Event-based integration using on-the-fly matching

Anders Moen Hagalisletto
Birkeland Innovation, University of Oslo
Norwegian Computing Center Norway
andersmo@ifi.uio.no

Steinar Kristoffersen
Østfold University College
Halden, Norway
steinkri@ifi.uio.no

1. INTRODUCTION

Integration of applications is costly and cumbersome. Integration strategies based on standardization, common interfaces or hubs alleviate the situation somewhat, but at the cost of reduced functionality at the interface to the least common multiple [2]. In this paper, we propose an approach that automates the integration by matching incoming messages at runtime. The core of the approach is an efficient pattern matching [1] algorithm based on abstract, event-based state machines, which we describe in this paper. The method is illustrated by two agents running a session of Blackjack on the Internet. Initially incompatible agents synchronize their implementations at runtime with minimal computational cost. On-the-fly event-based integration thus comes across as a promising technology for automating existing manual and expensive processes.

2. DISTRIBUTED BLACKJACK

As a case study we have specified a scenario consisting of mobile phones with *player* agents that connect to a server, hosting the role of a *dealer* at a casino. The players and the dealer all have their own, unique implementation of the application. They have different commands, and potentially different interpretations of the game's rules. For convenience we assume that some basic concepts, like *decks* of *cards* and *chips* (for betting) are uniformly represented as universal types, albeit with different representations of the commands that deal with these entities.

Blackjack is a simple game, which involves betting between the the player and the bank about getting a score of the cards that is closest to 21, by drawing cards from a deck comprising multiples of 52 standard playing cards. An ace scores either a 1 or 11, kings, queens and jacks count 10 and all other cards maintain their numerical value. The demonstrator presented in this paper follows standard Blackjack with minor and insignificant exceptions.

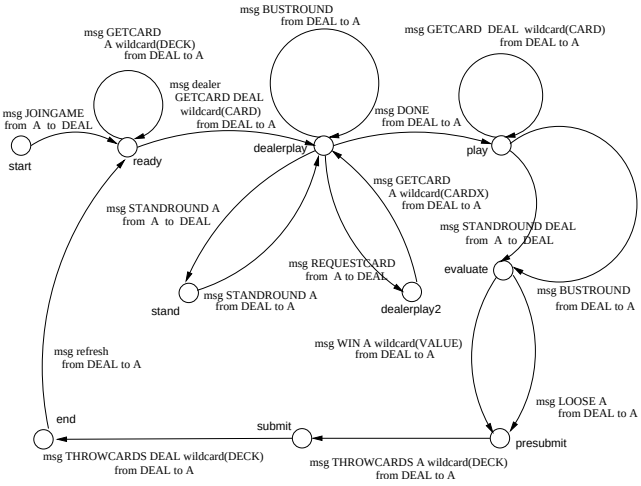

Figure 1: Dealer's application graph.

2.1 The application graphs

The application graphs, which represent the abstract behaviour of each application, can be conceptualized as finite state machines. Figure 1 shows a dealer's perspective of the Blackjack game. In the beginning the dealer is waiting for agents that register for a game session. The dealer *DEAL* must receive the message

$$\mathsf{msg}\ \text{JOINGAME}\ \mathsf{from}\ A\ \mathsf{to}\ DEAL$$

from the player A in order to proceed. The current game may include several players, therefore the dealer gives pairs of cards to all the participants as described by the reflexive transition in the ready state. After the initial dealing of cards the dealer picks two cards in a similar way ($\langle \mathsf{ready}, \mathsf{dealerplay} \rangle$), and in state dealerplay, there are three options. Either the player stands, busts or requests and gets one more card. When the players are done with their active part, the dealer notifies the players by sending a DONE message, and the initiative of the game is transferred to the dealer in state play. The dealer has now three options. Either to get one single card, since the current card value permits it, or stand since it is too risky to pick another card, or bust if the value of the deck is exceeding 21. In the two latter cases, the participants in the game are ready to evaluate. There are two options, either the player wins

or the player looses, and the result of the evaluation is sent to the player. Finally the table is closed, the players throw their cards $\langle \mathsf{presubmit}, \mathsf{submit} \rangle$ and then the dealer throws the cards $\langle \mathsf{submit}, \mathsf{end} \rangle$. The player might want to play another round, this is captured by the final transition,

$$\mathsf{msg}\,\text{refresh}\ \mathsf{from}\ \mathit{DEAL}\ \mathsf{to}\ A.$$

Observe that this final event is an *internal* event: Despite the fact that the message looks like a transmission from DEAL to A, no message is sent. The matching may ensure that the matches that have been made are not reset, so that they can instead be used for preparing the host application for the matching of potentially new rounds of Blackjack.

3. APPLICATION GRAPH MATCHING

When two agents synchronize their behaviour in a message-oriented framework, they rely on the application graphs to decide that a reduced subset of possible matches are, indeed, possibly legal.

Formally, an *application graph* is a four-tuple $A = \langle I, N, E, U \rangle$, consisting of the name of the application I, a finite set of nodes N, a set of transitions (edges) E, and a designated node U, called the current node. Hence formally $N = \{n_1, \ldots, n_k\}$, $U \in N$, and each transition $e \in E$ is of the form $\langle n_i, n_j, (\mathsf{msg}\, C\ \mathsf{from}\ a\ \mathsf{to}\ b) \rangle$, where $n_i, n_j \in N$, a, b are agent names and $C \in \mathscr{L}$. The message content is interpreted as a sequence of basic elements. Two sequences of message contents C_1 and C_2 are compared with respect to the current matching table $T = \langle T, b \rangle$ and the state of the application graph $A = \langle a, N, E, u \rangle$. Hence, first all the transitions starting from u is collected into the set of potential matching candidates. Then each transition is matched with the current concrete message, using the function $\mathsf{matchC?}$.

DEFINITION 1. *The function* $\mathsf{matchC?}$ *takes two message contents* C_1 *and* C_2*, and a matching table* T *as input and returns true if the contents match:*

$$\begin{array}{ll}
(i) & \mathsf{matchC?}(C_1, C_1, T) = \top \\
(ii) & \mathsf{matchC?}(e_1, e_2, T) = \mathrm{con}(\langle e_1, e_2 \rangle, T) \\
(iii) & \mathsf{matchC?}(e_1 _ C_1, e_2 _ C_2, T) = \\
 & \qquad \mathsf{matchC?}(e_1, e_2, T) \wedge \mathsf{matchC?}(C_1, C_2, T)
\end{array}$$

The first clause states that identical message contents match. The second clause says that two basic elements e_1 and e_2 match if the matching pair $\langle e_1, e_2 \rangle$ can be consistently added to the matching table T. The final clause states that composite message contents are matched from left to right. If a message matches with any of the application graph's current state-transitions or the existing matches that were made previously, then the matching can be executed. In practice this means that (i) the matching table is updated with possibly new matching pairs, (ii) that the application graph is adjusted to contain names of other agents and the indexes of any variables are reset and (iii) that the message is translated into the host component's language, based on the matching table E. The execution of matching inside an agent, denoted $\mathbb{M}(C_1, C_2, \langle b, T \rangle, A, W, t)$, matches two contents C_1 (message content) and C_2 (graph content) with respect to its application graph.

DEFINITION 2. *Suppose that* a *is the host agent and* b *is the communicating component. Then perform match is defined by:*

$$\begin{array}{ll}
(i) & \mathbb{M}(\epsilon, \epsilon, b_T, \langle a, N, E \cup \{\langle n_1, n_2, \mathsf{msg}\, C\ \mathsf{from}\ t_1^A\ \mathsf{to}\ t_2^A \rangle\}, n_1 \rangle, \\
 & \qquad W, \langle n_1, n_2, \mathsf{msg}\, C\ \mathsf{from}\ t_1^A\ \mathsf{to}\ t_2^A \rangle) = \\
 & \qquad \{b_T\} \cup \{\langle \mathrm{a}, N, E \cup \{\langle n_1, n_2, \mathsf{msg}\, C\ \mathsf{from}\ t_1^A\ \mathsf{to}\ t_2^A \rangle\}, n_2 \rangle\} \\
 & \qquad\qquad \textit{if } t_1^A = a \vee t_2^A = a \\
(ii) & \mathbb{M}(e _ C_1, e _ C_2, b_T, A, W, t) = \mathbb{M}(C_1, C_2, b_T, A, W, t) \\
(iii) & \mathbb{M}(e_1 _ C_1, e_2 _ C_2, \langle T, b \rangle, A, W, t) = \\
 & \qquad \mathbb{M}(C_1, C_2, \langle b, \{\langle e_1, e_2 \rangle\} \cup T \rangle, A, W, t) \\
 & \qquad\qquad \textit{if } e_1 \neq e_2 \wedge \neg \mathrm{wildcard?}(e_2) \\
(iv) & \mathbb{M}(e_1 _ C_1, \mathrm{wc}(w) _ C_2, \langle T, b \rangle, \\
 & \qquad \langle b, N, E \cup \{\langle n_1, n_2, \mathsf{msg}\, C\ \mathsf{from}\ t_1^A\ \mathsf{to}\ t_2^A \rangle\}, n_1 \rangle, \\
 & \qquad\quad \mathrm{wcg}(G), \mathsf{msg}\, C\ \mathsf{from}\ t_1^A\ \mathsf{to}\ t_2^A) = \\
 & \qquad \mathbb{M}(C_1, C_2, \langle b, \{\langle e_1, \star(w, G+1) \rangle\} \cup T \rangle, \\
 & \qquad\quad \langle b, N, E \cup \{\langle n_1, n_2, \\
 & \qquad\quad \mathsf{msg}\,\mathrm{sub}(\langle \star(w, G+1), \mathrm{wc}(w) \rangle, C)\ \mathsf{from}\ t_1^A\ \mathsf{to}\ t_2^A \rangle\}, n_1 \rangle, \\
 & \qquad\quad \mathrm{wcg}(G+1), \mathsf{msg}\,\mathrm{sub}(\langle \star(w, G+1), \mathrm{wc}(w) \rangle, C)\ \mathsf{from}\ t_1^A\ \mathsf{to}\ t_2^A) \\
 & \qquad\qquad \textit{if } t_1^A = a \vee t_2^A = a
\end{array}$$

The algorithm can be explained as follows: Clause (i): If both message contents are empty, then the matching has succeeded and the current pointer is moved to the next state n_2. The condition expresses that the active transition might involve send or receive events. If the message contents C_1 and C_2 are non-empty, then there are three cases: Clause (ii): In case of two matchable messages, then continue to match the remaining contents. Clause (iii): If the initial elements are different and the second element is not a wildcard, then it is added to the table before proceeding the match. Clause (iv): If the initial element in the the graph part of the content is a wildcard, then the wildcard is refreshed, and the message element is matched with the refreshed message element. This regards both send and receive events.

4. LESSONS LEARNED

A specification of the game Blackjack was used to calibrate the matching algorithm. A scenario with one player and a casino was implemented, including a network (asynchronous communication) and a standard stock of 52 cards. With the pertaining application graphs, an initial request from the client to join the game resulted in a simulation where the casino used all the cards in totally eight and a half rounds. In the end the casino ran out of cards. The matching session resulted in 9 matching tables, where the first eight captured the successful matchings, while the final one resulted in an incomplete match.

Future work will describe how the framework can be used to synchronize several agents that communicate using distinct notations for the data elements, as well as backtracking to repair mismatching, arising from nondeterminism.

5. REFERENCES

[1] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann Publishers, Inc, 1992.

[2] Jeff Sutherland and Willem-Jan van den Heuvel. Enterprise application integration and complex adaptive systems. *Commun. ACM*, 45(10):59–64, 2002.