Model-based monitoring of software components

Irène Grosclaude

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 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 1 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 donathing can be used for synchronization).

Typing of arcs: The arcs of type dataflow 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 it is 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
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: req::port_tr::tripOrder(triporder)
precondition
start[?e]:(on)
state1[?e]:(off)
triporder[?e]:(on)
t0[?e]:(?)
postcondition
start[?e]:(off)
state1[?e]:(on)
triporder[?e]:(off)
t0[?e]:(?)
```

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].

### REFERENCES


