Pervasive integration by autonomous agents

Anders Moen Hagalisletto
The Norwegian Computing Centre
Gaustadalléen 23
N-0323 Oslo, Norway
Email: anders.moen@nr.no

Steinar Kristoffersen
Østfold University College
Faculty of Information Technology
N-1757 Halden, Norway
Email: sk@hiof.no

Abstract—Integration of applications is costly and cumbersome. We propose an approach to automate integration among pervasive, autonomous agents, in which internal process specifications are matched with incoming messages at runtime. The core of the approach is an efficient matching module, which we demonstrate to be working well for a simple game of Blackjack played between agents that have different implementations. Finally, we outline some strategies for dealing with errors which arise when the algorithm has to choose between alternatives.

I. Introduction

According to forecasting companies, systems integration projects worldwide cost approximately 85 billions per year. This surpasses many estimates of how much the actual application development market in the first place is worth in total (AMR Research). Another crucial remark made by many observers concerns the high rate of failure in integration projects. According to Forrester, as many as 65% of the integration projects are not completed on time and budget. Although the rationale of integration as well as the risks pertaining to these projects seem to be well-known [5], there exists no proven approach to support it. State-of-the-art solutions based on COM/DCOM and Corba turn out to be inflexible [6]. More loosely coupled solutions based on web-services handle document structures well, but do not facilitate the distribution of objects [7].

Another approach to coping with the seemingly intractable nature of integration projects, is event-based software integration [1]. It is characterized by interacting components, which broadcast events pertaining to their integration needs. It still suffers, however, from the lack of a common, consistent model of integration [1].

We approach this problem by looking at integration as an implicit consequence of pattern matching the process behavior of each component. We are going to talk about agents exchanging messages, and will demonstrate ad-hoc, automatic integration, which extends an event-based model of integration to cover scenarios where mutual capabilities are not known in advance.

In order to do this, we have to clarify some initial assumptions about the underlying system that is modeled, and how the matching is configured:

(A) Component isolation: The systems to be integrated consist of isolated components. That is, there might insofar as we know at runtime, be no previous dependencies or shared knowledge between the components.

(B) Asynchronous communication: The components communicate using asynchronous messages transmissions over a network. The network might be a perfect link or a complex topology of links with delays, package loss, etc.

(C) Local view: There is no “objective” view on the process, the matching can only give a relative view on the surrounding environment. Everything the component knows about its environment, it has learned through communication. A component a might never know whether the other participating components have followed the rules of an application, it can only know whether another component b has interpreted a’s interactions in an appropriate way consistent with b’s rules, or not.

(D) Simple applications: The applications that can be handled in our integration framework are relatively simple, the high level specification of the applications at hand should be specified as event-driven finite state machines. For the purpose of this paper explicit loops may only occur once.

(E) Explicit reflection of behavior: The two components possess a high level representation of its external behavior. In practice this means that each component possesses an event-driven finite state machine that contains the messages to be communicated through an execution of the application.

(F) Always integrating: For convenience we assume that the components are always in an integration mode. This means that every received message is tested to be an event in the matching session of the host component. We further assume that there is only one application that is synchronized between the two components.

Two consequence of assumption C can be drawn: i An ongoing integration process can fail at anytime in the future. ii The agents can never be sure that the process is sound, by which we mean that the two participants follows each others intentions.

The paper is organized as follows: In Section II, we describe the architecture of the matching module, basic
terminology, algorithms and message flow. In Section III, we show how two agents might run an application of Blackjack, despite the fact that the language they use to represent the application and the syntax of message contents differ. In the matching framework, incoming messages might be misinterpreted: There might be situations where a received message is interpreted incorrectly, causing succeeding inadequate interactions between the agents. Techniques to repair mismatching of applications that are potentially synchronizable are discussed in Section IV, and a best effort protocol for error recovery is presented in detail. The approach is then evaluated in Section V. Finally in Section VI we draw some conclusions of our approach and point to some future research directions based on our framework.

II. Distributed Matching

When two agents synchronize their behavior in a message-oriented framework, then the notion of pattern matching, known from AI and logic [4], is lifted to a distributed setting. Pattern matching is well-understood and widely used for various purposes in computer science. The challenges with distributed pattern matching is contained in assumption C: Two agents or components maintain their own local view on the global system.

A. Message flow

Figure 1 shows the entire message flow from a component (agent) a to a component (agent) b. Let M denote the message to be transmitted. The agents have two sets of buffers (FIFO), input output and corresponding input-matching and output-matching buffers. The sender a first puts a message to be transmitted into the out buffer. The first elements are taken out and taken over by the output-matching buffer, and tagged with the operator matchmsg M , which reads “M is ready to be matched”. The rightmost (final) message is checked for potential match. If it does match, then the tag is removed and pushed out into the network. The receiver B picks up the message, and places it in the input buffer. The messages are then pushed input-matching buffer tagged with matchmsg M . If the message match with the current matching table and application graph, then the tag is removed and the message is ready for processing by client b.

Note that messages to be sent is also matched with the host component’s application graph. This is necessary since the application graph of the sender is not explicit about the receiver of the message and the concrete values to be transmitted is not hard-coded into the graph.

B. Technical preliminaries

In this section we present the basic concepts of the matching framework, the matching language and the basic matching data structures. A message is written

\[ \text{msg } C \text{ from } a \text{ to } b, \]

where a denotes the sender, b denotes the receiver and C denote the message content. The message content is a sentence in the matching language \( \mathcal{L} \). The language \( \mathcal{L} \), is the least language such that

\[ i \ L \]

\[ ii \ x_1, x_2, ..., x_n \ L \]

\[ iii \ x \text{ is a variable, then } w x \ L \]

\[ v \ v \in \mathcal{L} \]

\[ vi \text{ If } t_1, t_2 \mathcal{L}, \text{then } t_1 \circ t_2 \mathcal{L} \]

Concatenation

Constants, variables and wildcards are the basic elements of \( \mathcal{L} \). Constants are message elements that do not change during an execution of the application. Variables are of two kinds, standard variables and wildcard variables (or wildcards for short). Wildcards are reset for each assignment. Agent names, that are used in the from to header of a message are also constants. A matching pair, is a pair \( e_1, e_2 \), where both \( e_1 \) and \( e_2 \) are basic elements. The substitution operator, sub \( e_1, e_2, C \), takes a pair of basic elements \( e_1, e_2 \), and replaces each occurrence of \( e_2 \) with \( e_1 \) in the content C. A matching table is a set of matching pairs, \( T = e_1, e_2, ..., e_i, e_j \). We can define the function \( e \in_2 T \) : checks if element \( e \) occurs as a second-element in the matching table \( T \) and the function constmatch? \( e, T \) : checks whether element \( e \) is matched with a constant in \( T \). We let wildcard? \( e \) denote the boolean function that is true if \( e \) is a wildcard. An agent table is a pair \( b, M \), such that \( b \) is a agent name and \( T \) is a matching table. A transition is a triple \( n_1, n_2, l \), where \( n_1, n_2 \) are nodes and \( l \) is a label. A transition is reflexive, if \( n, n, l \). A labeled graph is a pair \( N, E \), where \( N \) is a set of nodes and \( E \) is a set of transitions. A path from a node \( n_1 \) to a node \( n_k \), written \( n_1 \rightarrow n_k \), is a sequence \( n_1, ..., n_k \), such that \( n_1, n_1, 1, l \). A graph is connected if for every \( x, y \), \( x \rightarrow y \) or \( y \rightarrow x \), or \( z x \rightarrow z \rightarrow y z \). A connected graph is cyclic if there are two distinct nodes \( x \) and \( y \), such that \( x \rightarrow y \) and \( y \rightarrow x \). A connected graph is merging if \( x y z x \rightarrow z \rightarrow y z \). An application graph is a four tuple \( A = I, N, E, U \), 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 \( N = n_1, ..., n_k \), \( U = N \), and each
1) Matching message content: 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 $T$, $b$ and the state of the application graph $A = a, N, E, u$. 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 $\text{matchC}$.  

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

$$\text{matchC}(C_1, C_2, T)$$

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 $e_1, e_2$ can be consistently added to the matching table $T$. The final clause states that composite message contents are matched from left to right. While the two first clauses are unquestionable, the final one, $\text{iii}$, is discussed in Section V.

2) Consistency: Which pairs that are treated as consistent with respect to a specific matching table turns out to be of crucial for the definition of matching. The notion of consistent augmentation of the matching table balance on a thin line: A too strong notion would typically rule out reasonable matchings, while a too weak matching criterion might cause invalid matches. Let $T$ be a matching table, and let $e_1, e_2$ denote a matching pair. Observe that the first element in a matching pair always originates from the concrete message to be processed, while the second element implicitly originates from the application graph possessed by the agent. We call the first element $e_1$, the message part, while the second element $e_2$ is called the graph part.

Definition 2: We say that a matching pair $e_1, e_2$ is consistent with respect to a matching table $T$, denoted $\text{con } e_1, e_2, T$, iff either

$$\text{matchC}(e_1, e_2, T)$$

Hence this means that a matching pair $E = e_1, e_2$ is consistent with the table $T$, $\text{i}$ if $E$ exists already, expressed by clause $\text{ii}$, the element in the application graph is a wildcard (hence everything should match), or finally $\text{iii}$ that the element in the application graph is not a wildcard, the graph part has not been matched before and the message part does not already match a constant in $T$, corresponding to clause.

Observation 1: $\text{con } e_1, e_2$.

Proof: It follows from $\text{ii}$ and $\text{iii}$, then there are two options: either $e_2$ is a wildcard or it is not. If
nullcard? $e_2$, con $e_1, e_2 , T$, in particular $T$. If not wildcard? $e_2$, then obviously $e_1, e_2$ and $e_2$, and additionally not constmatch? $e_1$, which implies con $e_1, e_2$.

Our notion of consistent augmentation of matching pairs is only one out of potentially several notions. If the matching table $T$ is interpreted as a mapping with domain $\text{Dom} e_1 e_1, e_2 T$ and codomain $\text{CoDom} e_2 e_1, e_2 T$, then we observe that $T$ is not even a function since we might have elements $e$ and $e$ such that both $e, e_2, e, e_2 T$. To some extent, the interpretation of consistency is a pragmatic issue. The notion at hand defines a class of applications, and might be unsuitable to other classes of applications.

3) Performing matching: If a message match the application graph and the current matching table, then the matching can be executed. In practice this means:

1) the matching table is updated with possibly new matching pairs,
2) the application graph is adjusted to contain names of other agents and the indexes of the wildcards are reset using the wildcard generator,
3) the message is translated into the host component’s language, based on the matching table $E$.

The execution of matching, denoted $\mathbb{M} C_1, C_2, b, T, A, W, t$, matches two contents $C_1$ (message content) and $C_2$ (graph content) with respect to an agent table $b, T$, an application graph $A$, a wildcard generator $W$, and the principal transition $t$ as input, and returns a revised agent table, application graph, and wildcard generator. The wildcard generator is of the form $\text{wcg} G$, where $G$ denotes a counter. The substitution function, denoted sub $e_1, e_2, C$, takes a pair $e_1, e_2$ and replaces each occurrence of $e_1$ with $e_2$ in the content $C$. Fresh wildcard variables can be generated by the function $\text{wc}_i G$, where $w_i$ denotes a concrete variable with explicit index $i$, $G$ denote a wildcard counter value, and denotes concatenation of indexes. An example of deployment is given the wildcard $\text{wc} X_4$ and counter value $18$, $X_4, 18 \text{ wc } X_4, 18$. Hence a new variable is generated with an index that contains the history of its instantiations: The wildcard variable $\text{wc} Y_4, 8, 13, 18$ is related to three previous wildcards, the context of the current wildcard can be traced back to the following variables: $\text{wc} Y_4, 8, \text{ wc } Y_4, 8, 13, \text{ wc } Y_4, 8, 13, 18$.

Definition 3: Suppose that $a$ is the host agent and $b$ is the communicating component. Then perform match is defined by:

\begin{align*}
    i & : M e, e, b, t, \\
    & \{ a, N, E \ n_1, n_2, \text{msg} C \text{ from } t_1^A \text{ to } t_2^A, n_1, \\
    & \text{ then } S, n_1, n_2, \text{msg} C \text{ from } t_1^A \text{ to } t_2^A, n_2, \\
    & \text{ if } t_1^A a, t_2^A a \}
\end{align*}

\begin{align*}
    ii & : M e, e, C_1, e, C_2, b, A, W, t, M C_1, C_2, b, A, W, t \\
    iii & : M e, e, C_1, e, C_2, T, b, A, W, t \\
    & \{ C_1, C_2, b, e_1, e_2 T, A, W, t, \}
\end{align*}

\begin{align*}
    iv & : M e, e, C_1, wc w, C_2, T, b, \\
    & \{ b, N, E \ n_1, n_2, \text{msg} C \text{ from } t_1^A \text{ to } t_2^A, n_1, \\
    & \text{wc } G, \text{msg} C \text{ from } t_1^A \text{ to } t_2^A, n_2, \\
    & \text{if } e_1 e_2 \text{ wildcard? } e_2 \}
\end{align*}

The algorithm can be explained as follows:

\begin{align*}
    i & : \text{If both message contents are empty, then the matching has succeeded and the current pointer is moved to the next state } n_2. \text{ The condition expresses that the active transition might involve send or receive events.} \\
    & \text{If the message contents } C_1 \text{ and } C_2 \text{ are non-empty, then there are three cases: }
    \\
    \text{ii } & : \text{In case both initial elements in the two message contents are identical, then continue to match the remaining contents.} \\
    \text{iii } & : \text{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.} \\
    \text{iv } & : \text{If the initial element in 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.} \\
\end{align*}

In the next section we describe how the framework can be used to synchronize two agents that communicate using distinct notations for the commands, and their view on the game defer slightly. The agents implement two versions of Blackjack and intend to run a session of the game without being synchronized in advance.

III. Distributed Blackjack

As a case study we have specified a scenario consisting of mobile phones that connect to a server, playing the role of casino. The players and the dealer all have their own implementation of the application, they have different commands, and potentially different high level understanding of the game. For convenience we assume that the notions of card and stock is uniform for every agent. The rules of the game follows standard Blackjack.
with some exceptions: The casino deals cards in a truly concurrent manner, in contrast to standard Blackjack where the dealer follows a certain order by dealing cards. Players initially ask for a virtual table in order to play the game.

Blackjack is a simple game, which involves betting between the player and the bank about how will get a score of 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 game starts by the player placing a bet, usually above some lower limit. The bank first deals two cards to each player, then two for himself. All the players’ cards' are dealt face up. The bank’s first card is face-down, the second is face-up. The players ask, in subsequent rounds to be “hit”, i.e., be dealt more cards one at a time, or to “stand”, after which they wait until all players and then the bank has finished. The player decides when to stop asking for cards, but of he passes the limit of 21 points, he is “bust” and the bet immediately collected by the bank.

The bank plays when all players have either asked to “stand” or been “busted” and starts by showing his face-down card. The bank usually plays according to house rules, which may e.g., stipulate that he has to keep drawing cards until he gets at least 16, and that he has to stand as soon as he reaches 17 or more. All players who have scored higher than the bank, and still lower than 22, are paid double their bet, unless they have 21 exactly to yield twice that. If the bank and a player scores the same sum, the bank wins.

There are local variations and more advanced rules, but for our pedagogical example the above simple interpretation will suffice.

A. The application graphs

The application graphs can be conceptualized as finite state machines. Figure 3, shows the dealer’s view on the Blackjack game. In the beginning the dealer is waiting for agents that register for a game session, the dealer DEAL...
In practice this means that an execution might involve a sequence of matching tables $T_1, T_2, T_3, \ldots$, instead of one table $T_n$. An obvious choice is to restart with an empty matching table, but this is avoided for two reasons: First, the components have learned to match several constants already. To re-learn the original matchups only requires extra computations, no new insights is gained by losing the components have learned to match several constants.

Second and even more important, if the constant matches for table $T_n$ is lost, then the next matching table $T_{n+1}$ might contain erroneous matchings of constant. The solution to this problem is to keep the matchings of constants, and erase the matchings of variables. If $T$ is a matching table, then let $T^e$ denote the subtable consisting of only constant matches. Thus an execution of a cyclic specification including a sequence of matching tables $T_{n\text{seq}} = T_1, T_2, T_3, \ldots$, where $T_1$ denotes the initial table, satisfies $T^e_{n+1} = T^e_n$ for each proper index $n$. We call this principle the principle of fact-learning.

The refresh command can be used at several places in the code, causing a semi-fresh instantiation of a matching-table. At each closing there should be a refreshment command causing a refreshment of matching tables.

We note that both Figure 3 and Figure 4 contain implicit cycles, for instance: $(\text{dealerplay} \rightarrow \text{dealerplay2})$ and $(\text{dealerplay2} \rightarrow \text{dealerplay})$. This cycle is implicit, since there is no explicit refreshment of the matching table.

2) Information events: In applications some events might occur at any state in the process. We call such events for information events. In case of the Blackjack application, information events can be, e.g., updating of new players or notification of other players move in the game.

Information events can be represented by adding reflexive incoming transition in all nodes: hence for all $n \in \mathbb{N}$,

\[
\text{msg InfoContent from } a \text{ to } b
\]

where $b$ denote the host component, and $a$ the information component.

---

### B. Prototyping in Maude

Maude is a declarative specification language [2] based on rewriting logic [3]. Maude specifications can be executed, which yields one out of potentially several traces. Mechanized analysis is possible through reachability analysis (using the search command) or model checking (by using Maude’s model checker). Maude is designed to support concurrency, which is a great advantage when specifying a distributed system like online remote Blackjack. Since Maude is based on term rewriting, the specifications are close to the pure specifications of the algorithms.

### C. Experimenting with the specification

The Blackjack specification 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 previous application graphs Figure 3 and Figure 4, 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. At 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.

When we start Maude and run the specification, it terminates after one million rewrites, with a long term containing the casino, the player and their buffers.

…..

rewrites: 934899 in 1289ms cpu
(1695ms real) (724838 rewrites/second)
result Configuration:

Dealertrace:

(msg joingame "Egil"

---

### Fig. 4. The application graph of the player.

### Fig. 5. One matching table of the dealer.
Maude returns a large term that consists of a trace by the dealer *Las Vegas*, which shows that *Egil* initially requested a game session. The request was accepted by *Las Vegas*, that replied with informing *Egil* that *Las Vegas* is a dealer, and stating that *Egil* is a new player to the game. The first actual Blackjack move occurs, *Las Vegas* sends the two initial cards to *Egil*: Ace and King of hearts.

IV. Error recovery

In this section we shall discuss some potential solutions to the fact that mismatches do occur quite frequently, due to the lack of global knowledge of the intended executions of the components involved. If we assume a strict matching algorithm like the one proposed in the paper, it is still possible that a matching session fails due to the problem of non-deterministic choices in the matching. Suppose that agent *a* receives a message $M$ \( \text{msg } e_{k_1} \rightarrow e_{k_2} \) from *a* to *b*. The receiver *b* might be in a state $s$ where two possible matches is possible, there might be arcs

\[
\text{message } m_1 \\
t_1 \quad n_1, n_2, \text{msg } e_1 \rightarrow e_2 \text{ from } a \text{ to } b \\
\]

and

\[
\text{message } m_2 \\
t_2 \quad n_1, n_3, \text{msg } e_3 \rightarrow e_4 \text{ from } a \text{ to } b \\
\]

Hence the receiver *b* might extend the matching table with either $T_1 \quad e_{k_1}, e_1, e_{k_2}, e_2$ based on transition $t_1$ or the match $T_2 \quad e_{k_1}, e_3, e_{k_2}, e_4$ based on $t_2$. The application graph can be of two main types: branching as in Figure 6 or merging as in Figure 7, where $n_2 \quad n_3$ (denoted $n_6$). A special case of the merge is the case when both the transitions are reflexive, that is $n_1 \quad n_2 \quad n_3$. Suppose that $T_1$ was the “correct” match and that *b* chose $T_2$. At some future time in the execution *b* might discover that something is wrong by not being able to interpret and synchronize the interactions with *a* in a satisfactory way. Several situations could occur:

(a) The receiver discovers the mismatch soon and restores the session successfully. This corresponds to the application graph in Figure 7, where the message $m_5$ is a send event from *b*, where the elements in $m_5$ depends on the elements in $m_2$.

(b) The receiver does not discover it and proceeds as if it has succeeded. The next event is a message sent to *a*, which is independent of the partial matches $T_1$ and $T_2$. The situation is covered by Figure 7, in which the elements $e_3$ and $e_4$ in $m_2$ is independent of the elements $e_5$ and $e_6$ in $m_5$.

(c) The receiver does not discover the mismatch and sends an improper message to *a* that erroneously interpret as sound. Hence both *a* and *b* have unhealthy matching tables. This corresponds to Figure 6; *b* mistakenly interprets the received message $M$ as a $m_2$ instance, and then sends message $m_4$. The initiator *a* is in a state such that $m_4$ can match (erroneously) the next transition.

(d) The receiver does not discover the mismatch and sends an improper message to *a* but *a* can not interpret the reply meaningfully. This is situation is similar to the previous one except that the initiator *a* cannot match $m_4$ successfully.

More intricate error situations can be constructed in this line: Agent *a* might also have non-deterministic choices on its side, etc. The easy way out for *b* in this case is to give up the matching and abort the session with *a*. But in some cases this is unfortunate. The most obvious case is when the interacting component might be unwilling to abort the session and restart from the beginning. In future work we will explore techniques that can be used to recover a erroneous session.

We also note that implicit cycles, like reflexive transitions and small loops do not cause any extra problem, since the choice element keeps track of these in a similar way as ordinary tree-structures. We say that an application graph is simple if it is merging and contains only one refresh event. A round is an trace in an application graph that starts and ends with the node $n$, where that $n_e, n, \text{refresh}$.

Fortunately the algorithm extends gently to simple application graphs:

Theorem 1: The matching protocol terminates for
simple application graphs in less than $R \times C \times B^p \times 2$ computations, where $C$ denotes a constant measuring local computations, while $R$ denotes the number of rounds.

The approach taken in this paper is quite flexible: The matching framework does not require a particular setup of components, the “distance” between the integrating components or the kind of “connectivity” they share do not add any particular difficulty to the synchronization mechanism.

Moreover, if it is desirable to adjust the integration process with user interaction, this can be supported by giving breakpoints for manual matching assignment and user-guided priorities at any step in the matching process.

V. Discussion

Several problems have been left unexplored in this paper. This paper is the first report from a broadly scoped and ambitious project, and we realize that potential future extensions now needs to be considered thoroughly in order for this research to progress.

A. The system and matching assumptions

Assumption A, component isolation, B asynchronous communication, and C local view, captures the state of the art model of distributed systems. Any strengthening of these assumptions would imply a weakening of the applicability of the matching framework. The three assumptions make the approach suitable to a variety of scenarios. The local view assumption can be weakened by observing that the integration module, as described in Figure 2, can be gently moved into a centralized matching hub on the link between the synchronizing components, since the matching components can operate relatively independent with the applications, as described in Figure 8.

The assumption about simple applications (D) represents an important restriction making matching feasible. Further investigations might generalize our approach to a wider class of applications.

The assumption that the components are in a always integrating mode (assumption F) is necessary in order to ascertain that no relevant event is is missed. It is however possible to weaken this assumption by requiring that the application relevant messages are filtered out on a particular integration channel. The obvious strengthening of this requirement is to permit more than one application to integrate at a time. A way to permit concurrent integration processes is to establish distinct separation channels for each application, such that the relevant messages are routed on the appropriate channel.

For practical application inside integration tools, assumption E, explicit reflection of behavior, is the most troublesome. If application graphs are specified manually, this might introduce potentially new errors into the integrating process. Without automation, the integration problem is moved to a practical problem of understanding, extracting, and specifying each application in formalisms that few software engineers know in advance or are capable and interested in learning.

B. Changing the matching algorithm

The matching algorithm has been designed in a pragmatic way by assuring that a synchronizable specification of Blackjack should be executed successfully. There are several ways that matching could be revised, and in the following section we shall discuss some alternatives.

1) Relaxing the matching algorithm: In the matching test (definition 1) and the matching function (definition 3), it was crucial that matching was performed in a linear way, starting by first matching the leftmost element in each content, and then proceed recursively to the right. This is although a rather strict requirement, by integrating realistic applications the order of the matching elements might be swapped: One agent might use

```
getcard wildcard CARD
```

while the other might use

```
 wildcard CARD GETCARD
```

The intention of each notation is the same: An agent gets a card. In order to make the matching possible, the basic elements should be independent of the ordering of the element. Intrinsically, this involves more computations than the straightforward left-to-right recursion, but whether it will give successful matching is an open problem at the moment.

2) Other notions of consistency: The notion of consistency used in the paper is the result of experimentation. The criteria for claiming a matching pair consistent presented in this paper is the following:

(C1) The subtable of constants forms a bijection
(C2) Anything can be matched to a wildcard

There are situations where these criteria do not hold. A counterexamples to Criterium 1, can be given by considering implementations of applications that use synonyms extensively. In case of the Blackjack application an agent might use several commands for getting cards, the player might use `getcard` or `receivecard`. If the casino only use one command this will be interpreted by the dealer as the following subtable: If the matching relation is injective\(^1\), then we can infer that `getcard = receivecard`.

\(^1\)Understood as $x \neq y \Rightarrow x \neq z \land y \neq z \land M \neq x \neq y$. 
A counterexample to Criterium 2, can be given by requiring that the value of the wildcard should be used further in the computation, that the wildcard cannot currently be deployed:

3) Deployment of wildcards: In the current algorithm, wildcards are used only once. For several purposes this is not good enough. In Blackjack, an example of deployment of wildcard variable would be to use the value for further computations: If the application graph should contain computations of the score from a set of cards, then the value of the wildcard values must be “stored” or assigned for further computations. This is not currently possible since each explicit occurrence of a wildcard variable in the application graph generates a fresh wildcard. This is, among other issues, on our list of future research problems that we are going to focus on in future work.

VI. Conclusion

As highlighted in Section V, there are several extensions of the approach that can be further explored. The most urgent problem to solve is to make the construction of application graphs more gentle.

Some minor extensions to the method itself can be made, based directly on the algorithms presented in this paper. We have described a binary matching relation, where one agent can match one single application with another single agent’s application. Applications in real life usually involve several participants. An extension of the method to one-to-many or many-to-many, can be made by extending the matching module with awareness about the different roles agents play in a game. One agent or component might run several different applications and be involved in on-the-fly matching of these application concurrently. This is possible if distinct matching channels are introduced, one channel for each application.

The matching algorithm is today only made for bilateral matching. Inasmuch as applications may exist which need to be matched against several different sets of terms simultaneously, this needs to be taken into consideration as well. The fundamental approach need to be further validated in future work, most basically to check that it can be extended to cover more complicated application types. We have started experimenting with applications that have less explicit ordering among terms, and which may contain potentially infinite recursive cycling. The results so far are encouraging, but more research is required to make the approach viable in real-world settings.

References