Semi-automated Construction of Messaging-Based Enterprise Application Integration Solutions

Pavol Mederly[*]
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 3, 842 16 Bratislava, Slovakia
mederly@fiit.stuba.sk

Abstract

Enterprise application integration, i.e. an endeavor to make independently developed information systems cooperate, is an important topic of enterprise computing for decades. Despite many efforts, both in industry and academic area, integration solutions development is still often a costly, error-prone process.

The goal of presented work is to make messaging-based integration solutions development and maintenance more efficient. In comparison to existing model-driven approaches that aim to generate code for integration solutions we are trying to reach a more advanced goal: to automate not only the code generation but the detailed design as well. In order to do this, we use artificial intelligence techniques, namely planning and constraint satisfaction. In this work we present a set of methods that – for a given integration solution abstract design and non-functional requirements (like throughput, availability, monitoring, minimal communication middleware usage, and so on) – create a suitable solution design and in some cases an executable code as well.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*computer-aided software engineering*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

[*]Recommended by thesis supervisor: Prof. Pavol Návrat. Defended at Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava on September 30, 2011.

Mederly, P. Semi-automated Construction of Messaging-Based Enterprise Application Integration Solutions. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 3, No. 3 (2011) 44-54

Keywords

enterprise application integration, messaging, integration patterns, constraint satisfaction, action-based planning

1. Introduction

Enterprise application integration, or EAI for short, deals with making independently developed and sometimes also independently operated applications in the enterprise to communicate and cooperate – in order to produce a unified set of functionality [11]. A software system that enables such cooperation is often called an *integration solution.*

Application integration and its accompanying area, service oriented computing, are considered to be "hot topics" in the information technology industry, because of their significance for enterprises seeking efficiency and flexibility in today's dynamic environment. For the same reasons, service oriented computing is an important research topic in academia as well [24].

In this work we deal with integration solutions based on *messaging* that utilize the Pipes and Filters architectural pattern, as described in [11]. These solutions can be characterized in the following way:

1. They consist of a set of components (filters) that communicate by sending and receiving messages through channels (pipes), primarily, although not necessarily, in a reliable and asynchronous way. These components carry out functionality related to the integration problem itself – like accessing information systems that have to be integrated or translating messages between formats they use – or auxiliary functionality necessary for the implementation of required flow of control and data within the solution, like duplicating, routing, logging, splitting, or aggregating messages.
2. As a primary, although not exclusive, mean of communication these solutions use messaging middleware, also called message-oriented middleware or message queuing. It is a preferred technology for creating integration solutions, because it allows creating solutions characterized by loose coupling of their components, both at design and execution time [11].

As in the case of any other software product development, also when creating an integration solution there are a couple of distinct software engineering activity types: one of commonly used classifications recognizes requirements specification, software design, implementation, validation, and operation and maintenance. There are several approaches that assist the developer with the implementation activities in the domain of messaging-based integration solutions, e.g. [26, 29]. Our basic question is: *Can we help the developer more?* Is it possible to provide any methods and tools that are useful during the *solution design*? As described in this paper, we have succeeded in using traditional artificial intelligence techniques, namely planning and constraint programming, in order to reach this goal.

The structure of this paper is as follows: In Section 2 we describe state of the art in the area of automating integration solution creation. Section 3 brings the formulation of our research problem. Then sections 4 to 7 describe our main result – methods for automating selected aspects of integration solutions design and their evaluation. The final section contains a conclusion and outlines possibilities for consequential research.

2. State of the Art

Technical issues of creating messaging-based integration solutions have been the subject of recent research efforts, undertaken mainly in the area of *model-driven integration.* These efforts are based on an idea of constructing integration solutions using an approach starting with an abstract description of the solution and stepwise enhancing it by adding more details, either manually by developers or automatically by model transformation and code generation techniques.

Building on the work on enterprise application patterns (EIPs) [11], Scheibler and Leymann have proposed an idea of executable enterprise application integration patterns [26]. They have enriched original EIPs with configurable parameters in order to be able to use them as elements of platform-independent models of integration solutions. Authors have provided a graphical editing environment for creating integration solution designs and generating executable code for a chosen platform, either Business Process Execution Language (BPEL) [3, 28] or a configuration language for specific integration tools: Apache ServiceMix [21] or Apache Camel [13]. The approach is limited to using XML as a message format, and WSDL (Web Services Description Language) as a means of describing interfaces of systems being integrated. Authors applied their idea also in an outsourced, software-as-a-service setting [27].

Frantz, Corchuelo, and Gonzáles have proposed Guaraná, a modeling language for EAI based on entities that are very similar to enterprise integration patterns [7]. The principle of their work is comparable to the one described above. Currently authors claim a support for Microsoft Workflow Foundation as a target platform [29] with some limitations due to conceptual mismatches between the world of messaging-based integration and workflow automation. However, their approach is independent on a specific platform, and translators to other platforms are conceivable. Moreover, authors declare they work on their own runtime system to execute integration solutions written in Guaraná [6].

Overall, these works present a step forward to making development of messaging-based integration solutions more efficient. Namely, they relieve a developer from writing detailed, platform-specific configuration and/or code, and allow him or her to concentrate on more abstract, platform-independent aspects. However, they do not take into account non-functional requirements, like throughput, availability, message processing latency, and so on. A developer has to design an integration solution that meets such requirements "manually", knowing details of a selected integration platform, and reflecting this knowledge in the platform-independent models. (Those, then, become – at least partially – dependent on a chosen platform.)

Moreover, many of the enterprise integration patterns (e.g. Transactional Client), are of a highly technical nature. Also some others, e.g. Recipients List and Publish-Subscribe Channel, capture design decisions at quite detailed level. Therefore if one uses EIPs alone as a tool for modeling the integration solution, the business and technical aspects of the solution are strongly tangled.

An interesting attempt aimed towards a lifting of the level of abstraction has been performed by Umapathy and Purao [31, 32]. They also have recognized the fact that using enterprise integration patterns to describe integration solutions is at too technical a level. Moreover, they claim that the mapping from an abstract specification of the solution (in a process-oriented language like Business Process Model and Notation, or BPMN for short) to a concrete design described by a set of EIPs is a cognitively demanding task. Therefore they have devised a system based on Speech Act theory that assists the designer with the mapping from models in BPMN to sets of EIPs. A weak point of their method is that the abstract specification of the solution (i.e. the annotated BPMN diagram) captures very little information, so the designs provided to the developer are of varying relevance and the model itself cannot be used to generate directly executable solutions.

Another relevant research result is the BIZYCLE integration process [1, 22], an outcome of a project whose goal was to investigate the potential of model-based software and data integration methodologies. It helps the integration developer at multiple levels, ranging from the questions of incompatible semantics of data down to the level of technical interoperability. However, the BIZYCLE process does not cover the main question of our research – namely, how to design an effective messaging-based integration solution for a given target integration platform.

3. Overall Goal and Hypotheses

Given the situation described above, we state the main goal of this dissertation in the following way:

To find a way of partially or fully automating the process of design and implementation of messaging-based integration solutions, in order to improve some of their characteristics.

We are going to research methods that will help the developer to find a detailed design of a messaging-based integration solution that would comply with a defined abstract design, non-functional requirements, design goals and environment characteristics.

In order to achieve this goal we plan to confirm or refute the following two hypotheses:

HYPOTHESIS 1. *It is possible to partially or fully automate the detailed design and implementation of messaging-based integration solutions, given their abstract design (control and data flow specification), non-functional requirements, design goals and environment characteristics, utilizing planning and constraint satisfaction methods.*

However, automating the design process is not a goal in itself. What is important is whether this automation brings any real benefits for developers – manifesting themselves e.g. in shorter time to produce a solution for a given integration problem, in reducing the number of defects in such a solution, or in its better maintainability.

Although in the future we want to characterize these benefits quantitatively by measuring e.g. an effort needed to construct an integration solution, in this dissertation we plan to concentrate on a simpler aspect: properties of source code. We are going to research the following hypothesis.

HYPOTHESIS 2. *Methods of partial or full automation of design and implementation mentioned in Hypothesis 1 can lead to more concise source code compared to traditional way of integration solution development.*

By a source code for our approach we understand the code used to specify the input for our methods. We can reasonably assume that concise source code is easier to create, will contain fewer defects, and is easier to maintain.

4. General Description of our Approach

On our journey to semi-automated construction of messaging-based integration solutions we have explored two approaches: planning and constraint programming. We have developed a set of four methods listed below.

The input for such a method for semi-automated integration solution construction is an integration problem that consists of:

1. *abstract design*, namely the specification of control and/or data dependencies between systems or services that have to be integrated, without any technical details related to the deployment of solution components or to their communication,
2. *non-functional requirements specification*: (a) mandatory non-functional properties the solution has to have, like required throughput, availability, manageability, and so on, (b) design goals that are to be achieved, like minimization of the use of messaging middleware, balancing CPU load, and so on;
3. *description of the environment*, consisting of: (a) properties of systems or services that have to be integrated, (b) properties of integration (or mediation) services and communication channels that are available – in part they are given by the integration platform that has to be used.

The output of such a method is an executable integration solution, or – at least – its detailed design. Concrete methods that implement this approach are the following:

1. *Message-level, planning-based method* (or, shortly, the ML/P method) is a method for designing integration solutions dealing mainly with aspects of throughput, availability, monitoring, message ordering, translating between different message contents and formats, and finding the best way of deployment at a coarse level. It does not deal with internal structure of messages (hence "message-level"). It uses action-based planning. [18]
2. *Data element-level, planning-based method* (or the DL/P method) is a method specifically aimed at managing data elements in messaging-based integration solutions. It uses action-based planning as well. [15]
3. *Message-level, constraint programming-based method* (or the ML/CP method) is the first of the methods using constraint programming. Its application area is very similar to the one of ML/P with a slightly extended set of design aspects it deals with. [20]
4. *Universal constraint programming-based method* (or the U/CP method) is the most comprehensive of our methods. In current version it solves almost all aspects covered by previous methods along with several additional ones, and goes into much more details when designing the solution. It is able to produce directly executable code for selected integration platforms. It uses constraint programming as well. [16, 19]

4.1 Input of the Methods

Our methods are devoted to designing an integration solution based on its abstract design, non-functional requirements specification and a description of target environment.

4.1.1 Abstract Design

The core of the abstract design is the flow of control and data that has to be implemented. This flow can be depicted in various ways; in the following we use UML (Unified Modeling Language) activity diagrams as well as our own textual, Java-like, domain-specific language developed as part of the U/CP method implementation.

As an example integration scenario used to illustrate the methods let us consider a hypothetical online retailer company "Widgets and Gadgets 'R Us" (WGRUS) [11]. This company buys widgets and gadgets from manufacturers and resells them to customers. The company wants to automate its purchase orders processing. Since parts of the whole process are implemented in disparate systems, our goal is to create an integration solution that would interconnect these systems in a required way (see Fig. 1 and 2).

Handling of purchase orders in the integration solution should look like this: Orders are being placed by company customers. After receiving an order, our solution ensures that the customer's credit standing as well as inventory is checked. If both checks are successful, goods are shipped

```
process ProcessOrder(Order order)
{
  fork-and-join
  {
    {
      Credit creditInfo = CheckCredit(order);
    }
    {
      for-each (Line orderLine in order) using "//wg:Lines/wg:Line"
      {
        LineInventoryInfo lineInventoryInfo;
        exclusive choice
        {
          case xpath("substring($orderLine/...) = 'W'") ratio 0.9:
            lineInventoryInfo = CheckWidgetInventory(orderLine);
          case xpath("substring($orderLine/...) = 'G'") ratio 0.09:
            lineInventoryInfo = CheckGadgetInventory(orderLine);
          default:
            reject "Invalid item type" sending orderLine;
        }
      }
      aggregate (lineInventoryInfo into InventoryInfo inventoryInfo);
    }
  }
  Status status = ComputeOverallStatus(order, creditInfo,
                                       inventoryInfo);
  if (xpath("$status/wg:Status = 'true'") ratio 0.85)
  {
    fork
    {
      Bill(order);
      Ship(order);
    }
  }
  else
  {
    forward-to "RejectedOrders" sending order;
  }
}
```

Figure 2: An example of specification of control and data flow, in the textual form.

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 split into individual order lines, and each of them is inspected to see if the item ordered is a widget, a gadget, 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. After checking the inventory, the information on individual order lines is aggregated into one message that concerns the order as a whole.

A basic unit of an abstract design is a *service*. It provides an access to a system that has to be integrated (in our case such services are e.g. `CheckCredit`, `CheckWidgetInventory`, and so on) or carries out some computation or data transformation (e.g. `ComputeOverallStatus`). Services are then connected together using language constructs that prescribe flow of both control and data among them. Examples of these constructs are `fork` and `fork-and-join` (used to split and optionally join the control flow, see the Scatter-Gather pattern [11]), `choice`, `if`, `else` (used to specify a decision point, see the Message Router pattern [11]), `for-each` and `aggregate` (used to ensure individual processing parts of a message, see Composed Message Processor pattern [11]).

4.2 Non-functional Requirements

Currently we deal with the following categories of requirements:

1. *Throughput*: We may require the solution to be able to continuously process a specified number of messages per time unit, e.g. per second or per minute.

2. *Availability*: We may require the solution to guarantee a specified level of availability, i.e. that it is able to process messages within defined time with a specified probability. In current version of the methods we do not use quantitative measures for availability; instead, for simplicity we have chosen a set of discrete values to denote "low", "normal" and "high" availability. Users of our methods have to decide for themselves what they understand by these qualitative levels.

3. *Message format*: At many places we may require that data have to be in a specified format, e.g. comma-separated values, fixed-length records, XML, JSON (JavaScript Object Notation), or other.

4. *Message ordering*: There can be a situation that the original ordering of messages is lost, typically in the case of parallel or alternative processing. It may be required that, at some places, messages should be present in their original order.

5. *Monitoring*: Sometimes there is a requirement that all messages going through a specific channel should be monitored, that means their content should be available to a monitoring tool.

6. *Message duplication*: When using messaging middleware, message duplication sometimes occurs. It can be a requirement that these duplicates not be present at specified points in a solution, i.e. they must be either not allowed to emerge, in the first place, or eliminated afterwards, if possible.

7. *Checkpointing*: Service containers occasionally fail. In such situations it is convenient to be able to resume message processing (after a container is restarted) from a known point, defined by the developer.

8. *Middleware usage*: Messaging middleware is sometimes an "expensive" resource, so it is a usual requirement that its use should be minimized, or even strictly limited to a defined value.

9. *CPU usage*: In the presence of multiple hosts it could be a requirement either to limit a CPU usage of each of them to a defined value, or to ensure that their usage is balanced (within given limits).

Some of these properties can be specified as design goals, e.g. it is possible to state we want to minimize the middleware usage, or balance the CPU usage as much as possible.

4.3 Description of Target Environment

In order to provide a correct concrete design, the method must have information about the target environment. First of all, it must know about properties of individual services that are referenced in the abstract design: their input/output characteristics (namely content and format of data items consumed/produced, and possibilities of their positioning in message parts), throughput and availability characteristics (depending on deployment), cost or resource usage data. The method must also know about available features of the execution platform, e.g. what types of message channels it can use, what integration services are available, and so on.

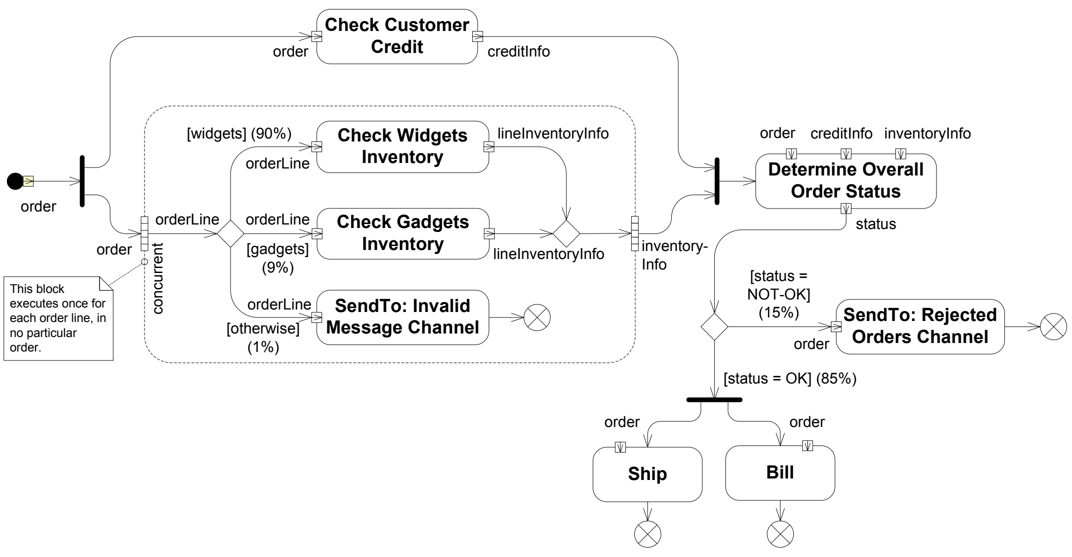


Figure 1: An example of specification of control and data flow, using UML.

4.4 Output of the Methods

We work with messaging-based integration solutions that follow the Pipes and Filters architectural pattern [11]: they receive messages that come through an input channel or channels, process them by a set of services connected by various channels, and put them into an output channel or channels. Besides services prescribed in the abstract design, the solution contains auxiliary services that implement the required control and data flow. These are implementations of various integration patterns, namely Wire Tap, Recipient List, Content Based Router, Splitter, Aggregator, Resequencer, Content Enricher, Message Translator, Composed Message Processor, Scatter-Gather, Competing Consumers, Message Dispatcher, and Transactional Client [11]. Channels can be a implemented in memory or they can use messaging middleware, and can be of both Point-to-Point and Publish-Subscribe types [11].

We have devised a representation of such an integration solution using a direct acyclic graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. Each vertex $v \in V$ can represent either a service or an auxiliary component (i.e. solution's input or output point, or a fork or join point). In order to model this fact we have defined a partial function $Service : V \to Services$ that, for each vertex $v \in V$ representing a service, gives this service, and for auxiliary vertices is undefined. Each edge $e_{ij} = (v_i, v_j) \in E$ represents a channel carrying messages from v_i to v_j. For a given vertex v we denote a set of its incoming edges $In(v)$ and a set of its outgoing edges $Out(v)$.

Each service and each channel have a set of properties. They are modeled usually by functions with a domain of V or E, respectively. We call these functions, along with the *Service* one, to be *property functions*. Property functions related to selected design aspects are shown in Tab. 1.

Using these functions we can formulate rules that characterize a correct integration solution. An example of such a rule is: "If a service's input channel is a topic, the service cannot be deployed into more than one service container at once." It can be formulated in the language of the first-order logic as

$$\forall v \in Dom(Service), e \in In(v) : \\ ContainerCount(v) > 1 \implies \\ ChannelType(e) \neq Topic.$$

Such rules are used by our methods to find a solution for a given integration problem.

4.5 Capabilities of Individual Methods

Individual methods differ in features they support, as shown in Tab. 2. Here the *message level* support of the *Channel content* aspect means that we restrict messages to carry only one kind of information, i.e. we are considering content of a message to be an indivisible unit. In contrast, *variable level* support of this aspect is more realistic: it allows messages to contain several data items, using a concept of *process variables* (like `order` or `creditInfo` in Fig. 2). *Coarse level* of the *Service deployment* aspect recognizes four basic modes of a service deployment (single thread, multiple threads in one process, multiple processes at one host, multiple hosts), whereas *Finest level* goes down to concrete numbers of threads in individual service containers. *Partial* support of the *Availability* aspect in U/CP means that the developer has to specify redundancy requirements explicitly, for example as "service `Bill` has to be deployed at two independent hosts", and the method creates a detailed design according to these requirements.

5. Methods Based on Planning

Methods ML/P and DL/P use action-based planning [25] to create a suitable design of an integration solution. We have chosen this approach because there is a strong similarity between creating an integration solution and planning in general: when constructing an integration solution, we are looking for a system, composed of services organized in a directed acyclic graph, that transforms input message flow(s) to output one(s), while when planning, we are looking for a sequence of actions transforming the world from an 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.

Table 1: Functions used to model basic aspects of messaging-based integration solutions.

Aspect	Vertex-related function(s)	Edge-related function(s)
All	$Service(v)$	-
Channel content	-	$Content(e)$, or $VariablePresence(e, var)$ and $VariablePosition(e, var, pos)$
Channel types	-	$ChannelType(e)$
Service deployment	$DeploymentMode(v)$, or $ThreadCount(v)$, $ContainerCount(v)$, $HostCount(v)$, and $Deployment(v)$	-
Monitoring	$MonTransparent(v)$	$Monitored(e)$
Message ordering	-	$Ordered(e)$, $OrderMarked(e)$
Message format	-	$Format(e)$ or $VariablePosition(e, var, pos)$
Message duplication	$TransClient(v)$, $Idempotent(v)$	$Duplicates(e)$
Checkpointing	-	$Checkpointed(e)$

Table 2: Support for design aspects by individual methods.

Aspect	ML/P	DL/P	ML/CP	U/CP
Channel content	Message level	Variable level	Message level	Variable level
Channel types	Yes	-	Yes	Yes
Service deployment	Coarse level	-	Coarse level	Finest level
Throughput	Yes	-	Yes	Yes
Availability	Yes	-	Yes	Partially
Monitoring	Yes	-	Yes	Yes
Message ordering	Yes	-	Yes	Yes
Message format	Yes	-	Yes	Yes
Message duplication	-	-	Yes	-
Checkpointing	-	-	-	Yes

An integration problem to be solved is transformed into input data for an action-based planner, written using Planning Domain Description Language (PDDL). The planner is then executed and its output, i.e. the plan, is transformed to an integration solution graph representation.

Integration problem encoding works as follows: Channels that are present in the integration solution being created correspond to the planner's states of the world – or, more exactly, states of the world reflect cut-sets of specific cuts of the solution graph (cuts that correspond to states in processing of messages). The state of the world changes as individual services and other components of the solution process their incoming message flow(s) and generate their outgoing one(s): an operator corresponding to such a component 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 then corresponds to the input flow(s) entering the solution, and the goal state corresponds to the expected output flow(s).

A state of the world is a conjunction of literals. Most important of these literals are those that characterize channels (message flows). We directly map properties of channels into these literals. As an example, let us consider the ML/P method. In this method we work with channel properties described by functions $Content$, $Format$, $Ordered$, $OrderMarked$, $Monitored$, and $ChannelType$. These functions are mapped to the following arguments of the `message` predicate we use to describe message flows: `Content`, `Format`, `Ordered`, `OrderMarked`, `Monitoring`, and `Channel`. In a similar way we map $VariablePosition$ function to the `data` predicate in the DL/P method.

Concerning properties of solution graph vertices, these are mapped to the planning operators names, and possibly also their parameters.

Finally, rules that describe a correct solution are transferred into operators' preconditions and effects. The challenge is to find a representation of states of the world, a set of operators, and formulation of their preconditions and effects that would cover relevant properties and rules of the domain of messaging-based integration solutions and still would be processable by available planners in a reasonable time.

As an example, let us consider the message flow present at a starting point of integration solution corresponding to abstract design depicted in Fig. 1. In the ML/P method it would be represented by the following atom:[1]

```
(message c_order xml ord ord_not_marked
not_mon ch_queue flow_1)
```

An operator corresponding to `CheckCredit` service deployed in fourth mode (multiple hosts), with monitoring required, written using PDDL is shown in Fig. 3.

Message flow present at its output would be, then, represented by the following atom:

```
(message c_order_crinfo xml unord ord_not_marked
mon_req ch_memory4 flow_1)
```

As for the methods' output side, the plan (a sequence of

[1]We are using the PDDL syntax here, see [8].

```
(:action CheckCredit_PL4_M
 :parameters (?ordered - t-ord ?orderMarked - t-ordm
              ?channel - t-ch ?flowID - t-flow)
 :precondition
   (and
     (message c_order xml ?ordered ?orderMarked mon ?channel ?flowID)
     (acceptable_input_channel_for_PL4 ?channel)
   )
 :effect
   (and
     (not (message c_order xml ?ordered ?orderMarked mon ?channel ?flowID))
     (message c_order_crinfo xml unord ?orderMarked mon_req ch_memory4 ?flowID)
   )
  )
```

Figure 3: Planning operator corresponding to `CheckCredit` service.

actions, i.e. operators applied) represents an integration solution we are looking for. Actions in the plan correspond to the vertices of the solution graph and action dependencies (in the form of predicate formulas) correspond to solution graph edges. The transformation from the plan to integration solution description is straightforward. For more information on ML/P and DL/P methods, please see [18] and [15], respectively.

6. Methods Based on Constraint Programming

The other two methods (ML/CP and U/CP) are based on a transformation of an integration problem into a constraint satisfaction problem (CSP) [14] in such a way that a solution of the CSP can be transformed back into a solution for this integration problem.

Principles of the transformation are the following:

1. Given the integration problem, we create a skeleton of the solution graph.
2. For each vertex and edge of this graph we create a set of CSP variables: in principle, value of each property function, applied to this vertex or edge, is represented by one or more variables.
3. Each design rule is represented using one or more constraints over respective CSP variables.

When creating a skeleton of the solution graph, the basic assumption is: The integration solution graph strongly resembles the control flow structure, which the solution has to implement – in particular, between each two business services connected by a control dependency in the abstract model, there is a message flow in the integration solution. The rationale behind this assumption is that sending of a message is the basic mechanism used to implement control and data flow between solution components, so it is natural to create a message flow between any two services connected by a control flow. In addition, we insert a sufficient number (controlled by the user) of empty slots that will potentially contain auxiliary integration services between each two services connected by a messaging channel.

After applying this rule, the skeleton of the integration solution graph would look like that presented at Fig. 4. Nodes (boxes, circles, and diamonds) are vertices of the graph and connections between them are edges. Circles with question marks represent slots for integration services added as described above.

We have created several advanced optimization techniques, which are necessary to speed up the process of solving CSPs for larger integration problems. Perhaps the most important one is the partitioning of the problem, both horizontally – to sets of integration processes, and vertically – to sets of design issues. Other techniques include choosing the appropriate general heuristics for the selection of variables to be considered when searching for a solution, as well as creating our own custom ones (like first choosing `ChannelType` variables when solving *Channel types* design issue). By applying these techniques we were able to obtain adequate performance of the U/CP method, as shown in the next section.

7. Evaluation

All four methods have been implemented as research prototypes, using several existing planners [2, 4, 5, 9, 10, 12, 23, 33] and the JaCoP constraint programming library [30], respectively. They have been evaluated using a set of integration problems taken from the literature as well as from our experience from real-life integration project at Comenius University in Bratislava, as described in [15, 16, 18, 19, 20]. Here we summarize the most important findings.

7.1 Performance of Methods Based on Planning and Constraint Programming

First of all, we have found that our methods based on constraint programming perform much better than their planning-based counterparts. For an example, see the comparison of ML/P and ML/CP methods shown in Tab. 3.

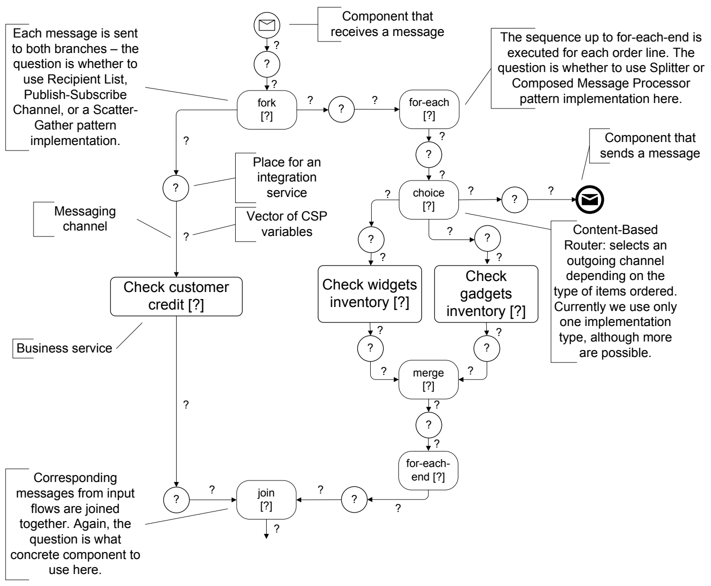
The *Problem description* column references the respective integration problem as it is present in [17]. The *Aspects* column shows which of the design issues the problem deals with, namely: monitoring (M), message formats (F), throughput (T), availability (A), message ordering (O), and duplicate messages elimination (D). The *Business services* and *Int. comp.* columns show the number of business services and integration components used in the optimal solution, respectively. The last three columns provide the CPU time needed to find a solution by the ML/P method (where applicable), and to find the optimal solution and to conclude that no better solution exists by the ML/CP method.

We assume that the better performance of methods based on constraint programming is due to the fact that these methods have to consider a smaller space of possible solutions (e.g. by limiting a number of integration services to be used, and by requiring an explicit specification of the control flow within the abstract design), as well as because we were able to provide advanced optimization techniques mentioned in the previous section. We believe that such techniques can be used in planning-based methods as well, albeit perhaps with significantly larger effort; however, without further research we cannot state anything for certain about it.

7.2 Evaluation of the U/CP Method

We have undertaken an extensive evaluation of our most advanced method, U/CP. In order to evaluate it we have created a set of eight integration scenarios. Two of them are taken from [11], the other six from a major integration project undertaken at Comenius University in Bratislava:

1. Widgets and gadgets order processing scenario (described above). [11]



Component that receives a message
Each message is sent to both branches – the question is whether to use Recipient List, Publish-Subscribe Channel, or a Scatter-Gather pattern implementation.
The sequence up to for-each-end is executed for each order line. The question is whether to use Splitter or Composed Message Processor pattern implementation here.
fork [?]
for-each [?]
Component that sends a message
Place for an integration service
choice [?]
Messaging channel
Vector of CSP variables
Content-Based Router: selects an outgoing channel depending on the type of items ordered. Currently we use only one implementation type, although more are possible.
Check customer credit [?]
Check widgets inventory [?]
Check gadgets inventory [?]
Business service
merge [?]
for-each-end [?]
Corresponding messages from input flows are joined together. Again, the question is what concrete component to use here.
join [?]

Figure 4: Graphical representation of a CSP constructed for the sample integration problem.

Table 3: Selected results of the evaluation of ML/CP and ML/P methods.

#	Problem description	Aspects	Business services	Integration comp.	ML/P (sec)	ML/CP, opt. solution (sec)	ML/CP, all solutions (sec)
1	WGRUS (D)	MFTA	5	10	0.43	0.17	0.44
2	WGRUS (J)	MFTAO	11	20	15.47	0.33	30.04
3	WGRUS (J)	MFTAOD	11	29	n/a	2.80	7,012.36
4	University (G)	MFTA	11	20	55.00	0.20	1.87

2. Loan broker. [11]
3. Transfer of data from academic information system and human records system to university's central database of persons.
4. Canteen menu web presentation.
5. Transfer of data about thesis and dissertations from academic information system (AIS) to external plagiarism detection system.
6. Transfer of data about defended thesis and dissertations as well as fulltexts of these works from AIS to university library information system.
7. Transfer of personal data from central database of persons to the information system of dormitories and canteens.
8. Transfer of students' admission confirmations from academic information system to central database of persons.

We have tried to answer the following questions:

Q1. Is the method usable, i.e.

a. Is it universal enough, i.e. can it be applied to a sufficiently wide set of problems?

b. Is it able to create integration solutions in reasonable time?

c. Does it create solutions that meet specified functional and non-functional requirements?

Q2. Is the method an improvement over existing approaches?

7.2.1 Applicability of the Method

We have answered *Q1a* by creating input data for our method for the above mentioned eight integration scenarios (besides others). We have found that the expressiveness of our control and data flow description language is largely sufficient. Only minor changes that would increase the practical usability of the language have been identified. Concerning the expressive power of the non-functional specification language, we have found it adequate, with some points to improve, like the ability to specify multiple usage profiles, global message ordering requirements, or message processing latency. These will be treated in the future.

7.2.2 Performance of the Method

Concerning the method's performance (question *Q1b*), we have conducted a set of experiments. Their results are summarized in Tab. 4.

Meaning of individual columns is the following: *P#* denotes the integration problem number. *Processes* column indicates the number of integration processes in the problem. *Components* shows the number of components (i.e. services, input/output directives, and control flow constructs elements) in the integration problem and solution, respectively. *Connections* denotes the number of

Table 4: Results of the U/CP method performance evaluation.

P#	Processes	Components	Connections	Variables	Containers	Best solution (sec)	All solutions (sec)
1	1	19 (25)	20 (26)	6	5	0.3	0.3
2	1	23 (47)	29 (53)	13	3	21.9	Timeout
3	23	125 (142)	129 (146)	85	6	30.4	30.5
4	1	8 (9)	7 (8)	7	1	0.8	0.8
5	1	9 (9)	8 (8)	9	1	0.9	0.9
6	9	74 (89)	91 (106)	67	1	4.4	4.4
7	1	22 (24)	23 (25)	9	1	0.3	0.3
8	1	18 (25)	19 (26)	10	1	0.6	0.6

connections (i.e. channels), again both in the integration problem and solution. *Variables* is the number of process variables present in the solution. *Containers* is the number of service containers the solution components can be deployed in. Finally, the *Best solution* column denotes the CPU time, measured in seconds, that was needed to find the optimal solution for a given integration problem, and the *All solutions* column gives the CPU time needed to conclude that no better solution exists. The *Timeout* value here means that the computation did not finish in allotted time of 600 seconds.

These results indicate that the method is performing well enough to be practically used. Moreover, we have verified, by code inspection and/or by performance testing, that the generated solutions have met stated functional and non-functional requirements (question $Q1c$).

7.2.3 Comparison to the Traditional Development of Integration Solutions

We have conducted a comparison of the development of integration solutions using our method to the development of such solutions using traditional tools, like Progress Sonic Workbench that we have used for several years.

We have found that our method addresses a major issue: the considerable complexity of messaging-based integration solutions that severely limits their understandability and maintainability. Solutions that are captured using our platform-independent, abstract control and data flow specification language are significantly more concise than solutions written using Progress Sonic-specific graphical-only language. As can be seen in Fig. 5, our method reduced the number of symbols necessary to implement an integration solution for a problem similar to #2 (Loan Broker) from 202 to 116 (i.e., by 43%). Likewise, the reduction for problem #4 (Canteen menu presentation) was from 82 to 39 (by 52%). Moreover, the character of the code has changed from highly technical, dealing e.g. with message parts content and formatting, to more conceptual, dealing with abstract notion of process variables instead. We expect to achieve a similar level of complexity reduction when comparing with other integration products, e.g. Apache Camel or FUSE Integration Developer; however, this has to be established experimentally yet. There are also other, less fundamental, areas of improvement, like removing the necessity of learning technical details of implementation platform concepts, better documentation and visualization of the integration solutions, and their automated deployment.

A drawback of using our method is that, although being quite universal, it uses a limited number of design constructs and makes concrete assumptions about the solutions being created. There are situations where this method would not find a solution as efficient or elegant as a developer would create "by hand". Experiments have shown a performance decrease of 41.3% in one particular test case (21.3% overall), due to some features missing in U/CP, in comparison to the native integration platform [19]. However, these experiments were aimed directly at measuring the performance of integration code and therefore used stub implementations of systems to be integrated. We expect that the performance decrease in integration scenarios involving real systems would be less significant. Nevertheless, we are working on adding identified missing features to the method and optimizing its run-time support components in order to enhance the performance of generated solutions.

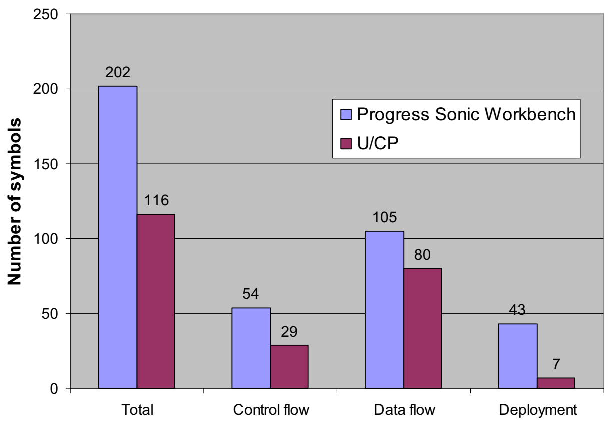


Figure 5: Number of symbols necessary to implement a sample scenario.

8. Conclusion

Overall goal of this thesis was to find a way of partially or fully automating the process of design and implementation of messaging-based integration solutions, in order to improve some of their characteristics. In order to fulfill it, we have formulated two hypotheses, mentioned in Section 3, and tried to accept or refute them.

As for the Hypothesis 1, we have constructed four methods (ML/P, DL/P, ML/CP and U/CP). All of them are based on our own abstract model of an integration solution using graphs with properties of their vertices and edges modeled as functions.

By evaluating these methods we have shown the following.

First of all, the process of creating detailed design of an integration solution can be partially or fully automated, given the abstract design, non-functional requirements, design goals and environment characteristics of such a solution.

Second, action-based planning can be used for designing integration solutions. Its use is advantageous in the sense that it does not require the developer to explicitly specify the control flow between individual services; it suffices to state their input/output requirements. On the other hand, experiments with the ML/P and DL/P methods have shown that (1) the time needed to find a suitable design is significantly longer than when using methods based on constraint satisfaction, (2) the number of design aspects that the planning-based methods were able to take into account is limited, and (3) the notion of solution optimality we were able to work with when using planning was rather coarse. The first two observations can be summarized in a way that our planning-based methods do not scale well with the problem size and the number of design aspects. However, we see a potential for improving these methods in the future, in particular by utilizing domain-specific knowledge within the planning process.

Third, constraint satisfaction can be used for designing integration solutions as well. Advantages of its use are:

1. U/CP method based on constraint satisfaction is able to construct integration solution designs quickly enough to be used as part of a design tool. We have implemented such a tool in the form of an Eclipse plugin and successfully used it to create several real-world integration solutions.
2. A transformation of abstract design rules into CSP constraints is more straightforward than their transformation to operators' preconditions and effects within our planning-based methods. This enabled us to implement a rich set of design aspects and metrics within the U/CP method, which could be further extended as necessary. Design metrics also provide a very flexible way of defining the criterion of solution optimality.
3. We were able to implement a partitioning scheme for the design problem. This provided us with a good performance as well as scalability with regards to the problem size and the number of design aspects considered.

Therefore, we have confirmed Hypothesis 1.

Concerning Hypothesis 2: Based on our subjective assessment, as well as on two case studies, which we have prepared, we can say that the source code that has to be created for the U/CP method is significantly more concise than the source code written directly for an integration platform. We can reasonably assume that more concise source code can positively influence other properties of the solution, namely the effort needed to create and maintain it, as well as the number of defects present.

Therefore, the main goal of this dissertation has been fulfilled.

8.1 Future work

As for future research, there are a number of questions worth looking at. We can roughly divide them to "more conceptual" and "more technical" groups. Among "more technical" future work directions there are, e.g. (1) evaluating the benefits of using the U/CP method more exactly, using for example quantitative measurements of the effort needed to create and maintain integration solutions, or (2) implementing additional design aspects, like message latency or global message sequencing.

Among conceptual questions there are, for example:

1. Is it possible to apply the approach used in our method in areas other than messaging-based integration solutions, namely e.g. in technical design of web service compositions?
2. Would it be helpful to try to combine planning and constraint programming together? Or, would other problem-solving techniques be more useful?
3. Is it possible to automate other aspects of integration solutions development, for example creation of transformation services – or to integrate our methods with existing frameworks in this area, like the BIZYCLE Model-Based Integration framework [1]?

Acknowledgements. This work was partially supported by the Slovak Research and Development Agency (under the contract No. APVV-0391-06), the Scientific Grant Agency of Slovak Republic (grant No. VG1/0508/09), the Cultural and Educational Grant Agency of the Slovak Republic (grant No. KEGA 345-032STU-4/2010), and it is the partial result of the Research & Development Operational Programme for the projects Support of Center of Excellence for Smart Technologies, Systems and Services II, ITMS 25240120029, and Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] H. Agt, G. Bauhoff, M. Cartsburg, D. Kumpe, R.-D. Kutsche, and N. Milanovic. Metamodeling foundation for software and data integration. In J. Yang, A. Ginige, H. C. Mayr, and R.-D. Kutsche, editors, *UNISCON*, volume 20 of *Lecture Notes in Business Information Processing*, pages 328–339. Springer, 2009.

[2] B. Bonet and H. Geffner. Heuristic Search Planner 2.0. *AI Magazine*, 22(3):77–80, 2001.

[3] B. Druckenmüller. Parameterization of EAI patterns. Master's thesis, University of Stuttgart, 2007. (in German).

[4] S. Edelkamp and S. Jabbar. MIPS-XXL : Featuring external shortest path search for sequential optimal plans and external branch-and-bound for optimal net benefit. In *6th International Planning Competition Booklet, Sydney, Australia, Sept. 2008*, 2008.

[5] S. Edelkamp and P. Kissmann. Optimal symbolic planning with action costs and preferences. In *Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 1690–1695, 2009.

[6] R. Z. Frantz. Runtime system, 2011. Retrieved February 14, 2011, from http://www.tdg-seville.info/rzfrantz/Runtime+System.

[7] R. Z. Frantz, R. Corchuelo, and J. González. Advances in a DSL for application integration. In *Actas del Taller de Trabajo Zoco'08 / JISBD Integración de Aplicaciones Web*, pages 54–66, 2008.

[8] A. Gerevini and D. Long. BNF description of PDDL3.0, 2005. Retrieved August 8, 2011, from http://zeus.ing.unibs.it/ipc-5/bnf.pdf.

[9] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, 20(1):239–290, 2003.

[10] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14(1):253–302, 2001.

[11] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Pearson Education, 2004.

[12] H. Kautz and B. Selman. SatPlan: Planning as satisfiability. In *Abstracts of the 5th International Planning Competition*, 2006.

[13] P. Kolb. Realization of EAI patterns in Apache Camel. Student research project, University of Stuttgart, 2008.

[14] M. Mach and J. Paralič. *Problems with Constraints: From Theory to Programming*. elfa, 2000. (in Slovak).

[15] P. Mederly. Semi-automated design of integration solutions: How to manage the data? In *6th Student Research Conference in Informatics and Information Technologies Proceedings*, pages 241–248. Slovak University of Technology in Bratislava, 2010.

[16] P. Mederly. A method for creating messaging-based integration solutions and its evaluation. *Information Sciences and Technologies Bulletin of the ACM Slovakia*, 3(2):91–95, 2011.

[17] P. Mederly and M. Lekavý. Report on evaluation of the method for construction of messaging-based enterprise integration solutions using AI planning, 2009. Retrieved August 8, 2011, from http://www.fiit.stuba.sk/~mederly/evaluation.html.

[18] P. Mederly, M. Lekavý, M. Závodský, and P. Návrat. Construction of messaging-based enterprise integration solutions using AI planning. In *Preprint of the Proceedings of the 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009, Krakow, Poland, October 12-14, 2009*, pages 37–50. AGH University of Science and Technology, 2009.

[19] P. Mederly and P. Návrat. Pipes and filters or Process manager: which integration architecture is "better"? In *Datakon 2011: Proceedings of the Annual Database Conference*. To appear (in Slovak).

[20] P. Mederly and P. Návrat. Construction of messaging-based integration solutions using constraint programming. In *Lecture Notes in Computer Science Vol. 6295: Advances in Databases and Information Systems: 14th East European Conference, ADBIS 2010 Novi Sad, Serbia, September 20-24, 2010 Proceedings*, pages 579–582. Springer, 2010.

[21] C. Mierzwa. Architecture of ESBs in support of EAI patterns. Master's thesis, University of Stuttgart, 2008. (in German).

[22] N. Milanovic, M. Cartsburg, R.-D. Kutsche, J. Widiker, and F. Kschonsak. Model-based interoperability of heterogeneous information systems: An industrial case study. In R. F. Paige, A. Hartman, and A. Rensink, editors, *ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*, pages 325–336. Springer, 2009.

[23] D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20(1):379–404, 2003.

[24] M. P. Papazoglou and W.-J. van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *VLDB Journal*, 16(3):389–415, 2007.

[25] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.)*. Prentice Hall, 2003.

[26] T. Scheibler and F. Leymann. From modelling to execution of enterprise integration scenarios: the GENIUS tool. In *Kommunikation in Verteilten Systemen (KiVS)*, pages 241–252. Springer, 2009.

[27] T. Scheibler, R. Mietzner, and F. Leymann. EAI as a service - combining the power of executable EAI patterns and SaaS. In *EDOC*, pages 107–116. IEEE Computer Society, 2008.

[28] T. Scheibler, R. Mietzner, and F. Leymann. EMod: platform independent modelling, description and enactment of parameterisable EAI patterns. *Enterprise IS*, 3(3):299–317, 2009.

[29] H. A. Sleiman, A. W. Sultán, R. Z. Frantz, and R. Corchuelo. Towards automatic code generation for EAI solutions using DSL tools. In A. Vallecillo and G. Sagardui, editors, *JISBD*, pages 134–145, 2009.

[30] R. Szymanek. JaCoP - Java constraint programming solver, 2011. Retrieved August 8, 2011, from http://www.jacop.eu/.

[31] K. Umapathy and S. Purao. Exploring alternatives for representing and accessing design knowledge about enterprise integration. In C. Parent, K.-D. Schewe, V. C. Storey, and B. Thalheim, editors, *ER*, volume 4801 of *Lecture Notes in Computer Science*, pages 470–484. Springer, 2007.

[32] K. Umapathy and S. Purao. Representing and accessing design knowledge for service integration. In *IEEE SCC*, pages 67–74. IEEE Computer Society, 2008.

[33] Z. Xing, Y. Chen, and W. Zhang. MaxPlan: Optimal planning by decomposed satisfiability and backward reduction. In *Proceedings of Fifth International Planning Competition, International Conference on Automated Planning and Scheduling (ICAPS'06)*, pages 53–56, 2006.

Selected Papers by the Author

P. Mederly, P. Návrat. Construction of messaging-based integration solutions using constraint programming. In *Lecture Notes in Computer Science Vol. 6295: Advances in Databases and Information Systems: 14th East European Conference, ADBIS 2010 Novi Sad, Serbia, September 20-24, 2010 Proceedings*, pages 579–582. Springer, 2010.

P. Mederly, M. Lekavý, M. Závodský, P. Návrat. Construction of messaging-based enterprise integration solutions using AI planning. In *Preprint of the Proceedings of the 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009, Krakow, Poland, October 12-14, 2009*, pages 37–50. AGH University of Science and Technology, 2009.

P. Mederly, M. Lekavý, P. Návrat. Service adaptation using AI planning techniques. In *Proceedings of the 2009 Fifth International Conference on Next Generation Web Services Practices, NWeSP 2009, 9-11 September 2009, Prague, Czech Republic*, pages 56–59. IEEE Computer Society, 2009.

P. Mederly, G. Pálos. Enterprise service bus at Comenius University in Bratislava. In *Proceedings of EUNIS 2008 "VISION IT - Visions for use of IT in higher education", Aarhus, June 23-27, 2008*, page 127, 2008.

P. Mederly and P. Návrat. Pipes and filters or Process manager: which integration architecture is "better"? In *Datakon 2011: Proceedings of the Annual Database Conference*. To appear (in Slovak).

P. Mederly, P. Návrat. Automated design of messaging-based integration solutions. In *Datakon 2010: Proceedings of the Annual Database Conference, October 16-19, 2010, Mikulov, Czech Republic*, pages 121–130. University of Ostrava, 2010. (in Slovak).

P. Mederly. Towards automated system-level service compositions. In *WIKT 2008, 3rd Workshop on Intelligent and Knowledge Oriented Technologies Proceedings*, pages 101–104. Slovak University of Technology in Bratislava, 2009.

P. Mederly. A method for creating messaging-based integration solution and its evaluation. *Information Sciences and Technologies Bulletin of the ACM Slovakia*, 3(2):91–95, 2011.

P. Mederly. Semi-automated design of integration solutions: How to manage the data? In *6th Student Research Conference in Informatics and Information Technologies Proceedings*, pages 241–248. Slovak University of Technology in Bratislava, 2010.

P. Mederly. Towards a model-driven approach to enterprise application integration. In *5th Student Research Conference in Informatics and Information Technologies Proceedings*, pages 46–53. Slovak University of Technology in Bratislava, 2009.