Construction of Messaging-Based Enterprise Integration Solutions Using AI Planning

Pavol Mederly, Marián Lekavý, Marek Závodský, and Pavol Návrat

Faculty of Informatics and Information Technologies,
Slovak University of Technology,
Ilkovičova 3, 842 16 Bratislava 4, Slovak Republic
{mederly, lekavy, zavodsky, navrat}@fiit.stuba.sk

Abstract. This paper presents a novel method of using action-based planning for construction of enterprise integration solutions that utilize messaging technologies. More specifically, the presented method is able to generate a sequence of processing steps needed to transform input message flow(s) to specified output message flow(s), taking into account requirements in areas of throughput, availability, service monitoring, message ordering, and message content and format conversions. The method has been implemented as a research prototype. It has been evaluated using scenarios taken from the literature as well as from real-world experience of the authors.

Keywords: Enterprise Application Integration, Enterprise Integration Patterns, Messaging, Action-Based Planning, STRIPS-like Planning

1 Introduction

Enterprise application integration deals with making independently developed and sometimes also independently operated applications in an enterprise to work together to produce a unified set of functionality [5]. It is a significant concern for many enterprises and a lot of resources are being spent in order to achieve its aims [2].

When creating an integration solution many issues have to be addressed. Some of them are related to correctly designing the *business logic* of the solution, i.e. algorithms such as "when a purchase order arrives, the system has to call 'check customer credit' and 'check inventory' services[1], and then, depending on the results, it should continue with order processing or return an error message to the client". Currently this business logic is, and very probably will continue to be, designed by human users, preferably by business analysts in cooperation with business owners.

Other issues to be addressed when designing an integration solution can be characterized as technical ones: these are concerned with differences in message transport

[1] Slightly simplifying, in this paper we use the term "service" to denote any software component providing business functionality that needs to be integrated and also to denote any component providing integration facilities like message routing, message format conversion, etc. The former are sometimes called business services, while the latter are called mediation (or integration) services.

protocols, application programming interfaces (APIs), message format and syntax, security requirements, availability and performance properties, logging and auditing requirements, etc. Resolution of these issues is driven by requirements and capabilities of participating services, by technical infrastructure available, and by business requirements. Although powerful tools in this area have appeared recently, e.g. those grouped under umbrella term “Enterprise Service Bus” [2], technical aspects are still being dealt with by people, mostly IT specialists.

In the long term, our research is directed towards automatic or semiautomatic resolution of these technical issues.[2] One of the first results achieved is a method of generating parts of messaging-based integration solutions using action-based planning approach. From many potential design aspects present in creating such solutions we have concentrated on the following ones: throughput, availability, service monitoring, message ordering, and message content and format conversions.

We have chosen the planning approach because there is a strong similarity between searching for an integration solution and planning in general: when constructing an integration solution we are looking for a sequence of services transforming input message flow to an output one, while when planning we are looking for a sequence of actions transforming the world from the initial state to a goal state. From the practical point of view it is reasonable to use existing planners capable of efficiently finding such sequences of actions (i.e. plans).

The remainder of this paper is organized as follows: Section 2 briefly characterizes messaging-based integration solutions, exemplifying some aspects of their design using a case study. In Section 3 the novel method of using an action-based planner to generate integration solutions is presented. Section 4 is devoted to the prototype implementation and the evaluation of the method. Section 5 describes related work. Section 6 closes this paper, giving some ideas on future work.

2 Messaging-Based Integration Solutions

The hypothetical online retailer company “Widgets and Gadgets ’R Us” buys widgets and gadgets from manufacturers and resells them to customers.[3] The company wants to automate purchase orders processing. Since parts of the whole process are implemented in disparate (and incompatible) systems, in order to achieve seamless operation, the company has to integrate systems involved in the process.

Handling of purchase orders looks like this: Orders are being placed by customers through three systems, namely through web interface, call center and fax gateway. Each order is stored in a separate message. A message containing an order is then translated from source system-specific data model to a common data model. After that, the customer’s credit standing as well as inventory is checked. If both checks are successful, goods are shipped to the customer and an invoice is generated. Otherwise, the order is rejected.

Due to historical reasons, information about stock levels is kept in two separate systems: Widgets Inventory and Gadgets Inventory. So each purchase order is inspected to

[2] As also described in “Business-driven automated compositions Grand Challenge” [10].

[3] This case study is taken from [5].

see if the items ordered are widgets, gadgets, or something else. Based on this information the request for checking inventory is sent to one of these systems or to a special message channel reserved for invalid orders.

The situation is shown in Fig. 1. We have chosen to use the Business Process Modeling Notation (BPMN) here as it is a technology-independent way of process modeling.

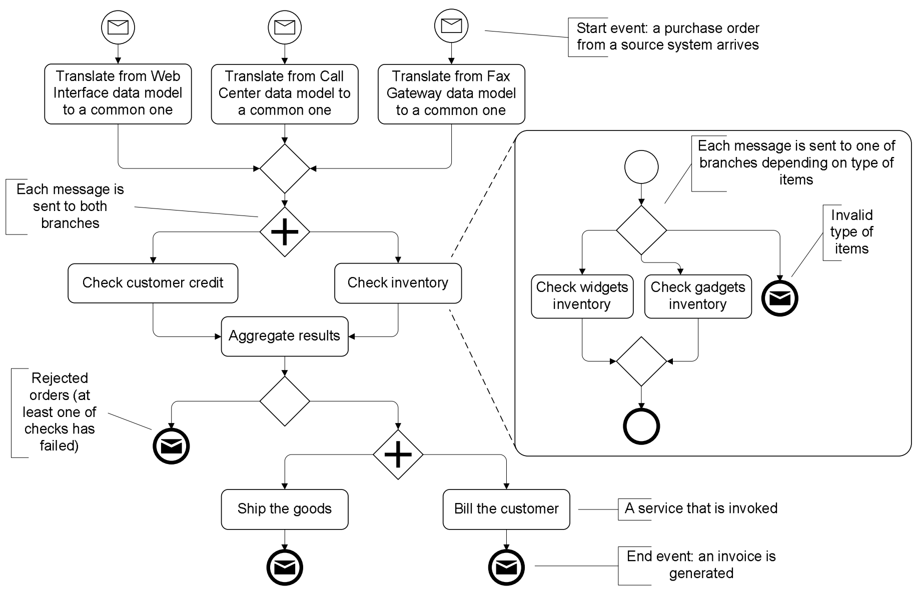
Start event: a purchase order from a source system arrives
Translate from Web Interface data model to a common one
Translate from Call Center data model to a common one
Translate from Fax Gateway data model to a common one
Each message is sent to both branches
Check customer credit
Check inventory
Aggregate results
Rejected orders (at least one of checks has failed)
Ship the goods
Bill the customer
A service that is invoked
End event: an invoice is generated
Each message is sent to one of branches depending on type of items
Invalid type of items
Check widgets inventory
Check gadgets inventory

Fig. 1. A sample integration scenario.

This kind of abstract description has to be implemented using concrete software entities. There are many technology approaches to do that. As stated above, in this paper we concentrate on messaging-based integration solutions, i.e. those that use an asynchronous messaging middleware (for example, a Java Message Service implementation) as a primary means for their components' communication.

This integration style is frequently used in practice. Below we shortly describe some of the design considerations that usually arise when creating solutions using this paradigm. These and other issues and their solutions are discussed in depth in the influential Hohpe and Woolf's Enterprise Integration Patterns book [5] in the form of patterns. We extensively use these patterns to describe integration solutions generated by our method.

Messaging-based integration solutions generally follow the Pipes and Filters pattern:[4] they receive messages at their input side (in one or more message flows), process them by a sequence of services connected by various channels, and put them on the output side (again, in the form of one or more message flows).

[4] Hohpe and Woolf's patterns are referenced by using names with capitalized words.

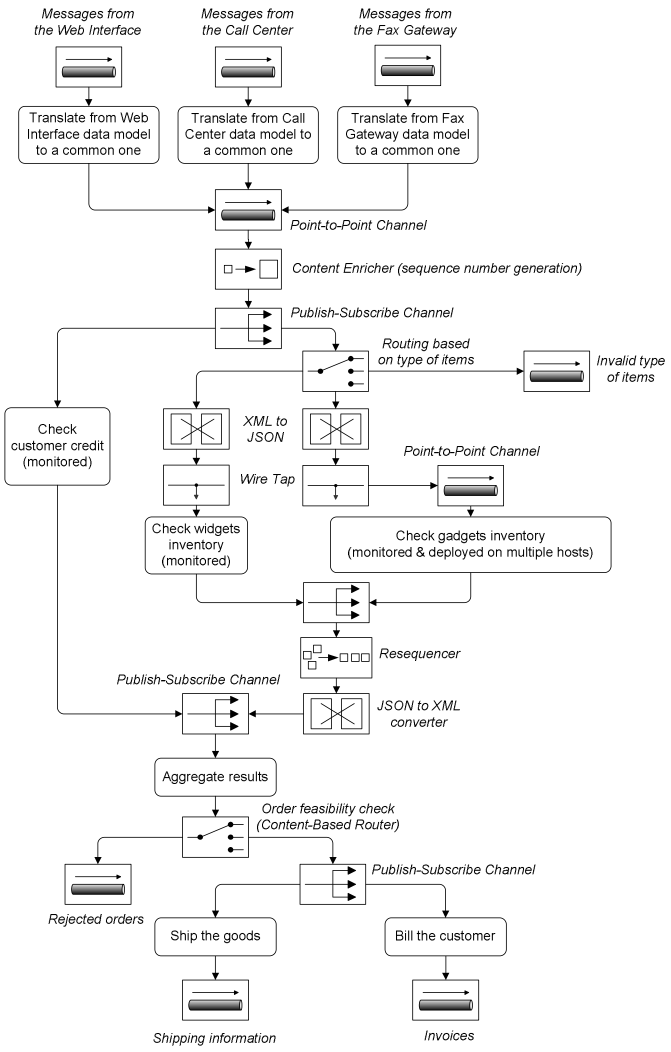
Messages from the Web Interface
Messages from the Call Center
Messages from the Fax Gateway
Translate from Web Interface data model to a common one
Translate from Call Center data model to a common one
Translate from Fax Gateway data model to a common one
Point-to-Point Channel
Content Enricher (sequence number generation)
Publish-Subscribe Channel
Routing based on type of items
Invalid type of items
Check customer credit (monitored)
XML to JSON
Wire Tap
Point-to-Point Channel
Check widgets inventory (monitored)
Check gadgets inventory (monitored & deployed on multiple hosts)
Resequencer
Publish-Subscribe Channel
JSON to XML converter
Aggregate results
Order feasibility check (Content-Based Router)
Publish-Subscribe Channel
Rejected orders
Ship the goods
Bill the customer
Shipping information
Invoices

Fig. 2. A messaging-based implementation of the sample integration scenario.

One of fundamental design choices is how individual services should be connected. A standard way of communication is through messaging middleware, using either Point-to-Point Channels (often called "queues") or Publish-Subscribe Channels (often called "topics" or "subjects"). The basic difference between these types of channels is that a message arriving at a Point-to-Point Channel is consumed by *exactly one* of receivers listening on this channel, while message arriving at a Publish-Subscribe Channel is consumed by *all* receivers listening on that channel. If services reside in the same address space, they can communicate via in-memory channels as well, eliminating the overhead of going through messaging middleware.

It is often the case that the *throughput and/or availability of a service* running in single thread of execution is not adequate. A typical solution is to deploy such a service in multiple threads, in multiple processes, or even on multiple hosts, using the Message Dispatcher and/or the Competing Consumers pattern. This limits the choices of an input channel for that service, e.g. not allowing topics to be used in some cases.

Another issue is that we might need to *monitor* some services – for example their quality-of-service attributes like throughput, response time, or availability. In most cases this means that we have to be able to "see" messages entering selected services and messages leaving them – typically using topics or a special Wire Tap service.

Yet another issue is that of message *ordering*. There are situations when it is important that the order of messages in the flow is kept unchanged. Unfortunately, in some cases, typically when processing messages in multiple threads, their order is not preserved. The solution is often based on a Content Enricher generating message sequence numbers paired with a Resequencer restoring original order of messages.

Almost all services require the messages to be in a specified *format*, e.g. comma-separated values, fixed-length records, XML, JSON (JavaScript Object Notation), or other. The integration architect has to employ specific converters appropriately.

Even though the basic process structure is designed by a business analyst, it is sometimes possible to choose from *alternative implementations of business services* available, depending e.g. on quality-of-service attributes, ease of access, and/or cost.

In order to see a concrete integration solution let us consider the following requirements: (1) while all services in our scenario work with messages in XML format, inventory checking ones use JSON format instead, (2) due to performance reasons the "Check gadgets inventory" service has to be deployed on multiple hosts, (3) we need to monitor correct functioning of credit checking service and both inventory checking services, and (4) the order of messages arriving at "Order feasibility check" service should be the same as original order of messages at the input side.

One of possible solutions implementing these requirements is shown in Fig. 2. It should be noted that this integration solution has been generated by the prototype implementation of our method. It is correct and optimal with respect to number of components. Placing Resequencer in one of the branches might look a bit strange, yet it is adequate – as the ordering of messages in the left branch is unchanged, the "Aggregate results" service produces a message flow with the original ordering of messages.

3 Description of the Method

Given a specification of an integration scenario (consisting of description of input and output message flows, services available and other requirements and constraints), the method uses a planner to generate an integration solution, i.e. a structure of services that will transform input message flow(s) to output one(s) while complying with specified requirements and constraints.

The *message flow* is a fundamental concept of the method. We do not keep track of individual messages processed by the integration solution – we observe message flows instead. A flow is characterized e.g. by content and format of messages it contains, their ordering, and so on (see below). A message flow is being transformed by services, can be split into alternative or parallel flows, and these flows can be eventually joined back into one flow later.

3.1 Input and Output of the Method

The input of the method consists of the following:

1. A specification of *input message flow(s)*. Each flow is described in terms of the content and format of messages in it and the type of channel it is present in. As an example, a flow can be specified as "a sequence of messages representing purchase orders entered via web interface, XML-formatted, available in a message queue". Note that the scenario described in our case study has three input message flows.
2. A specification of required *output message flow(s)*. This type of flow is specified in the same way as an input flow, with the possibility to add a requirement that messages should be in the same order as they were at the input. The scenario described in the case study has four output message flows.
3. A set of *services* that are available for processing the messages. These can be business and mediation services. Each service is described by a set of parameters, namely (1) input/output message content and/or format, (2) throughput and availability characteristics per deployment mode, (3) parameters related to monitoring and message ordering, (4) a cost of using the service.
4. A set of *other conditions* that have to be met, e.g. (1) the integration solution has to provide a sustained throughput of at least 100 messages per minute, (2) the availability of solution has to be "normal"[5], (3) each call to "Check customer credit", "Check widgets inventory" and "Check gadgets inventory" service has to be monitored.
5. A set of general *configuration options*, like whether to use action costs (or use a default cost of 1 for each action instead) or whether to take formatting, monitoring and message ordering aspects into account. This last option is very important for optimizing time taken to generate plans, as will be shown.

The output of the method is the plan that encodes a structure of services sought. Concrete examples of the method's input and output as implemented in the prototype can be found in [7].

3.2 Action-Based Planners

Due to similarities between searching for an integration solution and planning we conduct a search of the services to be used by employing a symbolical action-based planner.

Action-based (or STRIPS-like) planners, as descendants of the automated planner STRIPS [3], are based on the situation calculus. States of the world (situations) are described as conjunctions of grounded first-order predicate formulas; these formulas are literals. A state of the world can be modified by applying operators.

An operator is a triple $Op = (pre, del, add)$ where *pre* is a set of predicate formulas that must be satisfied in order for the operator to be invoked (a precondition), *del* is a set of predicate formulas that are deleted and *add* are predicate formulas that are added to

[5] We are considering 3 levels of availability: "low", "normal", and "high".

the description of the state of the world. Together, *del* and *add* represent the effect of the operator. The operators can be parameterized, i.e. predicate formulas in *pre*, *del*, *add* are allowed to contain free variables.

The planning problem consists of the planning domain (a set of operators) and the definition of the initial state and the goal state (states). The planner then tries to find a plan, consisting of operators that incrementally transform the world from the initial state to a goal state. Operators used in a plan are called actions and are usually required to have all their variables bounded. Although the plan is usually a sequence of actions, it is also possible to create plans with concurrent actions.

A frequently used algorithm of action-based planning is based on sequential adding of operators to the plan. Plan construction is guided by operators' preconditions and effects, usually employing some kind of heuristics. In our method, we only use the planner as a black box. The exact plan search algorithm is not important, as long as it provides correct results in acceptable time. More information on action-based planning can be found e.g. in [12].

3.3 Detailed Method Description

The principle of the method is the following: the integration problem to be solved is transformed into an input data for an action-based planner, written using the standard Planning Domain Description Language (PDDL). The planner is then executed and its output (i.e. the plan) is interpreted as a structure of services forming the integration solution. The situation is depicted in Fig. 3.

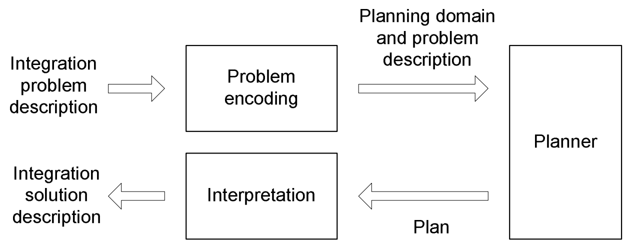
Integration problem description
Problem encoding
Planning domain and problem description
Planner
Integration solution description
Interpretation
Plan

Fig. 3. Basic principle of our method

Integration problem encoding works as follows: Message flows that are present in the integration solution correspond to the planner's states of the world. The state of the world changes as individual services (or other elements) of the solution process their incoming flow(s) and generate their outgoing one(s): an operator corresponding to such an element replaces predicate formula(s) corresponding to its input flow(s) in the state of the world by formula(s) corresponding to its output flow(s). The initial state of the world therefore corresponds to the input flow(s) entering the solution, and the goal state corresponds to the expected output flow(s).

The plan (a sequence of actions, i.e. operators applied) represents the integration solution we are looking for. Actions in the plan correspond to the technology elements of the solution and action dependencies (in the form of predicate formulas) correspond to connections (carrying message flows) between the elements. The transformation from the plan to integration solution description is straightforward.

In the following paragraphs we explore the problem encoding in more details.

State Description. Each message flow is described by the following predicate formula:

```
message (?Content ?Format ?Ordered ?OrderMarked
         ?Monitoring ?Channel ?FlowID)
```

`message` is a 7-ary predicate symbol. This predicate formula describes a message flow that fulfills the following conditions: (1) messages in the flow have content prescribed by the variable `?Content`[6], (2) messages in the flow are well formed according to a format specified by the variable `?Format`, (3) order of messages in the flow is (or is not, as indicated by the variable `?Ordered`) guaranteed to be identical to the order to messages in the input flow, (4) the original message ordering is (or is not, as indicated by the variable `?OrderMarked`) recorded somewhere, typically in a dedicated message property, (5) the messages are in a monitoring-related state prescribed by the variable `?Monitoring`, and (6) the messages are (or are not) currently available in the messaging middleware as prescribed by the variable `?Channel`. The last variable is used to distinguish among multiple identical flows coming out e.g. from a topic or from a Recipient List service.

If there is only one flow active, the state of the world contains only one `message` predicate formula. If there are multiple flows, like in our case study, there is one `messa-ge` predicate formula for each flow.

Operators. The most important operators we use are those directly derived from services available. Each service is transformed to up to four operators, one for each of the following modes of deployment (parallelism levels): (1) single thread, (2) single process, multiple threads, (3) single host, multiple processes, and (4) multiple hosts. Each mode of deployment provides specific levels of performance and availability, e.g. CreditCheck service in parallelism level 1 could achieve a throughput of 100 messages per minute and availability at the level of "normal".

What operators is the service transformed into is controlled by: (a) the list of allowed parallelism levels given in the service description, (b) comparing solution throughput and availability requirements (goals) to throughput and availability characteristics of this service deployed at a particular level of parallelism.[7]

As an example, the CreditCheck service that expects a message with content "Order" (`c_ord`) and transforms it into a "Order with Credit Information" (`c_ord_cr-`

[6] Actually in some cases we use this variable to encode part of contextual information as well (an example of such information is "the flow contains orders that have been rejected").

[7] With a slight simplification we assume that the necessary and sufficient condition for the solution meeting its throughput and availability goals is that each of services involved meets these throughput and availability goals individually. We also assume that the performance and availability of underlying messaging middleware is not a limiting factor. Providing more sophisticated treatment of these aspects is one of the topics of future work.

`info`), deployed on multiple hosts, with monitoring required, is represented by the following operator (symbols starting with `?` depict operator parameters):

```
operator: CreditCheck_PL4_M // parallelism level 4
                            // (multiple hosts)
pre: message (c_ord xml ?Ordered ?OrderMarked
              monitored ?Channel ?FlowID)
     acceptable_input_channel_for_PL4 (?Channel)
del: message (c_ord xml ?Ordered ?OrderMarked
              monitored ?Channel ?FlowID)
add: message (c_ord_crinfo xml unordered ?OrderMarked
              monitoring_requested channel_memory_PL4 ?FlowID)
```

Correct use of channels is controlled by `acceptable_input_channel_for-_PLx` predicates that allow e.g. for PL1 the use of a topic, a queue or a single-process in-memory channel. For PL4 it is allowed to use only a queue or an in-memory channel going out from a previous PL4-deployed service.[8] Transport of messages through messaging middleware is modeled by two special operators, `Queue` and `Topic`. If one of them is added after a service, it means that the service sends its output through this kind of channel. This allows modeling the fact that sending messages via these channels is more costly than using direct in-memory connections (when using a planner that supports action costs this can be expressed quantitatively).

Message ordering is accounted for by a set of two services: an OrderMarker service (implementing a Content Enricher pattern) that puts sequence numbers of messages into a dedicated message property, and a Resequencer service that reestablishes the order of messages using this property. When using multiple (alternative or parallel) message flows special care is taken with respect to message ordering and message order marking, but we have no space to cover the details here.

Monitoring is dealt with in the following way: services that have to be monitored require the value of `monitored` in the `?Monitored` parameter of `message` predicate formula in the precondition and set the value of `monitoring_requested` in the `?Monitored` parameters in its effects. This ensures that the input and output messages go either through a topic or through a WireTap service, because only operators corresponding to a topic and the WireTap are able to set the `monitored` value and "clear" the `monitoring_requested` one. Services that do not have to be monitored expect the value of `not_monitored` or `monitored` at the input and set the value of `not_monitored` at the output. As in message ordering aspect, the monitoring flag is also treated specially when using multiple flows.

For *joining alternative or parallel message flows*, a special "declarative" kind of operator is used. This operator does not correspond to any real service, it only serves as a declaration that message flows containing e.g. results from check of widgets inventory and results from check of gadgets inventory are merging in one queue into a message flow containing results from (an unspecified) inventory check. When using action costs these declarative operators are assigned the lowest possible cost. (When merging parallel flows, such declarative operator should be followed by a "real" Aggregate service, like the "Aggregate results" service in Fig. 2.)

[8] Assuming that this service runs in the same set of processes as its predecessor.

In order to shorten the time needed to find a plan it is possible to choose whether the solution should take aspects of monitoring, message formats, and message ordering into account. Parameters (of predicate formulas and operators) for disabled aspects are not created, so for example the `message` predicate can have an arity from 3 to 7, depending on the settings. As shown in the following section, this has strong performance implications.

4 Implementation and Evaluation

The method described above has been implemented in the form of a research prototype.

We have tried several planners. For practical reasons we have limited our search to those accepting PDDL as an input language. The selection of planners was guided by the results at the International Planning Competitions[9] and our previous experience.

We have selected four problems to demonstrate the evaluation results here: Problems 1 and 2 correspond to a part of Widgets and Gadgets order processing scenario described in Section 2. The part covered begins when orders from three sources are merged in a queue and ends as orders enter feasibility check. Problem 1 takes into account monitoring and throughput aspects. Problem 2 takes into account aspects of monitoring, message format, and throughput. Problems 3 and 4 capture the whole order processing scenario as described in the case study; Problem 3 does not take aspects of monitoring, message format, message ordering, and throughput into account, while Problem 4 does. These settings are summarized in Table 1.

Table 1. Description of problems selected for method evaluation.

# Scope	Aspects	Message predicate arity	Parameters/ operator	# of operators	Domain objects	Optimal plan length
1 reduced	M, T	4	3.67	21	21	15
2 reduced	M, T, F	5	4.12	25	23	19
3 full	–	3	2.50	22	28	26
4 full	M, F, O, T	7	6.81	36	39	36

Acronyms for aspects are: monitoring (M), message formats (F), message ordering (O), and throughput (T). “Message predicate arity” refers to the arity of the message predicate, “Parameters/operator” means average number of parameters of individual operators. These measures, along with number of operators and number of objects in the domain, very roughly indicate the size of state-space and plan-space that have to be searched – a major factor of complexity of the planning process.

The results (quality of solution found and CPU time needed to find it) for some of the planners are summarized in Table 2.[10]

[9] http://ipc.icaps-conference.org/

[10] These results are only informative: some planners provided settings affecting performance, e.g. possibility to choose heuristics, weighting factors, etc. We tried to find optimal settings, but in some cases it might be possible to find better settings.

Table 2. Characteristics and results of selected planners.

Planner	Domains solved	Plan search algorithm	Problem 1	Problem 2	Problem 3	Problem 4
Gamer	non-det, cost, seq	state, opt, conformant	O: 89.2s	O: 742.97s	O: 363.56s	Error
MIPS-XXL	cost, seq	state, opt	O: 1.74s	O: 6.00s	O: 177.4s	Error
HSP 2.0	seq	state, subopt	O: 0.15s	O: 0.43s	SO: 0.09s	O: 15.47s
FF 2.3	seq	state, subopt	O: 0.48s	O: 0.89s	SO: 0.07s	Error
SatPlan2006	par	SAT, opt	O: 7.35s	O: 12.57s	O: 2.67s	Error
MaxPlan	par	SAT, opt	O: 0.02s	O: 0.01s	O: 0.02s	Error

Acronyms for domains and search algorithms are: sequential plans (seq), parallel plans (par), action costs (cost), non-deterministic actions (non-det), state-space search (state), transformation to satisfiability problem (SAT), generating optimal plan (opt), not guaranteed to generate optimal plans (subopt). Acronyms for results are: optimal plan found (O), suboptimal plan found (SO), computation failed (Error).

These results show that our method is able to find solutions for practical integration problems using currently available planners. The majority of the planners had difficulties solving the most complex Problem 4. We suspect that some of them were not designed to work with such a large state-space as it was present in this problem. In our opinion, however, this is not a result of planning methods used, but more a result of implementation decisions made by the planners' designers.

When considering experiences from solving these and other integration problems, as the most suitable come HSP 2.0 planner (it is fast, although it produces suboptimal plans in some cases) and MIPS-XXL and Gamer (they are slower, but generate optimal sequential plans).

The planners are of different types, for example some generate sequential plans while other parallel ones. What type of planner do we actually need? Generally, it depends on what we want to optimize. Basically there are four (interrelated) possibilities:

1) integration solution complexity (number of components used in the solution),
2) latency (time needed for an integration solution to process a message),
3) throughput (number of messages processed by the solution per time unit),
4) resource consumption (e.g. network bandwidth, CPU time of message broker and/or application servers and so on).

At this moment, we use the criterion 1, corresponding to the shortest sequential plan. We can incorporate the criterion 4 by assigning costs to individual actions based on resource consumption. If we would like to optimize latency (criterion 2) we could use a parallel planner with durative actions (i.e. actions that have been assigned an execution time). For now we stay with the goal of finding the solution with the smallest complexity and leave the issue of exact optimality definition to be a subject of a future research. Therefore, for now optimal sequential planners with no other extensions are sufficient. The use of parallel planners in this context has a drawback in that they generally produce plans with the optimal makespan (number of steps of plan execution), not optimizing the number of involved actions. This way the solution usually contains more software components than necessary.

We have also created several additional scenarios stemming from real-life experience of the authors [6] and have verified that the solutions produced by the prototype implementation are correct and optimal in the sense of number of components.

More detailed evaluation report that includes descriptions of integration problems, PDDL files, output produced by individual planners as well as the discussion of results achieved by particular types of planners can be found in [7].

As a final remark, we can say that this method solves a defined subset of technical problems present in the creation of messaging-based integration solutions. We expect that the business-level decisions, e.g. the correct sequence of business services that have to be integrated or the business rules governing the mapping between incompatible message schemas and/or semantics, have to be provided by business users and are therefore out of scope of this method. Similarly, there are many technical aspects presently not dealt with by the method – e.g. creation of mediation components solving message schema and business protocol incompatibilities between systems being integrated, design issues related to security and reliability of the solution, issues of different communication protocols/mechanisms used, etc. Some of these are being worked on by the research community (e.g. [8]), while others are the subject of our future work. Finally, there are issues like dealing with unclear, changing and/or not correctly implemented components' interfaces that are possibly out of scope of any automated method for integration solution creation.

5 Related Work

We know of no existing research dealing with the problem of automatic creation of messaging-based integration solutions in a way similar to the one outlined above. More generally, however, there are several attempts to reduce human effort needed to solve technical issues present in construction of such solutions. Many of them are based on the idea of developing an abstract, platform-independent model of the integration solution and then incrementally refining it into platform-specific model(s) and/or executable code. For example, Scheibler and Leymann [14] based their platform-independent models on enterprise integration patterns [5] enriched by configurable parameters. These models can then be automatically transformed into executable code, either written in Business Process Execution Language or in a configuration language of a specific integration tool (Apache ServiceMix).

This and similar approaches require the developer to specify technology-related model elements manually – albeit at a higher level of abstraction, compared to directly writing platform-specific configuration or code. Our method, in contrast, generates such elements automatically.

The work of Umapathy and Purao [16] is directed towards choosing solution elements (described in the form of integration patterns) automatically. The authors have devised an inference engine based system that accepts an integration scenario described as annotated model written in Business Process Modeling Notation and offers the developer a set of possible enterprise integration patterns to implement individual parts of the scenario. The difference to our approach is that this system only provides an aid to the developer, offering him or her a set of more or less relevant alternatives to choose from. Our method, in contrast, automatically generates a directly executable solution. The human interaction can be added in the future, but it is not necessary for the method to work.

AI planners have been successfully used in various areas of software engineering. Some examples are test cases generation [4,13] or assembly of algorithms having specified characteristics, using components from a properly annotated domain-specific library [15]. Another example is the method of creating a plan for dynamic reconfiguration of distributed systems after failure [1], focusing mainly on a (re)distribution of components and connectors to individual machines and finding a correct sequence of starting machines, components and connectors. More related to the topic of this paper is the use of AI planners for web service composition, as described e.g. in [9] and [11]. Works in this area deal primarily with creating compositions at the business level, not paying attention to technical aspects, as we do.

6 Conclusion and Future Work

The research results presented here demonstrate that planning techniques are useful for solving technical problems related to enterprise application integration. Our method is able to create parts of integration solutions from the description of “what” has to be achieved, not “how” it should be done. Evaluation using a case study and a set of real-world integration scenarios has shown that the method presented is able to solve practically-sized problems. In comparison, existing works in this area either require the developer to specify technology-related elements manually or, if not, they are not able to generate an executable solution (only provide hints for the developer).

Our aims for the future include covering other aspects of messaging-based integration solutions, for example questions of assured message delivery (using Guaranteed Delivery and/or Transactional Client patterns) or a support for diverse message transport protocols. We also plan to provide a code-generation module that would read the plan and provide a partially or fully executable integration solution for selected integration tool(s). As mentioned above, we want to further elaborate the notion of solution optimality and aspects of throughput and availability.

We also plan to have a look at the possibility of using other techniques for automatic generation of the solution, like logic inference engines, and integrate them in a framework for automatic or semi-automatic resolution of technical issues present in enterprise application integration solutions construction.

Acknowledgments

This work was partially supported by the Slovak Research and Development Agency (contract No. APVV-0391-06) and by the Scientific Grant Agency of Republic of Slovakia, grant No. VEGA 1/0508/09.

References

1. Arshad, N., Heimbigner, D., Wolf, A. L.: Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. In: Proceedings of 15th IEEE International Conference on Tools with Artificial Intelligence, pp. 39–46. IEEE Computer Society (2003)
2. Chappell, D.A.: Enterprise Service Bus. O'Reilly Media, Inc., Sebastopol, CA (2004)
3. Fikes, R. E., Nilsson, N. J.: STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence 2, 189–208 (1971)
4. Fröhlich, P., Link, J.: Automated Test Case Generation from Dynamic Models. In: ECOOP 2000. LNCS, vol. 1850, pp. 472–491. Springer, Heidelberg (2000)
5. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Pearson Education, Inc., Boston, MA (2004)
6. Mederly, P., Pálos, G.: Enterprise Service Bus at Comenius University in Bratislava. In: EUNIS 2008 VISION IT: Visions for IT in Higher Education, p. 129. University of Aarhus, Aarhus, Denmark (2008) Full text available at: http://eunis.dk/papers/p98.pdf
7. Mederly, P., Lekavý, M..: Report on Evaluation of the Method for Construction of Messaging-Based Enterprise Integration Solutions Using AI Planning, http://www.fiit.stuba.sk/∼mederly/evaluation.html
8. Nezhad H. R. M., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-Automated Adaptation of Service Interactions, In: Proceedings of WWW 2007, pp. 993–1002, ACM (2007)
9. Oh, S.-Ch., Lee, D., Kumara, S. R. T.: A Comparative Illustration of AI Planning-based Web Services Composition, ACM SIGecom Exchanges 5(5), 1–10 (2005)
10. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., Krämer, B.J.: Service-Oriented Computing Research Roadmap, In: Cubera, F., Krämer, B.J., Papazoglou, M.P. (eds.) Dagstuhl Seminar Proceedings 05462. Internationales Begegnungs-und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
11. Rao, J., Su, X.: A Survey of Automated Web Service Composition Methods, In: SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2005)
12. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach (2nd ed.). Prentice Hall, Upper Saddle River, NJ (2003)
13. Scheetz, M., von Mayrhauser, A., France, R., Dahlman, E., Howe, A. E.: Generating Test Cases from an OO Model with an AI Planning System. In: Proceedings of 10th International Symposium on Software Reliability Engineering, pp. 250–259. IEEE Computer Society (1999)
14. Scheibler, T., Leymann, F.: A Framework for Executable Enterprise Application Integration Patterns. In: Mertins, K. et al. (eds.) Enterprise Interoperability III, pp. 485–497. Springer, Heidelberg (2008)
15. Troy, A. J., Eigenmann, R.: Context-Sensitive Domain-Independent Algorithm Composition and Selection. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pp. 181–192. ACM (2006)
16. Umapathy, K., Purao, S.: Representing and Accessing Design Knowledge for Service Integration. In: Proceedings of 2008 IEEE International Conference on Services Computing, pp. 67–74. IEEE Computer Society (2008)