A Method for Creating Messaging-Based 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
In this paper we present a method for creating messaging-based integration solutions. Input of this method consists of an abstract specification of control and data dependencies between components, specification of non-functional requirements, design goals, and target platform properties. The method produces a solution that meets specified requirements, tailored for the given platform (if such a solution exists). The output is in the form of a design as well as executable code. The evaluation has been carried out using a method prototype implementation, solving a set of integration problems taken from the literature as well as from real-life software projects.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; I.2.2 [**Artificial Intelligence**]: Automatic Programming —*Program transformation*

Keywords
enterprise application integration, messaging, non-functional requirements, constraint programming

1. Introduction
Application integration, aiming to enable cooperation of independently developed applications in an organization, is a significant issue in enterprise computing [2]. One of suitable approaches to creating integration solutions (i.e. software systems that connect individual applications) is *messaging*. Its main idea is that components of an integration solution communicate by passing messages, mostly in asynchronous way, using a kind of messaging middleware [2].

There are tools that enable developers to create such messaging-based integration solutions - for example, Progress Sonic ESB, Apache Camel, or Mule ESB[1]. These tools free developers from writing low-level code in traditional programming languages, replacing it by parameterizing pre-existing components (mostly adapters, business and integration services) and connecting them via appropriate channels.

However, despite existence of these powerful tools, development of integration solutions is still an expensive undertaking requiring highly-skilled software developers. Moreover, even though the tools use similar building blocks, rooted in enterprise integration patterns [2], there is no standard for describing messaging-based integration solutions yet. The result is that solutions are not portable among these tools (platforms).

Recently there have been several attempts in the academic community to cope with the problems mentioned above, for example [5, 6]. Their common trait is that they are based on model-driven development: a developer creates a platform-independent design model of an integration solution, chooses a target platform and the tool then transforms this model to executable code for the chosen platform. This solves the problem of portability among integration platforms, and partially makes the integration solution creation more efficient by raising the level of abstraction from the programming or configuration language of a particular tool to more abstract design-level language of the method.

Even so, some issues remain. In general, these works do not take into account non-functional requirements posed on an integration solution, like throughput, availability, message processing latency, and so on. A developer has to design such a solution "manually", reflecting the knowledge of details of selected integration platform in the platform-independent models. Those, then, become - at least partially - dependent on a chosen platform, and have to be revised after a platform is changed.

[*]Doctoral degree study programme in field Software Engineering. Supervisor: Professor Pavol Návrat, Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, STU in Bratislava. Work described in this paper was presented at the 7th Student Research Conference in Informatics and Information Technologies IIT.SRC 2011.

Mederly, P. A Method for Creating Messaging-Based Integration Solutions. Information Sciences and Technologies Bulletin of the ACM Slovakia, Special Section on Student Research in Informatics and Information Technologies, Vol. 3, No. 2 (2011) 91-95

[1]ESB is an acronym for Enterprise Service Bus, a kind of messaging-based integration tool or platform [1].

Therefore we have created a method that aims to produce a design of an integration solution, based on specification of control and data dependencies between its components, non-functional requirements, design goals, and target platform properties and then translates this design into executable code. The method builds on our previous works in this area, e.g. [3]; however, in comparison to them, it is much more advanced in the direction of practical usability. In this paper we shortly describe the method and its evaluation, carried out by using a prototype implementation to solve a set integration problems taken from the literature as well as from real-life software projects.

The remainder of this paper is organized as follows: Section 2 briefly characterizes our method for semi-automated creation of integration solutions. Section 3 is devoted to the evaluation of the method. Section 4 contains the conclusion and some ideas on future work.

2. A method for designing integration solutions

Input of our method consists of the following main parts:

1. Description of control and data dependencies between integration solution components.
2. Specification of non-functional requirements.
3. Design goals.
4. Description of the target environment.

Control dependencies constrain possible sequencing of integration solution components invocations. At this level of abstraction, these components are usually (1) services that either provide access to participating applications or transform data, and (2) instructions for receiving and sending messages from/to appropriate messaging channels. Our method works with the following control flow constructs: sequence, decision, fork (optionally with a join), for-each[2], and wait-for[3].

Data dependencies describe the flow of data among individual components. In messaging-based integration solutions, each component consumes one or more messages at its input side, and then produces zero or more messages at its output side. Each of these messages can contain one or more data elements that we call *process variables.* Our method expects to get an abstract specification of data dependencies in the form of process variables consumed and produced by each solution component. Then it finds the optimal positioning of these variables into physical messages that flow within the solution.

An example of control and data flow specification is shown in Figure 1, using a UML activity diagram. It describes our first case study, an order processing scenario of the fictitious retailer “Widgets and Gadgets ’R Us” taken from [2]. This integration solution has to process customers’ orders: for each incoming order it ensures that the customer’s credit standing as well as product 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 split to order lines and the availability of product in each order line is checked against appropriate system. Orders containing invalid order lines are rejected.

Please note that connections between components denote control dependencies only. Data dependencies are shown using activities’ input and output pins labeled by related input and output process variables.

Non-functional requirements can be specified in various areas, for example:

1. *Throughput* of the solution: a developer can specify how many messages per time unit the solution must be able to process. The method then finds a deployment of services into containers/threads such that this throughput would be achieved, knowing performance characteristics of individual services in particular containers.
2. *Availability* of the solution: in the current version of the method the developer can specify additional constraints on deployment of selected components, i.e. that a component has to be deployed on at least two distinct hosts (leading to increased availability).
3. *Message ordering*: sometimes the original order of messages has to be preserved, even in the presence of multithreading and/or alternative message flows.
4. *Monitoring*: messages flowing within an integration solution are generally not visible from the outside. Yet sometimes we need to monitor them at specified points, by using e.g. Publish/Subscribe Channels or the Wire Tap integration pattern.

Design goals deal with the question of what we want to optimize. Examples are number of solution components, number of messages and/or bytes transferred through messaging middleware or in all channels, number of message format transformations, host load, or number of threads.

The *target environment* is described with respect to integration services and communication channels it provides, properties of services (like throughput, deployment constraints, input/output characteristics, or use of system resources), and so on.

The method itself works by translating the integration problem into constraint satisfaction problem (CSP), solving the CSP and interpreting its solution as an integration solution design. From this design concrete executable code is more-or-less directly generated.

The transformation of an integration problem into CSP is the core of our method. The CSP is constructed so that each component and each channel in the solution is described by a vector of CSP variables and their values. But how do we know, in advance, what components and channels will the solution contain? Fortunately, it seems

[2]Executes a defined sequence of component invocations, once for each part of a given process variable.

[3]An arbitrary control dependency between two components.

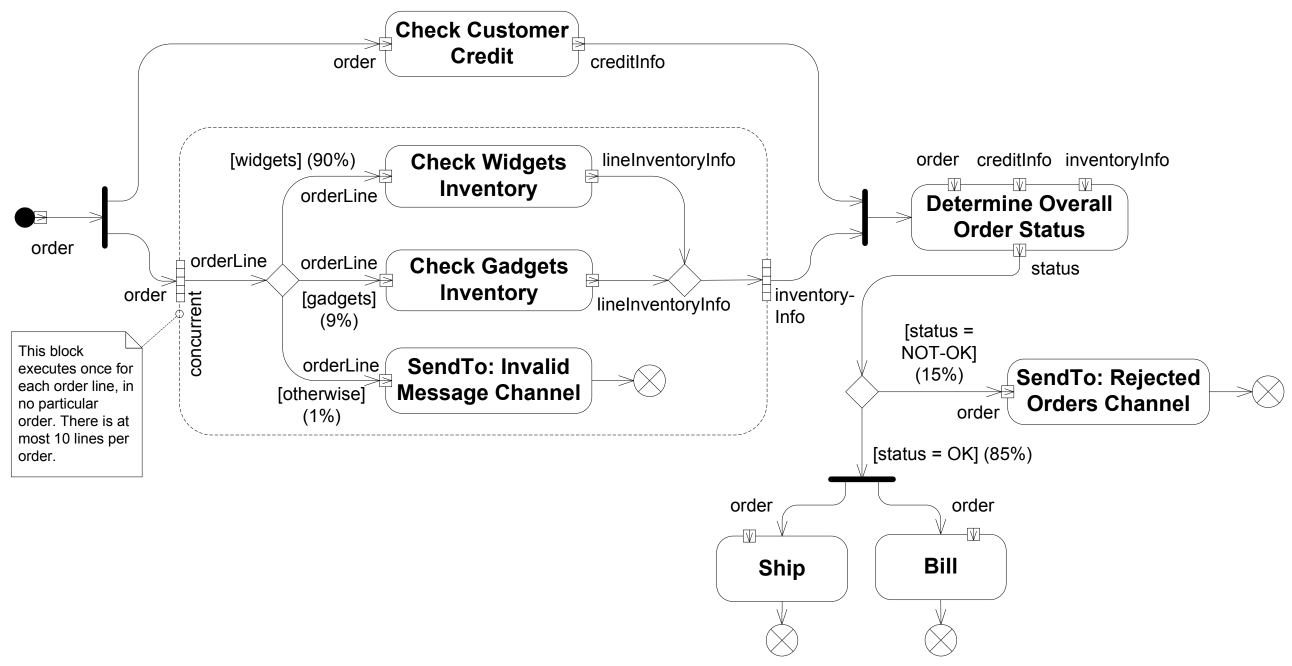

Figure 1: An example of control and data flow specification.

that we can safely assume that the basic structure of the solution corresponds to the structure of the control flow graph, with several additions[4]. These are namely: (1) between each two elements of the abstract control flow specification there can be inserted one or more places for integration services, and (2) if there are places of the solution that have the need of some data that is not possible to provide using existing channels, the method creates new places for channels.

So, when constructing the CSP, our method first creates an integration solution structure using the above mentioned idea, and then formulates a CSP for it. The CSP formulation consists of variables creation and constraints imposing. Variables corresponding to *components* reflect mainly the concrete implementation of a given control flow construct, integration service type that will be used, deployment of a component (e.g. number of threads for individual containers) and its throughput. Variables corresponding to *channels* reflect mainly the data transported within messages in that channel, their positions in the message (e.g. header, body, attachment), their format (e.g. XML, CSV, fixed-length records, etc.), type of channel (in-memory, messaging middleware point-to-point, publish/subscribe, etc.), state of messages with respect to monitoring, ordering, or checkpointing. Besides these, there are variables created for evaluating individual design goals, e.g. the cost of using a component or a channel, CPU load of a given host, and so on.

As an example, by applying the above algorithm to the control flow graph shown in Figure 1 we get a CSP, part of which is shown in Figure 2. The CSP is visualized as a graph whose nodes correspond to integration solution components (or potential components) and edges correspond to channels connecting them[5]. Components are of various types, indicated by different shapes whose meaning is explained in Figure 2. Question marks denote vectors of CSP variables.

The final step in CSP formulation is imposing *constraints* over CSP variables. In our case constraints reflect either general properties of the domain of messaging-based integration solutions (e.g. "if a component executes in more threads, message flow at its output is not ordered"), properties of concrete integration platform (e.g. "publish/subscribe channel can be used as an input for a component running in one container only"), or properties of components specific to given integration problem (e.g. "service S_1 has to be deployed in at least two containers" or "service S_2 in a process P expects to have variable v_1 at input and variables v_2 and v_3 at output").

The CSP can be created either for the solution as a whole, or, for performance reasons, the problem can be divided horizontally (to individual integration processes or process groups) and/or vertically (solving design aspects in related groups). Although the optimal solution found for partitioned problem is not guaranteed to be the optimal with respect to the original (unpartitioned) problem, our experiments have shown the difference is not significant.

The method can be easily extended by adding new design aspects, materializing in new types of non-functional requirements and/or components' attributes, CSP variables and constraints.

3. Evaluation

In order to evaluate our method we have created a set of integration scenarios. Two of them are taken from [2], the

[4]A rationale for this assumption is that the basic way of implementing a control dependency in messaging-based solutions is via sending messages through a channel that connects the respective components, as can be seen e.g. from the patterns published in [2].

[5]For simplicity we do not show potential channels (i.e. places where channels can be possibly created) here.

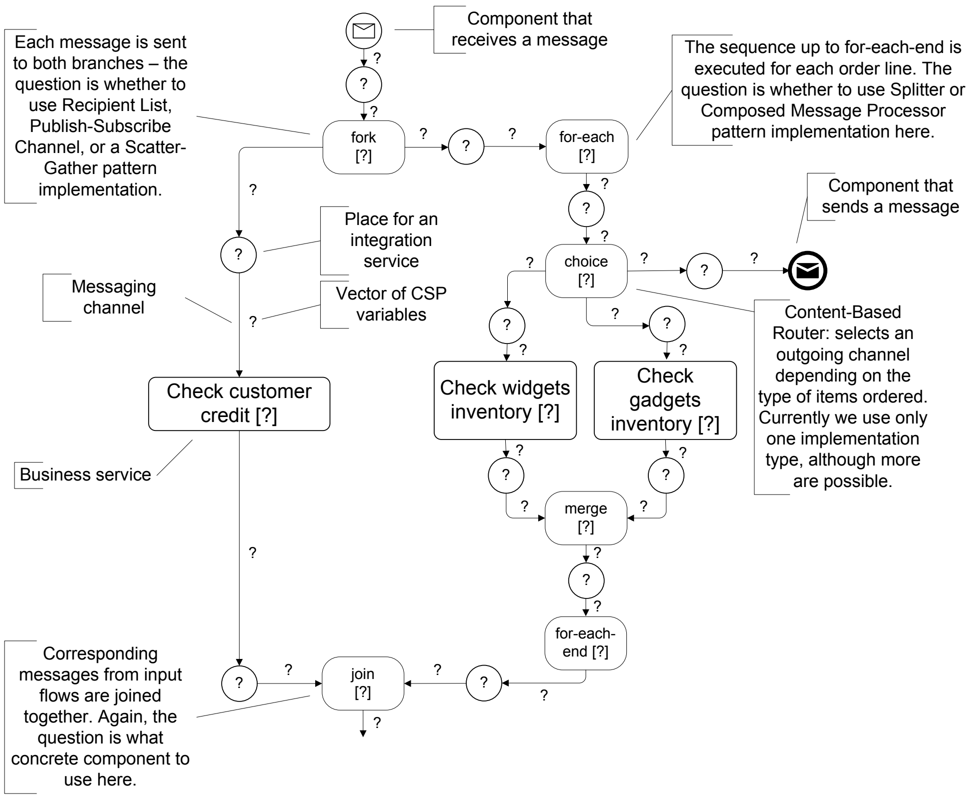
Component that receives a message
Each message is sent to both branches – the question is whether to use Recipient List, Publish-Subscribe Channel, or a Scatter-Gather pattern implementation.
The sequence up to for-each-end is executed for each order line. The question is whether to use Splitter or Composed 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 2: Graphical representation of a CSP constructed for the sample integration problem.

other three from a major integration project undertaken at Comenius University in Bratislava [4]:

1. Widgets and gadgets order processing scenario (described above) [2].
2. Loan broker [2].
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 to external plagiarism detection system.

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?

3.1 Results in the area of the method applicability

We have answered *Q1a* by creating input data for our method for the above mentioned five integration scenarios (besides others). We have found that the expressiveness of our control and data flow description language is largely sufficient; only minor additions, like optional variable presence and conditional fork branch execution, would be desirable. These are generally of a technical nature and will be implemented in the near future. Concerning the expressive power of the non-functional specification language, we have found it adequate, with the following points to improve:

1. We have found we need the ability to distinguish between hard limits (e.g. message sizes, number of iterations in for-each cycles, and so on) and usual (average) values.
2. We need the ability to specify multiple usage profiles, for example a profile corresponding to continuous processing of change events versus a profile for one-time processing of nightly full synchronization data set.
3. As for message ordering, it seems that although present specification constructs are useful, there are situations that require the notion of global message ordering, i.e. ordering dealing with more than one integration process.
4. It would be useful to be able to constrain or optimize the message processing latency.

The first two issues are of simple technical nature. The last two ones are more intricate and remain to be solved in the future.

Table 1: Results of the 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

3.2 Results in the area of the method performance

Concerning the method's performance (question $Q1b$) we have conducted a set of experiments. Their results are summarized in Table 1.

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 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, both by code inspection and by performance testing, that the generated solutions have met stated functional and non-functional requirements (question $Q1c$).

3.3 Comparison to the traditional development of integration solutions

We have conducted a qualitative 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 [4].

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 much more concise than solutions written using Progress Sonic-specific graphical-only language. (We are currently working on the quantification of this improvement.) This complexity reduction is relevant also when comparing with other integration products, e.g. Apache Camel or FUSE Integration Developer[6].

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".

[6]There are also other, less fundamental, areas of improvement, but we have no space to cover them here.

4. Conclusions

In this paper we have presented a method for semi-automatic construction of messaging-based integration solutions. Using five application scenarios we have shown that it is able to construct integration solutions that have specified functional and non-functional properties.

In the future we would like to enhance the usability of the method by providing a textual, programmer-friendly, domain-specific language for describing integration solutions' abstract designs. Besides executable code we plan to generate detailed design documentation as well. We would like to evaluate the method in more depth using quantitative comparison with the traditional development approach, as well as by applying it to additional scenarios taken from several real projects. We plan to use experiences we will gain in the further development of the method.

Acknowledgements. This work was partially supported by the Scientific Grant Agency of Slovak Republic, grant No. VG1/0508/09, and it is the partial result of the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] D. A. Chappel. *Enterprise Service Bus*. O'Reilly Media, 2004.

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

[3] 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, 2010 Proceedings*, pages 579–582. Springer, 2010.

[4] P. Mederly and G. Pálos. Enterprise Service Bus at Comenius University in Bratislava. In *Proc. of EUNIS 2008*, page 127, 2008.

[5] 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.

[6] H. Sleiman, A. Sultán, and R. Frantz. Towards automatic code generation for EAI solutions using DSL tools. In *XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2009), San Sebastián, Spain, September 8-11, 2009*, pages 134–145, 2009.