Model-based monitoring of software components

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 can hardly be split into separate entities with clear interaction, unless it is modeled it at a very fine level. But 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, and only interacting by message exchange. Since we suppose that the components are black-boxes, the state of a component can only be estimated by observing its external behavior. We present a supervision system which uses a model of the message exchange between the components to continuously evaluate the state of the components. The supervision consists in following the evolution of the components from their interaction models as messages are observed. Our approach detects component degradation or blocking and deals with the problem of message correlation and of possible uncertainty on the component evolution. The supervision is local to each component, so it is well adapted to component-based applications which are often large and evolve rapidly.

2 COMPONENT BEHAVIOR MODEL

The behavior of each component is modeled in a formalism based on Petri nets. The places of the Petri net represent the state of the component. The transitions correspond to the exchanged messages. Figure 2 shows the behavior model of a travel agent component in the interaction scenario described in figure 1. The Petri net is complemented as follows:

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

[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

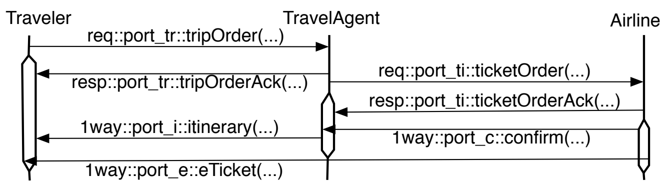
Traveler
TravelAgent
Airline
req::port_tr::tripOrder(...)
resp::port_tr::tripOrderAck(...)
req::port_ti::ticketOrder(...)
resp::port_ti::ticketOrderAck(...)
1way::port_i::itinerary(...)
1way::port_c::confirm(...)
1way::port_e::eTicket(...)

Figure 1. The ticket order application: normal interaction scenario.

Typing 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 different transactions.

Temporal constraints: An occurrence instant is associated with each transition. Temporal constraints between these instants can be associated to the transitions, they are checked each time the transition is fired. The *internal constraints* describe the time that normally takes a component to perform a task between two messages. 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 internal constraint *IC1 : t1-t0 in* $[$*20,100*$]$ can be associated with the transition *trans1*.

Exception handling: We differentiate the *functional* exceptions which appear in the interface specification of component methods from all the exceptions that may happen but that are not explicitly specified in the interface (in general problems due to runtime conditions like resource problems are not considered in the method specification). The functional exceptions are divided into two classes: the exceptions due to the client and the ones due to the server. To deal with the non-functional exceptions and complete the model, a generic type "*UnknownError*" is used. The model must describe if a component may send an error message as a result of a problem occurring when it is calculating and 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).

Messages introspection: Several processes may run in parallel inside the component, the correlation of the messages of a same data flow is guided by the order in which the messages occur, but also by indications which specify if the messages can be correlated by their content. Let's suppose in our example that the travel agent component successively receives two requests: *req::port_tr::tripOrder(triporder#1)* and *req::port_tr::tripOrder(triporder#2)*. If then it emits the message *resp::port_tr::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, the message description can be extended with variables corresponding to particular fields of the message parameters. Let's suppose that the *triporder* and the *ack* parameters contain the traveler

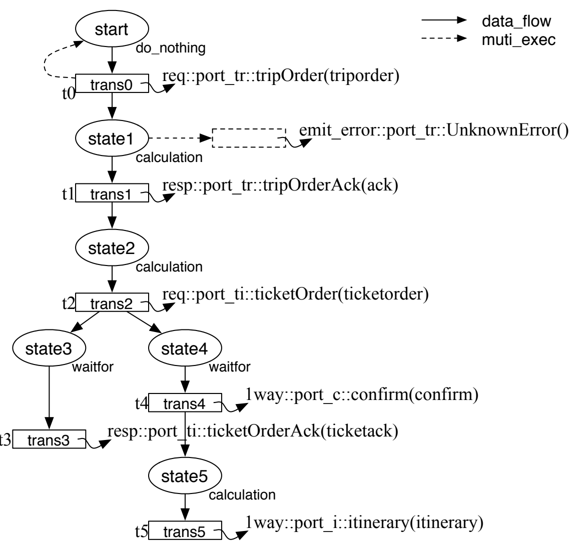
start
do_nothing
data_flow
muti_exec
t0
trans0
req::port_tr::tripOrder(triporder)
state1
emit_error::port_tr::UnknownError()
calculation
t1
trans1
resp::port_tr::tripOrderAck(ack)
state2
calculation
t2
trans2
req::port_ti::ticketOrder(ticketorder)
state3
waitfor
state4
waitfor
t4
trans4
1way::port_c::confirm(confirm)
t3
trans3
resp::port_ti::ticketOrderAck(ticketack)
state5
calculation
t5
trans5
1way::port_i::itinerary(itinerary)

Figure 2. Behavior model of the travel agent component.

ID, we could replace the first two transitions by:
req::port_tr::tripOrder(triporder, x = triporder.getID())
resp::port_tr::tripOrderAck(ack, x = ack.getID()).
The messages will be correlated only if they share the same *x*.

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

3 COMPONENT HISTORY CONSTRUCTION

The supervision is not done directly from the model presented above but from a translation of this model into a set of partial transitions called *tiles*, following the approach in [3] which we have extended in order to take into account attributes with arguments and with non-binary values. Each transition of the Petri net is translated into a tile which describes the partial state change when the transition is fired. The following tile corresponds to transition *trans0*. The tile precondition describes the necessary conditions for the transition to fire, the label corresponds to the observable event and the postcondition describes the values of the attributes after the transition. *t0* is the message occurrence time, *?e* the data flow identifier.

tile *trans0[?e]* ***label:*** *port_req::tr::tripOrder[?triporder,?t0]*

precondition	***postcondition***
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)*

The supervision consists in constructing the possible histories (i.e. the sets of sequential or concurrent state changes) which explain the observed messages. A history is constructed by instantiating and chaining the tiles as messages are observed. An example of possible history for the travel agent component is given in figure 3. As the state of a component may be ambiguous, the set of all possible histories is constructed. During the history construction, statistical information is collected about error emissions (according to the error type) and processing times to detect component degradation or blocking. A precise description of the translation of the Petri net into tiles and of the history construction and analysis can be found in [4].

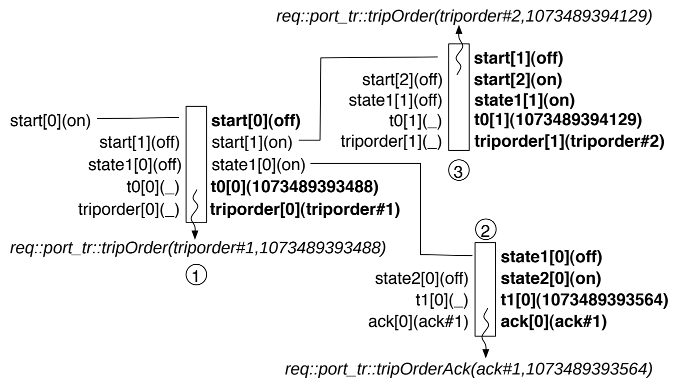

Figure 3. A possible history for the travel agent component. The circled figures indicate the message occurrence order.

4 RELATED WORK

Few works address the problem of black-box component monitoring. Closely related to our work is the work in [8, 5] on the supervision of event-driven, embedded real-time software. The normal component behavior is expressed in the ITU Specification and Description Language SDL. This approach (contrary to ours) is only effective for monitoring session-oriented services (sessions must execute sequentially). Our work is also related to the *oracle* problem for software testing [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”[7]. Works extend the component specification with an interaction protocol [2, 6]. The possibility of using the interaction models for runtime checking is suggested but these works mainly focus on behavior composition, on compatibility checking between interfaces or between interfaces and implementations. Finally, the work in [9], even if it requires the component source code, uses measures similar to ours (error rates, performance) to infer component health indicators.

REFERENCES

[1] L. Baresi and M. Young, ‘Test oracles’, Tech. report, Dept. of Computer and Information Science, Univ. of Oregon, http://cs.uoregon.edu/ michal/pubs/oracles.html, (2001).

[2] L. de Alfaro and T. A. Henzinguer, ‘Interface automata’, *Proc. of the* 9th *Annual Symposium on Foundations of Software Engineering (FSE’01).*

[3] 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*, (June 1998). Kluwer Academic Publishers, Boston.

[4] I. Grosclaude, ‘Model-based monitoring of component-based software systems (poster)’, *Int. workshop on model-based diagnosis DX’04.*

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

[6] F. Plasil and S. Visnovsky, ‘Behavior protocols for software components’, *IEEE Transactions on Software Engineering*, (2002).

[7] P. Popov, S. Riddle, A. Romanovsky, and L. Strigini, ‘On systematic design of protectors for employing ots items’, *Proc. of the* 27th *Euromicro Conference*, (2001).

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

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