESERT has been used to monitor applications running on both mobile and wired infrastructures.

1 Introduction

In this work we present the DESERT tool that allows the automatic generation of distributed monitoring systems for enhancing security and dependability of a component-based application at architectural level.

In our system model we assume a set of black-box components that interact with each other by exchanging messages. A message encodes information about the type of communication, i.e. a request or a reception, the kind of service and its parameters and the (returned) data. This architectural level model has shown to be flexible enough to model several types of distributed systems and communication patterns. For instance, in [1] we model mobile sensors applications. In this case components are sensor devices and communication is achieved by means of send and receive asynchronous invocations. In [2] we have modeled CORBA middleware based applications. In this case we have CORBA components that communicate by means of different types of service invocations (i.e., asynchronous, synchronous and deferred synchronous invocations).

At the architectural level we define an *anomalous component* as one that interacts with the remaining components in order to subvert the 'correct' system behavior. Anomalous component interactions can have different origins and their detection constitutes the basis to provide different functionalities of the system. For instance, anomalous interactions can originate by a malicious component that exploits other components vulnerabilities (see [3] for an extended survey).

R. de Lemos et al. (Eds.): Architecting Dependable Systems IV, LNCS 4615, pp. 210–236, 2007.

In this case the component detection constitutes the basis to build an Intrusion Detection System (IDS)[4,5,6,7,8] to enhance the system security functionality. In the field of dependable computing, anomalous components interactions can be a consequence of architectural mismatch [9] and/or of components faults that lead to system failure. In this case the detection mechanism can be the basis for error detection and system recovery [10]. In the field of performance evaluation, anomalies on components interactions can be a consequence of degraded response time, thus detection mechanisms can be used to provide reconfiguration mechanisms.

Today's monitoring tools [11] are a viable solution to detect anomalous components interactions. They are tools that: (i) gather information about applications, (ii) interpret the gathered information; (iii) respond appropriately (i.e. they can undertake different reaction policies). A monitoring system can be characterized by the functionalities it provides. Modern monitoring tools are used to increase security, dependability and performance (see Section 2 for a detailed survey). Moreover they can be part of the target system therefore they can add new system behaviors.

In this work we present the DESERT tool [12,2,1,13] that allows the generation of distributed monitoring systems for component based applications.

The monitoring definition is obtained starting from a DESERT program written in the DESERT definition language. The DESERT program contains both an interfaces descriptions part and a global automaton one (Figure 1 part 2). The interfaces descriptions part is obtained by means of an interface description language that permits to describe each component of the system in terms of its name and the services that it requires/provides. The global automaton part can contain different state machines (that in the following will be referred to as interaction properties) that are described by means of a DESERT state machine definition language. A state machine describes the correct messages exchange among components (i.e. the correct component communications). As we are going to see in Section 3.2, the state machines can model complex communication patterns but they are not suitable to describe temporal properties (i.e, the DESERT approach is appropriate for applications without timeliness requirements).

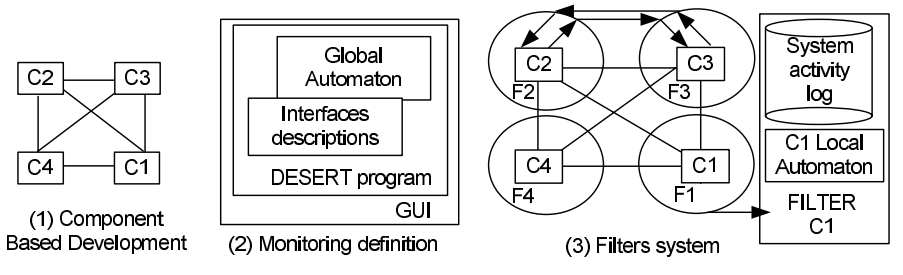

Fig. 1. DESERT phases

The DESERT program is a means to define a logically centralized monitoring tool. This monitoring tool: (i) gathers all messages exchanged among components; (ii) checks the messages consistency with respect to the property defined by the global automaton; (iii) in case of mismatches undertakes a set of reaction policies. Different reaction policies can be defined to provide different functionalities of the system. For instance, when a component misbehaves for malicious purposes isolation of this component can be the correct reaction to enhance the system security. In case the component performs anomalous interactions as a consequence of a fault, recovery reaction can improve the system dependability.

The implementation of the logically centralized monitoring system can pose problems of security, reliability and performance. Furthermore, already existing legacy distributed systems could not allow the addition of a new component which monitors the information flow in a centralized way.

To overcome these problems DESERT automatically decomposes the global automaton in a set of local automata that are assigned one for each component. A local automaton constitutes the basis to build a filter that is interposed between its component and the environment (see Figure 1 part 3). The filter captures all incoming/outgoing component messages and uses the local automaton to locally detect violation of the policy expressed by the global automaton. In other words, the filters taken as a whole system constitute a monitoring system that is "equivalent"(see [12,13] for a formal description) to the centralized one.

The DESERT tool implements both a front-end and a set of back-ends. The former, starting from the DESERT program and a component name (e.g. $C2$), produces the platform independent specification of the $C2$ local automaton. The latter translates the $C2$ local automaton specification in a specific filter implementation. For instance for distributed applications where the components communicate by using the CORBA middleware we have implemented a CORBA back-end. This back-end automatically produces a new CORBA component (the filter) that is interposed between the component communications and the environment.

We point out that the novelty of this work is the description of the DESERT definition language and the overview of the DESERT distribution basic steps. In fact in [12,13] we primarily focus on the proofs of correctness and completeness of the distribution process while in [2] and in [1] we only sketch how the approach can be suitable for different areas (i.e, enforcement and security respectively).

The paper is organized as follows. In the next section we present an overview of the monitoring system technology and we summarize the contribution of our architectural level monitoring technology. Section 3 introduces our monitoring definition language, in particular, Section 3.1 describes our interface description language and Section 3.2 the state machine specification language. Section 4 shows how interfaces and state machine can be used to define a logically centralized monitoring system. Section 5 summarizes the different reaction policies that can be set to generate different monitoring systems for different areas of applications. Section 6 sketches the generation of the distributed implementation of the logically centralized monitoring system. Section 7 describes different

case studies in which we have applied our monitoring approach. We show how our reaction policies can be tuned in order to output different monitoring systems used for different functionalities of the system. Finally, Section 8 provides conclusive remarks and future work.

2 Monitoring Tools at Glance: Concepts and Terminology

Monitoring systems have been around since the 1960s. Originally they were conceived with a centralized structure and used to debug centralized systems. Today's monitoring systems monitor distributed applications and themselves have a distributed architecture. Generally speaking, a monitoring system can be defined as tools that: (i) gathers system information; (ii) interprets the gathered information; (iii) after interpretation can undertake a set of reaction policies. Monitoring provides a solution for areas of growing concerns: lack of dependability, security and performance enhancement and tools to support distributed applications.

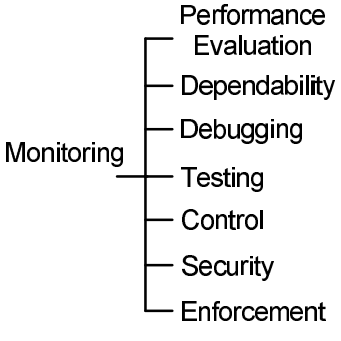

Fig. 2. Monitoring uses

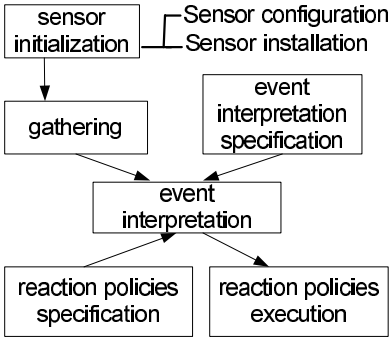

Fig. 3. Monitoring activities

In Figure 2 we show the primary uses of monitoring tools. Dependability includes cases in which interpretation involves detection of system errors and reaction policies are undertaken to enhance the system fault tolerance [11]. Performance evaluation includes detection of the system response time degradation. In this case reactions can include system reconfiguration, dynamic program tuning and on-line steering. Security involves the interpretation of information in order to detect attacks. In this case interpretation of information can be performed using a set of predefined intrusion signatures and/or the correct system behavior definition. The former are referred to as misuse detection systems [8,14,15,16] while the latter are referred to as anomaly detection systems [17,7,18,5,19,12,4]. Debugging and testing employs monitoring techniques to extract data values from an application being tested. Control includes cases in which the monitoring implements part of system functional requirements. Finally, enforcement [20] is the case in which the monitor interprets information in order to ensure desired system behaviors.

Despite the various monitoring system utilizations their definition involves the following standard activities: (i) sensor initialization; (ii) gathering; (iii) event interpretation specification; (iv) event interpretation; (v) reaction policies specification; (vi) reaction policies execution (see Figure 3). Each activity can be performed either by the user or by the monitoring system itself.

In order to describe the above activities in the following we introduce some basic concepts common to all monitoring systems.

Information arrives to the monitoring tools in the form of *events*. Events can regard the system states, interactions among system parts and system activities. We point out that events can be related to different layers of abstraction, i.e, hardware-level, process level and application level. A *sensor* is the monitoring element that locally gathers events. Sensor automatically sends events when they occur or it is the monitoring system itself that can request them.

Sensor initialization (see Figure 3) includes configuration and installation. Configuration is carried out by deciding what events a sensor will gather and the definition of additional sensor capabilities, e.g., local conditions checking. Sensor installation is carried out by placing the sensor code at the correct location. This is usually performed through code instrumentation[1] or conceiving the sensor as an external observer that sniffs all communications among system parts.

Gathering is the activity in which sensors collect events and forward them to the monitoring system. The gathering can be either off-line or in-line. The former is characterized by the fact that the gathering code uses the resources of the target system. The latter is characterized by the fact that the gathering code uses resources separated from the target system ones.

Event interpretation is the heart of the monitoring system, where the monitoring system interprets events. Event interpretation is achieved using the event interpretation specification that is usually defined by means of errors conditions description and/or description of correct system behaviors. Event interpretation can be categorized either as synchronous or asynchronous. Asynchronous is when events are interpreted after system execution. Synchronous when the system is suspended until event interpretation.

Reaction policies execution can take place after event interpretation and its implementation must comply with the reaction policies specification. Reaction policies specification are a consequence of the monitoring system uses. Earlier monitoring systems were only involved in logging and tracing reactions. Nowadays monitoring systems embed complex reactive utilities that can be undertaken after event interpretation.

Generally speaking, reaction policies can be either non-intrusive or intrusive.

Non-intrusive reaction policies do not affect the program behavior except for execution speed and program size. Logging and tracing are examples of these monitoring system reactions. Logging can be performed to record violations of the correct system behavior. Tracing can be viewed as a high-level logging utility

[1] Instrumentation requires code access and can be manually performed by the user or automatically by code analysis.

that records all sequences of interactions resulting in the anomalous system behavior.

Intrusive reaction policies affect, to some degree, the state, the configuration and/or the execution of the system (see [10] for an extended survey). For instances intrusive reaction policies are: (i) termination; (ii) shunning; (iii) reconfiguration; (iv) rollback; (v) rollforward. Termination refers to the monitoring system ability to terminate part (or the whole) system execution. Shunning is the case in which the monitoring system denies the traffic generated by a specific source or a set of sources. Reconfiguration can physically alter the location or functionality of network or system elements. For instance, in the field of dependability, reconfiguration can be useful to switch in spare components or reassign tasks among non-failed components. In the case of security reconfiguration can be used to isolate the attackers. Rollback brings the system back to a previous saved state. Rollforward brings the system, in a new 'safe' state.

In this paper we focus on monitoring systems that use formal specifications for event interpretation (in the following referred to as specification-based monitoring systems). Different names, that depend on the monitoring system uses, can refer to a specification-based monitoring system. In the field of security they are referred to as specification-based and anomaly-based IDSs [5,12,4]. In the field of dependability and correctness checking they can be referred to as software-fault monitors [11] and enforcement mechanisms [20],respectively. Despite these various uses, a specification-based monitoring system is usually interpreted as a tool that takes an application and a specification of software properties and checks that the execution meets the properties, i.e., that the properties hold for the given execution. A property can describe the correct communication among system parts, the correct system states and the correct system activities. A specification language is a language that is used to describe properties.

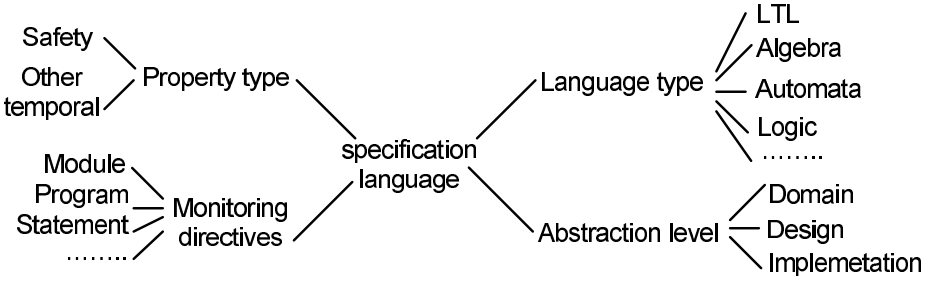

Fig. 4. Specification language categorization

In Figure 4 we show the elements that can characterize a specification language: (i) the language type; (ii) the abstraction level; (iii) property type; (iv) the monitoring directives. The type of language used to define a property can be based on algebra, automata, logic and so on. The abstraction level refers to the support that the language provides in order to specify the property and the knowledge about the domain, the design and the implementation of the system.

For instance, a language that provides support to specify properties for CORBA middleware would be classified domain-based. A language that allows the specification of properties in implementation independent fashion would be design based. Finally, properties that involve statements and variables of a system have to be defined by means of implementation-dependent language.

Two types of properties can be specified: safety and temporal ones. A safety property expresses that something bad never occurs. A temporal property includes progress and bounded liveness [11]. Monitoring directives specify that a property can be evaluated at different levels,i.e, program , module, statement and so on.

The DESERT tool allows the automatic generation of distributed specification-based monitoring systems for enhancing security and dependability in distributed black-box components applications. The black-box nature of the components imposes that our events can be observable messages exchanged among them. The DESERT language allows the definition of both the system model and the correct system behavior. The system model is provided by means of an interface description language. This language permits to describe each component of the system in terms of its name and the services that it requires/provides. The correct system behavior is provided by means of a global automaton that describes the correct messages exchange among components (i.e. interaction properties). As described in [20] automata can describe safety and bounded liveness properties.

A monitoring system, based on the global automaton, gathers all messages exchanged among components and verifies that such messages do not violate the policies expressed by the global automaton. Our monitoring system is off-line since it does not use the resources of the target system (in particular it is implemented as an external observer). Moreover, it is synchronous since messages are delivered only after interpretation.

Architectural level monitoring permits to obtain implementation-independent description, however DESERT provides the basic mechanisms to add design and implementation details into the properties descriptions. Moreover, as we describe in Section 5 DESERT allows the definition of reaction policies tailored for security and dependability purposes.

3 The DESERT Definition Language

In this section we describe the DESERT definition language that permits to specify both the system model and the global automaton. The system model specifies the components interfaces descriptions.

3.1 Components Interfaces Descriptions

This part is composed of a set of *component interface declarations*. A component interface declaration is composed of: (i) the component name; (ii) a list of services description.

A service description is either of the form $!serviceName(Parameters).returnType$ or $?serviceName1(Parameters1).returnType1$. The former declares that

the component is a client of the service $serviceName$ having the formal parameters $Parameters$ and the returned value type $returnType$. The latter declares that the component provides the service $serviceName1$ to the environment.

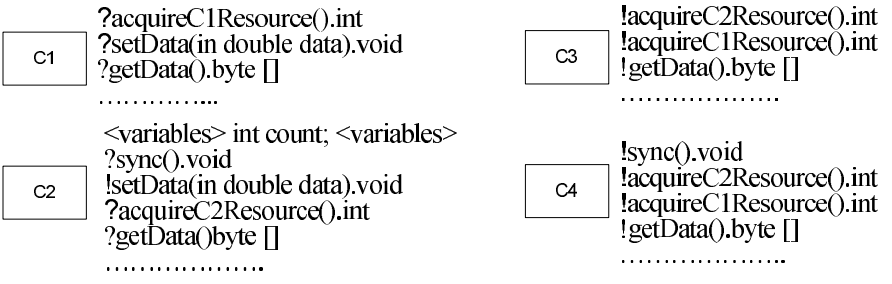

Fig. 5. Components interfaces

The $Parameters$ is constituted by a sequence of the type $T_1 D_1\ X_1\ ,\ T_2\ D_2\ X_2, \ldots, T_n\ D_n\ X_n$, with $n \geq 0$ ($n = 0$ means an empty sequence of parameters). T_i is a label that can take one of the following values: *in* and *out*. *in* specifies that the parameter X_i is an input service parameter (i.e., it must be provided to the service). *out* specifies that X_i is an output service parameter (i.e., it contains an output after the service execution). D_i is the X_i domain and defines the set of values that the parameter can take.

Suppose that a component is a client (server) of the service $!serviceName\ (Parameters)\ .returnType\ (?serviceName1(\ Parameters1).returnType1)$. The label $returnType$ ($returnType1$) defines the set of values that the component client (server) can receive (send) back after the $serviceName$ ($serviceName1$) synchronous service invocation. We point out that the type of service invocation (i.e. synchronous and asynchronous) is specified by means of a label that is prefixed to the service declaration. Optionally, a user can add variables declarations in the components interfaces descriptions.

In Figure 5 we sketch part of the case study that refers to a cooling water pipe distributed industrial application (see [2] for a detailed description). It concerns the monitoring of messages exchanged among a set of components that collect and correlate data on the amount of water that flows in different water pipes. This water is used to cool industrial machinery. The water pipes are monitored by the server components $C1$ and $C2$ that interact with *Programmable Logic Controllers* (PLCs) in order to obtain the data related to each water flow. The clients $C3$ and $C4$ request services on the servers in order to write/read the water flow data.

$C1$ can receive incoming requests of the $?acquireC1Resource()\ .int$ and $?set\ Data(in\ double\ data).void$ services in order to allow the exclusive access to the Area 1 data and to manually set its local data, respectively. $C2$ can receive incoming requests of the $?acquireC2Resource()\ .int$ and $?sync().void$ services in order to allow the exclusive access to the Area 2 data and to accept a synchronization request, respectively. Moreover, $C2$ can require the $!setData(in\ double$

data) *.void* service to $C1$ in order to send its data to $C1$. In particular $C2$ sends its data after the reception of a *sync*() request. $C3$ and $C4$ are client of the services exposed by the component $C1$ and $C2$. Notice that we relate the variable *count* to the component $C2$. We point out that this variable is used in the state machine definition, i.e., it is part of the monitoring system definition.

The goal of the overall application is to ensure consistency on the water flows data that are used for billing purposes. To this extent we define a state machine.

3.2 The Global Automaton

After the components interfaces descriptions, the DESERT program has to describe a state machine that defines the correct components communications.

In the following we introduce some notation useful to describe our automata. Let us suppose that the C interface declaration defines a service of the type $!serviceName(T_1\ D_1\ X_1, \ldots, T_n\ D_n\ X_n).returnType$ (i.e., C is a client of the *serviceName* service) and S exports the service $?serviceName(T_1\ D_1\ X_1, \ldots, T_n\ D_n\ X_n).returnType$ (i.e., S is a server of the serviceName service). The notation $C\ c, x;$ is used to declare two instances (i.e.,c and x) of the component C. The symbol '*' denotes an unknown type of component (see case study of Section 7 for examples).

In the following we describe the events that we monitor at architectural level. Let us consider the declarations $C\ c;$ and $S\ s;$:

- $!serviceName\ (X_1, \ldots, X_n)\_c\_s$ defines an event that can be observed when c performs the $serviceName\ (X_1, \ldots, X_n)$ service invocation on s. It is worth noticing that when the invocation is executed the parameter $X_1 \ldots X_n$ are suitable instantiated by c.
- $?serviceName\ (X_1, \ldots, X_n)\_c\_s$ defines an event that can be observed when s performs the $serviceName\ (X_1, \ldots, X_n)$ service-receive invocation on s.
- If the serviceName service is synchronous we can define the symmetrical invocation $!serviceName\_s\_c$ and $?serviceName\_s\_c$, i.e., the answer to the serviceName service observed on the server and client side, respectively.

The receive invocation $?serviceName(X_1, \ldots, X_n)\_c\_s$ has associated the default variables $?serviceName.X_i$, with $1 \leq i \leq n$, each of them contains the X_i parameter value when the invocation is performed at the s server side. The send invocation $!serviceName(X_1, \ldots, X_n)\_c\_s$ has associated the default variables $!serviceName.X_i$, with $1 \leq i \leq n$, each of them contains the X_i parameter value when the invocation is performed at the c server side. If the serviceName service is synchronous then the default variable !serviceName (?serviceName) contains the serviceName returned value sent (received) by s (c). We point out that all invocations that include the symbol $*$ define the same variables (see [13] for details).

Let p be one of the above invocations. Each transition of the global automaton can be labeled with a piece of information of the form $p[P]\{code\}$. The label

[P] is a predicate that the invocation p must verify, the field {code} a piece of code executed when the transition is performed. In particular, as we are going to see in section 7, a predicate is a means to avoid attacks/faults caused by invocations with malformed formats (e.g., sql injection and buffer overflow) while the code can be executed to perform more complex checks based on the component variables.

A state machine describes the system traces that an external observer should see in the case of correct components interactions. In particular, these traces are related only to the services invocations defined inside the global automaton alphabet.

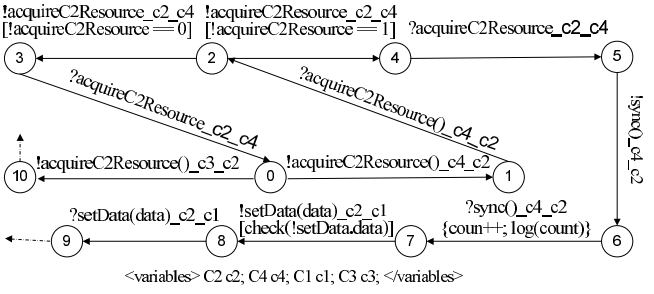

Fig. 6. Global automaton

Fig. 7. Simple property

In Figure 6 we show a portion of the state machine declaration that is related to our case study [13,2]. In this case we have the component instance $c1$ of the type $C1$, $c2$ of the type $C2$, $c3$ of the type $C3$ and $c4$ of the type $C4$. We can observe that in the state 0 both clients $c3$ and $c4$ have the possibility to perform an $acquireC2Resource()$ service invocation on the server $c2$. In the case that $c4$ performs the invocation the state machine moves from state 0 to 1. We point out that this invocation is observed at the $c4$ client side. In state 1 this invocation is observed at the $c2$ server and the global automaton state can be changed from 1 to 2. From the state 2 two possible transitions exit. One models the $c2$ $acquireC2Resource$ service response when its returned value is equal to 0. This is expressed by the condition $!acquireC2Resource == 0$ (e.g., this is the case of server busy). The other one models the $c2$ $acquireC2Resource$ service response when its returned value is equal to 1. If the latter is applied then the state machine moves to state 4, where the $c2$ services can be provided to the client $c4$. Notice that each time the service $sync()$ is received by $c2$ the related variable $count$ is updated. Moreover, the state machine does not include the description of the service $!getData().byte[]$ and other components services (see Figure 5). In Figure 7 we show a more simple policy in which concurrent services invocations are allowed. This is represented by multiple transitions that enter and exit from the same state.

We remark that different global automata can define different security policies and can be used to concurrently monitor the components. However, in the remaining we focus on a single automaton since all results can be easily extended to multiple concurrent automata.

In the following we sketch strengths and weaknesses of the DESERT definition language.

The DESERT language flexibility allows the definition of monitoring systems in different contexts. For instance, in the context of CUSPIS project [21] user services are implemented by a client (e.g. c) that performs invocations on finite set of servers $s_1, \ldots, s_n$. Servers can interact with each other and with further components. In this case a global system view is needed to ensure the correctness of interactions scattered over several components. We have defined a *server side* policy that characterizes client sessions in terms of both servers received invocations (e.g. $?serviceName(Parameters)\_c\_s_i$) and servers performed invocations (e.g. $!serviceName(Parameters)\_s_i\_s_j$). In the case of wireless sensor networks (see [1] for details), the automaton can contain only invocations of the form $!serviceName(listOfParameters)\_c\_s$ (i.e., a client side policy). This is the case in which mobile devices (e.g. sensors) send asynchronous service invocations to unknown servers, therefore, we have to write client side policies.

If the state machine describes all possible services invocations we may incur in the usual state explosion problem. However, the crucial property commonly only interests a subset of the global system behavior. Moreover, it is worth notice that the global automaton does not allow the definition of temporal constraints so that DESERT is a tool unsuitable to model interactions with timeliness requirements.

4 A Logically Centralized Monitoring System

A DESERT user defines the global automaton at the level of system integration, i.e., she defines the correct components communications by having a global system view. From the global automaton perspective it is easy to derive a logically centralized monitoring system. The logically centralized monitoring system does not have a concrete counterpart but its knowledge makes it easy to understand the filters system generation (i.e., the distributed monitoring system implementation) that is hidden by the DESERT tool. Therefore for the sake of simplicity we will describe the monitoring use and the reaction policies referring to a centralized approach. In Section 6 we show how the distributed implementation of the centralized approach is automatically generated.

In the following we use the case study of Section 3 to describe all actions that the centralized monitoring system can undertake, i.e., (i) send invocation acceptance; (ii) buffering action; (iii) receive invocation acceptance; (iv) forwarding action;(v) anomaly detection action.

The logically centralized monitoring system has a buffer used to store the components invocations it captures. Suppose that the monitoring system picks the invocation $!acquireC2Resource\ ()\_c4\_c2$ up from its local buffer and the global

automaton of Figure 6 is in state 0. Then the monitoring system can 'accept' this invocation since there is a 0-exiting transition labeled with it and there is not a predicate to be satisfied. Accept means that the monitoring system buffers the invocation ?*acquireC*2*Resource*() _*c*4_*c*2[2] and changes the automaton state to 1 (in the following this monitoring system activity will be referred to as *send invocation acceptance*). In state 1 there is the possibility that the monitoring system picks an invocation !*acquireC*2*Resource*()_*c*3_*c*2 up from the buffer (i.e., the client *c*3 requires the access to the same *c*2 resources). This invocation cannot be accepted since there is not a 1-exiting transition labeled with it. Therefore, the monitoring system checks the existence of a state reachable from 1 where the following conditions hold: (i) there is an exiting transition t labeled with the invocation !*acquireC*2*Resource*()_*c*3_*c*2; (ii) the predicate related to the transition t is satisfied (in our case 0). In this case the monitoring system puts the invocation back to process it later (*buffering action*). By continuing our example, the monitoring system can pick the invocation ?*acquireC*2*Resource*()_*c*4_*c*2 up from the buffer and accepts it by means of the 1-exiting transition. In this case the monitoring system forwards the *acquireC*2*Resource* invocation to the server *c*2 and changes the automaton state to 2 (*receive invocation acceptance*). Since the service *acquireC*2*Resource* is synchronous the monitoring system has to wait for the result ?*acquireC*2*Resource* to put it in the buffer. In any automaton state invocations that are not described in the global automaton are forwarded without any check, e.g. all getData() service invocations are forwarded, (*forwarding action*). An anomalous component invocation is detected when: (i) the invocation cannot be accepted in the current automaton state (e.g. q) and in any state reachable from q; (ii) the invocation was buffered and not consumed after a finite amount of time[3]; (iii) the invocation is related to services not present in the interfaces definitions (*anomaly detection action*).

After an anomaly detection action our centralized monitoring system can undertake a reaction policy. In the next section we describe all different reaction policies that can be set by means of the DESERT tool. Different reaction policies allow the generation of monitoring tools for different areas, e.g. security, dependability and enforcement (see Section 2).

5 The DESERT Reaction Policies and the Application Areas

In Figure 8 we show all reaction policies that can be set in order to generate monitoring systems for different uses.

[2] The buffering is needed since it is the global automaton that can (or cannot) define when the related receive invocation must be delivered. This is strictly related to wether or not the receive invocation is defined inside the global automaton alphabet.

[3] The amount of time must be chosen by the user. In particular it can be assigned at 'the global automaton level' (i.e, all invocations must be consumed after a fixed amount of time) and/or to each single invocation.

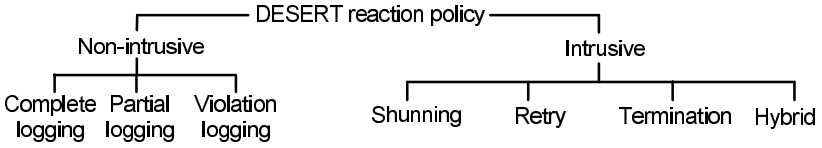

Fig. 8. DESERT reaction policies

The *logging* reaction policy permits to obtain a monitoring system that does not affect the system behavior. Different levels of logging can be set up. The *complete logging* allows the logging of all messages exchanged among components[4]. The *partial logging* permits to log all messages exchanged among components that belong to the global automaton alphabet. The *violation logging* permits to log all messages exchanged among components that violate the property expressed by the global automaton. For instance, the logging reaction policies can support off-line and on-line testing. In the former case complete logs can be produced and analyzed in order to detect traces that violate the case tests. In the latter case the monitoring system can produce a violation log in which all traces violating the global automaton policy are recorded. In other words we can test the run time system traces with respect to the global automaton ones.

The *Shunning* reaction policy can be partial and complete. In the partial shunning the monitoring system logs the information details of any anomalous message m that violates the property. When m has been logged, the monitoring discards m and does not deliver it to the receiver. In the complete shunning the monitoring system registers the sender of the message m and denies all future messages send by it, i.e., the monitoring system isolates the component that performed the violation. The shunning reaction policies can be used in the field of security [1] in order to isolate the component that attacks the system. However, it can generate components anomalous behaviors as a consequence of no returned value to them, therefore shunning cannot be applied in order to enhance the system fault tolerance.

In the *retry* reaction policy the monitoring system discards and logs any message m that mismatches the correct behavior. After m has been discarded the monitoring returns an *error* to the component that has sent m. An *error* is a value that a component recognizes either as an exception or a failure condition. The retry reaction policy can be a means to improve the system fault tolerance. The error detection mechanism is provided by our monitoring tool that detects a component misbehavior. Moreover, the error value returned can be a simple recovery mechanism to let the component try again. It is worth noticing that while the shunning policy can be always applied without having any type component knowledge the retry reaction requires that the component explicitly declares a handled returned error value.

[4] We log for each invocation: time, service name, parameters or returned value, sender and receiver.

Termination refers to the monitoring system ability to terminate the components generating the anomalous behavior. This reaction can be followed by a reinitialization phase in which components can be configured and restarted. Notice that in this case the monitoring system must have a mechanism to stop and restart a component execution.

Hybrid reaction policies include the case in which different reaction policies are assigned to different invocations. For instance, the shunning policy can be associated to each invocation related to services that are critical for the system security. The retry policy can be used for invocations that can be performed after the correct system login, i.e., this policy can be used to recover authorized components.

We can observe that both retry and shunning policies produce a monitoring system that acts like an enforcement mechanism (EM). As defined in [20] enforcement mechanisms compare a formal specification with the system steps. When there is a violation of the formal specification an EM can either terminate the system execution or replace an unacceptable execution step with an acceptable one.

6 The Distribution Process

The implementation of the logically centralized monitoring system is not practical in systems composed by a large number of distributed components and of interactions properties, where the parsing efficiency, scalability and failure can become relevant issues. Moreover, already existing legacy distributed systems could not allow the addition of a new component which monitors the information flow in a centralized way. The DESERT solution is an algorithm to automatically distribute the logically centralized monitoring system (i.e., the 'centralized' DESERT program) on each component of the system. It performs this generation by decomposing the global automaton in a set of local automata that are assigned one for each component of the system. A local automaton constitutes the basis to build a filter that locally monitors its component communications. The set of filters taken as a whole system constitutes a distributed monitoring system "equivalent" to the central one.

In Figure 9 we show the basic components of the DESERT tool that allow the generation of the monitoring system implementation. A graphical user interface allows the description of both components interfaces and state machines. These descriptions are stored in XML format.

The front end is composed of the following components: the local automaton generator, the parser and the semantic controller. The local automaton generator component takes in input the XML file and a component name (e.g. $C2$). It forwards the XML file to the parser and semantic component that performs all syntax and semantic checks, respectively. In the case that there are not errors the local automaton generator generates the XML specification of the $C2$ local automaton. We remark that this process can be performed locally on the host where $C2$ resides on.

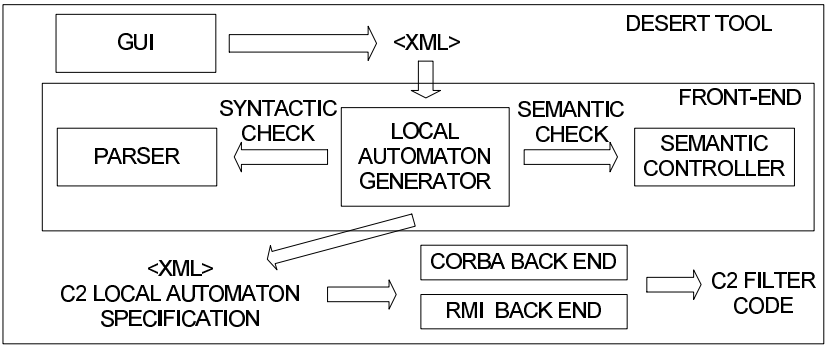

Fig. 9. The DESERT C2 filter generation

The $C2$ local automaton (in the following denoted with A_{C2}) is part of the global automaton enriched with transitions labeled with synchronization messages (see Figure 10) that in the following will be referred to as dependencies messages. These transitions are applied by the filter $C2$ to send (receive) information to (from) other filters. Dependencies allow the simulation of the centralized monitoring system. In [12,13] we show all formal proofs, we discuss the overhead introduced by such synchronization messages and we show how it does not constitute a problem since they are small in size (i.e., they are integer). Moreover, in [13] we also point out that both the time required to exchange the synchronization messages and their parsing can slow the application response time (i.e, the DESERT tool enhance security and dependability issues at the expense of the system response time).

The $C2$ local automaton specification is platform independent and may be translated into different filter implementations. For instance for distributed applications where the components communicate by using the CORBA middleware we have implemented a CORBA back-end. This back-end automatically produces a new CORBA component (the filter) that is interposed between the component communications and the environment. The filter exposes all services that the component requires and provides to the environment (see Figure 10). The entire process of filter generation is polynomial on the global automaton size. We point out that the filters work at the middleware level therefore we do not require components source code.

In the following we sketch the local automata generation and we discuss the filters actions. For the sake of presentation we introduce some notation. We use the notation $q' = \delta(q, p)$ to denote a global automaton rule that exits from the state q, enters in q' and is labeled with the invocation p. We denote with $q' = \delta_C(q, p)$ the same rule projected on the A_C local automaton. We denote with $k(q, q')(p)$ an integer that uniquely identifies the global automaton rule $q' = \delta(q, p)$. We use $P(q, q')(p)$ $(C(q, q')(p))$ to denote the predicate (code) related to the rule $q' = \delta(q, p)$.

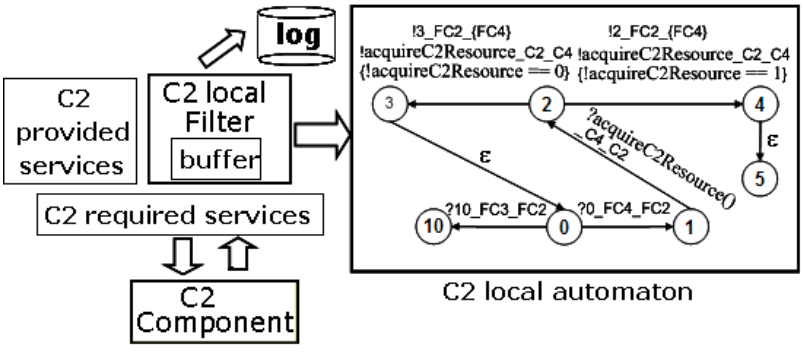

Fig. 10. The behavior of the run-time filters

Local automata are generated by performing two phases: local automata generation and dependencies generation.

In the local automata generation phase each rule of the global automaton is projected on a local automaton. Suppose that the global automaton defines the rule $q' = \delta(q, p)$ and p is an invocation locally observed on the component C. Then this phase adds the rule $q' = \delta(q, p)$, the predicate $P(q, q')(p)$ and the code $C(q, q')(p)$ to the A_C local automaton. In other words, looking at the global automaton, interactions that happen locally on a component C are projected on A_C. For instance in Figure 10 we show the A_{C2} local automaton related to our case study. It is worth noticing that it contains only rules labeled with invocations locally observed on the component $C2$.

The local automata obtained after this phase are not sufficient to realize the correct monitoring. A local automaton A_C can be constituted by disconnected sub-automata. The filter FC cannot be able to choose the right sub-automaton. Moreover, given a sub-automaton it cannot establish the next one. Our solution is to enrich local automaton with dependencies information and to link the sub-automata with ε-moves.

A dependency can be of the form $!k(q_1, q_2)(p1)\_FC\_\{FC1, \ldots, FCn\}$ and $?k(q_3, q_4)(p3)\_FCi\_FCj$. The former (outgoing dependency) is always related to the A_C rule $q_2 = \delta_C(q_1, p1)$ and is used by the filter FC to inform the filters $FC1, \ldots, FCn$ that it has applied such local rule. The latter (incoming dependency) is used by the filter FCj to receive the integer $k(q_3, q_4)(p3)$ sent by the filter FCi.

The dependencies generation phase is used to add transitions labeled with dependencies to the local automata. In the following we sketch the different sub-phases that compose the dependency generation.

In the first sub-phase the dependencies generation considers each state q of the global automaton that is exited by transitions projected on different local automata. Suppose that q is exited by the transitions $q_i = \delta(q, p_i)$, with $1 \leq i \leq n$, that are projected on the n different local automata ACi. In this case the dependencies generation phase considers each automata ACi and relates the outgoing

dependency $!k(q,q_i)(pi)\_FCi\_\{FC1,\ldots FCn\}$ to its rule $q_i = \delta_{Ci}(q,p_i)$, with $q \neq q_i$. Moreover, the phase considers each filter FCj, with $j \neq i$, and adds to A_{Cj} the rule $q_i = \delta_{Cj}(q,?k(q,q_i)(pi)\_FCi\_FCj)$. Suppose that the local automata of the filters $FC1,\ldots FCn$ are in state q and the filter F_{Ci} applies the A_{Ci} rule $q_i = \delta_{Ci}(q,p_i)$. In this case F_{Ci} has to parse the outgoing dependency related to this rule, i.e.,it sends the integer $k(q,q_i)(pi)$ to the filters $FC1,\ldots FCn$. Each filter F_{Cj}, with $j \neq i$, can accept the integer by applying the transition labeled with the related incoming dependency (i.e., $q_i = \delta_{Cj}(q,?k(q,q_i)(pi)\_FCi\_FCj)$). In other words, dependencies ensure that filters synchronize with each other so that exactly one q-exiting transition, labeled with an invocation, is accepted. This validates the constraint imposed by the global automaton. In the case that different filters, at the same time, want to apply a q-exiting rule a leader election can be performed to elect the one that will apply its local rule. We point out that synchronization among filters is required only when the states of the applied rules are different.

In the second sub-phase the dependencies generation considers each rule $q' = \delta(q,p)$, with $q \neq q'$, projected on a filter FC and all filters $FC1,\ldots FCn$ where a q'-exiting rule has been projected. The phase relates to the rule $q' = \delta(q,p)$ the dependency $!k(q,q')(p)\_FC\_\{FC1,\ldots FCn\}$ and for each local automaton of the filter FCi, with $1 \leq i \leq n$, defines the rule $q' = \delta_{Ci}(q,?k(q,q')(p)\_FC\_FCi)$. Each filter FCi applies this dependency when F_C has applied the rule $q' = \delta(q,p)$ and parsed the related dependency $!k(q,q')(p)\_FC\_\{FC1,\ldots FCn\}$. In this way the filters $FC1,\ldots FCn$ synchronize to the state q' and the ordering imposed by the global automaton is respected, i.e., q'-exiting rules can be applied only after the q-exiting rule is applied.

Finally, ε-moves can be added to each local automaton in order to correctly link eventually disconnected states.

The F_C-filter activities are similar to the ones of the logically centralized monitoring system. It checks that both C local invocations and incoming dependencies verify the policy defined by the local automaton. It has a buffer where it can store all C-local invocations and all incoming dependencies. Moreover, it can undertake all reactions policies defined by the logically centralized monitoring system. In the following we sketch the FC filter activities by assuming that its A_C local automaton is in state q.

Suppose that the filter FC picks the invocation $!servicName(parameters)\_C\_S$ from its buffer, such invocation labels a q-exiting transition and verifies the related predicate. Then FC updates the A_C state, forwards the invocation to the filter FS and parses the dependencies (if any) related to such rule (*send invocation acceptance*).

Suppose that the filter FC picks the invocation $?servicName(parameters)\_S\_C$ up from its buffer, such invocation labels a q-exiting transition and verifies the related predicate. Then FC updates the A_C state, forwards the invocation $seviceName$ to component S and parses the dependencies (if any) related to such rule (*receive invocation acceptance*). We point out that when the service is

synchronous the filter waits for the service answer and puts it back in its local buffer.

The filter forwards without any check, invocations that are not defined inside the global automaton alphabet (*forwarding action*). It puts back in the buffer, invocations that cannot be accepted in the current state q, but can be accepted in a state reachable from q (*buffering action*). It accepts each incoming dependency that labels a transition exiting from the current automaton state(*incoming dependency acceptance*). Finally, FC locally detects the anomalous interactions when: (i) the invocation cannot be accepted in the current automaton state (e.g. q) and in any state reachable from q; (ii) the invocation was buffered and not consumed after a finite amount of time; (iii) the invocation is related to services not present in the interfaces definitions; (iv) an incoming dependency cannot be accepted (*anomaly detection action*).

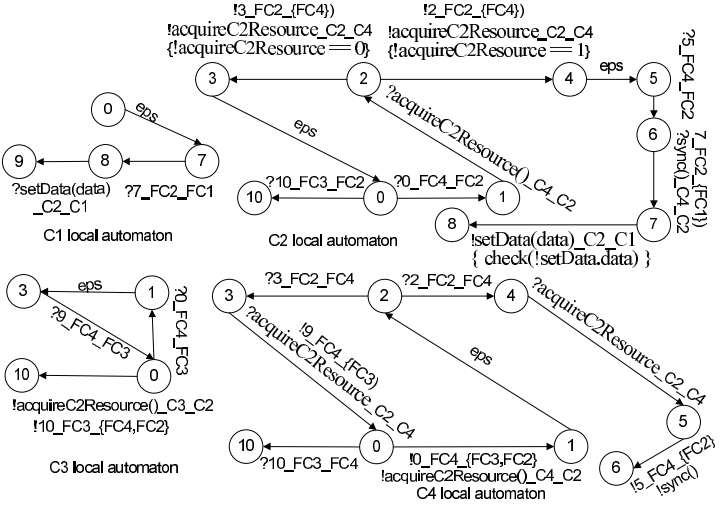

Fig. 11. The local automata

In Figure 11 we show part of the local automata related to our case study where we denote with eps an ε move. When the distributed monitoring system starts, all local automata are in state 0. Filter $FC3$ and $FC4$ synchronize so that exactly one of them performs its local invocation. Suppose that $FC4$ gains the right. Under this assumption $FC4$: (i) sends the integer 0 to the filters $FC3$ and $FC2$; (ii) sends the message $acquireC2Resource()$ to the filter $FC2$; (iii) changes the local automaton state to 1. Both filters $FC2$ and $FC3$ receive the dependency 0 and move to the state 1. We remark that any $FC3$ invocation has to be buffered so that mutual exclusion is ensured. We can observe that the service $!setData(data)\ \_C2\_C1$ can be provided after the chain of invocations $!acquireC2Resource()\ \_C4\_C2$, $?acquireC2Resource()\ \_C4\_C2$, $!acquireC2Resource\ \_C2\_C4$, $!sync()$, $?sync()$ is performed.

We remark that the local automata generation and the filters generation is hidden to the user by the DESERT tool. The user has to describe the centralized specification (i.e., system model and global automaton) and apply the DESERT tool in order to generate the filters system. As it is shown in [13,12] the filters simulate the logically centralized monitoring system. Filters realize a peer-to-peer minitoring that enhances security and fault tolerance w.r.t. the centralized implementation. This is consequence of the fact that a distributed implementation has not a single point of vulnerability. Moreover, when a filter fails unrelated filters can continue their activities.

7 The Case Studies

In this section we show different case studies where we have applied the DESERT tool. These case studies are related to different applications that run on both mobile and wired infrastructures.

In the following we sketch how DESERT can be applied to enhance the security in a component based application.

Component based software development (CBSD) aims to build a system from existing components. In contrast to traditional development, where system integration is often a marginal aspect, component integration is the centrepiece of CBSD. Developers have to face problems of components adaptation and ensure an acceptable security and dependability level. It is a widely accepted fact that components integration problems cannot be always addressed at development time. Components can be poorly documented so that the integration developers can make mistakes in the integration process. Components can contain bugs or malicious code, therefore security flaws are introduced. Components can be employed not exactly in the contexts for which they are intended, therefore, faults are introduced at the integration level. A component may have more functionalities than the developers know about and so he/she cannot understand the implications of introducing the component inside the system (see [3,9] for an extended survey).

Unsolved integration problems often result in the possibility of anomalous components interactions[5]. The DESERT tool can be used to generate monitoring systems providing a further layer that enhances the security of the integration code.

For instance in the case study presented in this paper we have monitored the components to ensure the consistency of water flow data. In this case study components are written in java so that tools to obtain the source code can be

[5] Notice that very often it is not possible to establish the nature of the component anomalous interactions. As it is described in [10] an external observer cannot distinguish when a component is interacting in anomalous way as a consequence of malicious intents or internal fault. However, from our point of view we can deal with such violations with different reaction policies. In particular, when the security is a crucial aspect we can isolate anomalous components. In the case that the fault tolerance must be enhanced we can recover such components.

used. This permits to analyze the components logic and produce rogue implementations that exploits bugs and overcomes the static security measures. In the simulation phase rogue clients were produced in order to: (i) exploit the components vulnerabilities; (ii) perform unauthorized access; and (iii) simulate race conditions. Furthermore, malicious clients were used to obtain fake water flow data. Local filters were able to discover such anomalous behavior (see [13] for details), apply the DESERT shunning reaction (see Section 5 for details), isolate the attackers and alert the system manager.

In [2] we have used DESERT to automatically assemble a set of components. In this context, one of the main goals is to compose and eventually adapt loosely coupled independent components to make up a system [22,23]. Building a distributed system from reusable or COTS components introduces a set of problems, mainly related to compatibility and communication. Often, components may have incompatible or undesired interactions. One widely used technique to deal with these problems is to use adaptors. They are additional components interposed between the components forming the system that is being assembled. The intent of the adaptors is to moderate the communication of the components in a way that the system complies only to a specific behavior. In [2] we use the *SYNTHESIS* tool to produce a global automaton specification (i.e. a centralized adaptor) that forces the system to exhibit only a set of *safe* or *desired* behaviors. For example, the adaptor forces the system to exhibit only the subset of deadlock-free and/or explicitly specified wanted behaviors. Such specification is automatically distributed and implemented by the DESERT tool by using the retry policy. In this case the monitoring system acts like an enforcement mechanism that ensures the policy described by the global automaton. The retry policy is used to enhance the system fault tolerance since it allows the anomalous components to continue their execution.

In [1] we have used DESERT to provide intrusion detection facilities in the the CoP protocol [24][6]. CoP is a protocol used for routing on mobile wireless sensor networks (WSNs).

In the following we summarize attacks that can be undertaken in wireless sensor networks (see [25] for an extended survey).

1. *Compromised Node:* Due to an external intervention, a sensor may be compromised and can be used to subvert the correct WSN behavior.

2. *False Node:* Additional fake nodes could be thrown in the sensed area sending false data or blocking the passage of true data.

3. *Node Malfunction or Outage:* A node in a WSN may malfunction and generate inaccurate or false data or it could just stop functioning hence compromising used paths.

4. *Message Corruption:* Attacks against the integrity of a message occur when an intruder inserts itself between the source and the destination and modifies the contents of a message.

[6] The research was partially funded by the European project COST Action 293, "Graphs and Algorithms in Communication Networks" (GRAAL). Preliminary results contained in this paper appeared in the [1].

5. *Denial of Service:* A denial of service attack may take several forms. It may consist in jamming the radio link or it could exhaust resources or misroute data.

Generally speaking, state machines can be used to face the above attacks (see [8,12,4] for details). For instance, messages corruption can be avoided by means of the predicates that define the correct message format. Denial of service can be detected by bounding the number of messages in each automaton path. Moreover, the automaton paths permit to describe the correct ordering among invocations. In the following we show how the DESERT tool has been used to address some of the above attacks by means of the CoP protocol [24]. To this extent we have enhanced the DESERT definition language with invocations that can contain the component type names. Suppose that C and S are two different types of components and their interfaces are defined by using the DESERT notation (see 3 for details). The invocation $!serviceName\ (X_1, \ldots, X_n)\_C\_S$ defines that one of the possible instances of the component C is sending the $serviceName$ asynchronous invocation to the component instances of the type S. The invocation $?serviceName\ (X_1, \ldots, X_n)\_C\_S$ defines that a set of instances of the component S can receive the asynchronous invocation of the $serviceName$ service.

In the field of location-awareness and clustering protocols like CoP, we model a mobile WSN by a set of sensors $AH = \{s_1, s_2, \ldots, s_k\}$. Let $L \subseteq AH$ be a subset $\{l_1, l_2, \ldots, l_m\}$ of sensors identifying the clusterhead of a given protocol P. In other words, the sensors in L characterize a set of areas $Ar = \{Ar_1, Ar_2, \ldots, Ar_m\}$ (clusters) where each area Ar_i represents the portion of the sensed area where the corresponding l_i plays the clusterhead role. There can be different roles according to P, let $R = \{C_1, C_2, \ldots, C_n\}$ be the set of roles. Each C_i has associated an interface that characterizes all messages sent/received by sensors playing that role. It is worth noticing that in this case role is used as synonymous of component type.

In Figure 12 we show the four types of roles that each sensor can play and the corresponding interfaces according to the described CoP protocol. Considering an area denoted with A_{VGN} we define the Out-range, the In-range and the Clusterhead roles that are played by sensors residing in it, and the Extern role representing the sensor playing the clusterhead role inside an adjacent A_{VGN} area.

The role Out-range models a sensor s located in a position that is inside the A_{VGN} but at a distance greater than ds (ds is a natural number) from the A_{VGN} center. This role defines an interface composed by the $?pos(double\ x, double\ y).void$ and $!send(double\ x, double\ y, int\ dest, char\ []\ msg).void$ asynchronous services. $?pos(double\ x, double\ y).void$ specifies that the sensor s can receive the incoming message $pos(double\ x, double\ y)$ used to set up its initial position. The parameters $double\ x$ and $double\ y$ are suitably instantiated with the coordinates of s. $!send(double\ x, double\ y, int\ dest, char\ []\ msg).void$ specifies that s can send the message msg towards the sink $dest$. The parameters x and y are instantiated with the current position of s and $dest$ is an integer that denotes a sink.

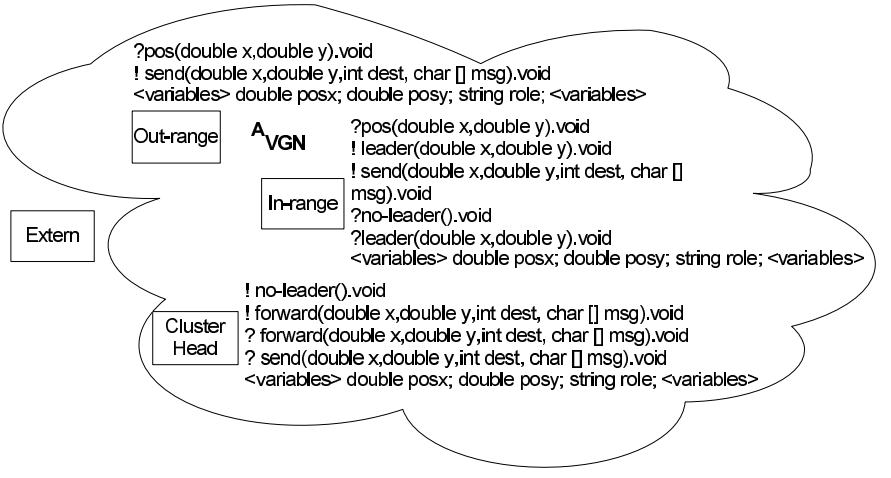

Fig. 12. The CoP roles

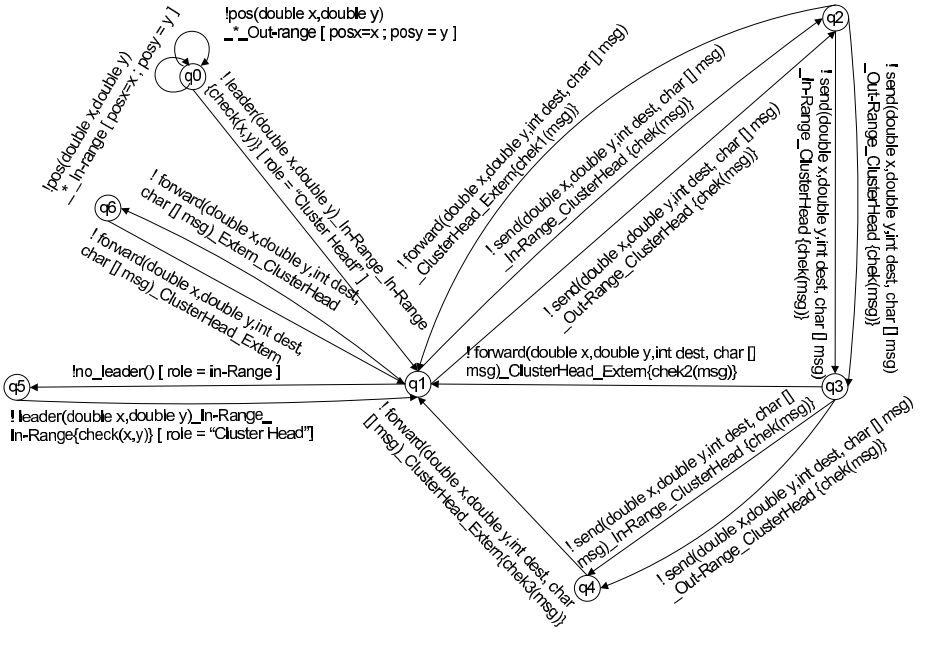

Fig. 13. Global Automaton

The role In-range models a sensor s located in a position inside an A_{VGN} and at distance at most ds from the corresponding center. This role adds to the Out-range role the following services: $!leader(double\ x, double\ y).void$, $?no-leader().void$ and $?leader(double\ x, double\ y).void$. The service $!leader(double\ x, double\ y).void$ specifies that the sensor s can send the message $leader(double\ x, double\ y)$ in order to become clusterhead. The parameters $double\ x$ and $double\ y$ are suitably instantiated with the coordinates of s. $?no-leader().void$ is implemented by s in order to accept the notification sent by the clusterhead when it leaves its role. $?leader(double\ x, double\ y).void$ is used by s to receive the notification of a sensor s' that requires to be clusterhead. The parameters $double\ x$ and $double\ y$ are suitably instantiated with the coordinates of s'.

The role Clusterhead is played by a sensor s providing the forward of messages towards the right sink. This role defines the following services: $!no-leader().void$, $?forward(double\ x, double\ y, int\ dest,\ char\ []\ msg).void$, $!forward(double\ x, double\ y, int\ dest,\ char\ []\ msg).void$ and $?send(double\ x, double\ y, int\ dest,\ char\ []\ msg).void$. The $!no-leader().void$ service specifies that the sensor s can send the asynchronous message $no-leader()$ to the environment. This message is sent by s in order to leave its clusterhead role due to its movement or to its draining battery. $?forward(double\ x, double\ y, int\ dest,\ char\ []\ msg).void$ implements the service used by s in order to receive the message msg. This message is forwarded by a clusterhead s' that resides in an area surrounding the one of s. The parameters x and y denotes the position of the clusterhead s' and $dest$ encodes the sink. The service $!forward(double\ x, double\ y, int\ dest,\ char\ []\ msg).void$ is used by s in order to forward the message msg towards the sink $dest$. The parameters x and y denotes the position of s. The service $?send(double\ x, doubley, int\ dest,\ char\ []\ msg).void$ is used by s in order to receive a message msg sent by a sensor s' residing inside the A_{VGN}. The parameters x and y denotes the position of s' and $dest$ is an integer that denotes a sink.

The role Extern models one of the clusterheads surrounding the current A_{VGN}.

All roles have associated the real numbers x and y used to store the current position of the sensor and the string $role$ that encodes the current role played by the sensor.

Starting from the description of the CoP protocol we now point out some basic properties that should be guaranteed in order to obtain a fair behavior of the protocol.

1. For each area A_{VGN} there must be at most one sensor playing as clusterhead.

2. When a finite amount of data has been collected by a clusterhead, it must be forwarded in the correct direction.

3. A clusterhead that changes its status to normal sensor due to a movement or because of the draining battery has to forward all the collected messages before its movement.

4. All messages forwarded by a clusterhead have to be received by the clusterhead of the adjacent VGN area.

5. When a clusterhead leaves its role a new sensor (if any in the area) has to take its role.

We formalize these properties by defining a state machine that will be given in input to our tool in order to produce the distributed "patch" for the sensors participating in the CoP protocol.

Figure 13 shows the Global Automaton related to the sensors based system of Figure 12. This automaton defines the correct sequences of messages inside each A_{VGN}. At the beginning (state $q0$) all the sensors are informed about their positions[7]. According to their position, each sensor sets its local variable *role*. The In-Range sensors candidate themselves to become leader. Once the ClusterHead has been elected, the system moves to state $q1$ and the real interaction can start. This transition, in practice, realizes property 1. Property 2 is realized by the path $q1, q2, q3$. In this example we fixed the "finite amount of data" by means of a maximum of three collected messages after which the clusterhead necessarily forwards them. Property 3 is realized by means of transition $q1, q5$, in fact, if the system state is $q1$ there are no messages stored in the clusterhead. When data is forwarded, it is received by the Extern role, i.e., some clusterhead of another A_{VGN} on the way to the specified sink.[8] And this realizes property 4. Finally, property 5 is valid by means of transition $q1, q5$. From $q5$, in fact, a new ClusterHead must be elected before any other communication can occur.[9] Note that, when a ClusterHead receives a forward (transition $q1, q6$), it necessarily has to forward it (transition $q5, q1$).

Concerning the predicates, $check1(msg)$ is used to verify the correct format of the message msg forwarded by the ClusterHead. This predicate permits to check that msg is not a buffer overflow attack. The $check2(msg)$ is similar to the above one, however it adds a test verifying that msg is equivalent to the compression of the two messages previously received by the ClusterHead. The predicate $check(x, y)$ verifies that the leader is at distance at most ds from the A_{VGN} center.

We have used the DESERT tool to automatically generate a filter for each sensor. Each filter is constituted by a few lines of code installed in the sensors. This realizes a distributed system that locally detects violation of the sensors interactions policies and is able to minimize the information sent among sensors in order to discover attacks across the network.

[7] It is worth notice that the position is informed by means of the invocation *pos* that exits from the state 0. In particular this invocation is performed by a component not modeled inside the system (i.e., the satellite component) therefore we use the symbol '*' in the sender field.

[8] Note that, in our example, while a clusterhead is collecting messages (i.e., the system is either in $q2$ or $q3$ or $q4$), it is not allowed to receive a forward. This, in fact, can happen only at $q1$. In order to not waste messages, this means that, according to the scheduling at the MAC layer, there is some time that is a priori set up. During such a time a clusterhead can wait for other messages without incurring in any forward.

[9] Again, in order to not waste forward messages we may think of a buffer for the In-Range roles in which a forward is temporarily stored till a new ClusterHead is elected.

In [1] we show how our method affects the performance of the CoP protocol. The experiments are performed running the powered protocol over hundreds of random instances of mobile WSNs. We show the overhead in terms of consumed energy and in terms of performed instructions by the filtered sensors. The experiments also show the estimated percentage reduction of the network lifetime respect to the original CoP protocol.

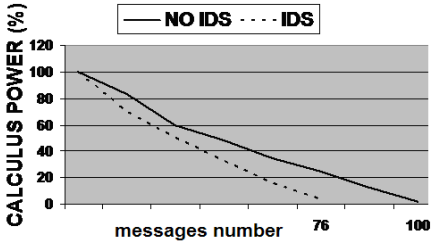

Fig. 14. Average of the residual computational power depending on messages exchanged inside an A_{VGN}

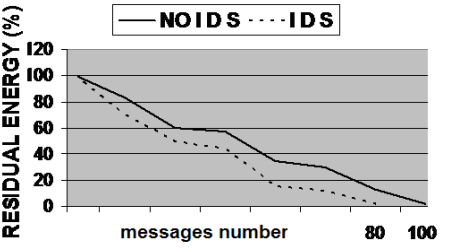

Fig. 15. Average of the residual energy depending on messages exchanged inside an A_{VGN}

In Figure 14 we show the lifetime of the system inside an A_{VGN}. Considering each kind of message of the sensors as a different set of instructions, we show the overhead in terms of percentage of computational power loss. The cost of ensuring the normal protocol behavior in terms of number of instructions is increased, on average, around 24%. Notice that transmission/receptions operations are much more expensive than local computations. According to the consumption values expressed in [26,24], transmitter and receiver electronics consume an equal amount of energy per bit, namely $5nJ/bit$. While the energy to support the signal above some acceptable threshold against power attenuation caused by the distance is just $100pJ/bit/m^2$.

In Figure 15 we show that, on average, the percentage of the draining of the sensors batteries inside an A_{VGN} is increased by around 20%.

Concerning the detection of attacks we detect any behavior that violates the property expressed by the global automaton. We use the DESERT shunning policy in order to isolate the attacker. In particular each malicious node is isolated by its filter and by the filters of all surrounding nodes[10].

8 Conclusions and Future Works

In this work we presented the DESERT tool that allows the automatic generation of distributed monitoring systems for enhancing security and dependability at architectural level.

[10] When the filter of a sensor is compromised other surrounding filters detect and discard the anomalous invocation. Moreover, they observe with each other in order to send exactly one alert towards the sink.

An architectural level definition language permits to specify both the system model and the correct system behavior. The system model is provided by means of an interface description language. The correct system behavior is provided by means of state machines. These 'centralized' specifications are used by the front-end and the back-end of the DESERT tool in order to generate a distributed monitoring system implementation. The monitoring system is constituted by one filter for each component that locally detects violation of the global specification.

DESERT has been used for applications running on both mobile and wired infrastructures.

In future work we are developing more complex detection and recovery mechanism. It can happen that the retry reaction policy causes a condition in which the same component retry several times as a consequence of the same condition error. Moreover, very often the component generating the anomalous behavior does not always correspond to the source that triggered the error[11]. In both cases we exploit the integration level view provided by the global automaton. We analyze the state reached in the global computation so that we can identify the sources of errors and we can recover several distributed components.

References

1. Inverardi, P., Mostarda, L., Navarra, A.: Distributed IDSs for enhancing security in mobile wireless sensor networks. In: IEEE International Workshop on Pervasive Computing and Ad Hoc Communications (IEEE PCAC'06), IEEE Computer Society Press, Los Alamitos (2006)
2. Inverardi, P., Mostarda, L., Tivoli, M., Autili, M.: Automatic synthesis of distributed adaptors for component-based system. In: Proceedings of the 21st Automated Software Engineering (ASE) Conference (2005)
3. Lindqvist, U., Jonsson, E.: A map of security risks associated with using cots. Computer 31, 60–66 (1998)
4. Orset, J.M., Alcalde, B., Cavalli, A.: An EFSM-based intrusion detection system for ad hoc networks. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, Springer, Heidelberg (2005)
5. Ko, C., Ruschitza, M., Levitt, K.: Execution monitoring of security-critical programs in distribute system: A specification-based approach. IEEE (1997)
6. White, G.B., Fisch, E.A., Pooch, U.W.: Cooperating security managers: A peer-based intrusion detectionn system. IEEE Network (1996)
7. Stillerman, M., Marceau, C., Stillman, M.: Intrusion detection for distributed applications. Communications of the ACM (1999)
8. Eckmann, S.T., Vigna, G., Kemmer, R.A.: Statl: An attack language for state-based intrusion detection. Journal of Computer Security 10, 71–104 (2002)
9. de Lemos, R., Gacek, C., Romanovsky, A.: Architectural Mismatch Tolerance. In: Architecting Dependable Systems. LNCS, vol. 2677, pp. 175–196. Springer, Heidelberg (2003)
10. Avizienis, A., Laprie, J., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transaction on Dependableand Secure Computing 1, 11–33 (2004)

[11] In this case a severe reaction policy would terminate several components.

11. Delgado, N., Gates, A.Q., Roach, S.: A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. IEEE Transactions on Software Engineering 30, 859–871 (2004)
12. Inverardi, P., Mostarda, L.: A distributed intrusion detection approach for secure software architecture. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 168–184. Springer, Heidelberg (2005)
13. Mostarda, L.: Distributed detection systems for secure software architectures, Ph.D, Thesis in computer Science, University of L'Aquila (2006)
14. Porras, P.A., Neumann, G.P.: Event monitoring enabling responses to anomolous live disturbances. Proc. of 20th NIS Security Conference (1997)
15. Snapp, S.R., Dias, J.B.G.V., Goan, T., Heberlein, L.T., Ho, C., Levitt, K.N., Mukherjee, B., Smaha, S.E., Grance, T., Teal, D.M., Mansur, D.: Dids (distributed intrusion detection system) - motivation architecture and early prototype. In: Proc. 14th National Security Conference vol. 1, pp. 361–370 (1997)
16. Vigna, G., Kemmerer, R.A.: Netstat: a network-based intrusion detection system. Journal Computer Security 7, 37–71 (1999)
17. Javitz, H.S., Valdes, A.: The nides statistical component description and justification. Technical report - Columbia University (1994)
18. Vaccaro, H., Liepins, G.: Detection of anomalous computer session activity. In: Proc. of the 1989 Synopsium on Security and privacy, pp. 280–289 (1989)
19. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Effecient decentralized monitoring of safety in distributed system. In: ICSE (2004)
20. Schneider, F.B.: Enforceable security policies. ACM Trans. on Information and System Security 3, 30–50 (2000)
21. European Commision 6th Framework Program - 2nd Call Galileo Joint Undertaking: Cultural Heritage Space Identification System (CUSPIS), `http://www.cuspisproject.info`
22. Crnkovic, I., Larsson, M.: Building reliable component-based Software Systems. Artech House, Boston, London (2002)
23. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (2004)
24. McCann, J.A., Navarra, A., Papadopoulos, A.A.: Connectionless Probabilistic (CoP) routing: an efficient protocol for Mobile Wireless Ad-Hoc Sensor Networks. In: IPCCC (2005)
25. Perrig, A., Stankovic, J., Wagner, D.: Security in wireless sensor networks. Commun. ACM 47, 53–57 (2004)
26. Heinzelman, W., Chandrakasan, A., Balakrishnan, H.: Energy-Efficient Communication Protocols for Wireless Microsensor Networks. In: Proc. of the Hawaiian Int. Conf. on Systems Science (2000)