RI08006 17 April 2008 Computer Science

IBM Research Report

Planning with Communicating Automata

Jana Koehler
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland

Biplav Srivastava
IBM India Research Laboratory
4, Block - C, Institutional Area
Vasant Kunj, New Delhi - 110 070, India.

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

Planning with Communicating Automata*

Jana Koehler
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
koe@zurich.ibm.com

Biplav Srivastava
IBM India Research Laboratory
4, Block - C, Institutional Area
Vasant Kunj, New Delhi - 110 070, India
sbiplav@in.ibm.com

Abstract

Until today, planning operators are mostly considered as atomic transitions that change the value of boolean or numeric state variables. This remains also true if nondeterministic effects are added to operator descriptions. Solely, the compound tasks used in HTN planners encapsulate more complex behavior, but it is difficult to model nondeterminism and iterations using HTN representations. In many real-world applications however, in particular in technical environments such as hardware and software systems (for example networks, server farms, or embedded controllers), more expressive planning operators that encapsulate nondeterministic and iterative behavior are needed.

In this paper, we propose to use nondeterministic communicating automata as the operators for a planning system. We show how to model such operators, define the planning problem with communicating automata and present a first, preliminary planner involving model checking techniques.

Introduction

Many real-world planning applications are currently emerging from the need to equip complex technical systems with more flexibility allowing deliberative and reactive behavior. A typical example is the planning and distributed execution of application systems based on *web services*. A web service as it is defined in WSDL, the Web Services Definition language (Christensen *et al.* 2001) encapsulates a particular piece of software that can be invoked via the Internet. WSDL is an XML-based language used to specify the location of the web service and the operations (or methods) that it exposes. A web service is characterized by *port type*, an element that specifies the operations and messages involved in interacting with the web service. Four types of communication are defined involving a service's operation (endpoint):

- One-way: The endpoint receives a message.
- Request-response: The endpoint receives a message, and sends a correlated message.
- Solicit-response: The endpoint sends a message, and receives a correlated message.
- Notification: The endpoint sends a message.

A WSDL *message* element defines the data elements of an operation. Each *message* can consist of one or more *parts*. The *parts* can be compared to the parameters of a function call in a traditional programming language. The *binding* element defines the *message* format and protocol details for each port. XML Schema syntax is used to define platform independent data types.

In a nutshell, a WSDL specification only tells us something about the syntax of messages that enter or leave a computer program. Today, the interaction with a web service is very limited in the sense that only single send and receive operations on messages are specified in WSDL. One can, however, imagine that this limitation opens up and allows to use web services to implement entire processes that receive different messages and return other messages. The functionality of those services needs to be described with some additional piece of information, either by some *semantic annotation* of what it does and/or by some *functional annotation*, how it behaves. Semantic annotations have been widely discussed in the semantic web community. By using pre-agreed ontologies, the meaning of the messages to be exchanged can be understood. To enable the semantic-based composition of services in order to support business-to-business or enterprise application integration, using ontologies is not enough. It is often assumed that a business process or application is associated with some explicit business goal definition that can guide a composition tool to select the right service (McIlraith & Son 2002). Unfortunately, we found that this is usually not the case. A business process model usually describes the processing of persistent data objects that can be created at runtime by some application system or a human user. The real "goal" of a business process can hardly be captured in such a model, but often remains implicit. However, a more practical goal of a business process is the correct handling or creation of data documents.

We can see services as the operators in a plan that encapsulate some behavior, but this behavior differs from

*This manuscript was last edited in Nov 2002.

what we find in PDDL operators (McDermott 2000). A functional annotation of a non-trivial service that goes beyond WSDL has to add some elementary information about the specific input-output behavior that occurs, i.e., in which order the ports become active and either wait for messages to arrive or in which order they will send messages. This behavior cannot be specified with sequences of actions, but requires loops, branching and nondeterminism.

Consequently, in this paper, we investigate how the functional annotation in form of a *communicating nondeterministic automaton* can support the automatic composition of services with a planning system. We found that modeling planning operators as communicating nondeterministic automata yields a powerful planning representation formalism that is applicable to many other planning problems we find in complex technical systems. Such a formalism has quite a number of interesting features:

- we obtain a very compact representation of quite complex behavior,
- allowing more than atomic state transitions in an operator supports a flexible decomposition of domains into planning operators,
- automata theory provides a solid theoretical foundation of this planning approach and provides many techniques that are helpful when developing automata-based planning and domain-analyzing systems,
- by considering special types of automata, we can either obtain tractable special cases, e.g., in the case of finite-state machines, or model complex planning problems involving durative actions, e.g., by using timed automata.

The paper is organized as follows: In the next section, we review the well-known trip planning scenario[1] from the perspective of synthesizing a network of communicating processes. Then we discuss the structure of planning operators based on automata and we define the associated planning problem more formally. The second part of the paper discusses a first prototype of our planner based on Promela/SPIN (Holzmann 1991) and introduces the dimensions along which different classes of automata-based planning problems can be defined. We conclude with a review of related work, an outline of other potential application areas, and the many questions that remain to be investigated.

An example Scenario

To explain the problem and discuss our solution, we discuss an example scenario of booking travel packages in a travel agency. Let us consider the evolution of this scenario from a simple, closed-world process to a dynamic, integrated solution. In the simple, closed-world travel agency, a customer talks to the travel agent who notes the customer's requests and generates a *tripReservationRequest* document that may contain several needed *flight* and *hotel reservations*. The travel agent performs all bookings and when he is done, he puts the *tripReservationRequest* either into the *cancelledRequests* or the *completedRequests* repository, see Figure 1.

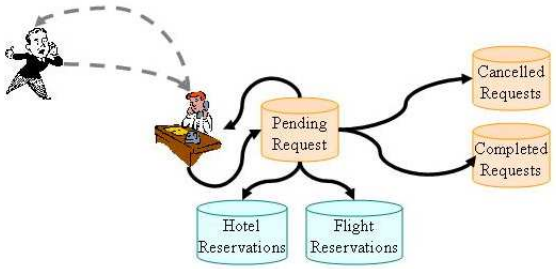

Figure 1: The closed-world travel agency.

Let us assume that our travel agency wants to cooperate with external specialized service providers that offer hotel and flight reservations, see Figure 2. This cooperation will require to reorganize the entire processing of customer requests. New services have to be integrated and all services must correctly interact with each other.

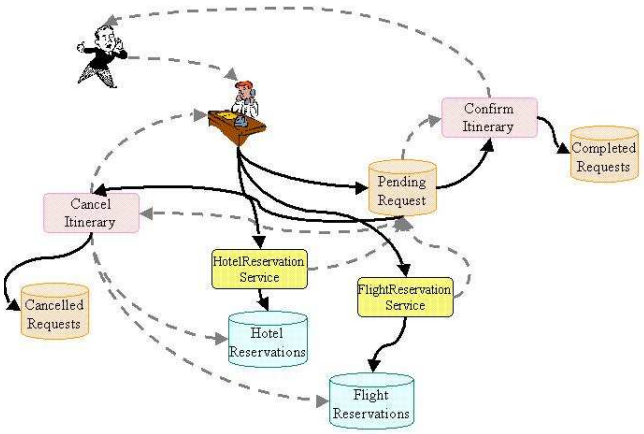

Figure 2: The open-world travel agency. Black solid lines show the transfer of reservation request data objects between the various operations in the process, while gray dashed lines show the sending of notification messages.

In this new process, upon receiving the customer order, the travel agent will still create a *tripReservationRequest*. We cannot of course know, what a customer desires, and we also want to implement a system that works for any customer request. In any case, some individual *flight* and *hotel requests* will result that are sent to the specialized service agencies. The *tripReservationRequest* is put into a *pendingRequests* repository

[1] See also http://www.w3.org/2002/04/17-ws-usecase.

where the document is accessible by the specialized services. The hotel and flight reservation services get triggered by the individual *hotelRequests* and *flightRequests*. Each of them works independently and tries to book the desired reservations. A reservation that is completed is put into a repository by the service. Although both services work independently, there is the need to coordinate their behavior, since they work on the same customer request. The coordination is achieved through updating the status of the pending *tripReservationRequest* by each service in case of a successful completion or failure. When both services have been successfully completed, a *confirmation service* can put all documents together and inform the customer about the completed trip plan. If one of the services fails, a *cancellation mechanism* should be available that performs several functions:

- it immediately tells the service providers to stop making reservations,
- it makes sure that all completed reservations are cancelled,
- it informs the travel agent to contact the customer.

In this example scenario, we see properties that are quite typical for real-world planning scenarios:

- a number of concurrent processes (we could also call them agents) is up and tries to jointly achieve some goal,
- the processes partially interleave with each other, synchronize or run fully independently of each other,
- each process itself is showing a complex behavior,
- certain planned operations may fail and require to recover from a failed execution.

We are looking for a very general approach that would allow us to model planning problems resulting from such a scenario at a very abstract design level. First, we want to describe the individual activities occurring in the process as our planning operators. Second, we want to compose these operators (either by hand or automatically) to generate a design specification of the process from which, in a next step, an implementation can be derived.

As could already be seen from the informal description of the scenario, our planning operators are quite different from "classical" PDDL style operators. In particular, they have to encapsulate iterating and nondeterministic behavior. We want to specify this behavior of each service independently of the others and then let the planning system explore the possible interactions within the services and determine their "orchestration".

Planning Operators as Automata

To illustrate our approach, let us discuss how automata can model our example scenario. Figure 3 shows the customer automaton. The customer will start from a state *calling* and send a message *request*.[2] Then he will enter a state *waiting*, which can only be left if the travel notification (either positive or negative) is received.

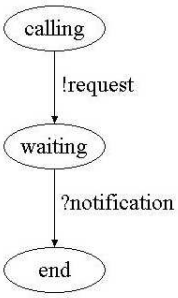

Figure 3: The customer behavior as a simple sequential automaton.

The behavior of the travel agent (Figure 4) is more complicated as it involves iteration and nondeterminism. The travel agent starts in the *receiving* state. When receiving a *request* message, he enters the *createHotelRequests* state. There are two nondeterministic transitions from this state: either by sending a *hotelRequest* and returning to the same state or by sending nothing anymore and entering the *createFlightRequests* state. Finally, the agents creates a *tripRequest* document summarizing all information.

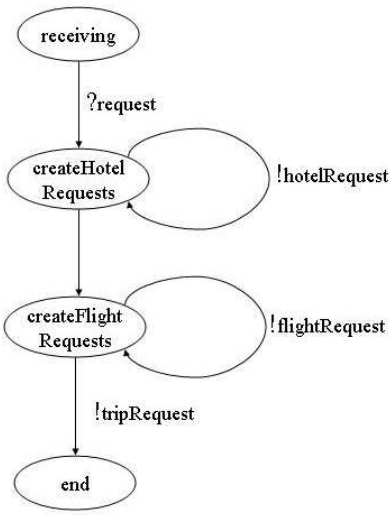

Figure 4: The travel agent behavior as an automaton with cyclic and non-deterministic transitions.

The *flightReservation* service (Figure 5) waits in the *receiving* state for incoming reservation requests or a cancellation event that will be sent by some cancellation mechanism. When a reservation request arrives, the service enters the *createFlightReservations* state and

[2]We use "!" to denote the sending of a message and "?" to denote the receiving of a message.

either sends a *reservation*, a *failure event* (in case that a flight reservation failed) or a *completion event* (in case it booked all requested reservations). If it receives a *cancellation event*, it immediately stops processing.

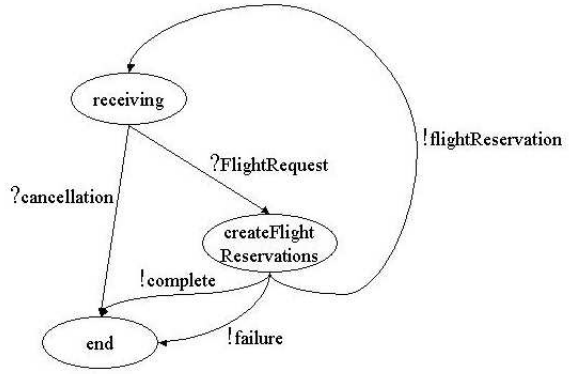

Figure 5: The behavior of the flight reservation service.

Let us define our planning operators more formally.

Definition 1 *A communicating automaton A is a tuple $\langle S, M, \delta, I, E\rangle$ with S being a finite set of states of the automaton, M being the set of messages that can be sent to or received from a communication channel by A, $\delta : S \times M \rightarrow 2^S$ is the transition function (which may be partial), $I \in S$ is the initial state of A, $E \subseteq S$ is the set of end states of A.*

We assume that there is a unique initial state, but allow multiple end states. Communicating automata exchange messages by writing them into or reading them from communication channels. Each process is modeled by using one automaton only and the channels are passive communication devices.

Definition 2 *A channel is a queue of finite or infinite capacity that can store messages of a particular data type.*

A message is an ordered set of constants in some data type, e.g., integers, booleans, strings, XML schemas. A message could also be a formula over some logical theory.

Definition 3 *A global state is the state of all automata and the contents of the communication channels.*

Definition 4 *The planning problem with communicating automata is given by the tuple $\langle \mathcal{A}, \mathcal{C}, \mathcal{I}, \mathcal{G}\rangle$ where*

- *$\mathcal{A}$ is a set of automata,*
- *$\mathcal{C}$ is a set of channels, which we currently assume to be global, i.e., accessible by all automata,*
- *$\mathcal{I}$ is the initial global state with all automata $A \in \mathcal{A}$ being in their initial state I and with an initial (possibly empty) distribution of messages in the channels,*
- *$\mathcal{G}$ is the goal represented as another communicating automaton A_G, which may be empty.*

Definition 5 *A plan A_P solves a planning problem $\langle \mathcal{A}, \mathcal{C}, \mathcal{I}, \mathcal{G}\rangle$ if and only if $A_P \subseteq (\mathcal{A} \cup A_G)$, $A_G \in A_P$ and all automata in A_P can execute in $\mathcal{I}$ and terminate in an end state under all possible executions.*

Note that a plan is a network of communicating automata that can encapsulate more than a single sequential behavior.

Our definitions capture the "propositional" variant of the problem where the set of automata is given and new instances of an automaton cannot be added to the search space (similar to the set of ground actions in a classical planning problem).

We can vary the problem in the sense that the set of automata must execute and terminate for a set of different initial message distributions that are sent by some initialization automaton, by considering different types of goal formulas, or by allowing automata with more than just a single initial state. Using automata to represent goal formulas has been proposed recently in (Dal-Lago, Pistore, & Traverso 2002), where the authors also address the problem of complex planning problems in nondeterministic domains. While we make an explicit use of automata to represent operators and goals, they define an intermediate goal representation language from which the automata can be automatically built and consider actions as atomic transitions.

Theorem 1 *Planning for communicating automata is undecidable for unbounded communication channels, but NP-hard if the channels are bounded.*

The results follow immediately from the theorems proven in (Brand & Zafiropulo 1983) who have proposed communicating finite-state machines as the foundation for the specification and verification of communication protocols. We extend their thoughts here to the problem of designing dynamic processes.

In the next section, we will present a first representation of our planning domain using the design specification language Promela (Holzmann 1991) that has an automata-theoretic semantics. Promela allows us to consider the simplest form of automata, namely finite state machines with designated end states communicating via bounded channels, i.e., the planning problem is decidable and a model checking approach can be used to generate plan.

Designing a planning domain with Promela

Promela (Holzmann 1991) is a design specification language that implements the approach given in (Brand & Zafiropulo 1983). We briefly review the major features of the language that are necessary to understand the code below.

Among the elementary data types are *int* (for integers) and *chan* for communication channels. A channel has finite capacity and holds messages of a specific type. For example *chan sent = [2] of { int,chan,int }* specifies that the channel named *sent* can hold up to two

messages being a sequence of an integer and a channel name followed by an integer. The boundedness of channels guarantees the finiteness of Promela models. We can use the first message field as the message type, e.g., define a constant *letter* and write *letter(5,3)* to denote a specific letter message.

Processes are specified using the *proctype* construct. A process specification contains a sequence of statements that are executed in the order they appear in the sequence. Basic statements allow to receive and send messages or perform simple calculations with integers. A statement can only be executed when it becomes executable, which depends on the type of the statement. A statement of the form $< channelname >$? $< message >$ denotes the attempt to read a message from a channel. It becomes executable only when a message of the declared format is contained in the channel. A statement of the form $< channelname >$! $< message >$ denotes the sending of a message to a channel and becomes executable only if the channel has enough capacity to hold that message. Important control statements are branching *if ... fi* and loops *do ... od*. Inside a branching or loop construct, alternative sequences of execution can be specified using the :: operator. The first statement in a "::"-branch is called the guard, since its executability determines whether the branch becomes executable or not. Several branches can be become executable in a state, which results in nondeterminism of the process.

To use Promela as our preliminary planning representation formalism, we begin by defining the message types that will be sent and the channels that have to be used to communicate those messages between the various automata. Each repository in our example becomes a global asynchronous communication channel that holds messages of a particular type and that can (in principle) be accessed by any automaton.

```
chan pendingRequests = [CAPACITY] of
        {int,int,chan,int,int,int,int,int}
```

The *pendingRequests* channel holds messages of the following structure: *tripReservation(id, custId, tStatus, hStatus, hcount, fStatus, fcount)*. Each message is of type *tripReservation*, contains an identification number *id* that denotes the particular customer request we process, followed by a channel variable *custId* that defines the return address under which the customer can be reached. The remaining variables define the status of processing the trip request (*tstatus*), the status of processing the associated hotel requests (*hstatus*), the number of different hotel stays associated with this customer request (*hcount*), the status of processing the flight requests (*fstatus*) and the number of flights (*fcount*).

We add two more channels that store the completed or cancelled trip requests having the same message structure:

```
chan cancelledRequests = [CAPACITY] of
        {int,int,chan,int,int,int,int,int}
chan completedRequests = [CAPACITY] of
        {int,int,chan,int,int,int,int,int}
```

Four channels are needed to exchange hotel and flight requests that have a simpler structure:

```
chan pendingHotels  = [CAPACITY] of
        {int,int,int,int};
chan processedHotels = [CAPACITY] of
        {int,int,int,int};
/*hotelReservation(id,hcount,hstatus)*/

chan pendingFlights = [CAPACITY] of
        {int,int,int,int};
chan processedFlights = [CAPACITY] of
        {int,int,int,int};
/*flightReservation(id,fcount,fstatus)*/
```

Finally we need a communication channel to transfer hotel and flight requests from the travel agency to the service providers, set up a channel to send the *cancelEvent* that stops one service in case of failure of the other, and we also need to define a communication channel between the customer and the travel agency:

```
chan cancelEvent = [1] of {int}; /*id*/

chan newRequest =[1] of {chan}; /*B2C channel*/

chan pendingHotels = [CAPACITY] of
        {int,int,int,int};

chan pendingFlights = [CAPACITY] of
        {int,int,int,int};
```

Now we can define our planning operators as a Promela process. The customer process uses a local variable *answer* to store the result of his trip request processing. The customer sends his return address (a local channel under which he wants to be informed) to the channel that will later be read by the travel agency. This will initiate a trip reservation process. We do not make any assumptions about what details else the customer communicates about his trip as this will be different for each customer.

```
proctype customer(){
  int  answer;
  chan result = [1] of {int};
  newRequest!result;
  result?answer}
```

The *travelAgent* process represents the travel agent creating the *tripReservationRequest* document from talking to the customer and preparing a nondeterministic number of *hotel* and *flight requests* that he sends to the service providers. The status of processing these requests is set to *pending*. The nondeterministic process has to respect the capacity of the channels, which is an artificial restriction that we have to obey due to the finiteness of Promela models.[3]

[3]A Promela process can also create a local channel and instantiate a new process on that local channel. This would allow planning operators to not only create new domain objects in the form of messages, but also to instantiate other

```
proctype travelAgent(){
 int  id=1; int  fcount=0; int hcount=0;
 chan custId;

 newRequest?custId;
 do
  :: nfull(pendingFlights)
     -> fcount = fcount+1;
     pendingFights!flightReservation(id,fcount,
                    pending);
  :: break
  :: full(pendingFlights)
     -> break
 od;
 do
  :: nfull(pendingHotels)
     -> hcount = hcount+1;
     pendingHotels!hotelReservation(id,hcount,
                   pending);
  :: break
  :: full(pendingHotels)
     -> break
  od;
  pendingRequests!tripReservation(id,custId
     pending,pending,hcount,pending,fcount)}
```

The flight reservation and hotel reservation services have an identical structure in our domain model and we only show the former. The flight reservation service contains a loop that reads from the *pendingFlights* channel *unless* a message arrives in the *cancelEvent* channel, i.e., this channel becomes non empty ("`nempty`"). The *unless* construct in Promela allows us to define priority events that can interrupt the execution of a process. If the channel still contains messages, the first message is read and either this request is added to the *processedFlights* channel with the status updated to *complete* or the service cannot complete the reservation and updates the *fstatus* variable in the *tripReservationRequest* from *pending* to *cancelled*. It also keeps some local variables to count the number of processed reservations. When the channel contains no more messages and the service has completed all incoming messages, it updates the *fstatus* variable from *pending* to *complete*. If a flight request cannot be completed, the service updates the status of the *tripReservationRequest* in the *pendingRequests* channel from *pending* to *cancelled*.

```
proctype reserveFlight(){
 int  id,leg,hstatus,hcount,fstatus,
      fcount;
 int  incoming=0; int completed=0;
 chan custId;

 {do
   :: nempty(pendingFlights)
      -> pendingFlights?flightReservation(id,leg,
             pending);
      incoming++;
      if
       :: processedFlights!flightReservation(id,
            leg,complete);
          completed++;
       :: pendingRequests?tripReservation(id,custId,
             pending,hstatus,hcount,fstatus,fcount);
          pendingRequests!tripReservation(id,custId,
             pending,hstatus,hcount,cancelled,fcount)
      fi
  :: empty(pendingFlights) && incoming == completed
     -> pendingRequests?tripReservation(id,custId,
          pending,hstatus,hcount,fstatus,fcount);
     pendingRequests!tripReservation(id,custId,
          pending,hstatus,hcount,complete,fcount);
     break
  :: empty(pendingFlights) && incoming != completed
     -> break
 od} unless nempty(cancelEvent)}
```

The *confirmItinerary* process listens to the *pendingRequests* channel and waits for the *hstatus* and *fstatus* to change to *complete*. In this case, it can read the *tripReservationRequest* message and send it to the *completedRequests* channel with *tStatus* also set to *complete*. It also uses the *custID* channel information and informs the customer about the completion of his reservation request.

```
proctype confirmItinerary(){
 int  id,hcount,fcount;
 chan custId;

 pendingRequests?tripReservation(id,custId,
       pending,complete,hcount,complete,fcount)
 -> completedRequests!tripReservation(id,custId,
       omplete,complete,hcount,complete,fcount);
 custId!complete}
```

The *cancelItinerary* service also watches the *pendingRequests* channel, but for either the *fstatus* or *hstatus* variable to change to *cancelled*. In this case, it first sends out the request identification number to the *cancelEvent* channel to which the services are listening. Then it cancels all hotel and flight reservations by reading out the reservation messages with status *pending* and writing them back with status *cancelled*. It finally sends the *tripReservationRequest* with status *cancelled* to the *cancelledRequests* channel to which the travelAgent could be listening in order to become active again (which we have not modeled here). In our model, it directly sends a negative notification to the customer.

```
proctype cancelItinerary(){
 int  hstatus,fstatus,
      id,hcount,fcount,stay,leg;
 chan custId;

 if
  :: pendingRequests?tripReservation(id,custId,
       pending,cancelled,hcount,fstatus,fcount)
  :: pendingRequest?tripReservation(id,custId,
       pending,hstatus,hcount,cancelled,fcount)
 fi;
 cancelEvent!id;
 do
  :: processedHotels?hotelReservation(id,
```

operators (or copies of themselves) during execution. These are two very interesting features, but they go beyond the scope of this paper and require further investigation.

```
          stay,complete);
   -> processedHotels!hotelReservation(id,
          stay,cancelled);
:: processedHotels?hotelReservation(id,
          stay,cancelled)
   -> break
:: empty(processedHotels)
   -> break
od;
do
:: processedFlights?flightReservation(id,
          leg,complete);
   -> processedFlights!flightReservation(id,
          leg,cancelled);
:: processedFlights?flightReservation(id,
          leg,cancelled)
   -> break
:: empty(processedFlights)
   -> break
od;
cancelledRequests!tripReservation(id,
      custId,cancelled);
custId!cancelled}
```

We have omitted a few details from our representation that are Promela specific. For example, we used *timeout* statements that will allow a process to terminate when no statements are executable anymore. Furthermore, we used *atomic* constructs to reduce the possible interleaving of processes and reduce the number of states in the state space. Of course, alternative representations of this domain could also be imagined.

Planning with Automata

Given our planning operators as automata and a set of communication channels, we now need to specify the initial distribution of messages in those channels to define a planning problem. For the moment we consider the special case that all channels are initially empty and that we also do not have a specific goal formula to satisfy. Thus, it remains to select a non-empty subset of automata that can execute and terminate in an end state.

We are currently experimenting with a simple search algorithm that indexes each process by its receive and send operations, nondeterministically selects an executable process and checks whether the resulting set of automata can terminate in an end state. Figure 6 summarizes our simple, preliminary planner.

The algorithm deals with the "propositional" case of our planning problem. For this special case, it is sound and complete when a sound and complete termination checking procedure is used and all possible subsets in $2^{\mathcal{A}}$ are systematically explored.

In our example, $\mathcal{A}$ contains six automata, of which only the customer automaton is initially executable, since it is the only one that writes to a communication channel and does not wait to receive a message. The customer process can execute and send its message. However, it cannot terminate since it waits to receive a message. Thus, our search algorithm needs

plan$(\mathcal{A}, \mathcal{C}, \mathcal{I}, A_G)$
  $A_P = A_G$
  do
    determine subset $\mathcal{A}'$ of executable automata
    if $\mathcal{A}' = \emptyset$
      then backtrack over previous choices,
          return failure if no such choice exist
      else
          nondeterministically select an $A \in \mathcal{A}'$
          if $A_P = A_P \cup A$ terminate in end states
            then return A_P, break
            else continue
          fi
    fi
  od

Figure 6: A preliminary planning algorithm.

to select another process that could provide that message to the channel the customer listens to. The customer can receive the required message either from the *confirmItinerary* or *cancelItinerary* process. Here, the search algorithm encounters a choice point, but none of the processes is executable. Only after adding the *travelAgent* process and both reservation services, the *confirmItinerary* or *cancelItinerary* process can execute. If only one of these two automata is added, the *customer* automaton can terminate, but not in all possible execution traces of the *reserveFlight* and *reserveHotel* automaton, i.e., only in some executions the *customer* will be able to receive the message it is waiting for. Thus, only when all automata are contained in the result set A_P we can verify that termination in some end state is guaranteed under all executions.

It is obvious that the planning problem involves a verification step that tests for termination in an end state after each selection process. This makes such a planning algorithm much more costly than a classical planner that only needs to test whether the goal state is contained in the current state. The search space is the Cartesian product of all individual operator automata and grows significantly with each automaton being added. We used the model checker SPIN (Holzmann 1991) to verify whether our incrementally growing set of automata terminates in valid end states. Thus, model checking plays an important role in this planning approach that we will further explore below.

The Role of Model Checking

By using a non-empty goal formula A_G either during the planning process or once a plan has been found, we can formulate additional requirements the communicating automata should satisfy. When adding A_G to the planning process, we will usually encode system *invariants*—"maintenance goals" in (Dal-Lago, Pistore, & Traverso 2002)—into the automaton representing A_G and make sure that those invariants, if true in the initial system state, remain true in all reachable system

states, independently of the execution sequence that leads to each specific state. To express the invariant that when the customer receives an answer, this answer should always either be *cancelled* or *complete*, the following monitor process can be added as a goal A_G:[4]

```
proctype maintenanceGoalMonitor(){
 do
  :: answer != pending
     -> assert(answer==complete) || answer==cancelled);
     break
 od
}
```

Alternatively, we can (once a set of automata has been determined), formulate requirements to demonstrate specific behaviors, i.e., sequential sub plans that are contained in our non-sequential plans.

In the planning as model checking paradigm, the usual approach is to add the negation of the requirements to the model and have the model checker produce a violating trace demonstrating such a behavior. In the travel agency scenario, one may be interested in the following requirements, which can be stated as LTL formulas:

- *Always, eventually, a request is completed or it is cancelled.*
  $\Box\,(\,\Diamond\,(\,requestCompleted \vee requestCancelled\,)\,)$
- *Always a cancelled hotel or flight will eventually lead to a cancelled request.*
  $\Box\,(\,(\,hotelCancelled \vee flightCancelled\,)$
  $\rightarrow \Diamond\, requestCancelled\,)$

A request is completed once the *customer* process can receive the message *answer=complete*. A request is cancelled once the *cancelEvent* channel contains a message. A hotel or flight reservation request is cancelled once a service updates the *hstatus* or *fstatus* message field to *cancelled*. In our model, one can show that both requirements are satisfied. By formulating the *negation* of the above requirements in form of a so-called *never* claim in the SPIN tool we can also extract a violating trace demonstrating a specific behavior that satisfies the requirement, see Figure 7.

Limitations of Model Checking

The travel agency scenario is also interesting from another angle since it involves the nondeterministic generation of multiple instances of *hotelRequest* and *flightRequest* messages. At planning time, we do not know how many requests will result from the *travelAgent* process, and therefore need a plan that works for any number of requests. At run time, it will turn out how many particular requests have been generated.

[4]As a slight change to our model, it requires the *answer* variable to be global such that the assertion can be formulated. The operator ! = stands for inequality, while || stands for disjunction.

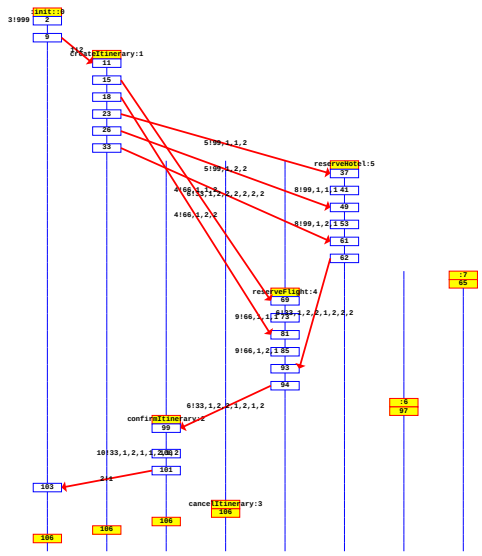

Figure 7: A plan for successfully booking a trip, which is obtained from a trace violating a *never* claim. The visualization in SPIN depicts vertical lifelines for the running processes with lines between processes visualizing messages being exchanged.

In our representation, we have the *travelAgent* process sent these requests to two global communication channels that are subsequently processed by the reservation services. Alternatively, we could also have modeled individual instances of the reservation services that are created at runtime to process each particular request message. The number of generated messages or process instances is nondeterministic and limited by a *predefined* bound in the Promela model. A model checker can thus only show that the processes behave correctly for that *specific* bound, but it cannot show that the processes would behave correctly for *arbitrary* finite bounds. Thus, when using a model checker we only know that our system behaves correctly for up to 5, 10, or 15, etc. hotel or flight reservation requests. The size of the search space is also significantly increasing, which sets a limit on possible values of bounds that can be investigated. For the basic case of a single flight and hotel request, the automata generate a state space of 725 states with 1094 transitions. With the bound set to 2 flights and hotels, we obtain already 8,435 states and 14,852 transitions. In the case of up to 5 flight and 5 hotel requests, 264,857 states and 526,492 transitions result making complete model checking almost infeasible. This challenges to think about different ways of showing *termination in end states* on which our planning approach heavily relies. A possible alternative we are currently exploring is to use a specialized inductive reasoner to prove termination for arbitrary values of *bound*, e.g., (Walther & Schweitzer 2001).

A further complication can occur if messages are not as simple as we consider them here, but could be concept descriptions based on some ontologies or arbitrary logical formulas. In this case, determining whether a receive or send operation is executable could already be a computationally expensive task. Figure 8 summarizes the three dimensions that determine the complexity of the planning problem.

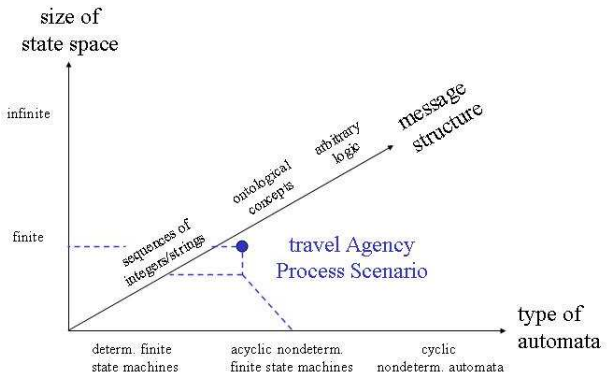

Figure 8: Three dimensions along which planning problems with communicating automata can be characterized.

Related Work

We see our work very closely related to the "planning as model checking" work pioneered in (Giunchiglia & Traverso 1999) that has been shown to yield powerful planning tools to address conformant planning problems (Cimatti & Roveri 2000), planning problems under uncertainty and partial observability (Bertoli, Cimatti, & Roveri 2001; Pistore & Traverso 2001; Jensen & Veloso 2000), but also classical planning problems (Edelkamp & Reffel 1999; Hölldobler & Störr 2000; Fourman 2000). In all these approaches the set of ground PDDL actions is considered when developing a semantic model of a planning problem. Similarly in HTN planning (Erol, Hendler, & Nau 1994; Nau *et al.* 1999), operators can express aggregate behavior that can be further refined, but how to express nondeterminism and iterations in compound tasks is still an unexplored issue.

Driven by the problem of specifying, validating and automatically synthesizing complex, reactive processes, we have proposed to use nondeterministic automata as a formalism to represent planning operators. A possible example of such operators are business services for which the automata could serve as functional annotations. So far, web service composition has been studied based on explicit semantic goals, e.g., (McIlraith & Son 2002) who uses Golog to generate complex plans with the services as atomic building blocks, or more from an interaction logic viewpoint, e.g., (Piccinelli *et al.* 2002) who consider only the information in the WSDL port type definitions.

We also see connections to the area of distributed planning (DesJardins *et al.* 1999). We specify a high-level design that may involve abstract processes, which can be refined as long as they meet the external specification, i.e., an operator itself can aggregate a distributed process or the operators can run in a distributed environment.

Planning with communicating automata is also closely related to the problem of synthesizing controllers, see for example (Barbeau, Kabanza, & St-Denis 1998).

Specifying goals as automata has been investigated in (Dal-Lago, Pistore, & Traverso 2002) and it was shown that automata can express goals that are neither expressible in LTL or CTL logics. In our first exploration of planning with communicating automata, we showed that goals represented in LTL can also be evaluated over a set of automata to extract a more conventional (sequential) plan from the violating trace. Although the usage of LTL is not necessarily required in our approach, we got the impression that LTL is nevertheless appropriate in the fully observable, technical environments that we investigate. In those environments, nondeterminism is used to represent different possibilities of which only *one* actually occurs, which is also the view of concurrency adopted in Promela and LTL, see also (Lamport 1980) for an in-depth discussion of the issue.

Conclusion and Future Work

In this paper, we proposed to represent planning operators as communicating automata encapsulating quite complex behavior and to generate plans from these automata. We borrowed a first, preliminary planning language from the design specification language Promela. The motivation driving this approach comes from the need to equip complex technical systems with more flexibility allowing deliberative and reactive behavior. As an example we discussed the composition and distributed execution of application systems based on web services. Many other application areas for this type of planning can be imagined: the *reactive scanning of networks* where the scanning operations have to be planned based on what information previous scan operations returned and where the behavior of a scan operation is described by an automaton, or the *autonomic managing of server farms* where a planning system should plan the migration of complex application systems to other server configurations (IBM Corporation 2001). Another application is a *smart service discovery* that allows applications to discover distributed services based on capabilities described in terms of a domain ontology and that guarantees that the invocation of the discovered service would realize the required results. Such a service discovery is feasible in specialized domains like bioinformatics where public ontologies exist and a large variety of services is available online.

We proposed a first planning algorithm that is sound and complete for a subclass of the class of planning

problems we defined. It shows that the computational properties of this approach are quite challenging: the search proceeds over two interleaved worst-case exponential spaces—the set of all subsets of automata and the global state space of a given set of communicating automata. We experimented with a model checking approach to verify the termination in end states and identified clear limitations of model checking for this problem.

Many questions have to be further explored ranging from the appropriateness of different representations and automata models to scaling search algorithms, better techniques to establish termination or the quality of automata-based plans.

References

Barbeau, M.; Kabanza, F.; and St-Denis, R. 1998. A method for the synthesis of controllers to handle safety, liveness, and real-time constraints. *IEEE Transactions on Automatic Control* 43(22):1543–1559.

Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Heuristic search + symbolic model checking = efficient conformant planning. In Nebel (2001), 467–472.

Biundo, S., ed. 1999. *Proceedings of the 5th European Conference on Planning*, LNAI. Springer.

Brand, D., and Zafiropulo, P. 1983. On communicating finite-state machines. *Journal of the ACM* 30(2):323–342.

Christensen, E.; Curbera, F.; Meredith, G.; and Weerawarana, S. 2001. The web services description language WSDL. http://www-4.ibm.com/ software/ solutions/ webservices/ resources.html.

Cimatti, A., and Roveri, M. 2000. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research* 13:305–338.

Dal-Lago, U.; Pistore, M.; and Traverso, P. 2002. Planning with a language for extended goals. In Dechter, R.; Kearns, M.; and Sutton, R., eds., *Proceedings of the 20th National Conference of the American Association for Artificial Intelligence*, 447–454. AAAI Press.

DesJardins, M.; Durfee, E.; Ortiz, C.; and Wolverton, M. 1999. A survey of research in distributed, continual planning. *AI Magazine* 20(4):13–22.

Edelkamp, S., and Reffel, F. 1999. Deterministic state space planning with BDDs. In Biundo (1999), 381–392.

Erol, K.; Hendler, J.; and Nau, D. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In Hammond, K., ed., *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, 249–254. AAAI Press, Menlo Park.

Fourman, M. 2000. Propositional planning. In Traverso (2000).

Giunchiglia, F., and Traverso, P. 1999. Planning as model checking. In Biundo (1999).

Hölldobler, S., and Störr, H.-P. 2000. Solving the entailment problem in the fluent calculus using binary decision diagrams. In Traverso (2000).

Holzmann, G. 1991. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey.

IBM Corporation. 2001. Autonomic computing - a manifesto. www.research.ibm.com/autonomic.

Jensen, R., and Veloso, M. 2000. OBBD-based universal planning for multiple synchronized agents in non-deterministic domains. In Chien, S.; Kambhampati, S.; and Knoblock, C., eds., *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling*, 167–176. AAAI Press, Menlo Park.

Lamport, L. 1980. Sometime is sometimes not never. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, 174–185.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–56.

McIlraith, S., and Son, T. C. 2002. Adapting golog for composition of semantic web services. In Fensel, D.; McGuiness, D.; and Williams, M.-A., eds., *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning*, 482–493. Morgan Kaufmann, San Francisco.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP - simple hierarchical ordered planner. In Dean, T., ed., *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 968–973. Morgan Kaufmann, San Francisco, CA.

Nebel, B., ed. 2001. *Proceedings of the 17th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, San Francisco, CA.

Piccinelli, G.; Emmerich, W.; Zirpins, C.; and Schuett, K. 2002. Web service interfaces for inter-organisational business processes - an infrastructure for automated reconciliation. In Wegmann, A., and Duddy, K., eds., *Proceedings of the 6th International IEEE Conference on Enterprise Distributed Object Computing*, 285–292. IEEE Press, Los Alamitos.

Pistore, M., and Traverso, P. 2001. Planning as model checking for extended goals in non-deterministic domains. In Nebel (2001), 479–484.

Traverso, P., ed. 2000. *AIPS Workshop on Model Theoretic Approaches to Planning*.

Walther, C., and Schweitzer, S. 2001. Verifun user guide. Technical Report VFR 02/01, Technical University Darmstadt.