Abstract

This paper presents an idea to apply Jackson’s JSP method, which is suitable for the transformation problem frame, into the state transition design, presented in the behavior problem frame in Jackson’s problem frames. First, properties of the problem domain are described in state transition rules of events and states. Next, the requirement is described by the relations of domain states which are occurred in the same period of time. Then domain state transition rules are combined into a requirement state transition, and finally, domain events are allocated into the requirement state transition as actions to obtain the machine specification.

1. Introduction

Before designing a software system, it is very important to sufficiently analyze the problem to be solved by the software system for reliable software development. The context equipment and physical entities controlled by the equipment, like gas and liquid, should be carefully analyzed to be completely understood what would happen when some event takes place, especially for embedded software systems. Currently Jackson’s problem frames [1] are widely recognized as an idea to carefully analyze the original problems before considering their solutions, which is essential for reliable software development.

Before the problem frames idea was published, Jackson had proposed the famous two design methodologies, JSP [2] and JSD [3]. The basic concept of JSP is constructing the program structure from its input and output data structures to obtain the solution structure reflecting the original problem solution. JSD, Jackson System Development, may be seen as an enlargement of JSP. In JSD, designer first analyzes the real world, represents it into a kind of concurrent sequential processes, and then allocates required functions into the processes.

Jackson proposed five problem frames [1], required behavior problem frame, commanded behavior problem frame, information display problem frame, simple work pieces problem frame and transformation problem frame. For each problem frame, its own suitable methodology may be applicable to designing the solution. From the problem frames point of view, JSP may be seen as a suitable methodology for the transformation frame, while JSD is for the information display frame. However, no suitable methodology, like JSP or JSD, for the other frames has been proposed.

JSP is only applicable to translation frame, and not to behavior frames which are appeared in embedded software systems, because behavior frame contains controlled domains which may not be treated as input/output data streams as JSP required. Also JSD is applicable to information display frame, and not to behavior frames, because JSD ignores a problem to affect the behavior of real world [4], which should be treated as a main problem in behavior frames.

In this paper, we propose an methodology suitable for behavior frames by applying JSP-like method into behavior frames. We modify JSP to treat controlled domains instead of input/output data streams, state transition diagrams instead of Jackson trees, or regular expressions, and states correspondence instead of data structures correspondence, still keep the original concept of JSP to obtain the solution structure reflecting the original problem structure.

Also, we restrict that the “requirement” in behavioral frame should be described by the domain states, while the “specification” should be described by the domain events, and “domain property” should be expressed with state transition diagram of the events and the states. By this restriction, the specification is obtained almost automatically from the requirement and domain properties. Also, the requirement errors which are unrecognizable by the domain properties and the lack of the specification to consider exceptional events will be checked automatically.

In the followings, the concept of Jackson’s problem
frames and JSP are described in section 2. Then the methodology with JSP flavor for behavior frames is presented with two simple examples, and some discussion and related works are described in section 4. Finally in section 5, we discuss further problems with concluding remarks.

2. Problem frames and JSP

Problem frames are applicable as a method to obtain the software requirements and specification by analyzing problem context, which is controlled by the software, before considering the function and solution of the software. The problem is described as a problem diagram including a machine as solution, domains as problem context, and requirement to express required domain behavior. Among the components, machine belongs to solution, while domains and requirement are problems. Components in problem diagram are connected with interface lines to show that the connected components share phenomena, like events and states, in common. Figure 1 shows problem diagram example.

![Figure 1. Problem diagram example](image)

The interface rule between machine and domains is called "specification," the interface rule between the requirement and domains is called "requirement," and interface rule for a domain is called the "domain property." The domain property is given and described as indicative, while requirement and specification are created by software developers to describe required things as optative. In problem analysis, given things and required things should carefully be classified.

To solve the problem of transformation frame type, Jackson’s JSP is suitably applicable. In JSP, input and output data streams become problem domains, and their structures are represented by Jackson tree diagrams, similar to regular expression, as domain properties. The requirement is expressed as rules of correspondences between input and output data structures to show which part of output is generated from which part of input. Then input and output structures are combined into one structure to keep the correspondences. Finally program parts required for the transformation process are allocated into adequate portions of the combined structure to obtain the final program. Figure 2 show the above concept of JSP in problem diagram.

![Figure 2. Concept of JSP in problem diagram](image)

The final program structure obtained by JSP will reflects the original problem structure since input and output data structures deeply reflect the problem structure. As described above, JSP only treats data streams as problem domains, it is not applicable to current embedded software, where several equipment should be treated as controlled domains. Still the basic concept of JSP where the final solution structure should reflect the original problem domain structures seems to be applicable to embedded software design. The rest of the paper describes an attempt to apply the JSP idea into state transition design of embedded software, or behavioral problem frame.

3. Methodology with JSP flavor for behavioral frame

To apply JSP into behavioral frame, input and output data streams are replaced to controlled domains, and domain properties are expressed with state transition diagrams instead of Jackson tree diagrams. We can consider the correspondence of domain structures as requirement, which guide to combine the domain structures to obtain a skeleton of state transition diagram as a solution structure. Finally necessary event outputs, as process fragments, are allocated into adequate portions of the skeleton state transition to obtain the solution as machine specification.
3.1. Concept of state transition version of JSP

As described, JSP consists of following four stages(Fig 2).

1. Define input and output data structures in regular expressions. (Domain properties)
2. Determine correspondence rules between the input and output structures. (Requirement)
3. Using the correspondence rules, combine the input and output structures into one program structure in regular expression. (Domain properties + Requirement → Specification)
4. Allocate necessary process fragments into the program structure. (Specification)

For state transition design, the elements of above JSP are replaced into the elements of behavioral frame, as follows;

- Input and output data streams are replaced to controlled domains
- Data structure diagram is replaced to state transition diagram
- Correspondence among data structure is replaced to correspondence among domain states.
- Program structure is replaced to machine specification in state transition diagram.
- Combining data structures is treated as combining state transition diagrams.
- Program fragment is considered as event output to control the domain

Here the correspondence of domain states means that problem domains are required to be corresponding states simultaneously. Now JSP-like methodology for behavioral frame may be described as follows,

1. Define domain structures in state transition diagrams as properties.
2. Determine correspondence rules among domain states to request the domains to be those states simultaneously as part of requirement.
3. Using the correspondence rules, combine the domain state transition diagrams into a state transition diagram skeleton.
4. Allocate necessary events output, as process fragments, into the skeleton to obtain the machine specification.

Figure 3 shows the concept of JSP-like methodology for the behavioral frame.

![Figure 3. JSP-like method for behavioral frame](image)

3.2. Domain property

Domain property is described for each domain to specify the property of the domain phenomena shared with machine interface and referred by requirement interface. Machine interface phenomena consists of events to control the domain by machine, as output events, and events to inform domain state by the domain sensor, as input events. On the other hand, the requirement interface phenomena mainly consist of domain states, since the requirement is usually described with the domain states which should be true simultaneously. Here event phenomena are occurred instantaneously while state phenomena become true during a period of time. In the paper, we restrict that machine interface contains only event phenomena and the requirement interface contains only state phenomena for controlled domain.

The domain property is described by a state transition to express the relations between events and states.

- \( Prop = (S, s_0, E, \delta) \)
  - \( S \) : set of states
  - \( s_0 \in S \) : initial state
  - \( E = I \cup O (I \cap O = \emptyset) \) : set of events
    - \( I \) : input events ( with '?' prefix )
    - \( O \) : output events ( with '!' prefix )
• $\delta \subseteq S \times E \times S$: state transition rules

For output event $!e_1$, output transition $(s_1, !e_1, s_2) \in \delta$ means that when the domain is in $s_1$, the machine may send $e_1$ event to the domain to change the state into $s_2$. The state $s_2$ is said to have output transition. For input event $?e_2$, input transition $(s_1, ?e_2, s_2) \in \delta$ means that when the domain is in $s_1$, the domain may send $e_2$ event to the machine, or machine should prepare to receive $e_2$ event from the domain. If it happens, the domain state will be in $s_2$. The state $s_1$ is said to have input transition. If state has input and output transitions, machine may send output event before input event occurs. From machine interface point of view, property state transition may be seen as a protocol description.

Figure 4 shows a simple example of a traffic light unit property state transition same as in [1]. The unit has four states, $Off$ (both lights are off), $Go$ (red is off and green is on) and $Bright$ (both are on), and two events, $RPulse$ to control red light and $GPulse$ to control green light, both are output events from machine. Note that Jackson introduce undefined state(?) instead of our $Bright$ state, while we treat the traffic light unit with two red and green lights which are controlled separately. Using bright state or not is depend on the requirement, not on the property.

For a more complicated example of property state transition, figure 5 show a example of caller telephone domain property in telephone switching system. When the domain is $Conversation$ state, $?onHook$ input event may occurred from the domain to make it $Idle$ state. On the same time, machine may send $!stopConv$ event to finish the conversation and move to $WaitonHook$ state. The $!stopConv$ event may be used to inform that the other party send $onHook$ to finish the conversation, however this is not belong to the domain property but to the requirement. When the domain is in $Dialing$ state, machine receives $?dial$ event without state changes. Simultaneously machine may send $!call$ event to change the state into $Calling$, or send $!busy$ or $!error$ to stop the $Dialing$. Note that, $!busy$ and $!error$ events change the state into the same state, $WaitonHook$, but generate deferent message tones. Also, the diagram shows that $?dial$ event is occurred only when the state is $Dialing$. This means that when the domain is not in $Dialing$ state, $?dial$ event is never occurred or ignored [1].

![Figure 4. Traffic light unit property](image)

**Figure 4. Traffic light unit property**

For a more complicated example of property state transition, figure 5 show a example of caller telephone domain property in telephone switching system. When the domain is $Conversation$ state, $?onHook$ input event may occurred from the domain to make it $Idle$ state. On the same time, machine may send $!stopConv$ event to finish the conversation and move to $WaitonHook$ state. The $!stopConv$ event may be used to inform that the other party send $onHook$ to finish the conversation, however this is not belong to the domain property but to the requirement. When the domain is in $Dialing$ state, machine receives $?dial$ event without state changes. Simultaneously machine may send $!call$ event to change the state into $Calling$, or send $!busy$ or $!error$ to stop the $Dialing$. Note that, $!busy$ and $!error$ events change the state into the same state, $WaitonHook$, but generate deferent message tones. Also, the diagram shows that $?dial$ event is occurred only when the state is $Dialing$. This means that when the domain is not in $Dialing$ state, $?dial$ event is never occurred or ignored [1].

![Figure 5. Example of simple caller telephone property](image)

**Figure 5. Example of simple caller telephone property**

### 3.3. Correspondence rules among domain states

Correspondence rules among domain states constraints that domains should be in those states simultaneously, as a part of requirement. A correspondence rule is defined as $\{d_1, s_1, d_2, s_2, ..., d_n, s_n\}$, where $d_i$ ($1 \leq i \leq n$) is problem domain name and $s_i$($1 \leq i \leq n$) is a state in domain di, showing that in some period of time $d_i$ should be in $s_i$($1 \leq i \leq n$) simultaneously. For example, in the two-way traffic problem [1], two signal light units, U1 and U2, appear as domains. The domain has four states, $Off$, $Stop$, $Go$ and $Bright$, as in Fig.4. The correspondence rules contains three rules, $\{U1.Stop, U2.Stop\}$, $\{U1.Go, U2.Stop\}$ and $\{U1.Stop, U2.Go\}$. This requests that unit state $Off$ and $Bright$ should not be used and $\{U1.Go, U2.Go\}$ should not be used, as requirement.

In telephone switching system problem, given domains are caller telephone TelA, callee telephone TelB, network NW and billing system BS. Here caller and callee are not explicitly designated as domains but considered as a part of
telephones. Figure 6 shows problem diagram of telephone switching system.

![Problem diagram for telephone switching system](image)

**Figure 6. Problem diagram for telephone switching system**

When a telephone call is made, TelA is in **Calling**, TelB is in **Ringing**, Network is in **Reserved** and Billing system is in **Stop** states, as state correspondence rule. When in conversation, TelA and TelB are in **Conversation**, Network is in **Connect** and Billing system is in **Billing**. Table 1 shows an example of state correspondence rules for telephone switching system requirement.

<table>
<thead>
<tr>
<th></th>
<th>Idle</th>
<th>Dialing</th>
<th>Calling</th>
<th>Converse</th>
<th>Wait A on Hook</th>
<th>Wait B on Hook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tel A</td>
<td>Idle</td>
<td>Dialing</td>
<td>Calling</td>
<td>Converse</td>
<td>Wait A on Hook</td>
<td>Wait B on Hook</td>
</tr>
<tr>
<td>Tel B</td>
<td>Idle</td>
<td>Dialing</td>
<td>Calling</td>
<td>Converse</td>
<td>Wait A on Hook</td>
<td>Wait B on Hook</td>
</tr>
<tr>
<td>NW</td>
<td>Idle</td>
<td>Dialing</td>
<td>Calling</td>
<td>Converse</td>
<td>Wait A on Hook</td>
<td>Wait B on Hook</td>
</tr>
<tr>
<td>BS</td>
<td>Idle</td>
<td>Dialing</td>
<td>Calling</td>
<td>Converse</td>
<td>Wait A on Hook</td>
<td>Wait B on Hook</td>
</tr>
</tbody>
</table>

**Table 1. State correspondence rules for telephone switching system**

Those examples show that the domain state correspondence rule may become simple but an effective way to express requirement.

### 3.4. Combining state transition diagrams

A set of state correspondence rules described in 3.3 determines the requirement to the problem domains to constraint the domain state combinations. For each rule, requirement states may be defined. The domain state transition diagrams will be combined into a requirement state transition diagram, as follows.

1. determine the requirement transition
   The order relations among requirement states, like state $S_1$ is followed directly by state $S_2$, should be determined. A requirement state should be selected as an initial state. By the order relations, all requirement states should be reached from the initial state. The order relation may be seen as transition rules among requirement states.
   
   For example in two-way traffic problem, four requirement states, BothStop1 and BothStop2 for $\{U_1.Stop, U_2.Stop\}$, Go1 for $\{U_1.Go, U_2.Stop\}$ and Go2 for $\{U_1.Stop, U_2.Go\}$, are introduced. The initial state is BothStop1, then BothStop1 $\rightarrow$ Go1, Go1 $\rightarrow$ BothStop2, BothStop2 $\rightarrow$ Go2 and Go2 $\rightarrow$ BothStop1 are the order relations, as shown in Fig. 7. Here, BothStop1 and BothStop2 have the same domain states, but different following states. All transitions in Fig. 7 are triggered by timeout events, since no input events from the domains occur.

2. check inconsistency between requirement and domain properties
   The requirement state transition in (1) should be inconsistent with the domain properties, or domain state transition diagrams. The requirement state transition is inconsistent with the domain properties, if and only if followings are satisfied, where $D_1, \ldots, D_n$ are domains, $D_i.s_j$ is domain state for $D_i$, $S_1, \ldots, S_m$ are
requirement states, \( D_i(S_k) \) is the state of domain \( D_i \) when whole system is in \( S_k \) requirement state, and \( S_a \rightarrow S_b \) is a requirement state transition.

(a) For any domain \( D_i \), \( D_i(S_b) \) is reachable from \( D_i(S_a) \) by \( D_i \) property.

(b) If \( e_1, e_2, \ldots, e_h \) is a sequence of events in \( D_i \), which should be occurred during the transition from \( D_i(S_a) \) to \( D_i(S_b) \), \( e_1 \) may be input or output event but all other \( e_j (2 \leq j \leq h) \) should be output events.

(c) No or only one domain may contain an input event for each requirement state transition.

By 1, 2 and 3, any requirement state transition may be specified, from machine point of view, by \( \text{state} \rightarrow \text{[input]/[output]} \ldots \rightarrow \text{state} \) which lead usual UML state machine form, where output events be treated as actions. Actions will be described in 3.5.

If in the above, \( e_1, e_2, \ldots, e_h \) contains more than one input event, new requirement states should be introduced just before each input event to wait for the input, and determine the domain states for the new requirement states, to obtain the requirement state transition inconsistent with domain properties. If in the above, \( D_i(S_b) \) is not reachable from \( D_i(S_a) \) in \( D_i \) property, the requirement may not be realizable in the problem domains. In this case, the requirement, or sometimes domain property, should be modified.

3. check completeness of requirement transition

For each requirement state \( S_j \) and for each input event \( e_i \) possibly occurred at \( S_j \), requirement state transition from \( S_j \) triggered by \( e_i \) should be defined. For each requirement state \( S_j \) and domain \( D_i \), all possible input events occurred at \( D_i(S_j) \) may be obtained by \( D_i \) property. If no transition is defined from \( S_j \) triggered by those input events, add transition from \( S_j \) to make requirement transition complete. For switching example, in additional to \( \text{dialing} \rightarrow \text{ringing} \), \( \text{dialing} \rightarrow \text{idle} \) should be added, since \( ?\text{onHook} \) input event may occur at \( \text{dialing} \) state by A telephone property shown in Fig. 5.

Figure 8 shows an example of requirement state transition obtained from table 1, which is inconsistent and complete with the domain properties. Here \( S_a \rightarrow D_i?e_k[c] \rightarrow S_b \) shows a transition from \( S_a \) to \( S_b \) triggered by input event \( e_k \) from domain \( D_i \) when condition \( c \) holds.

Note that, in [1], event \( e_k \) generated by domain \( D_i \) is described by \( D_i?e_k \), and \( e_k \) generated by machine \( M \) is described by \( M!e_k \). Here, we use \( D_i?e_k \) and \( D_i!e_k \) for input and output event of \( D_i \), respectively, from the machine viewpoint.

![Figure 8. Requirement state transition diagram for telephone switching system](image)

In the requirement state transition diagram, only input event or timeout event may appear as trigger event, while input and output events appear in domain property state transition diagram. Domain property describe the possibility of event occurrences at each state, as indicative. On the other hand, requirement state transition describe that when domains are such states, and such input event occurred at such domain, then domains will be requested to be in such states, as optative. Note that all input events in Fig.8 occur from A and B telephone domains reflecting the subscriber actions as onHook, offHook and Dial, which should be considered as problem phenomena. This guarantee that the requirement state transition diagram shown in Fig. 8 is purely a problem, not a solution.

### 3.5. Allocating output events as action fragments

In inconsistency checking between requirement and domain properties, output events which are necessary to change state of each domain may be obtained. This output events should be allocated to the corresponding requirement state transition to obtain the specification.

If more than one paths exist from \( D_i(S_a) \) to \( D_i(S_b) \) in domain \( D_i \), select one by considering more detail domain characteristics. For example in two-way traffic system, Figure 4 shows that there are two possible paths from \( \text{Stop} \) state to \( \text{Go} \) state, one through \( \text{Off} \) and one through \( \text{Bright} \). In [1], \( \text{Bright} \) is not permitted and only one path, \( \text{Stop} \rightarrow \text{Off} \rightarrow \text{Go} \), is possible. Even if \( \text{Bright} \) is permitted, we select \( \text{Stop} \rightarrow \text{Off} \rightarrow \text{Go} \) path since this seems to
be more economical.

The order of domain event output should be determined by detail requirement consideration. For example, from ringing to conversation state transition in telephone switching system, three domain output events are needed, to connect network, to start billing, and to stop ring tone A for conversation. Which event should be first? Telephone company may want to start billing first, while network engineer wish to connect network first. In many cases, the order of event output may be ignored since most of the events can be processed instantaneously much faster than human activities.

Requirement state transition with output events becomes "specification," which specify the machine behavior. The output event allocation stage may be said to generate specification from requirement and domain properties.

Figure 9 shows two-way traffic specification state transition diagram obtained from Fig. 4 and Fig. 7. Here initial actions, Sig1!RPulse and Sgg2!RPulse, are described to make both signal Stop in initial transition with no trigger (begin from small black circle).

**Figure 9. Specification state transition diagram for two-way traffic**

Figure 10 shows telephone switching specification state transition diagram obtained from Fig.5, Fig.6, Fig.8 and other domain properties. A condition box is added to check the result of dial event. This diagram simplify the real switching system just for understandability, still leave the basic concept of original problem. To specify more detail behavior, domain properties should be described in detail first, then requirement referring the elaborate properties should be described.

**Figure 10. Specification state transition diagram for telephone switching system**

4. Discussion and Related works

The above section expressed that our JSP-like method may be applicable to a behavioral problem frame where state transition diagrams be used for domain property description. Jackson [1] classifies problems into five frames, such as required behavior, commanded behavior, information display, simple work pieces and transformation frames. JSP is suitable for transformation frame and JSD is for information display frame. We claimed that our JSP-like method for state transition diagram design is applicable method for behavior frames.

If all domain properties could be described in state transition diagrams, the JSP-like method could show how to obtain the requirement inconsistent and complete to the problem domains, and how to obtain the specification which satisfies the requirement under the constraints of domain properties. Compared to original JSP, combining structures stage seems to be much complicated. However, if the correspondences between domain properties and domain state are determined carefully, the rest of stages including combining structures, checking completeness and allocating necessary output events, may be completed by a systematic way or may be obtained with semi-automatic computer aided tool.

The JSP-like method is similar to the concept in [5] where domain and properties are correspond to port and protocol, respectively. Here, we can explicitly express the context in which our system may be applicable, while [5] did not consider the problem context.

Research on automatically generating specification from requirement is given in [6] and [7]. Both [6] and [7] use first ordered predicate calculus, and CSP [8] like formal approaches to express domain property and requirement, and mathematical theorem-proving method is applied to transform requirement to specification. We believe that the state
transition concept, like UML state machine [9] and SDL state diagram [10], is more familiar and applicable than predicate calculus to average software engineers, especially in embedded software area where many problems may be classified in behavior frame.

5. Conclusion

A state transition design method based on the JSP concept has been described. The method is applicable for behavior problem frame in embedded software systems where domain properties are expressed with state transition diagram including domain states and events. The requirement may be described in terms of domain state correspondence rules showing which state of each domain could occur simultaneously. Using the correspondence rules, domain state transition diagrams are combined into a requirement state transition diagram, and finally necessary events, as action fragments, are allocated into the combined state transition diagram being similar to JSP.

We have applied our method to analysis and design of simple embedded software examples including automatic vending machine, ATM, simple telephone switching system, watching machine, and rice cooker, and recognized that the method is suitable for these problems since requirement have been effectively described in terms of domain states, while specification have been described in terms of events. Also, domain state correspondence rules have been very helpful to understand the required behavior.

We plan to apply the method to non trivial problems to evaluate the effectiveness and difficulties of the method. Also, as discussed previously, computer aided support system for the method will be developed to semi-automate the design processes.

References