Model-based monitoring of component-based software systems

Irène Grosclaude [1]

Abstract. The development of component-based software systems opens the possibility of using model-based diagnosis techniques for large software systems. We have developed a simple monitoring system based on the modeling of the external behavior of software components by Petri nets. With each component is associated a local controller which observes the messages received and sent by the component and compares them with the specified behavior. As the components interact, information is collected on error emission and time constraint violation to infer indicators about the state of components.

1 INTRODUCTION

Fault occurrence is unavoidable in any large software application. Nevertheless, attempts of using model-based techniques for the diagnosis of software faults are few. Even with the structural improvement brought by object oriented programming, software behavior modeling is often difficult because the software can hardly be split into separate entities with clear interaction, unless it is modeled it at a very fine level. But in our opinion, new perspectives are opened by the development of component-based software engineering which conceives a software application as a set of components – each with a well-specified role – assembled by explicit rules.

Component-based software engineering aims at optimizing software component reusability and at developing infrastructures for open and rapidly evolving applications in which separately developed components can easily be assembled, where new components can easily be added and old ones easily replaced (even during runtime). With the growing size and complexity of applications, a support to fault monitoring becomes essential. Even if the components are relatively reliable, errors can arise at runtime due to remaining undetected logical errors (programming errors), to resource allocation problems (unavailable resource, deadlock) or to component use errors (incorrect inputs, requests made in a wrong order). The interaction between components can lead to the propagation of a fault from one component to another: by propagation of an exception, or because a component is waiting for another component which is blocked or slowed down for example. To determine the components at the source of a problem, it is necessary to know explicitly the dependence links between components. These links are dynamic, they not only depend on the connections between components but on the messages effectively exchanged during runtime.

In this article we do not try to explain the exact cause of a fault, but we aim at determining on which component a fault has first been visible (we suppose that a fault is first observable on the component on which it occurs). We want to discriminate a component which has emitted exception or error messages from a component which has been impacted by such messages. Let's take for example the interaction scenario between three components described on the right part of figure 1. The scenario is the following: first the traveler sends a trip order to the travel agent, then the travel agent sends back to it an acknowledgment, determines the travel legs and sends a plane ticket order to the airline server. The airline is supposed to send an acknowledgment to the travel agent, then to send it a confirmation of the flights and to send an e-ticket to the traveler. Let's suppose that the airline server is not able to complete its task and sends an error message to the travel agent. If the travel agent component then sends a error message to the traveler, it must not be considered as responsible for the transaction failure. In the same way, if the airline server does not respond or takes a long time to respond to the travel agent, the latter is not responsible for the delay induced in the sending of the itinerary to the traveler.

The rest of this paper is organized as follows: after having introduced the concept of software component, we describe the model on which our supervision system is based and the way the evolution of each component is followed. Then we describe the statistical component state indicators that we calculate. Finally, we present the possible extensions of our approach and compare it to related works.

2 COMPONENT-BASED SOFTWARE SYSTEMS

The idea that software should be componentized is not new. Already in 1968 Douglas McIlroy suggested that software industry could, like in electronic component industry, see applications like assembly of prefabricated components, mostly bought "on the shelves". Even if there is no consensus at the present time on what exactly is a software component, several successful software component models exist, and integration infrastructures for binary components are available in the market place (ex: the Sun Microsystems' JavaBeans technology, the Microsoft's Component Object Model (COM) technology, the Corba Component Model).

We adopt in this article a very general definition of software component: a component has a particular functionality, it can be connected to other components by a set of ports and can be configured by means of a set of attributes. The ports are of two kinds: the client ports on which the component emits requests and the server ones on which the component receives requests. Ports are defined by a name and a type. This type determines the methods offered (server port) or required (client port) by the component. A client port and a server port can be connected if the server port is of the same type as the client port or of a more specific type. An assembly of components is valid if each client port is linked to a corresponding server port.

[1] France Telecom R&D, 2 avenue Pierre Marzin, 22307 Lannion, France, email: irene.grosclaude@francetelecom.com, tel: +33 (0)2 96 05 07 53, fax: +33 (0)2 96 05 19 56

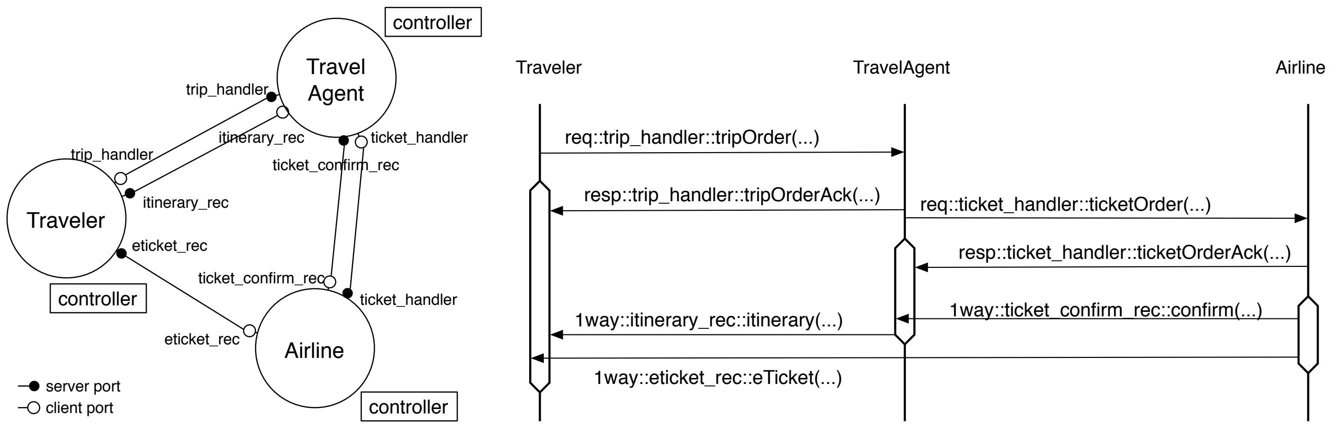

controller
Travel Agent
trip_handler
itinerary_rec
ticket_handler
ticket_confirm_rec
trip_handler
Traveler
itinerary_rec
eticket_rec
controller
ticket_confirm_rec
ticket_handler
eticket_rec
Airline
controller
server port
client port
Traveler
TravelAgent
Airline
req::trip_handler::tripOrder(...)
resp::trip_handler::tripOrderAck(...)
req::ticket_handler::ticketOrder(...)
resp::ticket_handler::ticketOrderAck(...)
1way::ticket_confirm_rec::confirm(...)
1way::itinerary_rec::itinerary(...)
1way::eticket_rec::eTicket(...)

Figure 1. The ticket order application. The left part of the figure shows the component assembly. The right part describes the normal interaction scenario.

Components communicate through the communication links set up between their ports. The interaction between components is limited to the messages exchanged through these links. We make no hypothesis on the communication type, synchronous or asynchronous, neither on the communication model: communication can correspond to method invocation or can be event-based. Components can be distributed on different computers. As already noticed, component-based applications can rapidly evolve as components can easily be added or replaced.

Figure 1 shows an illustrative example of component-based application taken from [6]. This example shows high-level (service level) components, but the method can be applied whatever the component granularity is.

3 SUPERVISION ALGORITHM

We want to develop a supervision system which continuously evaluates the state of the components. The supervision system must localize degradations in component execution, discriminating the degradations due to the component itself from the degradations resulting from problems on other components. The degradation of the state of a component manifests itself by a high rate of error message emission and/or reception, by an increase of the component response times or in the worst case by a complete blockage of the component.

We do not suppose that we have the source code of the components and we make no hypothesis on the used programming language. This makes an intrusive method impossible (to modify the code of the components in order to add observation points for example). The only available observations are the messages that the components exchange, which can be observed by interceptors set on each port of the components. The interceptors inform the component supervisor of the occurrence of a message, with its occurrence time.

Since many processes may be run in parallel inside the component, the monitoring system needs means to determine which sent or received messages are correlated. The correlation is done by using a model of the component behavior which describes in which (partial) order the correlated messages occur and which indicates if the messages can be correlated by their content (two messages can contain the same piece of information, a client or transaction ID for example, which can be used to correlate the messages).

By correlating the messages sent and received by a component, it is possible to follow the error message propagation inside this component. If in addition we know at anytime if a component is processing or if it is waiting for a message from another component, and if we know the normal processing time of components between two messages, we can determine if the component is the cause of a slow down or of a blocking.

3.1 Component behavior model

We have chosen a formalism based on Petri nets to model the behavior of each component. The places of the Petri net represent the state of the component. The transitions model the interactions with other components and are labeled by the exchanged messages. As an example, figure 2 shows the behavior model of the travel agent component (time constraints are not shown).

The message occurrence order is not sufficient information, the model is complemented as follows:

Typing of places: A place is of type *waitfor* if it corresponds to a state in which a component waits for a message from another component; it is of type *calculation* if it corresponds to a state in which a component is active (places of type *donothing* can also be used for synchronization).

Typing of arcs: We define two types of arcs: the arcs of type *data_flow* link the messages corresponding to the same transaction and the arcs of type *multi_exec* describe the synchronization between the treatment of different transactions.

Temporal constraints: An occurrence instant is associated with each transition. Temporal constraints between these instants can be added to transitions. These constraints are checked each time the transition is fired. They express time delays between the messages received or emitted by a component. We distinguish in particular two types of constraints: the internal constraints which describe the normal duration for the component to perform a task between two messages and the external constraints which describe the normal duration the component waits for a message from another component.

For example, if the time that the travel agent component takes to acknowledge a trip order is between 20 and 100 units of time, the constraint: *internal_constraint IC1 : t1-t0 in* [*20,100*] can be stated. If this constraint is associated with the transition *trans1*, it will be checked as soon as the acknowledgment message is sent.

Exception handling: We differentiate the exceptions that appear in the interface specification of component methods – we call them functional exceptions – from all the exceptions that may happen but that are not explicitly specified in the model (in general problems due to runtime conditions like resource problems are not considered in the method specification). The functional exceptions are divided into classes: the exceptions due to the client and the exceptions due to the server. To deal with the non-functional exceptions, a generic type "*UnknownError*" is used. The model must describe if the component may send an error message as a result of a problem occurring when it is calculating, and it must describe what happens if the component receives an unexpected error message when is it waiting for a response from another component (if the message is propagated to the client, if the execution aborts or not)[2].

Messages introspection: It can be interesting to compare the content of the messages in order to correlate the messages of the same data flow. Let's suppose in our example that the travel agent component successively receives two requests: *trip_handler::tripOrder(triporder#1)* and *trip_handler::tripOrder(triporder#2)*. If then it emits the message *trip_handler::tripOrderAck(ack)*, there is no way to know if this acknowledgment corresponds to the first request or to the second one. To clear up such ambiguities when the message content makes it possible, we offer the possibility of extending the message description with variables corresponding to particular fields of the message parameters. Let's suppose for example that the triporder and the ack parameters contain the traveler identification number, we could modify the model and replace the first two transitions by:

request::trip_handler::tripOrder(triporder,
x = triporder.getClientNumber())
response::trip_handler::tripOrderAck(ack,
x = ack.getClientNumber()).

As it can be noticed in this example, the description must contain the way of accessing the parameter field (here written as an object method call).

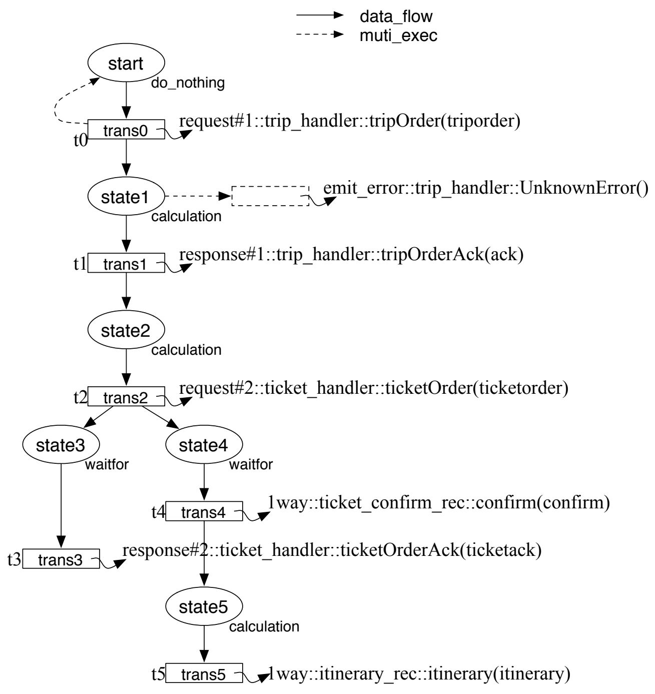
The good modeling of component behavior is a key point of our method. As we are aware that it could become its weakness, we are investigating methods to mine the models from execution traces, following approaches like [3, 7].

3.2 Component history construction

The supervision is local to each component and consists in following the evolution of the component from the model as the messages arrive. The supervision is not done directly from the Petri net model but from a translation of this model into a set of partial transitions called *tiles*, following the approach in [5] which we have extended in order to take into account attributes with arguments and with non-binary values. This way, we keep in memory the whole history of the component evolution in an homogenous structure, including the occurrence time of messages and their content.

3.2.1 Translation of the Petri net into tiles

Each transition of the Petri net is transformed into a tile which describes the partial state change when the transition is fired. Each tile



data_flow
muti_exec
start
do_nothing
t0
trans0
request#1::trip_handler::tripOrder(triporder)
state1
calculation
emit_error::trip_handler::UnknownError()
t1
trans1
response#1::trip_handler::tripOrderAck(ack)
state2
calculation
t2
trans2
request#2::ticket_handler::ticketOrder(ticketorder)
state3
waitfor
state4
waitfor
t4
trans4
1way::ticket_confirm_rec::confirm(confirm)
t3
trans3
response#2::ticket_handler::ticketOrderAck(ticketack)
state5
calculation
t5
trans5
1way::itinerary_rec::itinerary(itinerary)

Figure 2. Behavior model of the travel agent component.

is composed of a precondition which describes the necessary conditions on the values of a set of attributes for the transition to fire, of a label which corresponds to the observable event associated with the transition and of a postcondition which describes the values of the attributes after the transition firing. The same attributes are present in the pre and postcondition to prevent conflicts due to concurrent assignments (transitions are fired one by one).

The rules to construct a tile from a transition of our model are given in figure 3 (all the variables are prefixed by *?*, and *?_* is a notational convenience for variables that can take any value). An attribute is created for each state preceding (resp. following) the transition. This attribute changes from *on* to *off* (resp. from *off* to *on*) when the transition is fired. An attribute is also created for each parameter *arg* of the message in order to keep its runtime value in the associated variable *?arg*. This variable *?arg* is instantiated when a message is received (a response on a client port or a request/one-way on a server port). To each message is added an additional parameter which corresponds to its occurrence time. This parameter is instantiated at runtime by the message interceptor. The parameter *?e* is used to identify the different data flows.

The constraints associated with transitions are translated into *constraint checking actions*. The internal constraints are split into two actions: a *start_internal_constraint* action which is associated with the reference instant (this action is used for blockage detection) and a *end_internal_constraint* action which describes the constraint and which is associated with the second instant in the constraint. For example, the internal constraint *IC1 : t1-t0 in* [*20,100*] is split between the two tiles ***tile*** *transition0* and ***tile*** *transition1* given below.

The following tiles are obtained from the travel agent model (figure 2).

[2] This information could probably be at least partially added automatically from the type of nodes and from the context composed by the other messages.

	add to precondition	add to postcondition
state changes	state_{pre}[?e](on) state_{post}[?e](off)	state_{pre}[?e](off) state_{post}[?e](on)
time instant	t_i[?e](?_)	t_i[?e](?t_i)
parameter values		
• client port		
- request/one-way	arg[?e](?arg)	arg[?e](?arg)
- response	arg[?e](?_)	arg[?e](?arg)
• server port		
- request/one-way	arg[?e](?_)	arg[?e](?arg)
- response	arg[?e](?arg)	arg[?e](?arg)
next execution	state_{next}[?e+1](off)	state_{next}[?e+1](on)
label : port::methodName[?arg, ?t_i]		

Figure 3. Rules to transform the Petri net transition of fig 4 into a tile.

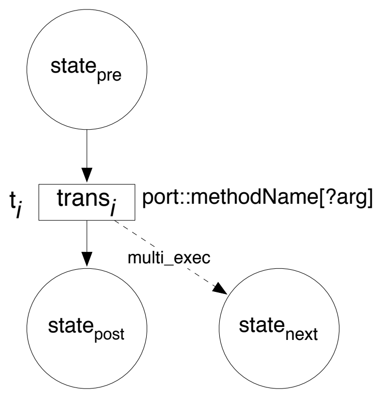


Figure 4. General form of a Petri net transition in a component behavior model.

***tile** transition0[?e]*
***label**: trip_handler::tripOrder[?triporder,?t0]*

pre	***post***
start[?e]:(on)	*start[?e]:(off)*
state1[?e]:(off)	*state1[?e]:(on)*
start[?e+1]:(off)	*start[?e+1]:(on)*
triporder[?e]:(?_)	*triporder[?e]:(?triporder)*
t0[?e]:(?_)	*t0[?e]:(?t0)*

***action**: start_internal_constraint*

***tile** transition1[?e]*
***label**: trip_handler::tripOrderAck[?ack,?t1]*

state1[?e]:(on)	*state1[?e]:(off)*
state2[?e]:(off)	*state2[?e]:(on)*
ack[?e]:(?ack)	*ack[?e]:(?ack)*
t1[?e]:(?_)	*t1[?e]:(?t1)*

***action**: end_internal_constraint[t1[?e]-t0[?e] in [20,100]]*
***action**: start_internal_constraint*

***tile** transition2[?e]*
***label**: ticket_handler::ticketOrder[?ticketorder,?t2]*

state2[?e]:(on)	*state2[?e]:(off)*
state3[?e]:(off)	*state3[?e]:(on)*
state4[?e]:(off)	*state4[?e]:(on)*
ticketorder[?e]:(?ticketorder)	*ticketorder[?e]:(?ticketorder)*
t2[?e]:(?_)	*t2[?e]:(?t2)*

***action**: end_internal_constraint[t2[?e]-t1[?e] in [10,50]]*

***tile** transition3[?e]*
***label**: ticket_handler::ticketOrderAck[?ticketack,?t3]*

state3[?e]:(on)	*state3[?e]:(off)*
ticketack[?e]:(?_)	*ticketack[?e]:(?ticketack)*
t3[?e]:(?_)	*t3[?e]:(?t3)*

***tile** transition4[?e]*
***label**: ticket_confirm_rec::confirm[?confirm,?t4]*

state4[?e]:(on)	*state4[?e]:(off)*
state5[?e]:(off)	*state5[?e]:(on)*
confirm[?e]:(?_)	*confirm[?e]:(?confirm)*
t4[?e]:(?_)	*t4[?e]:(?t4)*

***action**: start_internal_constraint*

***tile** transition5[?e]*
***label**: itinerary_rec::itinerary[?itinerary,?t5]*

state5[?e]:(on)	*state5[?e]:(off)*
itinerary[?e]:(?itinerary)	*itinerary[?e]:(?itinerary)*
t5[?e]:(?_)	*t5[?e]:(?t5)*

***action**: end_internal_constraint[t5[?e]-t4[?e] in [10,50]]*

***tile** transition6[?e]*
***label**: trip_handler::UnknownError[?t6]*

state1[?e]:(on)	*state1[?e]:(off)*
t6[?e]:(?_)	*t6[?e]:(?t6)*

***action**: end_internal_constraint[]*

3.2.2 History construction method

A history is a partially ordered set of instantiated tiles which describes the sequential or concurrent partial state changes that could have happened in the component and that explain the observed messages between the components. An example of possible history for the travel agent component is given in figure 5. As the state of a component may be ambiguous, the set of all possible histories is constructed locally for each component.

When a component is instantiated, an execution counter is set to zero and a first history is created. This history is initialized with *$start_i$[0](on)* for each initial place of name *$start_i$* and the constraint $\forall ?e$ *$state_j$[?e](off)* is stated for all other places *$state_j$*. In the following, we call *current constraints* the final values of the attributes after the firing of all the transitions. In figure 5, the attribute values constituting the current constraints are written in bold.

When a message is observed, the histories are updated in the following way:

- If the message can be unified with the label of a tile, the unification is propagated to the whole tile and the tile becomes a candidate to update the history;
- If a precondition of a candidate tile is unifiable with the current constraints of a history, the tile is added to an instance of the history and the unification is propagated to the history and to the tile (including the actions). Then the current constraints are updated with the values in the postcondition. If an action is associated with the tile, it is processed;
- If no tile can be added to a history, this history is canceled (since it means that this history is not consistent with the observations).

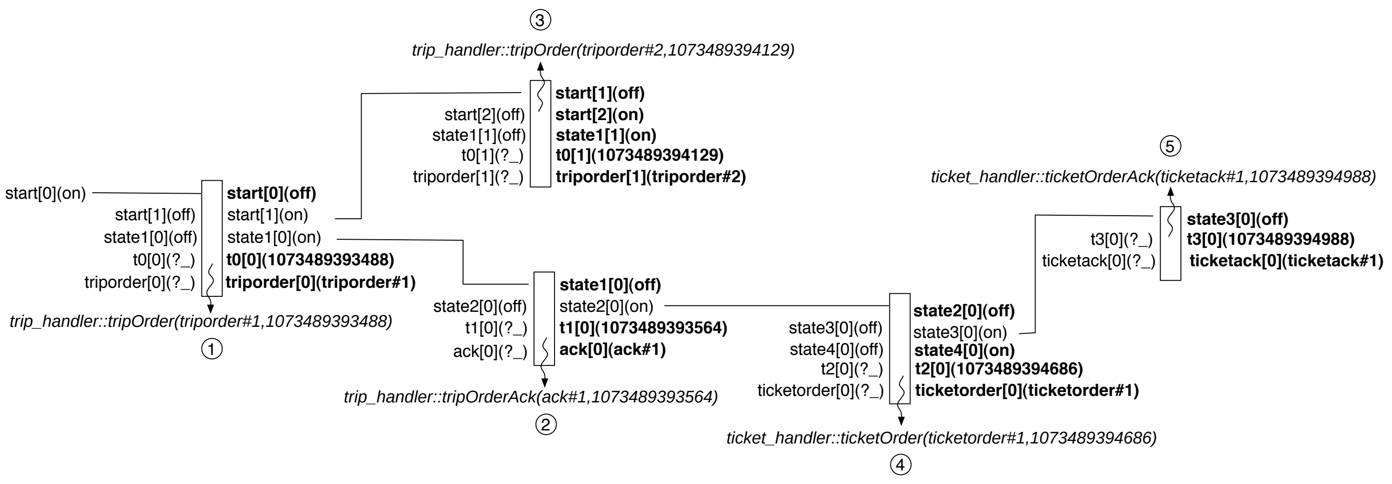


Figure 5. A possible history for the travel agent component. The circled figures indicate the message occurrence order.

It must be noticed that the histories must be regularly reduced to keep an acceptable size. Different policies can be used to delete completed executions, on the basis of the elapsed time from the last event, or by adding cleaning actions in the model for example.

3.3 Component state evaluation

We have shown in the previous section how we construct a representation of the component evolution. We show in this section how we exploit the constructed history in order to evaluate the component state.

3.3.1 General indicators

Our aim at the moment is to get general information on the state of the components. We use the following indicators:

- A set of error indicators: the first indicator is the rate of unexpected *UnknownError* emitted by the component. The two other indicators indicate the rate of functional errors imputed to the component: the rate of server-involved functional errors emitted by the component after the reception of a request and the rate of client-involved functional errors received by the component after it has emitted a request. These indicators are updated each time an emission or reception error occurs. As the evolution of an error rate may be more significant than its exact value, the derivatives of the rates are also calculated.
- Processing time indicators : the first indicator is the average delay of the component when it processes its internal tasks. This delay is calculated from the internal constraints given by the model. As this delay may not be very meaningful, specially if there is a large scale difference between the time intervals in the different constraints, we also calculate the average delay rate, and its derivative.

3.3.2 Component blockage detection

A component is considered as blocked if it emits none of the messages it is supposed to during an excessive duration. Such a situation can be detected by using the internal constraints in the behavior model. When a message is received which corresponds to the starting instant of internal constraints (the instant from which elapsed time is measured) a particular counter is incremented. This counter corresponds to the number of waited messages.When one of these messages is received, the counter is decremented. A component blockage is detected when the counter is positive and none of the waited messages has been received during an excessive time. This situation can be detected by using a timer. A maximal time is fixed, greater than all the maximal internal constraints of the component. This time is used to start and reset the timer. The timer is activated when the counter of start of internal constraint takes a non-zero value, and is restarted each time a waited message is received. If the time of this timer elapses, the component is considered as blocked.

4 FUTURE WORK

Firstly, we plan to improve the treatment of functional errors. At the present time, when a functional error is emitted in response to a request, it is imputed to the sender of the request or to the sender of the error message, depending on the error type. Things can in fact be more complicated. If the sender of a request receives an error message because it has sent a wrong parameter, to really impute the problem to that component, we must be sure that it has not previously received this wrong parameter from another component. To deal with this problem, we need a more precise specification of the cause of the functional errors and of the propagation of a parameter from one component to another. We also need information about the places where the parameters are assigned or modified.

The follow-up of parameters is included in a more general study about the global reconstruction of an interaction involving several components. We want to evaluate if a global history construction (by merging the component local histories) could bring useful information to explain errors. Dependency analysis (like program slicing) or an approach similar to consistency-based diagnosis could bring interesting information to localize the cause of an error in order to explain to a user why his request has failed.

5 RELATED WORK

Few works address the problem of black-box component runtime monitoring. Closely related to our work is the work in [12] on the supervision of event-driven, embedded real-time software. The normal component behavior is expressed in a formalism based on communicating finite state machines (the ITU Specification and Description Language SDL). The supervisor detects the discrepancies between the specified and the observed behavior. The approach deals with specification nondeterminism. The computational cost due to the possible large number of belief hypothesis is reduced by the use of a hierarchical supervisor. The difficult problem of supervisor resynchronization to determine the post-failure state of the target system is underlined but no general solution is proposed. Performance evaluation is detailed in [9]: from the specification of statistical response time requirements, start and stop interval monitoring directives are included in the model. The approach is only effective for monitoring session-oriented services (sessions must execute sequentially).

Our work is also related to the *oracle* problem for software testing, that is, the problem of providing a pass/fail judgment for a program execution[1]. Most of the approaches are based on executable assertions, but for black-box components, only few types of assertions remain applicable, mainly “consistency between arguments” and “dependency of return value from arguments”[11]. Since the efficiency of assertions depends on the precision with which they describe the program, their usefulness for components of any granularity is not certain. Works that extend the component specification with the component interaction protocol are closer to our work. The input/output behavior of components is specified by communicating finite-state machines, see [4] for example, or by enhanced regular expressions in [10]. The possibility of using the interaction models for runtime checking is suggested but these works mainly focus on component behavior composition, on compatibility checking between interfaces or between interfaces and implementations.

Finally, the work in [13] can be cited. Even if it requires the component source code, it uses measures similar to ours (error rates, performance) to infer component health indicators.

6 CONCLUSION

We have presented a software component monitoring approach based on the local interaction model of the components. This local approach is well adapted to component-based applications which are often large and evolve rapidly. The model is described in a graphical formalism based on Petri nets. We have shown how it can be translated into a set of transitions, or tiles, and how a representation of the component history is constructed by chaining these transitions as the components interact. During the history construction, statistical information is collected locally to each component about error emissions and processing times. This way, the component can exhibit information about its state which can be used for maintenance purpose.

Our intention is also to draw the DX community’s attention to existing works on component-based software system monitoring. Few works exist in the DX community about software diagnosis ([2, 8] to our knowledge). The main reason may be that until now software was not structured enough to enable the use of model-based diagnosis techniques, unless the software is modeled at the instruction-level. Component-based software infrastructures aim at allowing a rapid integration of separately developed software components. Because the developed applications are generally large, complex and highly dynamic, automatic diagnosis and management tools are required. Since these applications are strongly structured, we think that they could benefit of model-based techniques.

REFERENCES

[1] L. Baresi and M. Young, ‘Test oracles’, Technical Report CIS-TR-01-02, Dept. of Computer and Information Science, Univ. of Oregon, http://cs.uoregon.edu/ michal/pubs/oracles.html, (2001).

[2] L. Console, G. Friedrich, and D. Theseider Dupre, ‘Model-based diagnosis meets error diagnosis in logic programs’, *IJCAI*, 2, 1494–1499, (1993).

[3] J. E. Cook and A. L. Wolf, ‘Event-based detection of concurrency’, *Proc. ot the* 6^{th} *International Symposium on the Foundations of Software Engeneering (FSE-6)*, 35–45, (1998).

[4] L. de Alfaro and T. A. Henzinguer, ‘Interface automata’, *Proc. of the* 9^{th} *Annual Symposium on Foundations of Software Engineering (FSE’01)*, 109–120, (2001).

[5] E. Fabre, A. Aghasaryan, A. Benveniste, R. Boubour, and C. Jard, ‘Fault detection and diagnosis in distributed systems: an approach by partially stochastic petri nets’, *Journal of Discrete Event Dynamic Systems*, 8(2), (1998). Kluwer Academic Publishers, Boston.

[6] F. Laymann, ‘Web services flow language (wsfl 1.0)’, *IBM Software Group*, (2001).

[7] L. Maruster, T. Weijters, W. van der Aalst, and A. van der Bosch, ‘Process mining: discovering direct successors in process logs’, *Proc.s of the* 5^{th} *International Conference on Discovery Science 2002*, 364–373, (2002). LNCS.

[8] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, ‘Observations and results gained from the Jade project’, *DX-02*, (2002).

[9] B. R. Pekilis and Seviora R. E., ‘Automatic response performance monitoring for real-time software with nondeterministic behaviors’, *Performance Evaluation*, 53(1), 1–21, (2003). Elsevier Science.

[10] F. Plasil, and S. Visnovsky, ‘Behavior protocols for software components’, *IEEE Transactions on Software Engineering*, 28(11), 1056–1076, (2002).

[11] P. Popov, S. Riddle, A. Romanovsky, and L. Strigini, ‘On systematic design of protectors for employing OTS items’, *Proc. of the* 27^{th} *Euromicro Conference*, 22–29, (2001).

[12] T. Savor and R.E. Seviora, ‘Automatic detection of software failures: Issues and experience’, *Proc.* 10^{th} *Euromicro Workshop on Real-Time Systems*, (1998).

[13] J. Thai, B. Pekilis, A. Lau, and R. Seviora, ‘Aspect-oriented implementation of software health indicators’, 8^{th} *Asia-Pacific Conference on Software Engineering (APSEC 2001)*, 97–104, (2001).